CS 484/684 Computational Vision

Fall 2019, Yuri Boykov

Report any typos / errors to the TA, Towaki Takikawa at tovacinni@gmail.com

Homework Assignment #4 - Supervised Deep Learning for Segmentation

This assignment will test your understanding of applying deep learning by having you apply (fully supervised) deep learning to semantic segmentation, a well studied problem in computer vision.

You can get most of the work done using only CPU, however, the use of GPU will be helpful in later parts. Programming and debugging everything upto and including problem 5c should be fine on CPU. You will notice the benefit of GPU mostly in later parts (d-h) of problem 5, but they are mainly implemenmted and test your code written and debugged earlier. If you do not have a GPU readily accesible to you, we recommend that you use Google Colaboratory to get access to a GPU. Once you are satisfied with your code upto and including 5(c), simply upload this Jupyter Notebook to Google Colaboratory to run the tests in later parts of Problem 5.

Proficiency with PyTorch is required. Working through the PyTorch tutorials will make this assignment significantly easier. https://pytorch.org/tutorials/ (https://pytorch.org/tutorials/)

In [1]:
```
%matplotlib inline

# It is best to start with USE_GPU = False (implying CPU). Switch USE
_GPU to True only if you want to use GPU. However...
# we strongly recommend to wait until you are absolutely sure your CP
U-based code works (at least on single image dataset)
USE_GPU = True
```

In [2]:
```
# Python Libraries
import random
import math
import numbers
import platform
import copy

# Importing essential libraries for basic image manipulations.
import numpy as np
import PIL
from PIL import Image, ImageOps
import matplotlib.pyplot as plt
from tqdm import tqdm

# We import some of the main PyTorch and TorchVision libraries used f
or HW4.
# Detailed installation instructions are here: https://pytorch.org/ge
t-started/locally/
# That web site should help you to select the right 'conda install' c
ommand to be run in 'Anaconda Prompt'.
# In particular, select the right version of CUDA. Note that prior to
installing PyTorch, you should
# install the latest driver for your GPU and CUDA (9.2 or 10.1), assu
ming your GPU supports it.
# For more information about pytorch refer to
# https://pytorch.org/docs/stable/nn.functional.html
# https://pytorch.org/docs/stable/data.html.
# and https://pytorch.org/docs/stable/torchvision/transforms.html
import torch
import torch.nn.functional as F
from torch import nn
from torch.utils.data import DataLoader
import torchvision.transforms as transforms
import torchvision.transforms.functional as tF

# We provide our own implementation of torchvision.datasets.voc (cont
aining popular "Pascal" dataset)
# that allows us to easily create single-image datasets
from lib.voc import VOCSegmentation

# Note class labels used in Pascal dataset:
# 0:    background,
# 1-20: aeroplane, bicycle, bird, boat, bottle, bus, car, cat, chair,
cow, diningtable, dog, horse, motorbike,
#       person, pottedplant, sheep, sofa, train, TV_monitor
# 255: "void", which means class for pixel is undefined
```

In [3]:
```
# ChainerCV is a library similar to TorchVision, created and maintained by Preferred Networks.
# Chainer, the base library, inspired and led to the creation of PyTorch!
# Although Chainer and PyTorch are different, there are some nice functionalities in ChainerCV
# that are useful, so we include it as an excersice on learning other libraries.
# To install ChainerCV, normally it suffices to run "pip install chainercv" inside "Anaconda Prompt".
# For more detailed installation instructions, see https://chainercv.readthedocs.io/en/stable/install.html
# For other information about ChainerCV library, refer to https://chainercv.readthedocs.io/en/stable/
from chainercv.evaluations import eval_semantic_segmentation
from chainercv.datasets import VOCSemanticSegmentationDataset
```

In [4]:
```
# This colorize_mask class takes in a numpy segmentation mask,
#  and then converts it to a PIL Image for visualization.
#  Since by default the numpy matrix contains integers from
#  0,1,...,num_classes, we need to apply some color to this
#  so we can visualize easier! Refer to:
#  https://pillow.readthedocs.io/en/4.1.x/reference/Image.html#PIL.Image.Image.putpalette
palette = [0, 0, 0, 128, 0, 0, 0, 128, 0, 128, 128, 0, 0, 0, 128, 128, 0, 128, 0, 128, 128,
           128, 128, 128, 64, 0, 0, 192, 0, 0, 64, 128, 0, 192, 128, 0, 64, 0, 128, 192, 0, 128,
           64, 128, 128, 192, 128, 128, 0, 64, 0, 128, 64, 0, 0, 192, 0, 128, 192, 0, 0, 64, 128]

def colorize_mask(mask):
    new_mask = Image.fromarray(mask.astype(np.uint8)).convert('P')
    new_mask.putpalette(palette)

    return new_mask
```

In [5]:
```
# Below we will use a sample image-target pair from VOC training data
set to test your joint transforms.
# Running this block will automatically download the PASCAL VOC Datas
et (3.7GB) to DATASET_PATH if "download = True".
# The code below creates subdirectory "datasets" in the same location
as the notebook file, but
# you can modify DATASET_PATH to download the dataset to any custom d
irectory. Download takes a few minutes.
# On subsequent runs you may save time by setting "download = False"
 (the default value of this flag)

DATASET_PATH = 'datasets'

# Here, we obtain and visualize one sample (img, target) pair from VO
C training dataset and one from validation dataset.
# Note that operator [...] extracts the sample corresponding to the s
pecified index.
# Also, note the parameter download = True. Set this to False after y
ou download to save time on later runs.
sample1 = VOCSegmentation(DATASET_PATH, image_set='train', download =
False)[200]
sample2 = VOCSegmentation(DATASET_PATH, image_set='val')[20]

# We demonstrate two different (equivalent) ways to access image and
 target inside the samples.
img1, target1 = sample1
img2 = sample2[0]
target2 = sample2[1]

fig = plt.figure(figsize=(14,10))
ax1 = fig.add_subplot(2,2,1)
plt.title('sample1 - image')
ax1.imshow(img1)
ax2 = fig.add_subplot(2,2,2)
plt.title('sample1 - target')
ax2.imshow(target1)
ax3 = fig.add_subplot(2,2,3)
plt.title('sample2 - image')
ax3.imshow(img2)
ax4 = fig.add_subplot(2,2,4)
plt.title('sample2 - target')
ax4.imshow(target2)
```

Out[5]: <matplotlib.image.AxesImage at 0x7f17af540ed0>

Problem 1

Implement a set of "Joint Transform" functions to perform data augmentation in your dataset.

Neural networks are typically applied to transformed images. There are several important reasons for this:

1. The image data should is in certain required format (i.e. consistent spacial resolution to batch). The images should also be normalized and converted to the "tensor" data format expected by pytorch libraries.
2. Some transforms are used to perform randomized image domain transformations with the purpose of "data augmentation".

In this exercise, you will implement a set of different transform functions to do both of these things. Note that unlike classification nets, training semantic segmentation networks requires that some of the transforms are applied to both image and the corresponding "target" (Ground Truth segmentation mask). We refer to such transforms and their compositions as "Joint". In general, your Transform classes should take as the input both the image and the target, and return a tuple of the transformed input image and target. Be sure to use critical thinking to determine if you can apply the same transform function to both the input and the output.

For this problem you may use any of the `torchvision.transforms.functional` functions. For inspiration, refer to:

https://pytorch.org/tutorials/beginner/data_loading_tutorial.html (https://pytorch.org/tutorials/beginner/data_loading_tutorial.html)

https://pytorch.org/docs/stable/torchvision/transforms.html#module-torchvision.transforms.functional (https://pytorch.org/docs/stable/torchvision/transforms.html#module-torchvision.transforms.functional)

Example 1

This class takes a img, target pair, and then transform the pair such that they are in `Torch.Tensor()` format.

