ISRAILOV SARDOR

DEEP LEARNING PROJECT

"follow me"

Udacity

ISRAILOV SARDOR

INTRODUCTION:

The typical CNN is a convolutions followed by some fully connected layers followed by an activation function(softmax,relu, tanh…). But fully connected layers don't perserve spacial information. They work well as image classifier i.e identifies whether it's a car, dog etc. on a picture, but fails to tell us where is the object on the image. Normally we use techniques like forward,backpropagation, maxpooling, average pooling and so on.

If we change connected network, and insert more convolutions(deconvolutions) we can preserve spacial information by concatenating(layers).

Encoder: The role of encoder is to extract features from an image. Each conv_layer has its own level of features(dot, line, geometrical form etc…).

Skip connections is a way to retain some data by skipping a certain number of pairs (one encoder one decoder). These connections allows the network to use data from different resolutions from the network. We use skip connections technique to generalize the picture, because the role of encoding is to focused on a special set of features.

We used it in the decoder function. L'Etudiant étranger de Philippe Labro

Decoder: The role of a decoder is to preserve spacial information. We used transposed convolution that reverse the regular convolution layers, with a Bilinear upsampling(x2).

Batch normalization is also used in each FCN layer and it is based on the idea that, instead of just normalizing the inputs to the network, we normalize the inputs to layers within the network. Basically we subtract the mean, and force the standard deviation to 1(Gaussian distribution). WE have higher learning rates and deeper networks (avoid the saturation of nodes).

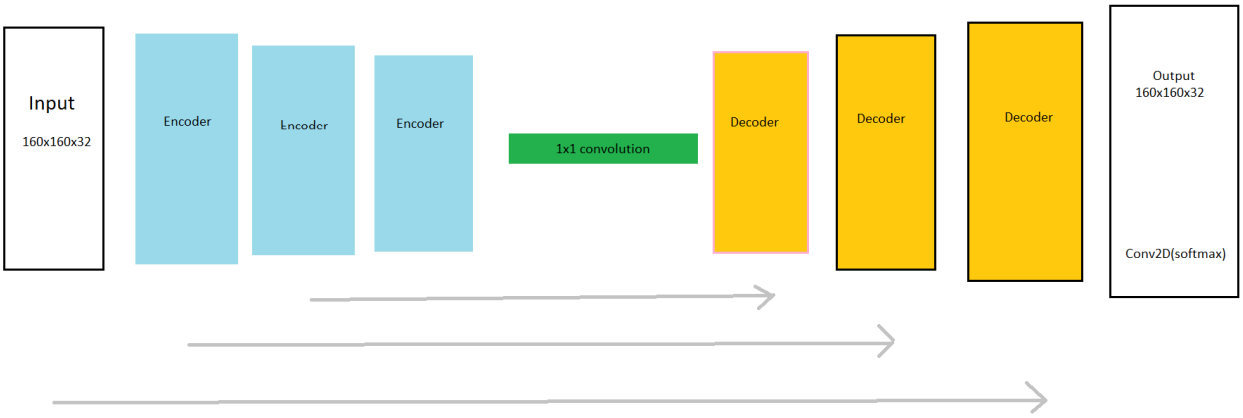
What is semantic segmentation: identifying several objects that we care about in an image and telling us where are they. Shortcoming of semantic segmentation is that it labels several objects as a whole if they are nearby.

Remarks: First of all, the commonly used size for filters for image processing is the power of 2. That's why I used in my encoders, the filter of this type.

Why **1x1 convolutions**? 1x1 convolutions are used to compute reductions before the expensive 3x3 and 5x5 convolutions. It reduces at the great extent the number of hyper parameters that we need to tune, that's why it's much more computationally efficient.

Basically, the architecture I found the most useful is the 3 layer encoder, a convolution and 3 layer decoder function.

