Deep Learning with Medical Images - LAB

Roman Zeleznik, M.S.
AAPM - American Association of Physicists in Medicine
Annual Meeting 2018

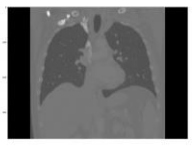
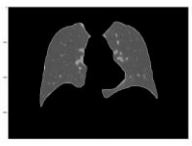
DANA-FARBER CANCER INSTITUTE | HARVARD MEDICAL SCHOOL | Computational Imaging & Bioinformatics Laboratory

Objectives

- Introduction to Deep Learning
- Lung segmentation in 56 test CT images using different deep learning networks
- Framework: TensorFlow[4], Keras[5], Python[6]
- Data: Lung Image Database Consortium image collection - LIDC-IDRI[1,2,3]
- Network types:
  - Fully Convolutional Network (FCN)[8]
  - U-Net[7]
- All tasks online using Jupyter Notebooks

Deep Learning with Medical Images - LAB

Lung segmentation in CT images


Details

- Each task in a separate Jupyter-Notebook
- Detailed description in each notebook
- Code already written – Fill in blanks (No programming skills needed)
- Due to time limitation, networks already trained. We can directly test them

Data Set

- 577 DICOM images with lung segmentation masks
- Training set: 289 images
- Validation set: 144 images
- Test set: 56 images
- Random subset of LIDC-IDRI data set
- Lung masks where automatically segmented – sufficiently accurate for training a network and further processing steps (e.g. nodule detection)

Tasks

- Connect to the online course
- Pre-process medical image data
- Segment lung with several Fully Convolutional Networks
- Segment lung with U-Net
- Compare results

References

1) Armato III, Samuel G., McLennan, Geoffrey, Bidaut, Luc, McNitt-Gray, Michael F., Meyer, Charles R., Reeves, Anthony P., ... Clarke, Laurence P. (2015). Data From LIDC-IDRI. The Cancer Imaging Archive.
2) Armato SG III, McLennan G, Bidaut L, McNitt-Gray MF, Meyer CR, Reeves AP, Zhao B, Aberle DR, Henschke CI, Hoffman EA, Kazerooni EA, MacMahon H, van Beek EJR, Yankelevitz D, et al.: The Lung Image Database Consortium (LIDC) and Image Database Resource Initiative (IDRI): A completed reference database of lung nodules on CT scans. Medical Physics, 38: 915–931, 2011.
3) Clark K, Vendt B, Smith K, Freymann J, Kirby J, Koppel P, Moore S, Phillips S, Maffitt D, Pringle M, Tarbox L, Prior F. The Cancer Imaging Archive (TCIA): Maintaining and Operating a Public Information Repository, Journal of Digital Imaging, Volume 26, Number 6, December, 2013, pp 1045-1057.
4) Tensorflow
5) Keras
6) Python
7) Ronneberger, O., Fischer, P. & Brox, T. U-Net: Convolutional Networks for Biomedical Image Segmentation. in Lecture Notes in Computer Science 234–241 (2015)
8) Long, J., Shelhamer, E. & Darrell, T. Fully convolutional networks for semantic segmentation. in 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2015). doi:10.1109/cvpr.2015.7298965

Server connection

URL: http://54.174.35.141:8000

UserName: Part before @ of email address used to register
e.g.: **examplename**@example.com
e.g.: **example.name**@example.com
e.g.: **example_name**@example.com

Password: AAPM badge number (6 digit number)

Server connection - Backup

URL: http://34.232.217.66:8000

UserName: Part before @ of email address used to register
e.g.: **examplename**@example.com
e.g.: **example.name**@example.com
e.g.: **example_name**@example.com

Password: AAPM badge number (6 digit number)

Welcome to the 2018 AAPM *Deep Learning with Medical Images* tutorial.

This one hour tutorial will give an overview of how Deep Learning can be applied to medical images. One important application of Deep Learning with medical images is image segmentation. This is used for many studies and often represents the basis for follow-up projects. This session will show you how to prepare medical images, test deep learning networks and finally how to evaluate the results.

If your are new to Jupyter notebooks, there are two different categories/boxes within the notebook. The first one are comments boxes and the second one are code boxes. To run code written in a code box simply click into the box and press 'Shift + Enter'. The results will then be shown below the code box. Optionally you can also click on the play rectangle, left of each code box (in some browsers this button only appears if the mouse hoovers over the code cell).

To verify that Jupyter is set up correctly and to follow an old coding tradition, we execute the first and simple line of code, greeting the world, respectively the conference today.

In [1]:
```
print 'Hello AAPM'
```

```
Hello AAPM
```

This session won't be a programming tutorial. Hence the code is already written requiring only its execution plus occasional parameter inserts.

Good luck and have fun with the following deep learning exercises!

Data preparation

Medical images are typically stored in a DICOM format. In this exercise we are using images from the LIDC data set [Link](). Each DICOM file contains one 2D image slice together with information about image size, resolution, location, patient information and much more. All DICOM files combined give a 3D image. To use these images for deep learning, we need to convert them. Furthermore, especially in big data sets, images can differ in size, resolution, scanner and so on. Therefore we also have to normalize them to match each other.

Note: If there are warnings shown after importing the libraries, we can ignore them. If you run the cell a second time, these warnings should be gone.

The reason for these warings are normally due to different library versions used. Python libraries change quite fast and can make it sometimes difficult to avoid warnings.

In [1]:
```
import os, pydicom, collections, multiprocessing, glob, shutil, csv, random
import numpy as np
import scipy.ndimage as ndimage
import matplotlib.pyplot as plt
```

DICOM data typically contains one folder per patient. Each patient folder contains one or more study folders and each study folder contains one one more series folder where the DICOM files are located. Therefore we need to find every series folder and save its path in a list for further processing. In this exercise we will load the test data from our network.