Solution:

In [6]:
```
class JointToTensor(object):
    def __call__(self, img, target):
        return tF.to_tensor(img), torch.from_numpy(np.array(target.co
nvert('P'), dtype=np.int32)).long()
```

In [7]:
```
# Check the transform by passing the image-target sample.

JointToTensor()(*sample1)
```

Out[7]:
```
(tensor([[[0.0431, 0.0510, 0.0353,  ..., 0.3137, 0.3725, 0.3490],
          [0.0196, 0.0431, 0.0235,  ..., 0.3294, 0.3569, 0.3294],
          [0.0392, 0.0510, 0.0471,  ..., 0.3412, 0.3765, 0.3608],
          ...,
          [0.9412, 0.9961, 1.0000,  ..., 0.9647, 0.9686, 0.9725],
          [1.0000, 0.9686, 0.9961,  ..., 0.9608, 0.9647, 0.9686],
          [1.0000, 0.9490, 1.0000,  ..., 0.9725, 0.9725, 0.9843]],

         [[0.0392, 0.0471, 0.0196,  ..., 0.1176, 0.1765, 0.1647],
          [0.0157, 0.0392, 0.0078,  ..., 0.1294, 0.1608, 0.1333],
          [0.0353, 0.0471, 0.0314,  ..., 0.1294, 0.1765, 0.1608],
          ...,
          [0.0157, 0.0667, 0.0706,  ..., 0.6549, 0.6588, 0.6588],
          [0.0784, 0.0431, 0.0667,  ..., 0.6510, 0.6510, 0.6549],
          [0.0745, 0.0235, 0.0784,  ..., 0.6627, 0.6627, 0.6706]],

         [[0.0314, 0.0392, 0.0157,  ..., 0.0118, 0.0706, 0.0549],
          [0.0078, 0.0314, 0.0039,  ..., 0.0235, 0.0549, 0.0275],
          [0.0275, 0.0392, 0.0275,  ..., 0.0275, 0.0706, 0.0549],
          ...,
          [0.0549, 0.0980, 0.0941,  ..., 0.2824, 0.2863, 0.2863],
          [0.1176, 0.0824, 0.0980,  ..., 0.2784, 0.2784, 0.2824],
          [0.1216, 0.0627, 0.1098,  ..., 0.2902, 0.2902, 0.2980]]]),
 tensor([[ 0,  0,  0,  ...,  0,  0,  0],
         [ 0,  0,  0,  ...,  0,  0,  0],
         [ 0,  0,  0,  ...,  0,  0,  0],
         ...,
         [15, 15, 15,  ...,  0,  0,  0],
         [15, 15, 15,  ...,  0,  0,  0],
         [15, 15, 15,  ...,  0,  0,  0]]))
```

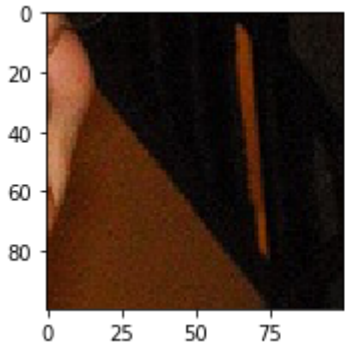
Example 2:

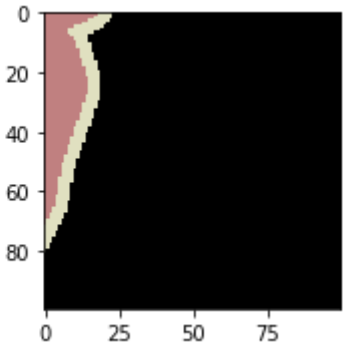
This class implements CenterCrop that takes an img, target pair, and then apply a crop about the center of the image such that the output resolution is $\text{size} \times \text{size}$.

Solution:

In [8]:
```
class JointCenterCrop(object):
    def __init__(self, size):
        """
        params:
            size (int) : size of the center crop
        """
        self.size = size

    def __call__(self, img, target):
        return (tF.five_crop(img, self.size)[4],
                tF.five_crop(target, self.size)[4])

img, target = JointCenterCrop(100)(*sample1)
fig = plt.figure(figsize=(12,6))
ax1 = fig.add_subplot(2,2,1)
ax1.imshow(img)
ax2 = fig.add_subplot(2,2,2)
ax2.imshow(target)
```

Out[8]: <matplotlib.image.AxesImage at 0x7f17af3e2ed0>





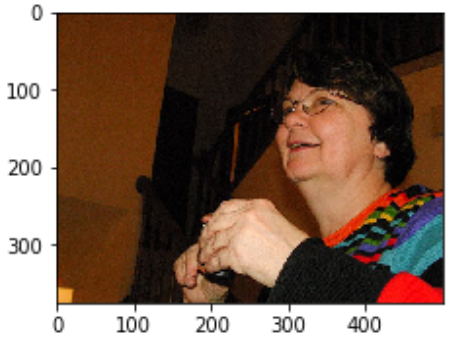
(a) Implement RandomFlip

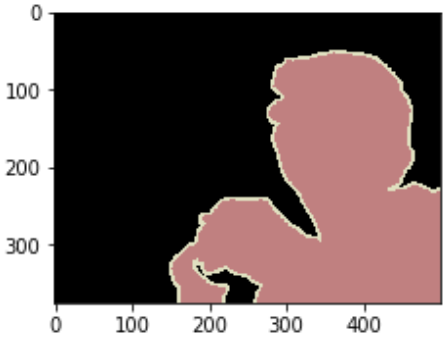
This class should take a img, target pair and then apply a horizontal flip across the vertical axis at random.

Solution:

In [9]:
```
class JointRandomFlip(object):
    def __call__(self, img, target):
        return (tF.hflip(img), tF.hflip(target))

img, target = JointRandomFlip()(*sample1)
fig = plt.figure(figsize=(12,6))
ax1 = fig.add_subplot(2,2,1)
ax1.imshow(img)
ax2 = fig.add_subplot(2,2,2)
ax2.imshow(target)
```

Out[9]: <matplotlib.image.AxesImage at 0x7f17af329910>





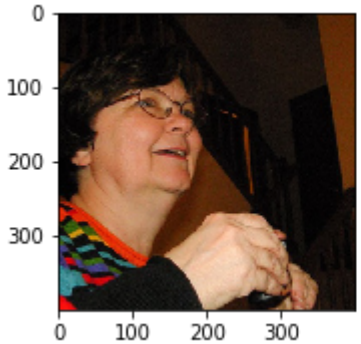
(b) Implement RandomResizeCrop

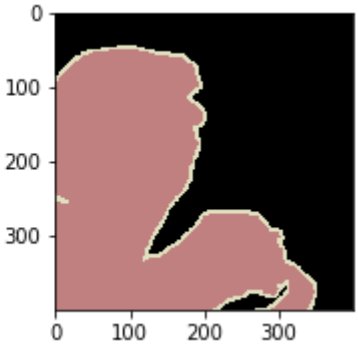
This class should take a img, target pair and then resize the images by a random scale between $[\text{minimum\_scale}, \text{maximum\_scale}]$, crop a random location of the image by $\min(\text{size}, \text{image\_height}, \text{image\_width})$ (where the size is passed in as an integer in the constructor), and then resize to $\text{size} \times \text{size}$ (again, the size passed in). The crop box should fit within the image.

Solution:

In [10]:
```
class JointRandomResizeCorp(object):
    def __init__(self, minimum_scale, maximum_scale, size):
        self.minimum_scale = minimum_scale
        self.maximum_scale = maximum_scale
        self.size = size

    def __call__(self, img, target):
        minimum_scale = self.minimum_scale
        maximum_scale = self.maximum_scale
        size = self.size

#       1. resize image to scale between [minimum_scale, maximum_scal
e]
        scale = np.random.uniform(minimum_scale, maximum_scale)
        newSize = (int(img.size[1] * scale), int(img.size[0] * scale
))
        img = tF.resize(img, newSize)
        target = tF.resize(target, newSize)

#       2. crop a random location of the image by min(size, image_hei
ght, image_width)
        img_width, img_height = img.size
        crop = min(size, img_height, img_width)
        x = random.randint(0, img.size[1] - crop + 1)
        y = random.randint(0, img.size[0] - crop + 1)

#       3. resize to size x size
        return (tF.resized_crop(img, x, y, crop, crop, size),
                tF.resized_crop(target, x, y, crop, crop, size))

img, target = JointRandomResizeCorp(0.9, 1.4, 400)(*sample1)
fig = plt.figure(figsize=(12,6))
ax1 = fig.add_subplot(2,2,1)
ax1.imshow(img)
ax2 = fig.add_subplot(2,2,2)
ax2.imshow(target)
```

Out[10]: <matplotlib.image.AxesImage at 0x7f17af271590>





(c) Implement Normalize

This class should take a img, target pair and then normalize the images by subtracting the mean and dividing variance.

