U-Net: Convolutional Networks for Face and Structure Image Segmentation

Anuja Garg
Acropolis Institute of Technology and Research, Indore, MP, India

Abstract. There is large approbation that successful training of deep networks requires many thousand annotated training samples. In this paper, to use the available annotated samples more efficiently I present a network and training strategy that relies on the strong use of data augmentation. The architecture consists of a contracting path to capture context and a symmetric expanding path that enables precise localization. We show that such a network can be trained end-to-end from very few images and outperforms the prior best method (a sliding-window convolutional network), using the same network trained on images having animals or other structures. The full implementation (based on TensorFlow) and the trained networks are available at: https://drive.google.com/open?id=1XSQILUQV1Xlzw1cV0NdTEqncLnfJVOnA.

I.Introduction

In the last five years, deep convolutional networks have outperformed the state of the art in many visual recognition tasks, e.g. [4,2]. The success of Convolutional Neural Networks was limited due to the size of the available training sets and the size of the considered networks, although they have already existed for a long time [5]. The breakthrough by Krizhevsky et al. [4] was due to supervised training of a large network with 8 layers and millions of parameters on the ImageNet dataset with 1 million training images. Since then, even larger and deeper networks have been trained [8].

The typical use of convolutional networks is on classification tasks, where the output to an image is a single class label. However, in many visual tasks, especially in face and structure image processing, the desired output should include localization, i.e., a class label is supposed to be assigned to each pixel. Moreover, thousands of training images are usually beyond reach in structgure tasks. Hence, Ciresan et al. [1] trained a network in a sliding-window setup to predict the class label of each pixel by providing a local region (patch) around that pixel as input. First, this network can localize. Secondly, the training data in terms of patches is much larger than the number of training images.
Obviously, the strategy in Ciresan et al. [1] has two drawbacks. First, it is quite slow because the network must be run separately for each patch, and there is a lot of redundancy due to overlapping patches. Secondly, there is a trade-o between localization accuracy and the use of context. Larger patches require more max-pooling layers that reduce the localization accuracy, while small patches allow the network to see only little context. More recent approaches [7,3] proposed a classifier output that takes into account the features from multiple layers. Good localization and the use of context are possible at the same time.

In this paper, a more elegant architecture is built upon, the so-called “fully convolutional network" [6]. We modify and extend this architecture such that it works with very few training images and yields more precise segmentations of faces and structures; see Figure 1. By supplementing a usual contracting network by successive layers, where pooling operators are replaced by up sampling operators, there is an increase in the resolution of the output.

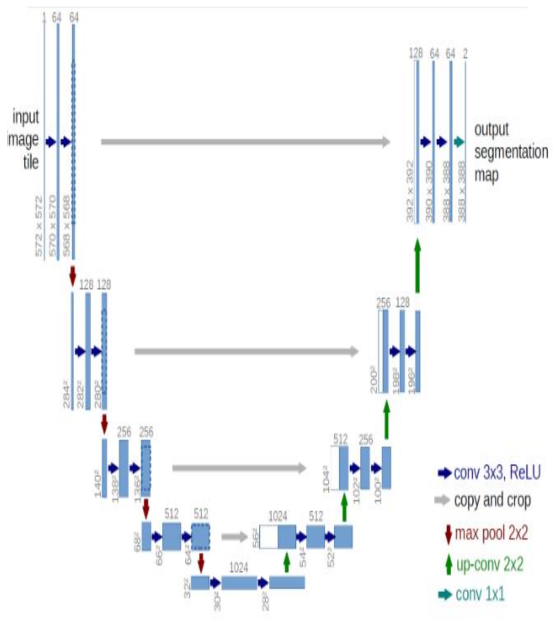


Fig. 1. U-net architecture (example for 64x64 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

II.Network Architecture

The network architecture is illustrated in Figure 1. It consists of a contracting path (left side) and an expansive path (right side). The contracting path follows the typical architecture of a convolutional network. It consists of the repeated application of two 3x3 convolutions (unpadded convolutions), each followed by a rectified linear unit (ReLU) and a 2x2 max pooling operation with stride 2 for downsampling. At each downsampling step we double the number of feature channels. Every step in the expansive path consists of an upsampling of the feature map followed by a 2x2 convolution (“up-convolution") that halves the number of feature channels, a concatenation with the correspondingly cropped feature map from the contracting path, and two 3x3 convolutions, each followed by a ReLU. The cropping is necessary due to the loss of border pixels in every convolution. At the final layer a 1x1 convolution is used to map each 64-component feature vector to the desired number of classes. In total the network has 23 convolutional layers.