In [2]:
```
inputFolder = 'data/LIDC Test DICOM'
dicomFolders = []
patientFolders = glob.glob(inputFolder+'/*')
for patient in patientFolders:
    studyFolders = glob.glob(patient+'/*')
    for study in studyFolders:
        seriesFolders = glob.glob(study+'/*')
        for series in seriesFolders:
            dicomFolders.append(series)
print 'Found', len(dicomFolders), 'image folders for testing.'
```

```
Found 3 image folders for testing.
```

We now can process each folder. As data pre-processing can take some time and often needs a lot of computational power we will only process one example folder. The rest of the folders are already processed.

In [3]:
```
randomFolder = random.choice(dicomFolders)
print 'The randomly chosen folder is:', os.path.basename(randomFolder)
dicomFiles = glob.glob(randomFolder+'/*.dcm')
print 'The folder conatins', len(dicomFiles), 'image slices'
```

```
The randomly chosen folder is: 1.3.6.1.4.1.14519.5.2.1.6279.6001.513023675145186449943177283498
The folder conatins 137 image slices
```

In the next step we load the DICOM files and save them to a Python dictionary. As we can not be sure that the DICOM files are saved in consecutive order and that python is reading them in the correct order, we need to sort the 2D slices depending on their real world coordinates to gain a correct 3D CT image.

In [4]:
```
dicomDictionar = {}
for dicomFile in dicomFiles:
    dicomSlice = pydicom.read_file(dicomFile)
    sliceLocation = getattr(dicomSlice, 'ImagePositionPatient', '')
    sliceLocation = sliceLocation[2]
    dicomDictionar[sliceLocation] = dicomSlice
dicomDictionar = collections.OrderedDict(sorted(dicomDictionar.items(), reverse=True))
```

For the following image processing we will need some image information from the DICOM header information.

In [5]:
```
sliceThickness = dicomSlice.SliceThickness
pixelSpacing = dicomSlice.PixelSpacing
sliceSpacing = abs(dicomDictionar.keys()[1] - dicomDictionar.keys()[2])
slope = dicomSlice.RescaleSlope
intercept = dicomSlice.RescaleIntercept
```

Our image data is currently stored in an image object. For easier processing we will convert this object to a 3D numpy array. The pixel values of medical images are typically saved in Hounsfield Units. DICOM images from different scanners can have an offset and/or scale for the pixel values which we have to add/multiply here as well.

In [6]:
```
imageCube = []
for location, dicomSlice in dicomDictionar.iteritems():
    image = dicomSlice.pixel_array * slope + intercept
    imageCube.append(image.astype(np.int16))
imageCube = np.asarray(imageCube)
```

Now we can display our image for the first time. We will plot three direction views of our 3D image. We will see that two of the images will look deformed. This is correct.

Question: What is the reason for the image deformation?

In [7]:
```
fig,ax = plt.subplots(1,3,figsize=(64,16))
ax[0].imshow(imageCube[imageCube.shape[0]/2,:,:],cmap='gray')
ax[1].imshow(imageCube[:,imageCube.shape[1]/2,:],cmap='gray')
ax[2].imshow(imageCube[:,:,imageCube.shape[2]/2],cmap='gray')
plt.show()
plt.close(fig)
```

As described earlier, DICOM images within a data set can have different image resolutions and sizes. We can print our current image features.

In [8]:
```
print 'Image size:', imageCube.shape ,'(Z,Y,X)'
print 'Image resolution:', sliceSpacing, pixelSpacing[0], pixelSpacing[1],'(Z,Y,X)'
```

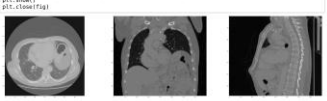
```
Image size: (137, 512, 512) (Z,Y,X)
Image resolution: 2.5 0.781250 0.781250 (Z,Y,X)
```

The next step is to normalize our image to get the same resolution in all three directions. Our intended resolution is 1 millimeter per 1 pixel. Afterwards we print our image resolution and size again. **Note: This step can take a moment.**

In [ ]:
```
spacing = np.array([sliceSpacing, pixelSpacing[0], pixelSpacing[1]], dtype=np.float32)
factor = spacing / [1,1,1]
imageCube = ndimage.interpolation.zoom(imageCube, factor, order=3)
print 'Image size:', imageCube.shape ,'(Z,Y,X)'
print 'Image resolution: 1 1 1 (Z,Y,X)'
```

We can plot our image again to see how the normalization affected it.

In [10]:
```
fig,ax = plt.subplots(1,3,figsize=(64,16))
ax[0].imshow(imageCube[imageCube.shape[0]/2,:,:],cmap='gray')
ax[1].imshow(imageCube[:,imageCube.shape[1]/2,:],cmap='gray')
ax[2].imshow(imageCube[:,:,imageCube.shape[2]/2],cmap='gray')
plt.show()
plt.close(fig)
```

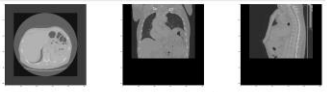


Now our image has a resolution of 1mm in each direction. But the image could still be different to other images. Therefore we extend/crop our image to 512x512x512pixel. We set the added pixel outside the image to -1500. This is a smaller value then any DICOM image value.

In [11]:
```
imageCube512 = np.zeros(shape=(512, 512, 512), dtype=np.int16)-1500
startX = int((512 - imageCube.shape[2])/2)
startY = int((512 - imageCube.shape[1])/2)
startZ = int((512 - imageCube.shape[1])/2)

imageCube512[max(imageCube.shape[0], 512),
             startY:(startY + imageCube.shape[1]),
             startX:(startX + imageCube.shape[2])] = imageCube[:min(imageCube.shape[0], 512),:,
:]
```

And plot the image again

In [12]:
```
fig,ax = plt.subplots(1,3,figsize=(64,16))
ax[0].imshow(imageCube512[imageCube512.shape[0]/2,:,:],cmap='gray')
ax[1].imshow(imageCube512[:,imageCube512.shape[1]/2,:],cmap='gray')
ax[2].imshow(imageCube512[:,:,imageCube512.shape[2]/2],cmap='gray')
plt.show()
plt.close(fig)
```

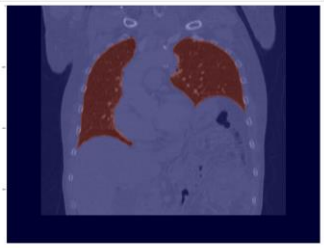


As working with 3D images requires very strong graphics cards and a lot of time, we will extract one 2D slice from our 3D image for further processing.