Solution:

In [11]:
```
norm = ([0.485, 0.456, 0.406],
        [0.229, 0.224, 0.225])

class JointNormalize(object):
    def __init__(self, mean, std):
        self.mean = mean
        self.std = std

    def __call__(self, tensor, target):
        return (tF.normalize(tensor, self.mean, self.std), target)

tensor = JointToTensor()(*sample1)
JointNormalize(*norm)(*tensor)
```

Out[11]:
```
(tensor([[[-1.9295, -1.8953, -1.9638,  ..., -0.7479, -0.4911, -0.593
8],
          [-2.0323, -1.9295, -2.0152,  ..., -0.6794, -0.5596, -0.679
4],
          [-1.9467, -1.8953, -1.9124,  ..., -0.6281, -0.4739, -0.542
4],
          ...,
          [ 1.9920,  2.2318,  2.2489,  ...,  2.0948,  2.1119,  2.129
0],
          [ 2.2489,  2.1119,  2.2318,  ...,  2.0777,  2.0948,  2.111
9],
          [ 2.2489,  2.0263,  2.2489,  ...,  2.1290,  2.1290,  2.180
4]],

         [[-1.8606, -1.8256, -1.9482,  ..., -1.5105, -1.2479, -1.300
4],
          [-1.9657, -1.8606, -2.0007,  ..., -1.4580, -1.3179, -1.440
5],
          [-1.8782, -1.8256, -1.8957,  ..., -1.4580, -1.2479, -1.317
9],
          ...,
          [-1.9657, -1.7381, -1.7206,  ...,  0.8880,  0.9055,  0.905
5],
          [-1.6856, -1.8431, -1.7381,  ...,  0.8704,  0.8704,  0.888
0],
          [-1.7031, -1.9307, -1.6856,  ...,  0.9230,  0.9230,  0.958
0]],

         [[-1.6650, -1.6302, -1.7347,  ..., -1.7522, -1.4907, -1.560
4],
          [-1.7696, -1.6650, -1.7870,  ..., -1.6999, -1.5604, -1.682
4],
          [-1.6824, -1.6302, -1.6824,  ..., -1.6824, -1.4907, -1.560
4],
          ...,
          [-1.5604, -1.3687, -1.3861,  ..., -0.5495, -0.5321, -0.532
1],
          [-1.2816, -1.4384, -1.3687,  ..., -0.5670, -0.5670, -0.549
5],
          [-1.2641, -1.5256, -1.3164,  ..., -0.5147, -0.5147, -0.479
8]]]),
 tensor([[ 0,  0,  0,  ...,  0,  0,  0],
         [ 0,  0,  0,  ...,  0,  0,  0],
         [ 0,  0,  0,  ...,  0,  0,  0],
         ...,
         [15, 15, 15,  ...,  0,  0,  0],
         [15, 15, 15,  ...,  0,  0,  0],
         [15, 15, 15,  ...,  0,  0,  0]]))
```

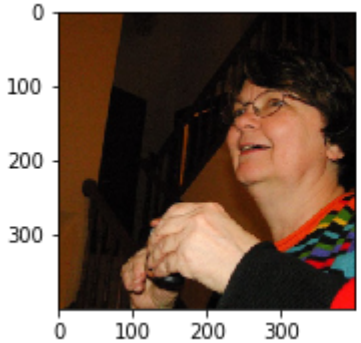
(d) Compose the transforms together:

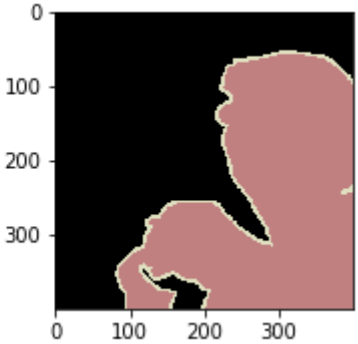
Use `JointCompose` (fully implemeted below) to compose the implemented transforms together in some random order. Verify the output makes sense and visualize it.

In [12]:
```
# This class composes transofrmations from a given list of image tran
sforms (expected in the argument). Such compositions
# will be applied to the dataset during training. This cell is fully
 implemented.

class JointCompose(object):
    def __init__(self, transforms):
        """
        params:
           transforms (list) : list of transforms
        """
        self.transforms = transforms

    # We override the __call__ function such that this class can be
    # called as a function i.e. JointCompose(transforms)(img, target)
    # Such classes are known as "functors"
    def __call__(self, img, target):
        """
        params:
            img (PIL.Image)    : input image
            target (PIL.Image) : ground truth label
        """
        assert img.size == target.size
        for t in self.transforms:
            img, target = t(img, target)
        return img, target
```

In [13]:
```
# Student Answer:
transforms = [JointRandomResizeCorp(0.9, 1.4, 400), JointRandomFlip
()]

img, target = JointCompose(transforms)(*sample1)
fig = plt.figure(figsize=(12,6))
ax1 = fig.add_subplot(2,2,1)
ax1.imshow(img)
ax2 = fig.add_subplot(2,2,2)
ax2.imshow(target)
```

Out[13]: <matplotlib.image.AxesImage at 0x7f17af1b2410>





(e) Compose the transforms together: use `JointCompose` to compose the implemented transforms for:

1. A sanity dataset that will contain 1 single image. Your objective is to overfit on this 1 image, so choose your transforms and parameters accordingly.

2. A training dataset that will contain the training images. The goal here is to generalize to the validation set, which is unseen.

3. A validation dataset that will contain the validation images. The goal here is to measure the 'true' performance

In [14]:
```
norm = ([0.485, 0.456, 0.406],
        [0.229, 0.224, 0.225])

# Student Answer:
sanity_joint_transform = JointCompose([JointToTensor(), JointNormaliz
e(*norm)])

train_joint_transform = JointCompose([JointRandomResizeCorp(0.9, 1.2,
512), JointRandomFlip(),
                                      JointToTensor(), JointNormalize
(*norm)])

val_joint_transform = JointCompose([JointToTensor(), JointNormalize(*
norm)])
```

This code below will then apply `train_joint_transform` to the entire dataset.

In [15]:
```
# Apply the Joint-Compose transformations above to create three datas
ets and the corresponding Data-Loaders.
# This cell is fully implemented.

# This single image data(sub)set can help to better understand and to
debug the network training process.
# Optional integer parameter 'sanity_check' specifies the index of th
e image-target pair and creates a single image dataset.
# Note that we use the same image (index=200) as used for sample1.
sanity_data = VOCSegmentation(
    DATASET_PATH,
    image_set = 'train',
    transforms = sanity_joint_transform,
    sanity_check = 200
)

# This is a standard VOC data(sub)set used for training semantic segm
entation networks
train_data = VOCSegmentation(
    DATASET_PATH,
    image_set = 'train',
    transforms = train_joint_transform
)

# This is a standard VOC data(sub)set used for validating semantic se
gmentation networks
val_data = VOCSegmentation(
    DATASET_PATH,
    image_set='val',
    transforms = val_joint_transform
)

# Increase TRAIN_BATCH_SIZE if you are using GPU to speed up trainin
g.
# When batch size changes, the learning rate may also need to be adju
sted.
# Note that batch size maybe limited by your GPU memory, so adjust if
you get "run out of GPU memory" error.
TRAIN_BATCH_SIZE = 16

# If you are NOT using Windows, set NUM_WORKERS to anything you want,
e.g. NUM_WORKERS = 4,
# but Windows has issues with multi-process dataloaders, so NUM_WORKE
RS must be 0 for Windows.
NUM_WORKERS = 0

sanity_loader = DataLoader(sanity_data, batch_size=1, num_workers=NUM
_WORKERS, shuffle=False)
train_loader = DataLoader(train_data, batch_size=TRAIN_BATCH_SIZE, nu
m_workers=NUM_WORKERS, shuffle=True)
val_loader = DataLoader(val_data, batch_size=1, num_workers=NUM_WORKE
RS, shuffle=False)
```

Problem 2

(a) Implement encoder/decoder segmentation CNN using PyTorch.

You must follow the general network architecture specified in the image below. Note that since convolutional layers are the main building blocks in common network architectures for image analysis, the corresponding blocks are typically unlabeled in the network diagrams. The network should have 5 (pre-trained) convolutional layers from "resnet" in the encoder part, two upsampling layers, and one skip connection. For the layer before the final upsampling layer, lightly experiment with some combination of Conv, ReLU, BatchNorm, and/or other layers to see how it affects performance.

You should choose specific parameters for all layers, but the overall structure should be restricted to what is shown in the illustration above. For inspiration, you can refer to papers in the citation section of the following link to DeepLab (e.g. specific parameters for each layer): http://liangchiehchen.com/projects/DeepLab.html (http://liangchiehchen.com/projects/DeepLab.html). The first two papers in the citation section are particularly relevant.

In your implementation, you can use a base model of choice (you can use `torchvision.models` as a starting point), but we suggest that you learn the properties of each base model and choose one according to the computational resources available to you.

Note: do not apply any post-processing (such as DenseCRF) to the output of your net.

