CS 484/684 Computational Vision

credits: many thanks for the design of this assigenemnt go to Towaki Takikawa

Homework Assignment #5 - 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/

In [1]:
```
%matplotlib inline

# It is best to start with USE_GPU = False (implying CPU). Switch USE_GPU to Tru
# we strongly recommend to wait until you are absolutely sure your CPU-based cod
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 for HW4.
# Detailed installation instructions are here: https://pytorch.org/get-started/l
# That web site should help you to select the right 'conda install' command to b
# In particular, select the right version of CUDA. Note that prior to installing
# install the latest driver for your GPU and CUDA (9.2 or 10.1), assuming your G
# 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 (containing popu
# 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, dinin
#       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 Prefe
# Chainer, the base library, inspired and led to the creation of PyTorch!
# Although Chainer and PyTorch are different, there are some nice functionalitie
# 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" insi
# For more detailed installation instructions, see https://chainercv.readthedocs
# For other information about ChainerCV library, refer to https://chainercv.read
from chainercv.evaluations import eval_semantic_segmentation
from chainercv.datasets import VOCSemanticSegmentationDataset
```

In [4]:

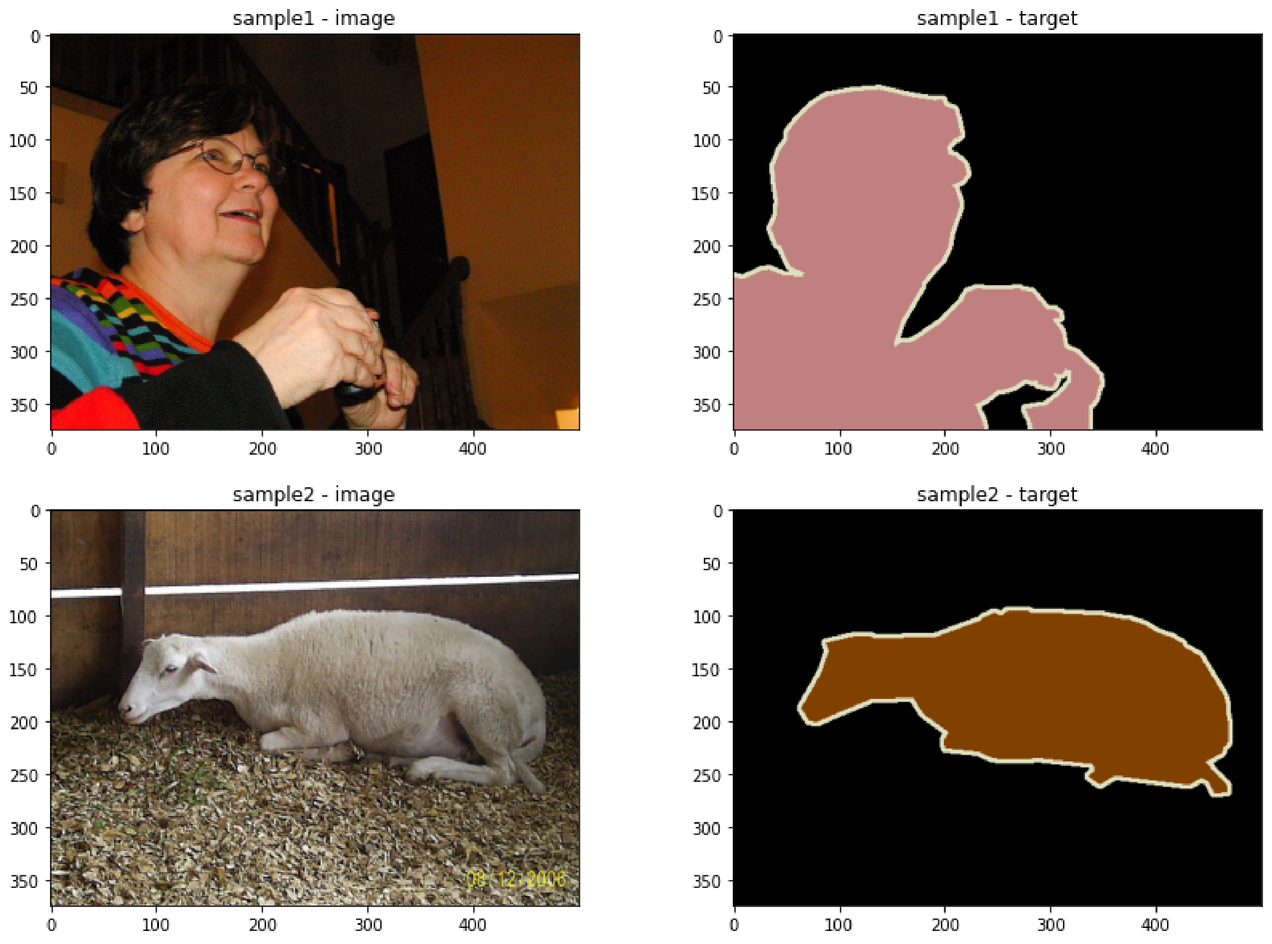
```
# This colorize_mask class takes in a numpy segmentation mask,
#  and then converts it to a PIL Image for visualization.
#  Since by default the numpy matrix contains integers from
#  0,1,...,num_classes, we need to apply some color to this
#  so we can visualize easier! Refer to:
#  https://pillow.readthedocs.io/en/4.1.x/reference/Image.html#PIL.Image.Image.p
palette = [0, 0, 0, 128, 0, 0, 0, 128, 0, 128, 128, 0, 0, 0, 128, 128, 0, 128, 0
           128, 128, 128, 64, 0, 0, 192, 0, 0, 64, 128, 0, 192, 128, 0, 64, 0, 1
           64, 128, 128, 192, 128, 128, 0, 64, 0, 128, 64, 0, 0, 192, 0, 128, 19

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 dataset to test
# Running this block will automatically download the PASCAL VOC Dataset (3.7GB)
# The code below creates subdirectory "datasets" in the same location as the not
# you can modify DATASET_PATH to download the dataset to any custom directory. D
# On subsequent runs you may save time by setting "download = False" (the defaul

DATASET_PATH = 'datasets'

# Here, we obtain and visualize one sample (img, target) pair from VOC training
# Note that operator [...] extracts the sample corresponding to the specified in
# Also, note the parameter download = True. Set this to False after you download
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 insi
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 0x7f969948be90>
```