In [13]:
```
imgSlice = np.zeros((512,512))
imgSlice = np.copy(imageCube512[:, int(imageCube512.shape[1]/2), :])
```

Finally we can load the lung segmentation for our image and display it afterwards. Keep in mind that the masks were generated automatically using some simple functions and therefore the masks won't be very accurate.

```
In [14]: maskFile = 'data/LIDC_Test_NPY/'+os.path.basename(randomFolder)+'.npy'
         mask = np.load(maskFile)[2]
         fig,ax = plt.subplots(1,1,figsize=[32,32])
         ax.imshow(imgSlice, cmap='gray')
         ax.imshow(mask, cmap='jet', alpha=0.4)
         plt.show()
         plt.close(fig)
```



This is all we need to do for image preparation. The next step would be to save the image and process the rest of the images. As this is already done for all images, we can skip this step and go to the next exercise.

Data Citation

Armato III, Samuel G., McLennan, Geoffrey, Bidaut, Luc, McNitt-Gray, Michael F., Meyer, Charles R., Reeves, Anthony P., ... Clarke, Laurence P. (2015). Data From LIDC-IDRI. The Cancer Imaging Archive. http://doi.org/10.7937/K9/TCIA.2015.LO9QL9SX

TCIA Citation

Clark K, Vendt B, Smith K, Freymann J, Kirby J, Koppel P, Moore S, Phillips S, Maffitt D, Pringle M, Tarbox L, Prior F. The Cancer Imaging Archive (TCIA): Maintaining and Operating a Public Information Repository, Journal of Digital Imaging, Volume 26, Number 6, December, 2013, pp 1045-1057. (paper)

The authors acknowledge the National Cancer Institute and the Foundation for the National Institutes of Health, and their critical role in the creation of the free publicly available LIDC/IDRI Database used in this study.

Image augmentation

Deep Learning requires a high number of input objects (in our case images). As big data sets are especially in medical imaging rare, we need a way to **artifically enlarge our data set to reduce over-fitting on image data**. One established way to achieve this is data augmentation. In this process the input images are modified so that they keep important features but also differ enough to look different from the original image. Popular ways of image augmentation are **rotating, translating, flipping or altering color/grey-value intensities.**

We start again with importing needed python libraries and loading our test images.

In [6]:
```
import random, keras, logging, os, glob
import numpy as np
from scipy.ndimage import rotate
import matplotlib.pyplot as plt

inputDir = 'data/LIDC_Test_NPY'
imageFiles = glob.glob(inputDir+'/*.npy')
print 'Found', len(imageFiles), 'images to test.'

testData = {}
for imageFile in imageFiles:
    studyID = os.path.basename(imageFile)
    testData[studyID] = list(np.load(imageFile))
print 'Loaded', len(testData), 'image files.'
```

```
Found 56 images to test.
Loaded 56 image files.
```

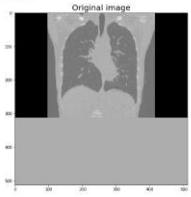
In this tutorial we apply rotation as well as translation to our input images. Therefore we define two functions. Each function takes the image as input as well as the boundaries for modifying the images. Within this boundaries we choose a random number for the angle to rotate and distance to translate.

In [2]:
```
def ranRotate(image, augRot):
    ranAngle = random.choice(range(augRot[0], augRot[1]+1, augRot[2]))
    image = rotate(image, ranAngle, cval=-1.0, reshape=False, order=0)
    return image, ranAngle
```

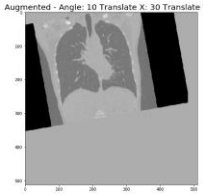
In [3]:
```
def ranTranslate(image, augTrans):
    ranX = random.choice(range(augTrans[0], augTrans[1]+1, augTrans[2]))
    ranY = random.choice(range(augTrans[0], augTrans[1]+1, augTrans[2]))

    imageTranslated = np.zeros(image.shape)-1

    # Input coordinates
    iX = [-ranX if ranX < 0 else 0, image.shape[1] if ranX < 0 else image.shape[1] - ranX]
    iY = [-ranY if ranY < 0 else 0, image.shape[0] if ranY < 0 else image.shape[0] - ranY]

    # Output coordinates
    oX = [0 if ranX < 0 else ranX, image.shape[1] + ranX if ranX < 0 else image.shape[1]]
    oY = [0 if ranY < 0 else ranY, image.shape[0] + ranY if ranY < 0 else image.shape[0]]

    imageTranslated[oY[0]:oY[1], oX[0]:oX[1]] = image[iY[0]:iY[1], iX[0]:iX[1]]

    return imageTranslated, ranX, ranY
```

To test our functions above, we choose a random image and plot it as a reference.

In [4]:
```
studyID = random.choice(testData.keys())
image = testData[studyID][0]
fig,ax = plt.subplots(1,1,figsize=[8,8])
ax.imshow(image, cmap='gray')
ax.set_title('Original image',fontsize= 20)
plt.show()
```



In [5]:
```
augRot = [-15,15,5]
augTrans = [0,40,10]

for i in range(10):
    imgRot, angle = ranRotate(np.copy(image), augRot)
    imgTrans, x, y = ranTranslate(imgRot, augTrans)

    fig,ax = plt.subplots(1,1,figsize=(8,8))
    ax.imshow(imgTrans, cmap='gray')
    titleText = 'Augmented - Angle: ' + str(angle) + ' Translate X: ' + str(x) + ' Translate Y:
' + str(y)
    ax.set_title(titleText, fontsize= 20)
    plt.show()
```



Training with data augmentation:

When a deep network is trained, image augmentation is typically done *on the fly*. That means, an image is loaded, randomly modified and then used to train the network. The drawback of this method is the computational power needed for the image modifications. These can significantly slow the training down! A good way to keep deceleration low is to train the network using the computers graphics card (GPU) and in parallel augment the images for the next epoch using the computers main processor (CPU).