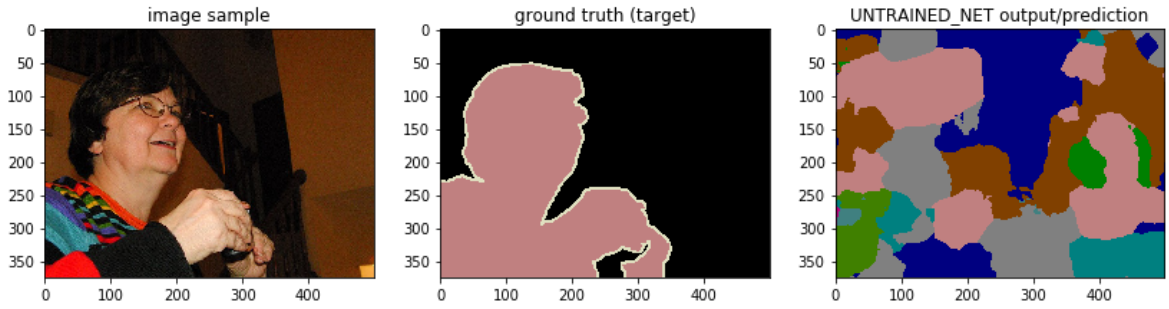
In [16]:
```
import torchvision.models as models

class MyNet(nn.Module):
    def __init__(self, num_classes, criterion=None):
        super(MyNet, self).__init__()
        # Implement me
        self.num_classes = num_classes
        self.resnet18 = models.resnet18(pretrained = True)
        self.conv1 = nn.Conv2d(576, 256, 5, 1)
        self.conv2 = nn.Conv2d(256, num_classes, 1, 1)
        self.bn = nn.BatchNorm2d(256)

        self.criterion = criterion

    def forward(self, inp, gts=None):
        # Implement me
        inp_shape = inp.shape
        resnet18 = self.resnet18

#       1. 5 layers of resnet
#       https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py#L198 ->
#       https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py#L207

        inp = resnet18.conv1(inp)
        inp = resnet18.bn1(inp)
        inp = resnet18.relu(inp)
        inp = resnet18.maxpool(inp)

#       output for concat
        concat_copy = inp.clone()
        concat_copy_shape = concat_copy.shape

        inp = resnet18.layer1(inp)
        inp = resnet18.layer2(inp)
        inp = resnet18.layer3(inp)
        inp = resnet18.layer4(inp)

#       2. First Upsample
        inp = F.interpolate(inp, size = (concat_copy_shape[2], concat_copy_shape[3]), mode = 'bilinear',
                            align_corners = True)

#       3. Concat
        inp = torch.cat((inp, concat_copy), 1)

#       4. Some layers of conv, relu, bn
        inp = self.conv1(inp)
        inp = F.relu(inp)
        inp = self.bn(inp)
        inp = self.conv2(inp)
        inp = F.relu(inp)

#       5. Second Upsample
        inp = F.interpolate(inp, size = (inp_shape[2], inp_shape[3]),
```

```
mode = 'bilinear', align_corners = True)

        lfinal = inp
        if self.training:
            # Return the loss if in training mode
            return self.criterion(lfinal, gts)
        else:
            # Return the actual prediction otherwise
            return lfinal
```

(b) Create UNTRAINED_NET and run on a sample image

In [17]:
```
untrained_net = MyNet(21).eval()
sample_img, sample_target = JointNormalize(*norm)(*JointToTensor()(*s
ample1))
untrained_output = untrained_net.forward(sample_img[None])

fig = plt.figure(figsize=(14,10))
ax = fig.add_subplot(1,3,1)
plt.title('image sample')
ax.imshow(sample1[0])
ax = fig.add_subplot(1,3,2)
plt.title('ground truth (target)')
ax.imshow(sample1[1])
ax = fig.add_subplot(1,3,3)
plt.title('UNTRAINED_NET output/prediction')
ax.imshow(colorize_mask(torch.argmax(untrained_output, dim=1).numpy()
[0]))
```

Out[17]: `<matplotlib.image.AxesImage at 0x7f179c6b7fd0>`



Problem 3

(a) Implement the loss function (Cross Entropy Loss). Do not use already implemented versions of this loss function.

Feel free to use functions like `F.log_softmax` and `F.nll_loss` (if you want to, or you can just implement the math).

In [18]:
```
# Student Answer:

class MyCrossEntropyLoss():
    def __init__(self, ignore_index):
        self.ignore_index = ignore_index

    def __call__(self, untrained_output, sample_target):
        softmax = F.log_softmax(untrained_output, dim = 1)
        return F.nll_loss(softmax, sample_target, ignore_index = self
.ignore_index)
```

(b) Compare against the existing CrossEntropyLoss function on your sample output from your neural network.

In [19]:
```
criterion = nn.CrossEntropyLoss(ignore_index=255)

print(criterion(untrained_output, sample_target[None]))

my_criterion = MyCrossEntropyLoss(ignore_index=255)

print(my_criterion(untrained_output, sample_target[None]))
```

```
tensor(3.0173, grad_fn=<NllLoss2DBackward>)
tensor(3.0173, grad_fn=<NllLoss2DBackward>)
```

Problem 4

(a) Use standard function `eval_semantic_segmentation` (already imported from chainerCV) to compute "mean intersection over union" for the output of UNTRAINED_NET on sample1 (`untrained_output`) using the target for sample1. Read documentations for function `eval_semantic_segmentation` to properly set its input parameters.

In [20]:
```
# Write code to propely compute 'pred' and 'gts' as arguments for fun
ction 'eval_semantic_segemntation'
pred = torch.argmax(untrained_output, dim=1).numpy()[0]

gts = torch.from_numpy(np.array(sample1[1].convert('P'), dtype=np.int
32)).long().numpy()
gts[gts == 255] = -1

conf = eval_semantic_segmentation(pred[None], gts[None])

print("mIoU for the sample image / ground truth pair: {}".format(conf
['miou']))
```

```
mIoU for the sample image / ground truth pair: 0.019176565539505815

/home/tempo/anaconda3/lib/python3.7/site-packages/chainercv/evaluatio
ns/eval_semantic_segmentation.py:91: RuntimeWarning: invalid value en
countered in true_divide
  iou = np.diag(confusion) / iou_denominator
/home/tempo/anaconda3/lib/python3.7/site-packages/chainercv/evaluatio
ns/eval_semantic_segmentation.py:168: RuntimeWarning: invalid value e
ncountered in true_divide
  class_accuracy = np.diag(confusion) / np.sum(confusion, axis=1)
```

(b) Write the validation loop.

In [21]:
```
# Modified from Classification notebook
def validate(val_loader, net):

    iou_arr = []
    net.eval()
    val_loss = 0
    correct = 0
    with torch.no_grad():
        for i, data in enumerate(val_loader):
            inputs, masks = data

            if USE_GPU:
                # Write me
                inputs = inputs.cuda()
                masks = masks.cuda()
                net = net.cuda()
            else:
                # Write me
                pass

            # Write me
            output = net(inputs)
            val_loss += MyCrossEntropyLoss(ignore_index = 255)(output
, masks)
            preds = torch.argmax(output, dim = 1).cpu().numpy()

            gts = torch.from_numpy(np.array(masks.cpu(), dtype = np.i
nt32)).long().numpy()
            gts[gts == 255] = -1
            # Hint: make sure the range of values of the ground truth
is what you expect

            conf = eval_semantic_segmentation(preds, gts)

            iou_arr.append(conf['miou'])

    return val_loss, (sum(iou_arr) / len(iou_arr))
```

(c) Run the validation loop for UNTRAINED_NET against the sanity validation dataset.

In [22]:
```
%%time
print("mIoU over the sanity dataset:{}".format(validate(sanity_loader
, untrained_net)[1]))
```

```
mIoU over the sanity dataset:0.01917636711676755
CPU times: user 2.95 s, sys: 717 ms, total: 3.67 s
Wall time: 3.03 s
```

Problem 5

(a) Define an optimizer to train the given loss function.

Feel free to choose your optimizer of choice from https://pytorch.org/docs/stable/optim.html (https://pytorch.org/docs/stable/optim.html).

In [23]:
```
# Modified from Classification notebook
def get_optimizer(net):
    # Write me
    optimizer = torch.optim.SGD(net.parameters(),
                                lr=0.001,
                                weight_decay=1e-5,
                                momentum=0.5,
                                nesterov=False)
    return optimizer
```

(b) Write the training loop to train the network.