To allow a seamless tiling of the output segmentation map, it is important to select the input tile size such that all 2x2 max-pooling operations are applied to a layer with an even x- and y-size.

III.Dataset Preparation

After downloading dataset, we have two folders. The first one is images which contains the raw images and annotation which contains the masks as a binary folder image.
By using image generator function, we prepare the dataset.

```
def image_generator(files, batch_size = 32, sz = (25
6, 256)):

 while True:

   #extract a random batch
   batch = np.random.choice(files, size = batch_size)


   #variables for collecting batches of inputs and out
puts
   batch_x = []
   batch_y = []


   for f in batch:

     #get the masks. Note that masks are png files
     mask = Image.open(f'annotations/trimaps/{f[:-
4]}.png')
     mask = np.array(mask.resize(sz))


     #preprocess the mask
     mask[mask >= 2] = 0
     mask[mask != 0 ] = 1

     batch_y.append(mask)
```

```
        #preprocess the raw images
        raw = Image.open(f'images/{f}')
        raw = raw.resize(sz)
        raw = np.array(raw)

        #check the number of channels because some o
f the images are RGBA or GRAY
        if len(raw.shape) == 2:
          raw = np.stack((raw,)*3, axis=-1)

        else:
          raw = raw[:,:,0:3]

        batch_x.append(raw)

    #preprocess a batch of images and masks
    batch_x = np.array(batch_x)/255.
    batch_y = np.array(batch_y)
    batch_y = np.expand_dims(batch_y,3)

    yield (batch_x, batch_y)
     batch_size = 32

all_files = os.listdir('images')
shuffle(all_files)

split = int(0.95 * len(all_files))

#split into training and testing
train_files = all_files[0:split]
test_files  = all_files[split:]

train_generator = image_generator(train_files, batch
_size = batch_size)
test_generator  = image_generator(test_files, batch_s
ize = batch_size)
x, y= next(train_generator)
plt.axis('off')
img = x[0]
msk = y[0].squeeze()
msk = np.stack((msk,)*3, axis=-1)

plt.imshow( np.concatenate([img, msk, img*msk], a
xis = 1))
```

IV.IoU metric

The intersection over union (IoU) metric is a simple metric used to evaluate the performance of a segmentation algorithm. Given two masks x_{true}, x_{pred} we evaluate

IoU = $y_{true} \cap y_{pred} / y_{true} \cup y_{pred}$

```
def mean_iou(y_true, y_pred):
    yt0 = y_true[:,:,:,0]
    yp0 = K.cast(y_pred[:,:,:,0] > 0.5, 'float32')
    inter = tf.count_nonzero(tf.logical_and(tf.equal(yt
0, 1), tf.equal(yp0, 1)))
    union = tf.count_nonzero(tf.add(yt0, yp0))
    iou = tf.where(tf.equal(union, 0), 1., tf.cast(inter/u
nion, 'float32'))
    return iou
```

V.Model

```
def unet(sz = (256, 256, 3)):
  x = Input(sz)
  inputs = x

  #down sampling
  f = 8
  layers = []

  for i in range(0, 6):
    x = Conv2D(f, 3, activation='relu', padding='same'
) (x)
    x = Conv2D(f, 3, activation='relu', padding='same'
) (x)
    layers.append(x)
    x = MaxPooling2D() (x)
    f = f*2
  ff2 = 64

  #bottleneck
```

```
 j = len(layers) - 1
 x = Conv2D(f, 3, activation='relu', padding='same')
(x)
 x = Conv2D(f, 3, activation='relu', padding='same')
(x)
 x = Conv2DTranspose(ff2, 2, strides=(2, 2), paddin
g='same') (x)
 x = Concatenate(axis=3)([x, layers[j]])
 j = j -1

 #upsampling
 for i in range(0, 5):
   ff2 = ff2//2
   f = f // 2
   x = Conv2D(f, 3, activation='relu', padding='same'
) (x)
   x = Conv2D(f, 3, activation='relu', padding='same'
) (x)
   x = Conv2DTranspose(ff2, 2, strides=(2, 2), paddi
ng='same') (x)
   x = Concatenate(axis=3)([x, layers[j]])
   j = j -1


 #classification
 x = Conv2D(f, 3, activation='relu', padding='same')
(x)
 x = Conv2D(f, 3, activation='relu', padding='same')
(x)
 outputs = Conv2D(1, 1, activation='sigmoid') (x)

 #model creation
 model = Model(inputs=[inputs], outputs=[outputs])
 model.compile(optimizer = 'rmsprop', loss = 'binary
_crossentropy', metrics = [mean_iou])

 return model
model = unet()
model.summary()
```