Alternatively, the images can be modified prior to the network training and every modified image stored on the hard drive. Due to the huge number of possible augmentations, this approach is only feasible for very small data sets and data sizes.

For *on the fly* augmentation keep in mind to use efficient functions. Search for different libraries to perform the modifications and compare the time needed per step. There can be tremendous differences in time needed for image modifications, especially with large images.

Lung segmentation using a fully convolutional network - Change settings again

As the first two network settings did not perform good enough, we need to try again.

As always, we start with importing some necessary libraries. Remark: If there is a **FutureWarning** shown after importing the libraries, we can ignore it.

In [1]:
```
import os, sys, csv, shutil, glob, random, tables
import numpy as np
import matplotlib.pyplot as plt
from keras import backend as K
import KerasModel
K.set_image_dim_ordering('tf')
```

```
Using TensorFlow backend.
```

As in the exercise before, we need to load our input data.

In [2]:
```
inputDir = 'data/LIDC_Test_NPY'
imageFiles = glob.glob(inputDir+'/*.npy')
print 'Found ', len(imageFiles), 'images to test.'

testData = {}
for imageFile in imageFiles:
    studyID = os.path.basename(imageFile)
    testData[studyID] = list(np.load(imageFile))
```

```
Found 56 images to test.
```

Again, we define the function for the dice coefficient.

In [3]:
```
def dice(yTrue, yPred):
    intersect = np.sum(yTrue.flatten() * yPred.flatten())
    return (2. * intersect + 1.0) / (np.sum(yTrue.flatten()) + np.sum(yPred.flatten()) + 1.0)
```

The code for the model in this exercise

```
inputs = Input([512, 512, 1], name='model_input')
conv1 = Conv2D(64, (3,3), activation='relu', padding='same', name='conv_1')(inputs)
pool1 = MaxPooling2D(pool_size=(2,2), name='pool_1')(conv1)
conv2 = Conv2D(128, (3,3), activation='relu', padding='same', name='conv_2')(pool1)
pool2 = MaxPooling2D(pool_size=(2,2), name='pool_2')(conv2)
conv3 = Conv2D(256, (3,3), activation='relu', padding='same', name='conv_3')(pool2)
conv4 = Conv2D(512, (3,3), activation='relu', padding='same', name='conv_4')(conv3)
conv5 = Conv2D(1, (1, 1), name='conv_8')(conv4)
upcn1 = Conv2DTranspose(1, (64, 64), strides=(4, 4), padding='same', name='upconv_1')(con
v5)
act = Activation('sigmoid', name='act')(upcn1)
model = Model(inputs=inputs, outputs=act)
model.compile(optimizer=Adam(lr=0.000001), loss=diceLoss, metrics=[dice])
```

We only change one parameter again, this time for the transpose layer, also known as **de**-convolutional layer. We change the kernel size from (3,3) to (64,64).

We need to load our model into the notebook. Afterwards we load our saved weights for this network.

In [4]:

```
model = KerasModel.getConv2d_3()
weightFile = 'data/trainOutput_06/weights/weights-926--0.65.hdf5'
model.load_weights(weightFile)
```

With everything set up, we can now test our network. The following code will loop over each study, load the image and the manually segmented mask. It then predicts a lung mask and saves the result. Furthermore it prints for each study the dice coefficient.

Remark: This step will take a few seconds to minutes as it predicts a mask for each of the 56 test images.

In [5]:

```
diceList = []
for studyID, patientData in testData.iteritems():
    img = patientData[0]
    msk = patientData[2]
    prd = model.predict(img[np.newaxis, :, :, np.newaxis])

    testData[studyID].append(prd)
    resultDice = dice(msk, prd)
    diceList.append(resultDice)
    print studyID, resultDice
```

```
1.3.6.1.4.1.14519.5.2.1.6279.6001.120196332569034738680965284519.npy 0.7209355642642862
1.3.6.1.4.1.14519.5.2.1.6279.6001.627998298349675613581885874395.npy 0.7748670956131675
1.3.6.1.4.1.14519.5.2.1.6279.6001.307921770358136677021532761235.npy 0.4825931229166011б
1.3.6.1.4.1.14519.5.2.1.6279.6001.152684536713461901635595118048.npy 0.7585172918421273
1.3.6.1.4.1.14519.5.2.1.6279.6001.283569726884265181140892667131.npy 0.4731126878812б276
1.3.6.1.4.1.14519.5.2.1.6279.6001.419601611032172899567156073142.npy 0.6629946320723527
1.3.6.1.4.1.14519.5.2.1.6279.6001.248425363469507808613979846863.npy 0.6650326778580856
1.3.6.1.4.1.14519.5.2.1.6279.6001.227796349777753378641347819780.npy 0.7271028736876762
1.3.6.1.4.1.14519.5.2.1.6279.6001.143782059748737055784173697516.npy 0.5111485123128839
1.3.6.1.4.1.14519.5.2.1.6279.6001.118924792893660585177487888076.npy 0.7623324998858834
1.3.6.1.4.1.14519.5.2.1.6279.6001.217589936421986638139451480826.npy 0.6770357445045142
1.3.6.1.4.1.14519.5.2.1.6279.6001.185154482385982570363528682299.npy 0.6971164178116455
1.3.6.1.4.1.14519.5.2.1.6279.6001.453946099750629491201946672990.npy 0.5696724081682528
1.3.6.1.4.1.14519.5.2.1.6279.6001.156181515927698299507377083373.npy 0.515029799559787
1.3.6.1.4.1.14519.5.2.1.6279.6001.309564220265302089123180126785.npy 0.5391371733299075
1.3.6.1.4.1.14519.5.2.1.6279.6001.665044273312514967270866833277.npy 0.5368878566945418
1.3.6.1.4.1.14519.5.2.1.6279.6001.129982010889624423230394257528.npy 0.6929036986969676
1.3.6.1.4.1.14519.5.2.1.6279.6001.483655032093002252444764787700.npy 0.6362864235747805
1.3.6.1.4.1.14519.5.2.1.6279.6001.187966156856911682643615997798.npy 0.6918434920683075
1.3.6.1.4.1.14519.5.2.1.6279.6001.187451715205085403623595258748.npy 0.6489717869681354
1.3.6.1.4.1.14519.5.2.1.6279.6001.199975006921901879512837687266.npy 0.6319199019574555
1.3.6.1.4.1.14519.5.2.1.6279.6001.192256506776434538421891524381.npy 0.7716654128873657
1.3.6.1.4.1.14519.5.2.1.6279.6001.223098610241551815995595311693.npy 0.7417227739356824
1.3.6.1.4.1.14519.5.2.1.6279.6001.270951128717816232360812849541.npy 0.7635255007010148
1.3.6.1.4.1.14519.5.2.1.6279.6001.121993590721161347818774929286.npy 0.7442839063448997
1.3.6.1.4.1.14519.5.2.1.6279.6001.905371958588660410240398317235.npy 0.7471742287448864
1.3.6.1.4.1.14519.5.2.1.6279.6001.125356649712550043958727288500.npy 0.6845055050219254
1.3.6.1.4.1.14519.5.2.1.6279.6001.157537616155071783765339942952.npy 0.5095874938179879
1.3.6.1.4.1.14519.5.2.1.6279.6001.219087313261026510620926082729.npy 0.5838033074095479
1.3.6.1.4.1.14519.5.2.1.6279.6001.218476624578721885561483687176.npy 0.648236457548473
1.3.6.1.4.1.14519.5.2.1.6279.6001.153732973534937692357111055819.npy 0.6037522929915162
1.3.6.1.4.1.14519.5.2.1.6279.6001.121391737347333465796214915391.npy 0.6443049293231871
```