The network at a glance:




```
Inputs Tensor("input_1:0", shape=(?, 160, 160, 3), dtype=float32)
Encoder 1 Tensor("batch_normalization/batchnorm/add_1:0", shape=(?, 80, 80, 32), dtype=float32)
Encoder 2 Tensor("batch_normalization_2/batchnorm/add_1:0", shape=(?, 40, 40, 64), dtype=float32)
Encoder 3 Tensor("batch_normalization_3/batchnorm/add_1:0", shape=(?, 20, 20, 128), dtype=float32)
Conv Layer Tensor("batch_normalization_4/batchnorm/add_1:0", shape=(?, 20, 20, 8), dtype=float32)
Decoder 1 Tensor("batch_normalization_5/batchnorm/add_1:0", shape=(?, 40, 40, 128), dtype=float32)
Deoder 2 Tensor("batch_normalization_6/batchnorm/add_1:0", shape=(?, 80, 80, 64), dtype=float32)
Decoder 3 Tensor("batch_normalization_7/batchnorm/add_1:0", shape=(?, 160, 160, 32), dtype=float32)
```

We can make the network very deep. WE can see the symmetry in this kind of networks: we downsample: use convolutions, maxpooling etc. and then we upsample, and finally, Output is the same size as the input image.

We use FCN, because we can take the image of any sizes and use the related features of two nearby patches. We use the stride of 2, in encoding, because it downsamples by the factor of 2.

The first strategy is to stack up convolutional neural networks, but the deeper the network gets, the more there is the risk of overfitting. Moreover, it's more computationally expensive.

There are two ways to construct the model and determine weights:

1) On the native machine(very slow even if I had Msi with 4GB GPU)
2) On the cloud server.

Finally, I tried both methods with different learning rates and match sizes, though there are some techniques that exist to fine-tune the hyper-parameters. Although, reculting the training data is as important as building correct FCN architecture I focused on finetuning hyperparameters of my FCN.

ISRAILOV SARDOR

Strategy for tuning hyperparameters:

First of all, we need to choose the architecture for the network. If it's too shallow, it won't work properly (we won't be satisfied with results). If it's too deep, then the model may be tending to overfit and learn the features. I included hyper parameters and model weights for 5 tries, though I tried more times. The thing is that when I saw it didn't start as good as I expected, I stopped and started again with better hyper parameters. I put in the parameters by intuition developed, but there are many optimization methods over there including Bayesian optimization.

The model with 2 encoders, a convolution and 2 decoders.

0^{th}: learn_rate:0.001, batch_size=50, num_epochs =100; score=0.373340485903

The model with 3 encoders, a convolution and 3 decoders.

1^{st}: learn_rate:0.001, batch_size=100, num_epochs =10;

2^{nd}: learn_rate:0.0008, batch_size=100, num_epochs = 10;

3^{st}: learn_rate:0.001, batch_size=100, num_epochs = 30; reLU instead of softmax.

4^{th}: learn_rate:0.001, batch_size=50, num_epochs =100; score=0.437

Reminder: the strategy is used is bottleneck convolutions that reduce the size, and then instead of applying fully connected layers, we apply 1x1 convolution with decoders to up sample to the image.

Implementation:

First we import the necessary libraries in the RoboND environment.

In [1]:
```
import os
import glob
import sys
import tensorflow as tf

from scipy import misc
import numpy as np

from tensorflow.contrib.keras.python import keras
from tensorflow.contrib.keras.python.keras import layers, models

from tensorflow import image

from utils import scoring_utils
from utils.separable_conv2d import SeparableConv2DKeras, BilinearUpSampling2D
from utils import data_iterator
from utils import plotting_tools
from utils import model_tools
```

Tf and keras are used for deep learning. Utils contain helper function for convolutions and plotting.