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/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.convert('P'),
```

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]:

```python
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 0x7f96986c6b90>
```

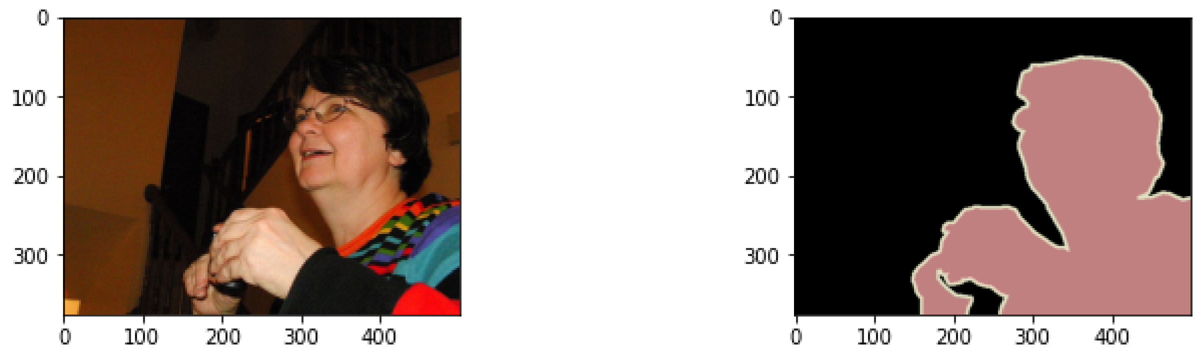
(a) Implement RandomFlip

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):
        if random.random() > 0.5:
            img = tF.hflip(img)
            target = tF.hflip(target)
        return img, 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 0x7f969865e950>
```



(b) Implement RandomResizeCrop

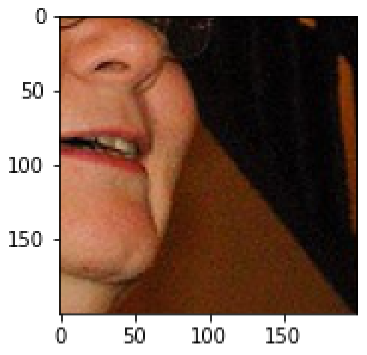
This class should take a img, target pair and then resize the images by a random scale between $[\text{minimum\_scale}, \text{maximum\_scale}]$, crop a random location of the image by $\min(\text{size}, \text{image\_height}, \text{image\_width})$ (where the size is passed in as an integer in the constructor),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