Let's print our average dice coefficient:

In [6]:

```
print 'Average dice coefficient:', str(round(np.sum(diceList)/len(diceList)*100, 1)) + '%.'
```

```
Average dice coefficient: 63.9%.
```

Now we should be somewhere in the 80ies - Great! Lets take a look at an image.

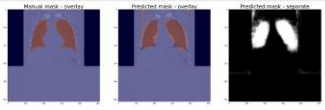
First, threshold our results:

In [7]:

```
for studyID, patientData in testData.iteritems():
    testData[studyID][3][testData[studyID][3]>0.9] = 1
    testData[studyID][3][testData[studyID][3]<1.0] = 0
```

Finally we can plot a manually segmented and predicted mask. Therefore we randomly choose a study ID and plot the results.

```
In [8]: studyID = random.choice(testData.keys())
        image = testData[studyID][0]
        mask = testData[studyID][2]
        pred = testData[studyID][3][0,:,:,0]

        fig,ax = plt.subplots(1,3,figsize=[32,16])
        ax[0].imshow(image, cmap='gray')
        ax[0].imshow(mask, cmap='jet', alpha=0.4)
        ax[1].imshow(image, cmap='gray')
        ax[1].imshow(pred, cmap='jet', alpha=0.4)
        ax[2].imshow(pred, cmap='gray')

        ax[0].set_title('Manual mask - overlay',fontsize= 30)
        ax[1].set_title('Predicted mask - overlay',fontsize= 30)
        ax[2].set_title('Predicted mask - separate',fontsize= 30)

        plt.show()
```



As you can see, the results are not perfect but for the third attempt, very promising! Let's see if we can do even better...

Lung segmentation using a fully convolutional network - Change settings again-again

One last attempt with this network, promise!

As always, we start with importing some necessary libraries. Remark: If there is a **FutureWarning** shown after importing the libraries, we can ignore it.

```
In [0]: import os, sys, csv, shutil, glob, random, tables
        import numpy as np
        import matplotlib.pyplot as plt
        from keras import backend as K
        import KerasModel
        K.set_image_dim_ordering('tf')
```

Again, loading the images and defining the dice coefficient.

```
In [9]: inputDir = 'data/LIDC Test NPY'
        imageFiles = glob.glob(inputDir+'/*.npy')
        print 'Found', len(imageFiles), 'images to test.'

        testData = {}
        for imageFile in imageFiles:
            studyID = os.path.basename(imageFile)
            testData[studyID] = list(np.load(imageFile))

        def diceLoss(y_true, y_pred):
            intersect = np.sum(y_true.flatten() * y_pred.flatten())
            return (2. * intersect + 1.0) / (np.sum(y_true.flatten()) + np.sum(y_pred.flatten()) + 1.0)
```

```
Found 56 images to test.
```

The code for the model in this exercise

```
inputs = Input((512, 512, 1), name='model_input')
conv1 = Conv2D(64, (5,5), activation='relu', padding='same', name='conv_1')(inputs)
pool1 = MaxPooling2D(pool_size=(2,2), name='pool_1')(conv1)
conv2 = Conv2D(128, (5,5), activation='relu', padding='same', name='conv_2')(pool1)
pool2 = MaxPooling2D(pool_size=(2,2), name='pool_2')(conv2)
conv3 = Conv2D(256, (5,5), activation='relu', padding='same', name='conv_3')(pool2)
conv4 = Conv2D(512, (3,3), activation='relu', padding='same', name='conv_4')(conv3)
conv5 = Conv2D(1, (1, 1), name='conv_5')(conv4)
upcn1 = Conv2DTranspose(1, (64, 64), strides=(4, 4), padding='same', name='upconv_1')(con
v5)
act = Activation('sigmoid', name='act')(upcn1)
model = Model(inputs=inputs, outputs=act)
model.compile(optimizer=Adam(lr=0.000001), loss=diceLoss, metrics=[dice])
```

So we know from our previus attempts that the change we made for the convolution did make our results worse. On the other hand the change we made for the de-convolution increased our results. So it makes sense to revert the convolution settings and keep the de-convolution settings.

```
In [10]: model = KerasModel.getConv2d_4()
         weightFile = 'dataTrainOutput_80/weights/weights-003--0.81.hdf5'
         model.load_weights(weightFile)
```

With everything set up, we can now test our network. The following code will loop over each study, load the image and the manually segmented mask. It then predicts a lung mask and saves the result. Furthermore it prints for each study the dice coefficient.