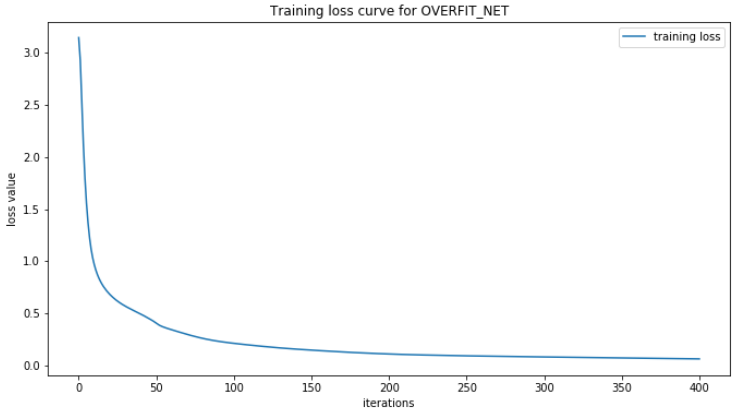
In [24]:
```
# Modified from Classification notebook
def train(train_loader, net, optimizer, loss_graph):

    for i, data in enumerate(train_loader):

        inputs, masks = data

        if USE_GPU:
            # Write me
            inputs = inputs.cuda()
            net = net.cuda()
            masks = masks.cuda()

        # Write me
        optimizer.zero_grad()
        main_loss = net(inputs, gts = masks)
        loss_graph.append(main_loss.item())
        main_loss.backward()
        optimizer.step()

    return main_loss
```

(c) Create OVERFIT_NET and train it on the single image dataset.

Single image training is helpful for debugging and hyper-parameter tuning (e.g. learning rate, etc.) as it is fast even on a single CPU. In particular, you can work with a single image until your loss function is consistently decreasing during training loop and the network starts producing a reasonable output for this training image. Training on a single image also teaches about overfitting, particualrly when comparing it with more thorough forms of network training.

In [25]:
```
%%time
%matplotlib notebook

# The whole training on a single image (20-40 epochs) should take onl
y a minute or two on a CPU (and a few seconds on GPU).
# Below we create a (deep) copy of untrained_net and train it on a si
ngle training image (leading to gross overfitting).
# Later, we will create a separate (deep) copy of untrained_net to be
trained on full training dataset.
# NOTE: Normally, one can create a new net via declaration new_net =
 MyNet(21). But, randomization of weights when new nets
# are declared that way creates *different* untrained nets. This note
book compares different versions of network training.
# For this comparison to be direct and fair, it is better to train (d
eep) copies of the exact same untrained_net.
overfit_net = copy.deepcopy(untrained_net)

# set loss function for the net
overfit_net.criterion = nn.CrossEntropyLoss(ignore_index=255)

# You can change the number of EPOCHS
EPOCH = 401

# switch to train mode (original untrained_net was set to eval mode)
overfit_net.train()

optimizer = get_optimizer(overfit_net)

print("Starting Training...")

loss_graph = []

fig = plt.figure(figsize=(12,6))
plt.subplots_adjust(bottom=0.2,right=0.85,top=0.95)
ax = fig.add_subplot(1,1,1)

for e in range(EPOCH):
    loss = train(sanity_loader, overfit_net, optimizer, loss_graph)
    ax.clear()
    ax.set_xlabel('iterations')
    ax.set_ylabel('loss value')
    ax.set_title('Training loss curve for OVERFIT_NET')
    ax.plot(loss_graph, label='training loss')
    ax.legend(loc='upper right')
    fig.canvas.draw()
    if(e % int(EPOCH / 10) == 0):
        print("Epoch: {} Loss: {}".format(e, loss))

%matplotlib inline
```

```
Starting Training...
```

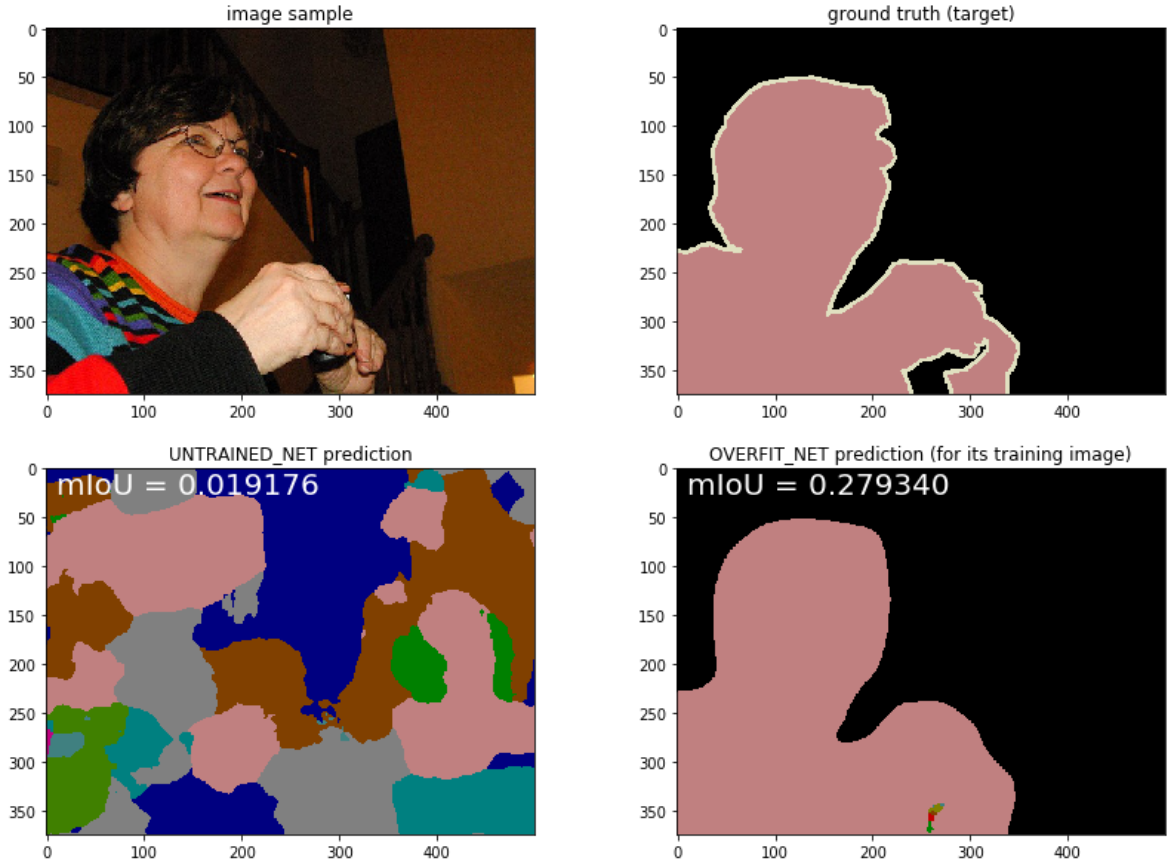



```
Epoch: 0 Loss: 3.1440441608428955
Epoch: 40 Loss: 0.4935438632965088
Epoch: 80 Loss: 0.26063480973243713
Epoch: 120 Loss: 0.1824187934398651
Epoch: 160 Loss: 0.13936130702495575
Epoch: 200 Loss: 0.11089920997619629
Epoch: 240 Loss: 0.09628084301948547
Epoch: 280 Loss: 0.08673789352178574
Epoch: 320 Loss: 0.07921924442052841
Epoch: 360 Loss: 0.07195495069026947
Epoch: 400 Loss: 0.06422226130962372
CPU times: user 3min 40s, sys: 8.15 s, total: 3min 48s
Wall time: 57.3 s
```

Qualitative and quantitative evaluation of predictions (untrained vs overfit nets) - fully implemented.