Afterwards, we define separable convolution with relu activation function and batch normalizations.

```
def separable_conv2d_batchnorm(input_layer, filters, strides=1):
    output_layer = SeparableConv2DKeras(filters=filters,kernel_size=3, strides=strides,
                                        padding='same', activation='relu')(input_layer)

    output_layer = layers.BatchNormalization()(output_layer)
    return output_layer
```

Then, we have 2d convolutions with batch normalizations:

```
def conv2d_batchnorm(input_layer, filters, kernel_size=3, strides=1):
    output_layer = layers.Conv2D(filters=filters, kernel_size=kernel_size, strides=strides,
                                 padding='same', activation='relu')(input_layer)

    output_layer = layers.BatchNormalization()(output_layer)
    return output_layer
```

We also have a bilinear upsample function that upsamples by a factor of 2, we use this function for decoding. There are several strategies for that: unpooling, “bed of nails”,max unpooling(at the same position as before max pooling). For example, x2 upsample:

```
Input Shape: (1, 4, 4, 3)
Output Shape: (1, 8, 8, 3)
```

In [3]:
```
def bilinear_upsample(input_layer):
    output_layer = BilinearUpSampling2D((2,2))(input_layer)
    return output_layer
```

Afterwards we construct our network, in my case it's 3 encoder, convolution and 3 decoder network, followed by 2D convolution with the softmax activation function.

def fcn_model(inputs, num_classes):…

return layers.Conv2D(num_classes, 1, activation='softmax', padding='same')(last_decoder)

We define our training model and then start training using training set images. We define the learning rate, batch size and the number of epochs. The learning rate was chosen to be 0.01. If we choose a large value for it we can converge faster to minimize the loss function, but at some moment our loss function may start to increase. If we choose

Model.fit_generator()//trains the model on the dataset. "Fits the model on data yielded batch-by-batch by a Python generator", which takes several parameters one of which is workers: in my case:8 because 8-core processor.

In [4]:
```
def encoder_block(input_layer, filters, strides):

    # TODO Create a separable convolution layer using the separable_conv2d_batchnorm() function.
    output_layer = separable_conv2d_batchnorm(input_layer, filters, strides=strides)

    return output_layer
```

Results:

ISRAILOV SARDOR

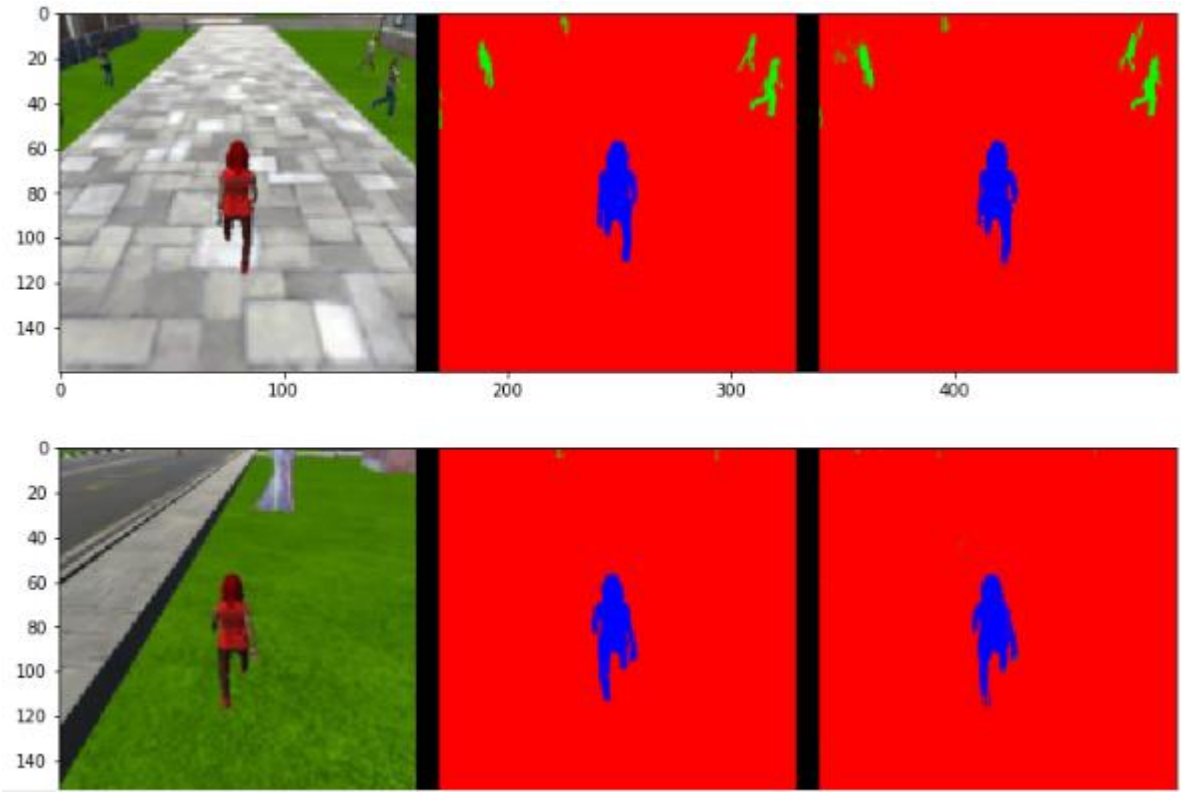


Figure 1(while following)

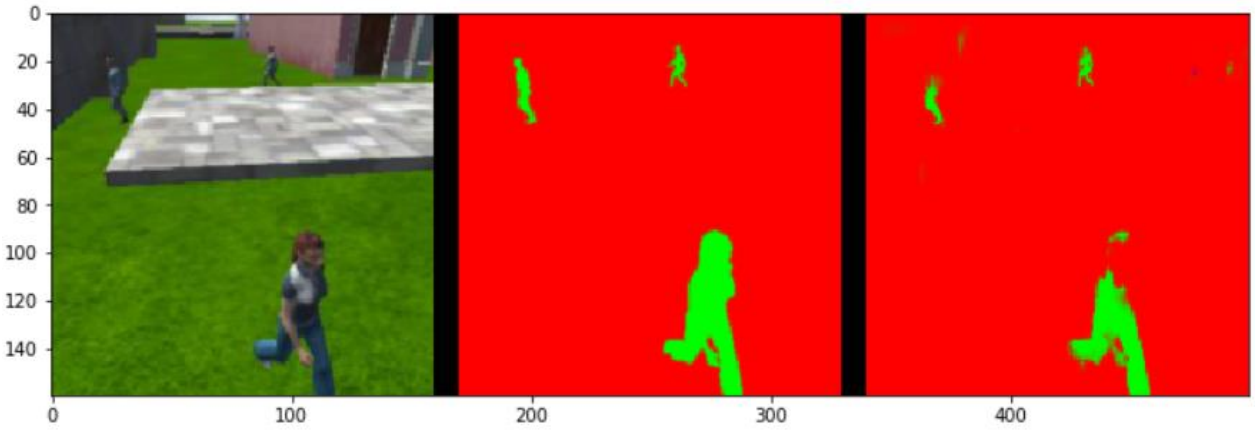


Figure 2(while without a target)

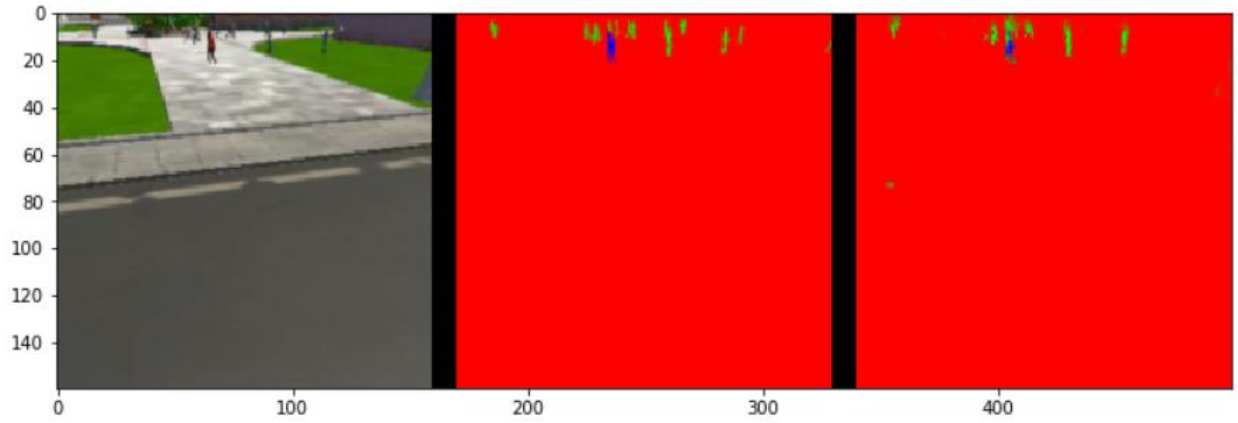


Figure 3(while a patrol with target)