Remark: This step will take a few moments to minutes as it predicts a mask for each of the 56 test images.

```
In [11]: diceList = []
for studyID, patientData in testData.iteritems():
    img = patientData[0]
    msk = patientData[2]
    prd = model.predict(img[np.newaxis, :, :, np.newaxis])

    testData[studyID].append(prd)
    resultDice = dice(msk, prd)
    diceList.append(resultDice)
    print studyID, resultDice
```

```
1.3.6.1.4.1.14519.5.2.1.6279.6001.120196332569034738680965284519.npy 0.8523769072606797
1.3.6.1.4.1.14519.5.2.1.6279.6001.627998298349675613581885874395.npy 0.8792399380839451
1.3.6.1.4.1.14519.5.2.1.6279.6001.307921770358136677021532761235.npy 0.7291867087820415
1.3.6.1.4.1.14519.5.2.1.6279.6001.152684536713461901635595118048.npy 0.859189499436744
1.3.6.1.4.1.14519.5.2.1.6279.6001.283569726884265181140892667131.npy 0.6670013982946713
1.3.6.1.4.1.14519.5.2.1.6279.6001.419601611032172899567156073142.npy 0.8178736652400505
1.3.6.1.4.1.14519.5.2.1.6279.6001.248425363469507808613979846863.npy 0.8392758942662605
1.3.6.1.4.1.14519.5.2.1.6279.6001.227796349777753378641347819780.npy 0.8517001176639475
1.3.6.1.4.1.14519.5.2.1.6279.6001.143782059748737055784173697516.npy 0.7367847505755742
1.3.6.1.4.1.14519.5.2.1.6279.6001.118924792893605851774878880764.npy 0.8973545949698952
1.3.6.1.4.1.14519.5.2.1.6279.6001.217589936421986638139451480826.npy 0.8309842789697318
1.3.6.1.4.1.14519.5.2.1.6279.6001.185154482385982570363528682299.npy 0.845401623777637
1.3.6.1.4.1.14519.5.2.1.6279.6001.453946099750629491201946672990.npy 0.760023107651786
1.3.6.1.4.1.14519.5.2.1.6279.6001.156181515927698299507377083373.npy 0.681128820820636
1.3.6.1.4.1.14519.5.2.1.6279.6001.309564220265302089123180126785.npy 0.7709111989772900
1.3.6.1.4.1.14519.5.2.1.6279.6001.665044273312514967278866833277.npy 0.7294601097742229
1.3.6.1.4.1.14519.5.2.1.6279.6001.129982010889624423230394257528.npy 0.8345401971053592
1.3.6.1.4.1.14519.5.2.1.6279.6001.483655032093002252444764787700.npy 0.820885891990259
1.3.6.1.4.1.14519.5.2.1.6279.6001.187966156856911682643615997798.npy 0.8383252110185342
1.3.6.1.4.1.14519.5.2.1.6279.6001.187451715205085483623595258748.npy 0.8137451441870326
1.3.6.1.4.1.14519.5.2.1.6279.6001.199975006921901879512837687266.npy 0.8080857029035368
1.3.6.1.4.1.14519.5.2.1.6279.6001.192256506776434538421891524301.npy 0.8887979018502972
1.3.6.1.4.1.14519.5.2.1.6279.6001.223098610241551815995595311693.npy 0.8496033110860276
1.3.6.1.4.1.14519.5.2.1.6279.6001.270951128717816232360812849541.npy 0.8649910580835792
1.3.6.1.4.1.14519.5.2.1.6279.6001.121993590721161347818774929286.npy 0.8681286418263169
1.3.6.1.4.1.14519.5.2.1.6279.6001.905371958588660410240398317235.npy 0.857296638330159
1.3.6.1.4.1.14519.5.2.1.6279.6001.125356649712550043958727288500.npy 0.8440373351499693
1.3.6.1.4.1.14519.5.2.1.6279.6001.157537616155071703765339942952.npy 0.738474125102762
1.3.6.1.4.1.14519.5.2.1.6279.6001.219087313261026510628926082729.npy 0.7910727764425446
1.3.6.1.4.1.14519.5.2.1.6279.6001.218476624578721885561483687176.npy 0.8087327380648133
1.3.6.1.4.1.14519.5.2.1.6279.6001.153732973534937692357111055819.npy 0.73325906545079
1.3.6.1.4.1.14519.5.2.1.6279.6001.121391737347333465796214915391.npy 0.8098411820120132
```

Let's print our average dice coefficient:

```
In [12]: print 'Average dice coefficient:', str(round(np.sum(diceList)/len(diceList)*100, 1)) + '%.'
```

```
Average dice coefficient: 80.6%.
```

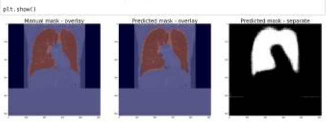
Slightly better! How do our images look like?

Thresholding the mask ...

```
In [13]: for studyID, patientData in testData.iteritems():
    testData[studyID][3][testData[studyID][3]>0.9] = 1
    testData[studyID][3][testData[studyID][3]<1.0] = 0
```

Finally we can plot a manually segmented and predicted mask. Therefore we randomly choose a study ID and plot the results.

```
In [14]: studyID = random.choice(testData.keys())
image = testData[studyID][0]
mask = testData[studyID][2]
pred = testData[studyID][3][0,:,:,0]

fig,ax = plt.subplots(1,3,figsize=[32,16])
ax[0].imshow(image, cmap='gray')
ax[0].imshow(mask, cmap='jet', alpha=0.4)
ax[1].imshow(image, cmap='gray')
ax[1].imshow(pred, cmap='jet', alpha=0.4)
ax[2].imshow(pred, cmap='gray')

ax[0].set_title('Manual mask - overlay',fontsize= 30)
ax[1].set_title('Predicted mask - overlay',fontsize= 30)
ax[2].set_title('Predicted mask - separate',fontsize= 30)

plt.show()
```



Ok, so this result looks slightly better then the previous one. Not perfect, but very promising.