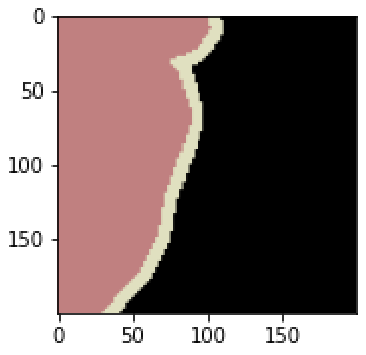
Solution:

In [10]:
```
class JointRandomResizeCrop(object):
    def __init__(self, scale_range, size):
        self.lowScale, self.highScale = scale_range[0], scale_range[1]
        self.size = size

    def __call__(self, img, target):
        # Resize
        rScale = np.random.uniform(self.lowScale, self.highScale)
        img = tF.affine(img, 0, [0, 0], rScale, [0, 0])
        target = tF.affine(target, 0, [0, 0], rScale, [0, 0])
        # Crop and then resize
        img_width, img_height = np.array(img.size)[-2:]
        min_size = min(self.size, img_height, img_width)
        rTop, rLeft = np.random.randint(
            0, img_height - min_size + 1), np.random.randint(0, img_width - min_
        img = tF.resized_crop(img, rTop, rLeft, min_size, min_size, self.size)
        target = tF.resized_crop(
            target, rTop, rLeft, min_size, min_size, self.size)

        return img, target


img, target = JointRandomResizeCrop((1, 3), 200)(*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 0x7f969857b5d0>
```





(c) Implement Normalize

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

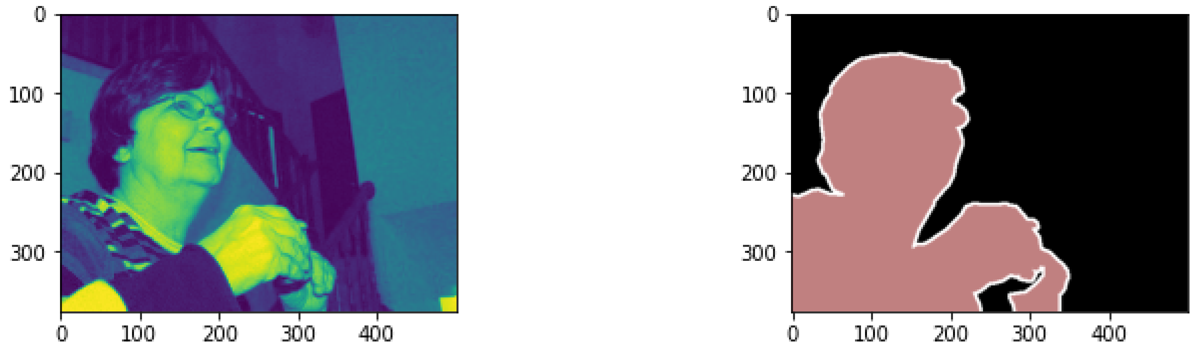
Solution:

In [11]:

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

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


    def __call__(self, img, target):
        img = tF.normalize(img, self.mean, self.variance)

        return img, target

img, target = JointNormalize(*norm)(*JointToTensor()(*sample1))
fig = plt.figure(figsize=(12,6))
ax1 = fig.add_subplot(2,2,1)
ax1.imshow(img[0])
```

```
ax2 = fig.add_subplot(2,2,2)
ax2.imshow(colorize_mask(target.numpy()))
```

Out[11]:
```
<matplotlib.image.AxesImage at 0x7f9698414e90>
```



(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.

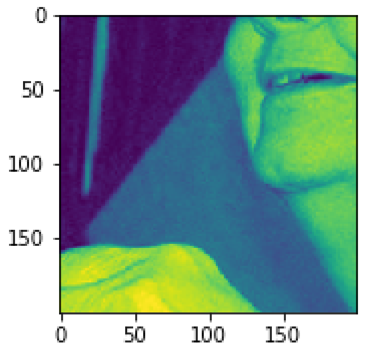
In [12]:
```
# This class composes transofrmations from a given list of image transforms (exp
# will be applied to the dataset during training. This cell is fully implemented