jupyter model_training Dernière Sauvegarde : 09/10/2018 (auto-sauvegardé)

File Edit View Insert Cell Kernel Widgets Help

Exécuter Code

```
false_neg = false_neg1 + false_neg2 + false_neg3

weight = true_pos/(true_pos+false_neg+false_pos)
print(weight)
```

```
0.7382478632478633
```

Entrée [20]:

```
# The IoU for the dataset that never includes the hero is excluded from grading
final_IoU = (iou1 + iou3)/2
print(final_IoU)
```

```
0.586133509094
```

Entrée [21]:

```
# And the final grade score is
final_score = final_IoU * weight
print(final_score)
```

```
0.432711810667
```

Figure 4(the best result)

In conclusion, I used the best results to follow the target in the simulator mode!

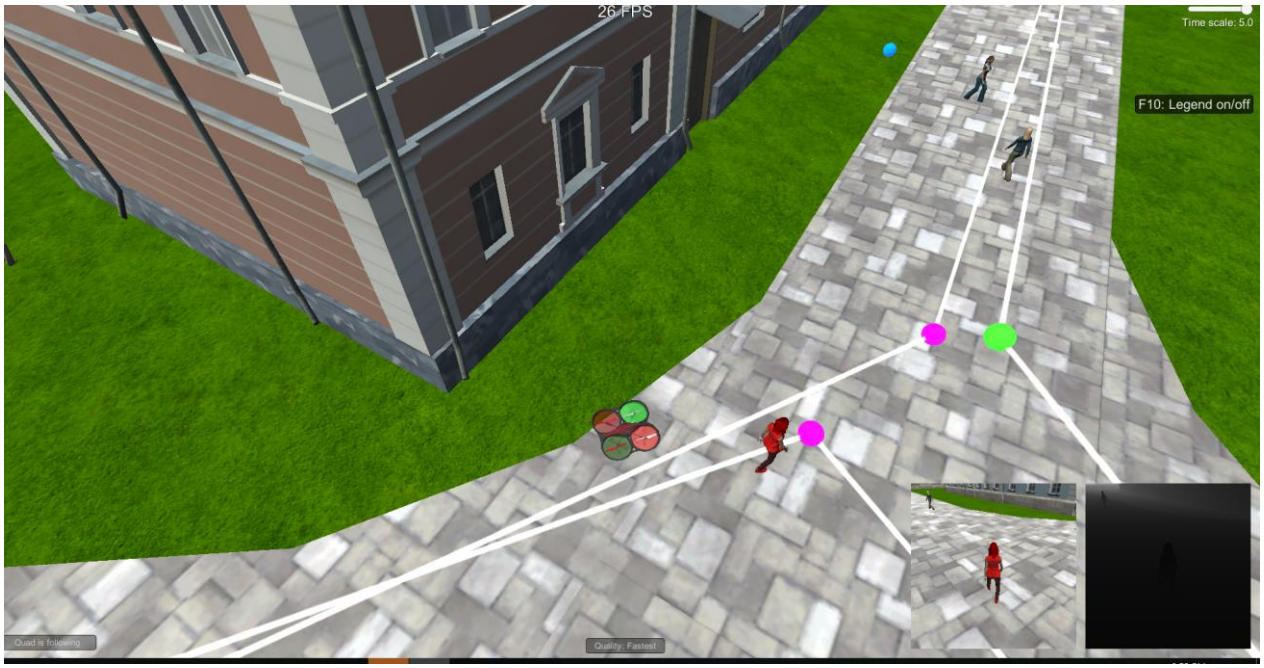


Figure 5(following)

Future Enhancements:

1) As stated above, there are many optimization methods for the network, one of which is Bayesian optimization.
2) Adding more layers with skip connections (encoder decoder) will improve the accuracy. But in this case, it will take much more time to train the network.