Hint: Try to change the threshold from 0.9 to something smaller or bigger and plot the result again. Did it improve the mask?

Ok, let's recap our findings here. We optimized our fully convolutional network as good as possible and the results very promising. If for example we would use this network to find a bounding box over the lung in a complete chest x-ray, this results would already be sufficient. Of course there are other parameters we can try to modify and re-run our network again to improve our results even more but for now we can leave this network as is and try something else. Time to change the network...

Lung segmentation using a U-Net

In this last exercise we will try to segment the lung in our test set with a **U-Net**[1]. The network architecture is shown in a box below. A U-Net is a network type that was developed for **image segmentation** and can be trained from **very few images.**

As always, we start with importing necessary libraries. And again, if there is a **FutureWarning** shown after importing the libraries, we can ignore it. We then load our testing images and define the dice coefficient again.

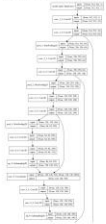
In [1]:
```
import os, sys, csv, shutil, glob, random, tables
import numpy as np
import matplotlib.pyplot as plt
from keras import backend as K
import KerasPadUnet
K.set_image_dim_ordering('tf')

inputDir = 'data/LIDC_Test_NPY'
imageFiles = glob.glob(inputDir+'/*.npy')
print 'Found ', len(imageFiles), 'images to test.'

testData = {}
for imageFile in imageFiles:
    studyID = os.path.basename(imageFile)
    testData[studyID] = list(np.load(imageFile))

def dice(yTrue, yPred):
    intersect = np.sum(yTrue.flatten() * yPred.flatten())
    return (2. * intersect + 1.0) / (np.sum(yTrue.flatten()) + np.sum(yPred.flatten()) + 1.0)
```

```
Using TensorFlow backend.
Found 50 images to test.
```

The trained U-Net

The following image shows the architecture of our U-Net. **Note the direct connections between earlier and later layers.** These are the secret of the success of the U-Net. If the image you see is to small, you can try to open th emage from here link.

Note: If you think that the U-Net looks like a C-Net, well, that's because the U-Net developers plot their network architecture from left to right while the image below was created using Tensorflow which plots the network architecture from top to bottm. Hence we need to rotate the image bellow by 90 degrees counter clock wise to make it look like a U-Net.



The code for the model in this exercise

```
inputs = Input([512, 512, 1], name='model_input')
conv1 = Conv2D(32, (3,3), activation='relu', padding='same', name='conv_1_1')(inputs)
conv1 = Conv2D(64, (3,3), activation='relu', padding='same', name='conv_1_2')(conv1)
pool1 = MaxPooling2D(pool_size=(2,2), name='pool_1')(conv1)

conv2 = Conv2D(64, (3,3), activation='relu', padding='same', name='conv_2_1')(pool1)
conv2 = Conv2D(128, (3,3), activation='relu', padding='same', name='conv_2_2')(conv2)
pool2 = MaxPooling2D(pool_size=(2,2), name='pool_2')(conv2)

conv3 = Conv2D(128, (3,3), activation='relu', padding='same', name='conv_3_1')(pool2)
conv3 = Conv2D(256, (3,3), activation='relu', padding='same', name='conv_3_2')(conv3)
pool3 = MaxPooling2D(pool_size=(2,2), name='pool_3')(conv3)

conv4 = Conv2D(256, (3,3), activation='relu', padding='same', name='conv_4_1')(pool3)
conv4 = Conv2D(512, (3,3), activation='relu', padding='same', name='conv_4_2')(conv4)

up5 = UpSampling2D(size=(2,2), name='up_5')(conv4)
up5 = concatenate([up5, conv3], axis=3, name='conc_5')
conv5 = Conv2D(256, (3,3), activation='relu', padding='same', name='conv_5_1')(up5)
conv5 = Conv2D(256, (3,3), activation='relu', padding='same', name='conv_5_2')(conv5)

up6 = UpSampling2D(size=(2,2), name='up_6')(conv5)
up6 = concatenate([up6, conv2], axis=3, name='conc_6')
conv6 = Conv2D(128, (3,3), activation='relu', padding='same', name='conv_6_1')(up6)
conv6 = Conv2D(128, (3,3), activation='relu', padding='same', name='conv_6_2')(conv6)

up7 = UpSampling2D(size=(2,2), name='up_7')(conv6)
up7 = concatenate([up7, conv1], axis=3, name='conc_7')
conv7 = Conv2D(64, (3,3), activation='relu', padding='same', name='conv_7_1')(up7)
conv7 = Conv2D(64, (3,3), activation='relu', padding='same', name='conv_7_2')(conv7)

conv8 = Conv2D(1, (1, 1), name='conv_8')(conv7)
act = Activation('sigmoid', name='act')(conv8)
```

We need to load our model into the notebook as well as the weights.

In [2]:
```
model = KerasModel.getUnet2d()
weightFile = 'data/trainOutput_05_06_09_0a/weights/weights-953--0.98.hdf5'
model.load_weights(weightFile)
```

With everything set up, we can now test our network. The following code will loop over each study, load the image and the manually segmented mask. It then predicts a lung mask and saves the result. Furthermore it prints for each study the dice coefficient.

Remark: This step will take a few seconds to minutes as it predicts a mask for each of the 56 test images.