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
```

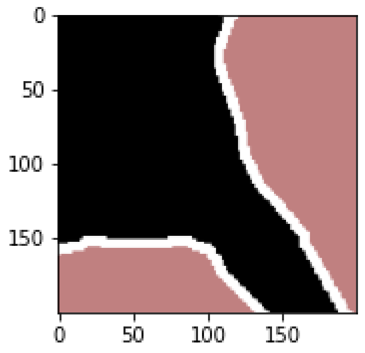
In [13]:
```
# Student Answer:

img, target = JointCompose([
    JointRandomFlip(),
    JointRandomResizeCrop((1, 3), 200),
    JointToTensor(),
    JointNormalize(*norm),
])(*sample1)
fig = plt.figure(figsize=(12,6))
ax1 = fig.add_subplot(2,2,1)
ax1.imshow(img[0])
```

```
ax2 = fig.add_subplot(2,2,2)
ax2.imshow(colorize_mask(target.numpy()))
```

Out[13]:

```
<matplotlib.image.AxesImage at 0x7f96993958d0>
```





(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]:

```
# Student Answer:

sanity_joint_transform = JointCompose([
    JointToTensor(),
    JointNormalize(*norm),
])

train_joint_transform = JointCompose([
    JointRandomFlip(),
    JointRandomResizeCrop((1, 3), 200),
    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 datasets and the
# This cell is fully implemented.

# This single image data(sub)set can help to better understand and to debug the
# Optional integer parameter 'sanity_check' specifies the index of the image-tar
# 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 segmentation ne
train_data = VOCSegmentation(
    DATASET_PATH,
    image_set = 'train',
    transforms = train_joint_transform
)

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

# Increase TRAIN_BATCH_SIZE if you are using GPU to speed up training.
# When batch size changes, the learning rate may also need to be adjusted.
# Note that batch size maybe limited by your GPU memory, so adjust if you get "r
TRAIN_BATCH_SIZE = 4

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

sanity_loader = DataLoader(sanity_data, batch_size=1, num_workers=NUM_WORKERS, s
train_loader = DataLoader(train_data, batch_size=TRAIN_BATCH_SIZE, num_workers=N
val_loader = DataLoader(val_data, batch_size=1, num_workers=NUM_WORKERS, shuffle
```

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 (residual blocks) 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. 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.

Solution:

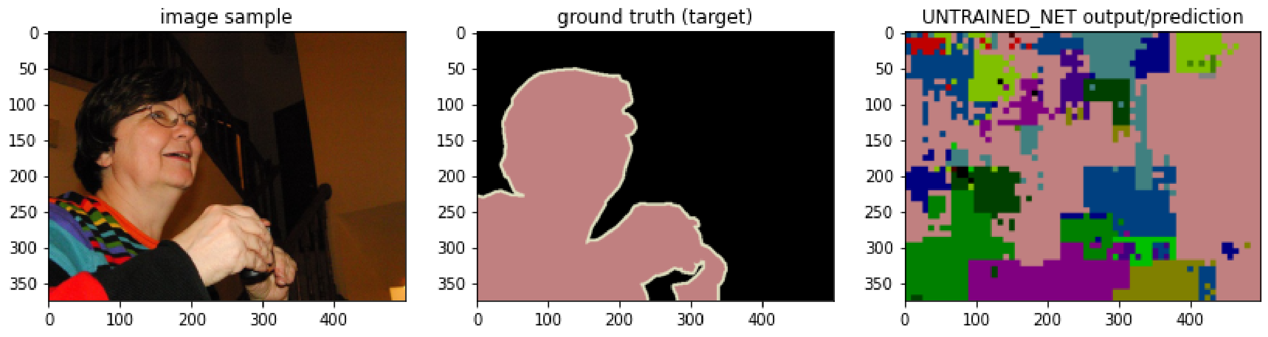
In [16]:
```
import torchvision.models as models

class MyNet(nn.Module):
    def __init__(self, num_classes, criterion=None):
        super(MyNet, self).__init__()
        modules = list(models.resnet18(pretrained=True).children())
        self.encoder = nn.ModuleList(modules[0:8])
        self.decoder = nn.ModuleList([
            nn.Conv2d(576, 21, (3,3), stride=2),
            nn.BatchNorm2d(576, affine=True),
        ])

    def forward(self, inp, gts=None):

        # Encoder

        # Conv Layer 1
        out = self.encoder[0](inp)
        out = self.encoder[1](out)
        out = self.encoder[2](out)
        # Conv Layer 2
        out = self.encoder[3](out)
        skip_out = self.encoder[4](out)
        # Conv Layer 3
        out = self.encoder[5](skip_out)
        # Conv Layer 4
        out = self.encoder[6](out)
        # Conv Layer 5
        out = self.encoder[7](out)