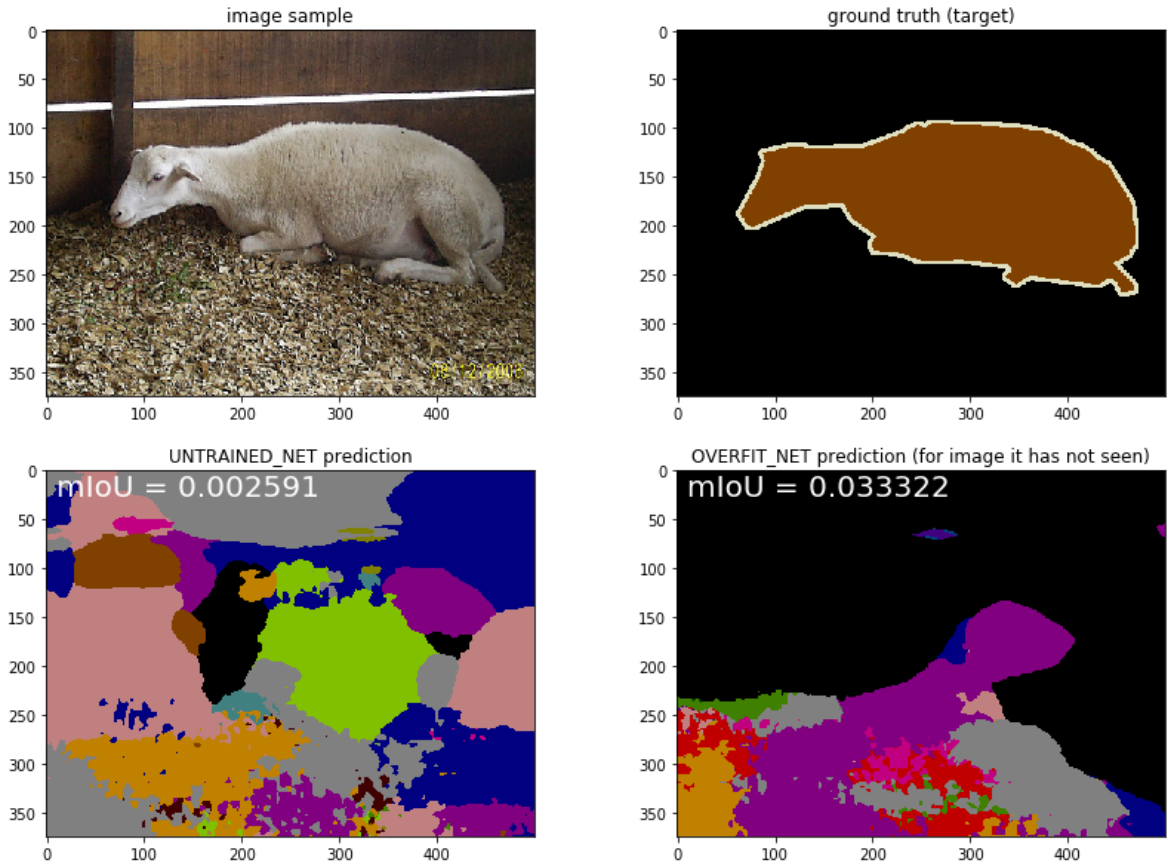
In [26]:
```
# switch back to evaluation mode
overfit_net.eval()

sample_img, sample_target = JointNormalize(*norm)(*JointToTensor()(*s
ample1))
if USE_GPU:
    sample_img = sample_img.cuda()
sample_output_O = overfit_net.forward(sample_img[None])
sample_output_U = untrained_net.forward(sample_img[None])

# computing mIOU (quantitative measure of accuracy for network predic
tions)
if USE_GPU:
    pred_O = torch.argmax(sample_output_O, dim=1).cpu().numpy()[0]
    pred_U = torch.argmax(sample_output_U, dim=1).cpu().numpy()[0]
else:
    pred_O = torch.argmax(sample_output_O, dim=1).numpy()[0]
    pred_U = torch.argmax(sample_output_U, dim=1).numpy()[0]

gts = torch.from_numpy(np.array(sample1[1].convert('P'), dtype=np.int
32)).long().numpy()
gts[gts == 255] = -1
conf_O = eval_semantic_segmentation(pred_O[None], gts[None])
conf_U = eval_semantic_segmentation(pred_U[None], gts[None])


fig = plt.figure(figsize=(14,10))
ax1 = fig.add_subplot(2,2,1)
plt.title('image sample')
ax1.imshow(sample1[0])
ax2 = fig.add_subplot(2,2,2)
plt.title('ground truth (target)')
ax2.imshow(sample1[1])
ax3 = fig.add_subplot(2,2,3)
plt.title('UNTRAINED_NET prediction')
ax3.text(10, 25, 'mIoU = {:_>8.6f}'.format(conf_U['miou']), fontsize=
20, color='white')
ax3.imshow(colorize_mask(torch.argmax(sample_output_U, dim=1).cpu().n
umpy()[0]))
ax4 = fig.add_subplot(2,2,4)
plt.title('OVERFIT_NET prediction (for its training image)')
ax4.text(10, 25, 'mIoU = {:_>8.6f}'.format(conf_O['miou']), fontsize=
20, color='white')
ax4.imshow(colorize_mask(torch.argmax(sample_output_O, dim=1).cpu().n
umpy()[0]))
```

Out[26]: <matplotlib.image.AxesImage at 0x7f17961b9c50>



In [27]:
```
sample_img, sample_target = JointNormalize(*norm)(*JointToTensor()(*s
ample2))
if USE_GPU:
    sample_img = sample_img.cuda()
sample_output_O = overfit_net.forward(sample_img[None])
sample_output_U = untrained_net.forward(sample_img[None])

# computing mIOU (quantitative measure of accuracy for network predic
tions)
if USE_GPU:
    pred_O = torch.argmax(sample_output_O, dim=1).cpu().numpy()[0]
    pred_U = torch.argmax(sample_output_U, dim=1).cpu().numpy()[0]
else:
    pred_O = torch.argmax(sample_output_O, dim=1).numpy()[0]
    pred_U = torch.argmax(sample_output_U, dim=1).numpy()[0]

gts = torch.from_numpy(np.array(sample2[1].convert('P'), dtype=np.int
32)).long().numpy()
gts[gts == 255] = -1
conf_O = eval_semantic_segmentation(pred_O[None], gts[None])
conf_U = eval_semantic_segmentation(pred_U[None], gts[None])

fig = plt.figure(figsize=(14,10))
ax1 = fig.add_subplot(2,2,1)
plt.title('image sample')
ax1.imshow(sample2[0])
ax2 = fig.add_subplot(2,2,2)
plt.title('ground truth (target)')
ax2.imshow(sample2[1])
ax3 = fig.add_subplot(2,2,3)
plt.title('UNTRAINED_NET prediction')
ax3.text(10, 25, 'mIoU = {:_>8.6f}'.format(conf_U['miou']), fontsize=
20, color='white')
ax3.imshow(colorize_mask(torch.argmax(sample_output_U, dim=1).cpu().n
umpy()[0]))
ax4 = fig.add_subplot(2,2,4)
plt.title('OVERFIT_NET prediction (for image it has not seen)')
ax4.text(10, 25, 'mIoU = {:_>8.6f}'.format(conf_O['miou']), fontsize=
20, color='white')
ax4.imshow(colorize_mask(torch.argmax(sample_output_O, dim=1).cpu().n
umpy()[0]))
```

Out[27]: <matplotlib.image.AxesImage at 0x7f1795fe7ed0>



Run the validation loop for OVERFIT_NET against the sanity dataset (an image it was trained on) - fully implemented

In [28]:
```
%%time
print("mIoU for OVERFIT_NET over its training image:{}".format(valida
te(sanity_loader, overfit_net)[1]))
```

```
mIoU for OVERFIT_NET over its training image:0.2793395039194985
CPU times: user 172 ms, sys: 15.5 ms, total: 187 ms
Wall time: 52.3 ms
```

WARNING: For the remaining part of the assignment (below) it is advisable to switch to GPU mode as running each validation and training loop on the whole training set takes over an hour on CPU (there are several such loops below). Note that GPU mode is helpful only if you have a sufficiently good NVIDIA gpu (not older than 2-3 years) and cuda installed on your computer. If you do not have a sufficiently good graphics card available, you can still finish the remaining part in CPU mode (takes a few hours), as the cells below are mostly implemented and test your code written and debugged in the earlier parts above. You can also switch to Google Colaboratory to run the remaining parts below.

You can use validation-data experiments below to tune your hyper-parameters. Normally, validation data is used exactly for this purpose. For actual competitions, testing data is not public and you can not tune hyper-parameters on in.

(d) Evaluate UNTRAINED_NET and OVERFIT_NET on validation dataset.

Run the validation loop for UNTRAINED_NET against the validation dataset:

In [29]:
```
%%time
# This will be slow on CPU (around 1 hour or more). On GPU it should
 take only a few minutes (depending on your GPU).
print("mIoU for UNTRAINED_NET over the entire dataset:{}".format(vali
date(val_loader, untrained_net)[1]))
```

```
mIoU for UNTRAINED_NET over the entire dataset:0.004496984206238499
CPU times: user 3min 50s, sys: 12.4 s, total: 4min 3s
Wall time: 1min
```

Run the validation loop for OVERFIT_NET against the validation dataset (it has not seen):

In [30]:
```
%%time
# This will be slow on CPU (around 1 hour or more). On GPU it should
 take only a few minutes (depending on your GPU).
print("mIoU for OVERFIT_NET over the validation dataset:{}".format(va
lidate(val_loader, overfit_net)[1]))
```

```
mIoU for OVERFIT_NET over the validation dataset:0.05855447182023666
CPU times: user 3min 54s, sys: 12.6 s, total: 4min 6s
Wall time: 1min 1s
```

(e) Explain in a few sentences the quantitative results observed in (d), (f), and (g):

Student answer:

As the name suggests overfit_net has a great mIoU. But it will not perform well on unseen data. Running untrained_net and overfit_net on validation set gives us bad results, as expected, since the first network is untrained, and the second one is overfitted to one image.