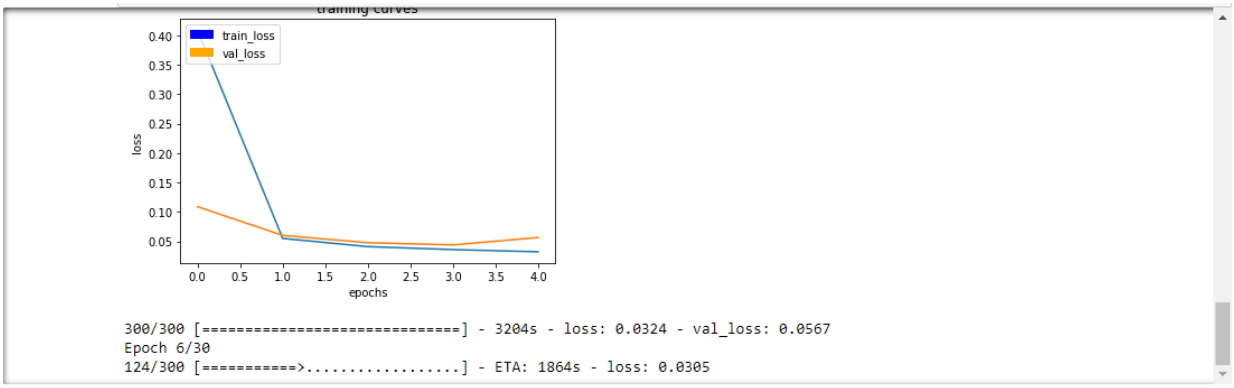
Annexes:

Conv2DKeras:

2D convolution layer (e.g. spatial convolution over images).

This layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs. If `use_bias` is True, a bias vector is created and added to the outputs. Finally, if `activation` is not `None`, it is applied to the outputs as well.

Example of overfitting: when the model learns the actual train data, and performs worse on the validation data:




```
300/300 [==============================] - 3204s - loss: 0.0324 - val_loss: 0.0567
Epoch 6/30
124/300 [===========>..................] - ETA: 1864s - loss: 0.0305
```

```
In [*]: # Save your trained model weights
        weight_file_name = 'model_weights'
        model_tools.save_network(model, weight_file_name)
```