        # Decoder

        # Upsample 1
        out = nn.Upsample(size=skip_out.size()[-2:])(out)
        # Skip Connection
        out = torch.cat([skip_out, out], 1)
        # Convs/ReLUs/BNs
        out = self.decoder[1](out)
        out = self.decoder[0](out)
        out = nn.ReLU()(out)
        # Upsample 2
        lfinal = nn.functional.interpolate(out, inp.size()[-2:])

        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()(*sample1))
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 0x7f9697340810>
```



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, actual, expected):
        m = nn.LogSoftmax(dim=1)
        loss = nn.NLLLoss(ignore_index=self.ignore_index)
        out = loss(m(actual), expected)
        return out
```

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

In [19]:
```
criterion = nn.CrossEntropyLoss(ignore_index=255)
print(untrained_output.size())
print(sample_target[None].size())

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

my_criterion = MyCrossEntropyLoss(ignore_index=255)
```

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

```
torch.Size([1, 21, 375, 500])
torch.Size([1, 375, 500])
tensor(3.0273, grad_fn=<NllLoss2DBackward>)
tensor(3.0273, 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 function 'eval

pred = torch.argmax(untrained_output, dim=1).numpy()
gts = np.array([sample_target.numpy()])

conf = eval_semantic_segmentation(pred, gts)

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

```
mIoU for the sample image / ground truth pair: 0.008983535274574947
/usr/local/lib/python3.7/dist-packages/chainercv/evaluations/eval_semantic_segme
ntation.py:91: RuntimeWarning: invalid value encountered in true_divide
  iou = np.diag(confusion) / iou_denominator
/usr/local/lib/python3.7/dist-packages/chainercv/evaluations/eval_semantic_segme
ntation.py:168: RuntimeWarning: invalid value encountered in true_divide
  class_accuracy = np.diag(confusion) / np.sum(confusion, axis=1)
```

(b) Write the validation loop.

In [21]:

```
def validate(val_loader, net):

    iou_arr = []
    val_loss = 0

    if USE_GPU:
        net.cuda()

    with torch.no_grad():
        for i, data in enumerate(val_loader):

            inputs, masks = data

            if USE_GPU:
                inputs, masks = inputs.cuda(), masks.cuda() # add this line

            my_net_outputs = net.forward(inputs)

            # Loss
            my_criterion = MyCrossEntropyLoss(ignore_index=255)
            batch_loss = my_criterion(my_net_outputs, masks)
            val_loss += batch_loss.item()

            if USE_GPU:
                preds = torch.argmax(my_net_outputs, dim=1).cpu().numpy()
                gts = masks.cpu().numpy()
            else:
                preds = torch.argmax(my_net_outputs, dim=1).numpy()
                gts = masks.numpy()

            # Hint: make sure the range of values of the ground truth is what yo

            gts[gts == 255] = preds[gts == 255]
            # computing mIOU (quantitative measure of accuracy for network predi
            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
```

```
mIoU over the sanity dataset:0.03331327293420222
CPU times: user 2.17 s, sys: 1.16 s, total: 3.33 s
Wall time: 3.33 s
/usr/local/lib/python3.7/dist-packages/chainercv/evaluations/eval_semantic_segme
ntation.py:91: RuntimeWarning: invalid value encountered in true_divide
  iou = np.diag(confusion) / iou_denominator
/usr/local/lib/python3.7/dist-packages/chainercv/evaluations/eval_semantic_segme
ntation.py:168: RuntimeWarning: invalid value encountered in true_divide
  class_accuracy = np.diag(confusion) / np.sum(confusion, axis=1)
```

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.

In [23]:
```
def get_optimizer(net):
    optimizer = torch.optim.Adam(net.parameters())
    return optimizer
```

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

In [24]:
```
def train(train_loader, net, optimizer, loss_graph):

    main_loss = 0

    if USE_GPU:
        net.cuda()

    for i, data in enumerate(train_loader):
        inputs, masks = data

        if USE_GPU:
            inputs, masks = inputs.cuda(), masks.cuda()

        # Loss
        optimizer.zero_grad()
        batch_loss = net.forward(inputs, masks)
        batch_loss.backward()
        optimizer.step()

        main_loss += batch_loss.item()

        loss_graph.append(batch_loss.item()) # Populate this list to graph the l

    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 only a minute
# Below we create a (deep) copy of untrained_net and train it on a single traini
# Later, we will create a separate (deep) copy of untrained_net to be trained on
# NOTE: Normally, one can create a new net via declaration new_net = MyNet(21).
# are declared that way creates *different* untrained nets. This notebook compar
# For this comparison to be direct and fair, it is better to train (deep) copies
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 = 200

# 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()
    print("Epoch: {} Loss: {}".format(e, loss))


%matplotlib inline
```

```
Starting Training...