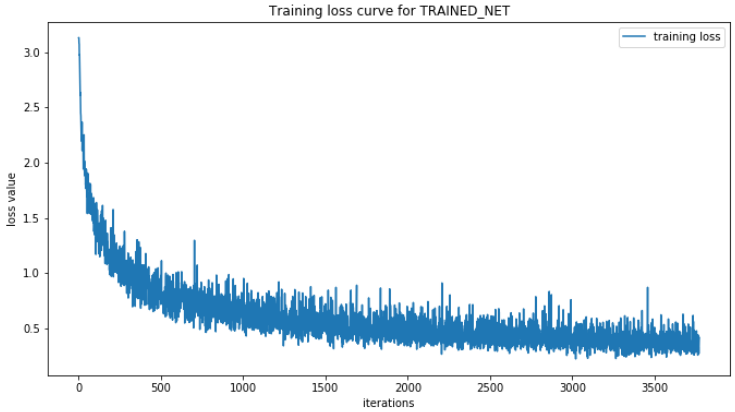
For trained_net, after running it for ~10 epochs, loss is around 0.1 - 0.5. running it for more epochs wont necessarily give lower loss or a higher mIoU. To get better results, after about 10 epochs, we should lower the learning rate to get better results

Comparing trained_net and overfit_net, overfit_new will definately do better on the image it overfit on than trained_net. For all images, trained_net will be better

(f) Create TRAINED_NET and train it on the full training dataset:

In [31]:
```
%%time
%matplotlib notebook

# This training will be very slow on a CPU (>1hour per epoch). Ideall
y, this should be run in GPU mode (USE_GPU=True)
# taking only a few minutes per epoch (depending on your GPU and batc
h size). Thus, before proceeding with this excercise,
# it is highly advisable that you first finish debugging your net cod
e. In particular, make sure that OVERFIT_NET behaves
# reasonably, e.g. its loss monotonically decreases during training a
nd its output is OK (for the image it was trained on).
# Below we create another (deep) copy of untrained_net. Unlike OVERFI
T_NET it will be trained on a full training dataset.
trained_net = copy.deepcopy(untrained_net)

# set loss function for the net
trained_net.criterion = nn.CrossEntropyLoss(ignore_index=255)


# You can change the number of EPOCHS below. Since each epoch for TRA
INED_NET iterates over all training dataset images,
# the number of required epochs could be smaller compared to OFERFIT_
NET where each epoch iterates over one-image-dataset)
EPOCH = 41

# switch to train mode (original untrained_net was set to eval mode)
trained_net.train()

optimizer = get_optimizer(trained_net)

print("Starting Training...")

loss_graph = []

fig = plt.figure(figsize=(12,6))
plt.subplots_adjust(bottom=0.2,right=0.85,top=0.95)
ax = fig.add_subplot(1,1,1)

for e in range(EPOCH):
    loss = train(train_loader, trained_net, optimizer, loss_graph)
    ax.clear()
    ax.set_xlabel('iterations')
    ax.set_ylabel('loss value')
    ax.set_title('Training loss curve for TRAINED_NET')
    ax.plot(loss_graph, label='training loss')
    ax.legend(loc='upper right')
    fig.canvas.draw()
    print("Epoch: {} Loss: {}".format(e, loss))

%matplotlib inline
```

```
Starting Training...
```

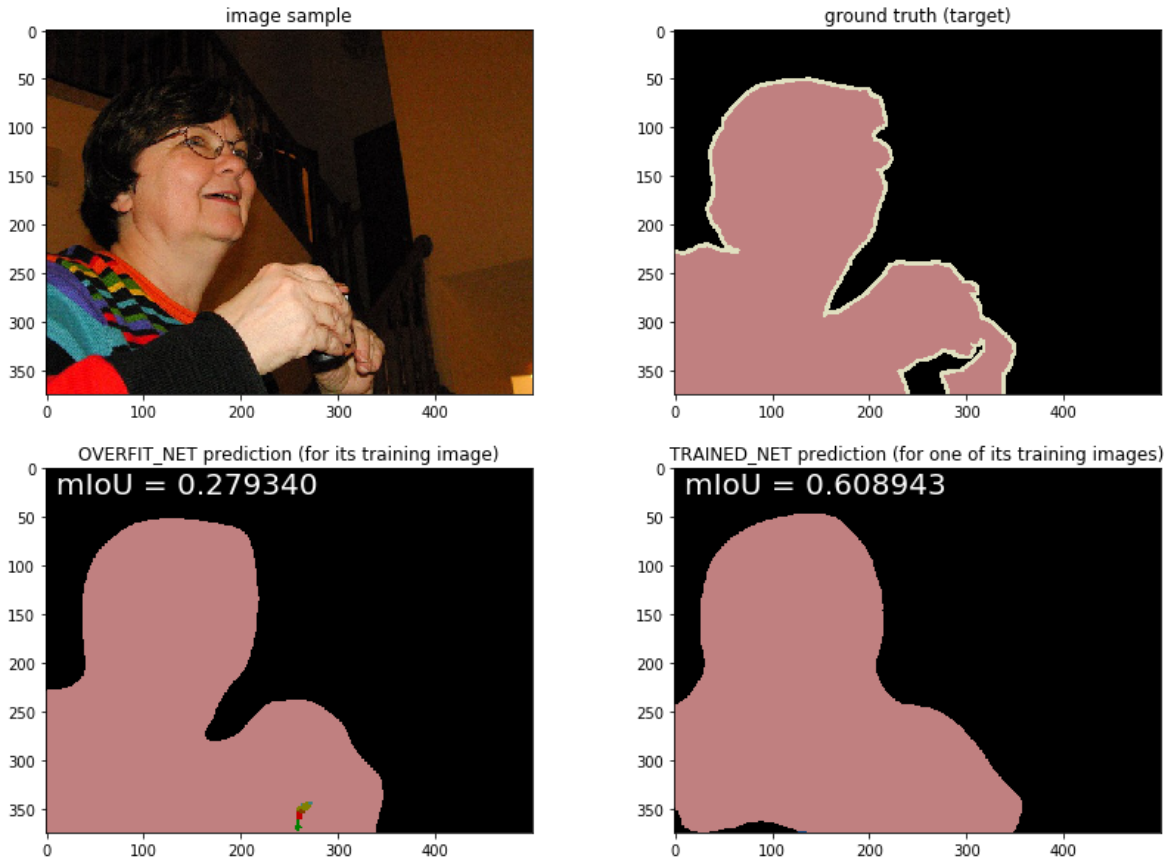


```
Epoch: 0 Loss: 1.547663688659668
Epoch: 1 Loss: 1.1788172721862793
Epoch: 2 Loss: 1.2515555620193481
Epoch: 3 Loss: 1.1394387483596802
Epoch: 4 Loss: 0.9269146919250488
Epoch: 5 Loss: 0.7210343480110168
Epoch: 6 Loss: 0.922466516494751
Epoch: 7 Loss: 0.6717900633811951
Epoch: 8 Loss: 0.7762126922607422
Epoch: 9 Loss: 0.6219239830970764
Epoch: 10 Loss: 0.6511335968971252
Epoch: 11 Loss: 0.6623179316520691
Epoch: 12 Loss: 0.564846396446228
Epoch: 13 Loss: 0.6812073588371277
Epoch: 14 Loss: 0.7872440814971924
Epoch: 15 Loss: 0.7050537467002869
Epoch: 16 Loss: 0.8706715703010559
Epoch: 17 Loss: 0.49757999181747437
Epoch: 18 Loss: 0.4273013174533844
Epoch: 19 Loss: 0.5540392994880676
Epoch: 20 Loss: 0.4865323007106781
Epoch: 21 Loss: 0.6828572750091553
Epoch: 22 Loss: 0.37220674753189087
Epoch: 23 Loss: 0.910569965839386
Epoch: 24 Loss: 0.44033169746398926
Epoch: 25 Loss: 0.3514823615550995
Epoch: 26 Loss: 0.4375629127025604
Epoch: 27 Loss: 0.5298530459403992
Epoch: 28 Loss: 0.4038936197757721
Epoch: 29 Loss: 0.4329994320869446
Epoch: 30 Loss: 0.45243391394615173
Epoch: 31 Loss: 0.31359484791755676
Epoch: 32 Loss: 0.4150051474571228
Epoch: 33 Loss: 0.37078210711479187
Epoch: 34 Loss: 0.48440858721733093
Epoch: 35 Loss: 0.3509039282798767
Epoch: 36 Loss: 0.5577936768531799
Epoch: 37 Loss: 0.4484555423259735
Epoch: 38 Loss: 0.5008074045181274
Epoch: 39 Loss: 0.28901341557502747
Epoch: 40 Loss: 0.4149225652217865
CPU times: user 2h 3min 43s, sys: 7min 30s, total: 2h 11min 13s
Wall time: 33min 42s
```