VI.Callbacks

Simple functions to save the model at each epoch and show some predictions

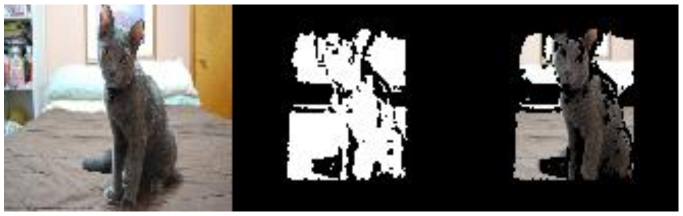
```
def build_callbacks():
        checkpointer = ModelCheckpoint(filepath='une
t.h5', verbose=0, save_best_only=True, save_weight
s_only=True)
        callbacks = [checkpointer, PlotLearning()]
        return callbacks

# inheritance for training process plot
class PlotLearning(keras.callbacks.Callback):

    def on_train_begin(self, logs={}):
        self.i = 0
        self.x = []
        self.losses = []
        self.val_losses = []
        self.acc = []
        self.val_acc = []
        #self.fig = plt.figure()
        self.logs = []
    def on_epoch_end(self, epoch, logs={}):
        self.logs.append(logs)
        self.x.append(self.i)
        self.losses.append(logs.get('loss'))
        self.val_losses.append(logs.get('val_loss'))
        self.acc.append(logs.get('mean_iou'))
        self.val_acc.append(logs.get('val_mean_iou'))
        self.i += 1
        print('i=',self.i,'loss=',logs.get('loss'),'val_loss=',
logs.get('val_loss'),'mean_iou=',logs.get('mean_iou'),
'val_mean_iou=',logs.get('val_mean_iou'))

        #choose a random test image and preprocess
        path = np.random.choice(test_files)
        raw = Image.open(f'images/{path}')
        raw = np.array(raw.resize((256, 256)))/255.
        raw = raw[:,:,0:3]
        s
        #predict the mask
        pred = model.predict(np.expand_dims(raw, 0))

        #mask post-processing
        msk  = pred.squeeze()
        msk = np.stack((msk,)*3, axis=-1)
        msk[msk >= 0.5] = 1
        msk[msk < 0.5] = 0

        #show the mask and the segmented image
        combined = np.concatenate([raw, msk, raw* ms
k], axis = 1)
        plt.axis('off')
        plt.imshow(combined)
        plt.show()
```

VII.Training

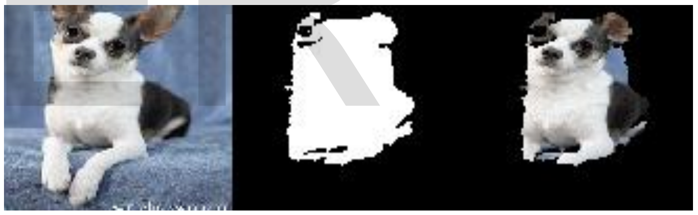
The input images and their corresponding segmentation maps are used to train the network with the stochastic gradient descent implementation of Case [9]. Due to the unpadded convolutions, the output image is smaller than the input by a constant border width. To minimize the overhead and make maximum use of the GPU memory, we favor large input tiles over a large batch size and hence reduce the batch to a single image. Accordingly we use a high momentum (0.99) such that a large number of the previously seen training samples determine the update in the current optimization step.

```
train_steps = len(train_files) //batch_size
test_steps = len(test_files) //batch_size
model.fit_generator(train_generator,
                    epochs = 30, steps_per_epoch = train_st
eps,validation_data = test_generator, validation_step
s = test_steps,
                    callbacks = build_callbacks(), verbose =
 0)
```