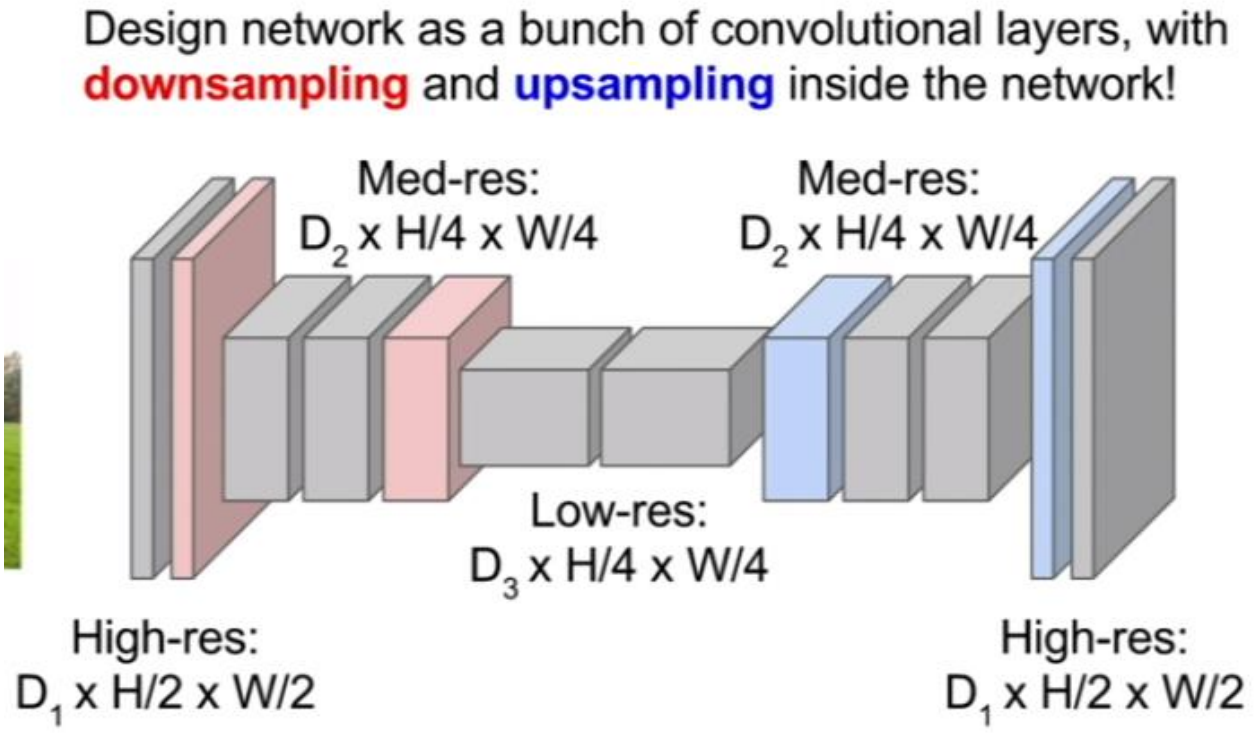
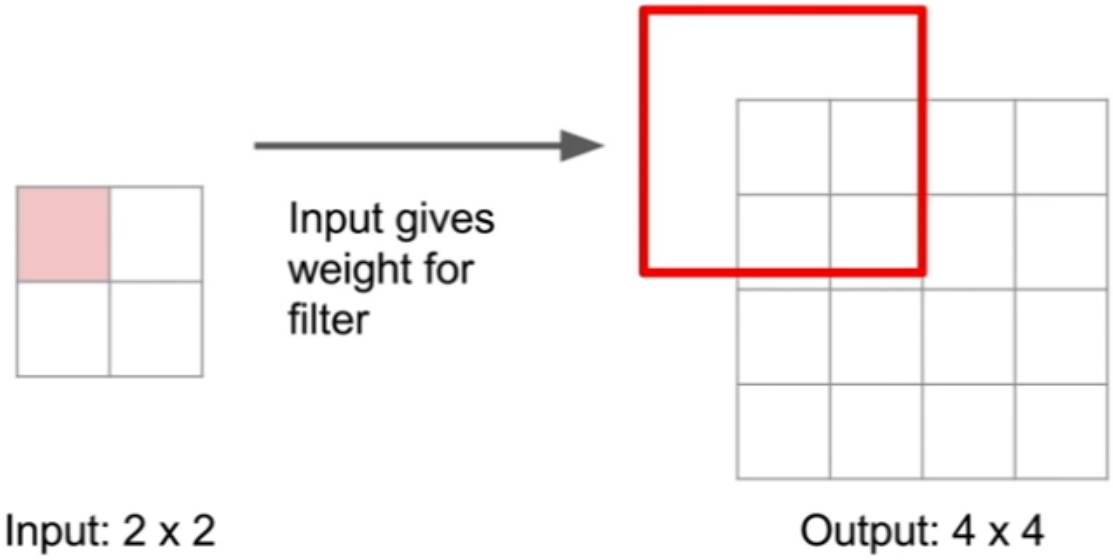


Figure 6(stanford CS231)

Transpose convolution: (learnable upsampling)

ISRAILOV SARDOR



Sources:

1) Stanford CS231
2) https://arxiv.org/pdf/1511.00561.pdf
3) https://www.analyticsvidhya.com/blog/2018/10/understanding-inception-network-from-scratch/
4) https://towardsdatascience.com/a-simple-guide-to-the-versions-of-the-inception-network-7fc52b863202 (inception and GoogLeNet)