(g) Qualitative and quantitative evaluation of predictions (OVERFIT_NET vs TRAINED_NET):

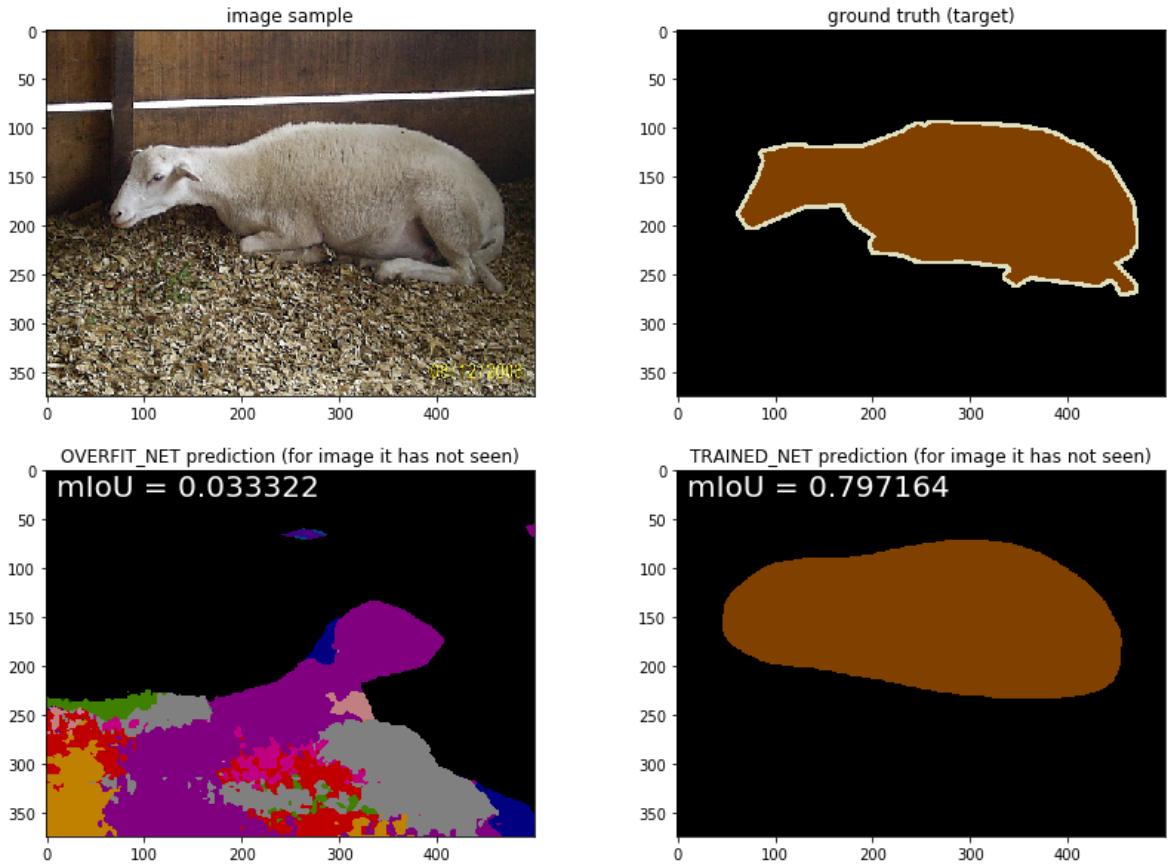
In [32]:
```
# switch back to evaluation mode
trained_net.eval()

sample_img, sample_target = JointNormalize(*norm)(*JointToTensor()(*s
ample1))
if USE_GPU:
    sample_img = sample_img.cuda()
sample_output_O = overfit_net.forward(sample_img[None])
sample_output_T = trained_net.forward(sample_img[None])

# computing mIOU (quantitative measure of accuracy for network predic
tions)
pred_T = torch.argmax(sample_output_T, dim=1).cpu().numpy()[0]
pred_O = torch.argmax(sample_output_O, dim=1).cpu().numpy()[0]
gts = torch.from_numpy(np.array(sample1[1].convert('P'), dtype=np.int
32)).long().numpy()
gts[gts == 255] = -1
conf_T = eval_semantic_segmentation(pred_T[None], gts[None])
conf_O = eval_semantic_segmentation(pred_O[None], gts[None])


fig = plt.figure(figsize=(14,10))
ax1 = fig.add_subplot(2,2,1)
plt.title('image sample')
ax1.imshow(sample1[0])
ax2 = fig.add_subplot(2,2,2)
plt.title('ground truth (target)')
ax2.imshow(sample1[1])
ax3 = fig.add_subplot(2,2,3)
plt.title('OVERFIT_NET prediction (for its training image)')
ax3.text(10, 25, 'mIoU = {:_>8.6f}'.format(conf_O['miou']), fontsize=
20, color='white')
ax3.imshow(colorize_mask(torch.argmax(sample_output_O, dim=1).cpu().n
umpy()[0]))
ax4 = fig.add_subplot(2,2,4)
plt.title('TRAINED_NET prediction (for one of its training images)')
ax4.text(10, 25, 'mIoU = {:_>8.6f}'.format(conf_T['miou']), fontsize=
20, color='white')
ax4.imshow(colorize_mask(torch.argmax(sample_output_T, dim=1).cpu().n
umpy()[0]))
```

Out[32]: <matplotlib.image.AxesImage at 0x7f17886f0c50>



In [33]:
```
sample_img, sample_target = JointNormalize(*norm)(*JointToTensor()(*s
ample2))
if USE_GPU:
    sample_img = sample_img.cuda()
sample_output_O = overfit_net.forward(sample_img[None])
sample_output_T = trained_net.forward(sample_img[None])

# computing mIOU (quantitative measure of accuracy for network predic
tions)
pred_O = torch.argmax(sample_output_O, dim=1).cpu().numpy()[0]
pred_T = torch.argmax(sample_output_T, dim=1).cpu().numpy()[0]
gts = torch.from_numpy(np.array(sample2[1].convert('P'), dtype=np.int
32)).long().numpy()
gts[gts == 255] = -1
conf_O = eval_semantic_segmentation(pred_O[None], gts[None])
conf_T = eval_semantic_segmentation(pred_T[None], gts[None])


fig = plt.figure(figsize=(14,10))
ax1 = fig.add_subplot(2,2,1)
plt.title('image sample')
ax1.imshow(sample2[0])
ax2 = fig.add_subplot(2,2,2)
plt.title('ground truth (target)')
ax2.imshow(sample2[1])
ax3 = fig.add_subplot(2,2,3)
plt.title('OVERFIT_NET prediction (for image it has not seen)')
ax3.text(10, 25, 'mIoU = {:_>8.6f}'.format(conf_O['miou']), fontsize=
20, color='white')
ax3.imshow(colorize_mask(torch.argmax(sample_output_O, dim=1).cpu().n
umpy()[0]))
ax4 = fig.add_subplot(2,2,4)
plt.title('TRAINED_NET prediction (for image it has not seen)')
ax4.text(10, 25, 'mIoU = {:_>8.6f}'.format(conf_T['miou']), fontsize=
20, color='white')
ax4.imshow(colorize_mask(torch.argmax(sample_output_T, dim=1).cpu().n
umpy()[0]))
```

Out[33]: <matplotlib.image.AxesImage at 0x7f178858bd50>



(h) Evaluate TRAINED_NET on validation dataset.

Run the validation loop for TRAINED_NET against the validation dataset (it has not seen):

In [34]:
```
%%time
# This will be slow on CPU (around 1 hour). On GPU it should take onl
y a few minutes (depending on your GPU).
print("mIoU for TRAINED_NET over the validation dataset:{}".format(va
lidate(val_loader, trained_net)[1]))
```

```
mIoU for TRAINED_NET over the validation dataset:0.45497497420924926
CPU times: user 4min 53s, sys: 16.2 s, total: 5min 9s
Wall time: 1min 17s
```

Problem 6

For the network that you implemented, write a paragraph or two about limitations / bottlenecks about the work. What could be improved? What seems to be some obvious issues with the existing works?

The biggest limitation of this is the hardware available. The network used is resnet 18, but given better hardware, we could use resnet 34, 50, 101 and 152, which are deeper networks which could give better results. For the current network, after 5-10 epochs, the loss and mIoU start to oscillate. Which means for us to improve the network, we need to decrease the learning rate, but that would be extremly time consuming and require better hardware. Another way to improve would be to use newer models, or detect boundaries of object while training. We could detect boundaries or refine the boundary information to get better results