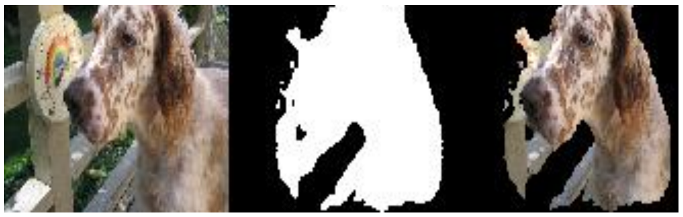
```
OUTPUT:
i= 1 loss= 0.5812949411673088 val_loss=
0.4535805203697898 mean_iou=
0.01489582893853234 val_mean_iou=
0.38134072856469586
```



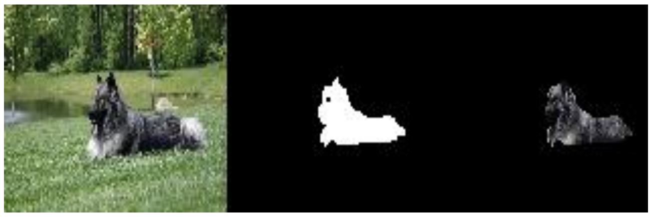
```
i= 2 loss= 0.4611788935040774 val_loss=
0.4059526297179135 mean_iou=
0.44138037224542603 val_mean_iou=
0.47776194594123145
```



```
i= 3 loss= 0.435364905421592 34 val_loss=
0.420459739186547 mean_iou=
0.4549277193470088 val_mean_iou=
0.5115999552336606
```



```
i= 4 loss= 0.41323080182619837 val_loss= 0.39397893168709497 mean_iou= 0.48104962339139967 val_mean_iou= 0.4626355333761735
```



VIII. Prediction with new sample

```
!wget http://r.ddmcdn.com/s_f/o_1/cx_462/cy_245/cw_1349/ch_1349/w_720/APL/uploads/2015/06/caturday-shutterstock_149320799.jpg -O test.jpg

raw = Image.open('test.jpg')
raw = np.array(raw.resize((256, 256)))/255.
raw = raw[:,:,0:3]

#predict the mask
pred = model.predict(np.expand_dims(raw, 0))

#mask post-processing
msk  = pred.squeeze()
msk = np.stack((msk,)*3, axis=-1)
msk[msk >= 0.5] = 1
msk[msk < 0.5] = 0

#show the mask and the segmented image
combined = np.concatenate([raw, msk, raw* msk], axis = 1)
plt.axis('off')
plt.imshow(combined)
plt.show()
```

The model predictions are efficient upto 93%

Conclusion

Image sementation using the u-net architecture achieves good performance on very different structure segmentation applications. It only needs very few annotated images and has a very reasonable training time on Google Colaboratory GPU. The u-net architecture can thus be applied easily to many more tasks.

References

1. Ciresan, D.C., Gambardella, L.M., Giusti, A., Schmidhuber, J.: Deep neural networks segment neuronal membranes in electron microscopy images. In: NIPS. pp. 2852{2860 (2012)
2. Girshick, R., Donahue, J., Darrell, T., Malik, J.: Rich feature hierarchies for accurate object detection and semantic segmentation. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2014)
3. Hariharan, B., Arbelez, P., Girshick, R., Malik, J.: Hypercolumns for object seg-mentation and ne-grained localization (2014), arXiv:1411.5752 [cs.CV]
4. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classi cation with deep convolutional neural networks. In: NIPS. pp. 1106{1114 (2012)
5. LeCun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W., Jackel, L.D.: Backpropagation applied to handwritten zip code recognition. Neural Computation 1(4), 541{551 (1989)
6. Long, J., Shelhamer, E., Darrell, T.: Fully convolutional networks for semantic segmentation (2014), arXiv:1411.4038
7. Seyedhosseini, M., Sajjadi, M., Tasdizen, T.: Image segmentation with cascaded hierarchical models and logistic disjunctive normal networks. In: Computer Vision (ICCV), 2013 IEEE International Conference on. pp. 2168{2175 (2013)
8. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition (2014), arXiv:1409.1556 [cs.CV]
9. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T.: Caffe: Convolutional architecture for fast feature embedding (2014), arXiv:1408.5093 [cs.CV]
10. U-Net: Convolutional Networks for Biomedical Image Segmentation 18 May 2015 Olaf Ronneberger, Philipp Fischer,Thomas Brox