In [3]:
```
diceList = []
for studyID, patientData in testData.iteritems():
    img = patientData[0]
    msk = patientData[2]
    prd = model.predict(img[np.newaxis, :, :, np.newaxis])

    testData[studyID].append(prd)
    resultDice = dice(msk, prd)
    diceList.append(resultDice)
    print studyID, resultDice
```

```
1.3.6.1.4.1.14519.5.2.1.6279.6001.120196332569034738680965284519.npy 0.989377929440089
1.3.6.1.4.1.14519.5.2.1.6279.6001.627998298349675613581885874395.npy 0.98768520062600253
1.3.6.1.4.1.14519.5.2.1.6279.6001.307921770358136677021532761235.npy 0.9868836095317021
1.3.6.1.4.1.14519.5.2.1.6279.6001.152684536713461901635595118048.npy 0.9878859238619769
1.3.6.1.4.1.14519.5.2.1.6279.6001.283569726884265181140892667131.npy 0.9824510188132849
1.3.6.1.4.1.14519.5.2.1.6279.6001.419601611032172899567156073142.npy 0.9887095901004328
1.3.6.1.4.1.14519.5.2.1.6279.6001.240425363469507880613979846063.npy 0.9754350524680478
1.3.6.1.4.1.14519.5.2.1.6279.6001.227796349777753378641347819780.npy 0.9895705121825947
1.3.6.1.4.1.14519.5.2.1.6279.6001.143782059748737055784173697516.npy 0.9888560605032344
1.3.6.1.4.1.14519.5.2.1.6279.6001.118924792893665851774878880764.npy 0.9853184107676759
1.3.6.1.4.1.14519.5.2.1.6279.6001.217589936421986638139451480826.npy 0.9787819387239986
1.3.6.1.4.1.14519.5.2.1.6279.6001.185154482385982570363528682299.npy 0.9819177216076671
1.3.6.1.4.1.14519.5.2.1.6279.6001.453946099750629491201946672990.npy 0.9632771135720539
1.3.6.1.4.1.14519.5.2.1.6279.6001.156181515927698299507377083373.npy 0.9876215683874248
1.3.6.1.4.1.14519.5.2.1.6279.6001.309564220265302089123180126785.npy 0.9749671350956369
1.3.6.1.4.1.14519.5.2.1.6279.6001.665044273312514967278866833277.npy 0.9837262520883004
1.3.6.1.4.1.14519.5.2.1.6279.6001.129982010889624423230394257528.npy 0.9816821113850981
1.3.6.1.4.1.14519.5.2.1.6279.6001.483655032093002252444764787700.npy 0.9857144257649336
1.3.6.1.4.1.14519.5.2.1.6279.6001.187966156856911682643615997790.npy 0.9864362745599984
1.3.6.1.4.1.14519.5.2.1.6279.6001.187451715205085403623595258748.npy 0.982442743957972
1.3.6.1.4.1.14519.5.2.1.6279.6001.199975006921901879512837687266.npy 0.9833832172832242
1.3.6.1.4.1.14519.5.2.1.6279.6001.192256506776434538421891524301.npy 0.9601985283502027
1.3.6.1.4.1.14519.5.2.1.6279.6001.223098610241551815995595311693.npy 0.9946871109139985
1.3.6.1.4.1.14519.5.2.1.6279.6001.270951128717816232360812849541.npy 0.9829623262486653
1.3.6.1.4.1.14519.5.2.1.6279.6001.121993590721161347818774929286.npy 0.9876310947174969
1.3.6.1.4.1.14519.5.2.1.6279.6001.905371958588660410240398317235.npy 0.9871553365516943
1.3.6.1.4.1.14519.5.2.1.6279.6001.125356649712550043958727288500.npy 0.9875339827839269
1.3.6.1.4.1.14519.5.2.1.6279.6001.157537616155071783765339942952.npy 0.9854187686541638
1.3.6.1.4.1.14519.5.2.1.6279.6001.219087313261026510628926082729.npy 0.9507624489277372
1.3.6.1.4.1.14519.5.2.1.6279.6001.218476624578721885561483687176.npy 0.983361554771356
1.3.6.1.4.1.14519.5.2.1.6279.6001.153732973534937692357111055819.npy 0.9925364165377226
1.3.6.1.4.1.14519.5.2.1.6279.6001.121391737347333465796214915391.npy 0.9583291957104855
```

Lets print our average dice coefficient again:

```
In [4]: print 'Average dice coefficient:', str(round(np.sum(diceList)/len(diceList)*100, 1)) + '%.'
```

```
Average dice coefficient: 98.0%.
```

The result should be in the upper nineties. This is a very good result and particularly impressive when compared to our first network output.

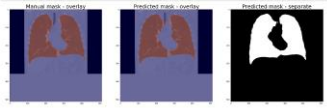
Again, we need to threshold our predicted mask.

```
In [5]: for studyID, patientData in testData.iteritems():
            testData[studyID][3][testData[studyID][3]>0.9] = 1
            testData[studyID][3][testData[studyID][3]<1.0] = 0
```

Finally we can plot a manually segmented and predicted mask. Therefore we randomly choose a study ID and plot the results.

Hint: Run this cell multiple times to display different results.

```
In [6]: studyID = random.choice(testData.keys())
        image = testData[studyID][0]
        mask = testData[studyID][2]
        pred = testData[studyID][3][0,:,:,0]

        fig,ax = plt.subplots(1,3,figsize=(32,16))
        ax[0].imshow(image, cmap='gray')
        ax[0].imshow(mask, cmap='jet', alpha=0.4)
        ax[1].imshow(image, cmap='gray')
        ax[1].imshow(pred, cmap='jet', alpha=0.4)
        ax[2].imshow(pred, cmap='gray')

        ax[0].set_title('Manual mask - overlay',fontsize= 30)
        ax[1].set_title('Predicted mask - overlay',fontsize= 30)
        ax[2].set_title('Predicted mask - separate',fontsize= 30)

        plt.show()
```



The shown results look very good and a lot better than the output of the previous tests!

Question: Display several results. What are the obvious problems of our prediction and how could we solve them?

Now, it is important to note here: A U-Net is not fundamentally the better network! But for our small data set and our given problem it definitely performed great!

References:

[1] Ronneberger O., Fischer P., Brox T. (2015) U-Net: Convolutional Networks for Biomedical Image Segmentation. In: Navab N., Hornegger J., Wells W., Frangi A. (eds) Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015. Lecture Notes in Computer Science, vol 9351. Springer, Cham