Epoch: 0 Loss: 3.1816470623016357
Epoch: 1 Loss: 2.0592970848083496
Epoch: 2 Loss: 0.9451185464859009
Epoch: 3 Loss: 0.6013822555541992
Epoch: 4 Loss: 0.3977959454059601
Epoch: 5 Loss: 0.2004314661026001
Epoch: 6 Loss: 0.14893510937690735
Epoch: 7 Loss: 0.12099331617355347
Epoch: 8 Loss: 0.09427690505981445
Epoch: 9 Loss: 0.0766497403383255
Epoch: 10 Loss: 0.06621570140123367
Epoch: 11 Loss: 0.06051842123270035
Epoch: 12 Loss: 0.053481604903936386
Epoch: 13 Loss: 0.047669667750597
Epoch: 14 Loss: 0.0394187867641449
Epoch: 15 Loss: 0.03885165974497795
Epoch: 16 Loss: 0.035011131316423416
Epoch: 17 Loss: 0.031055212020874023
Epoch: 18 Loss: 0.02929956652224064
Epoch: 19 Loss: 0.027198435738682747
Epoch: 20 Loss: 0.025892790406942368
Epoch: 21 Loss: 0.023565785959362984
Epoch: 22 Loss: 0.023122552782297134
Epoch: 23 Loss: 0.022164613008499146
Epoch: 24 Loss: 0.02104584500193596
Epoch: 25 Loss: 0.020383376628160477
Epoch: 26 Loss: 0.01954648084938526
Epoch: 27 Loss: 0.01870143599808216
Epoch: 28 Loss: 0.017857814207673073
Epoch: 29 Loss: 0.017335759475827217
Epoch: 30 Loss: 0.016720512881875038
Epoch: 31 Loss: 0.016047729179263115
Epoch: 32 Loss: 0.015547150745987892
Epoch: 33 Loss: 0.014971883036196232
Epoch: 34 Loss: 0.014546239748597145
Epoch: 35 Loss: 0.014137049205601215
Epoch: 36 Loss: 0.013649926520884037
Epoch: 37 Loss: 0.013333899900317192
Epoch: 38 Loss: 0.013057093136012554
Epoch: 39 Loss: 0.012738065794110298
Epoch: 40 Loss: 0.012453677132725716
Epoch: 41 Loss: 0.012163698673248291
Epoch: 42 Loss: 0.011883311904966831
Epoch: 43 Loss: 0.011640924029052258
Epoch: 44 Loss: 0.011379316449165344
Epoch: 45 Loss: 0.011130832135677338
Epoch: 46 Loss: 0.01093739178031683
Epoch: 47 Loss: 0.010735765099525452
Epoch: 48 Loss: 0.01052180863916874
Epoch: 49 Loss: 0.010334216989576817
Epoch: 50 Loss: 0.010155456140637398
```

```
Epoch: 51 Loss: 0.009976810775697231
Epoch: 52 Loss: 0.009807716123759747
Epoch: 53 Loss: 0.00963891763240099
Epoch: 54 Loss: 0.009481657296419144
Epoch: 55 Loss: 0.00934867188334465
Epoch: 56 Loss: 0.009219040162861347
Epoch: 57 Loss: 0.009087053127586842
Epoch: 58 Loss: 0.00896763987839222
Epoch: 59 Loss: 0.008855302818119526
Epoch: 60 Loss: 0.008744047954678535
Epoch: 61 Loss: 0.008638049475848675
Epoch: 62 Loss: 0.008534963242709637
Epoch: 63 Loss: 0.008435928262770176
Epoch: 64 Loss: 0.008346385322511196
Epoch: 65 Loss: 0.008261865004897118
Epoch: 66 Loss: 0.008179903030395508
Epoch: 67 Loss: 0.00810568779706955
Epoch: 68 Loss: 0.008036956191062927
Epoch: 69 Loss: 0.007968379184603691
Epoch: 70 Loss: 0.007902087643742561
Epoch: 71 Loss: 0.007839294150471687
Epoch: 72 Loss: 0.0077789598144590855
Epoch: 73 Loss: 0.007721965201199055
Epoch: 74 Loss: 0.00766785629093647
Epoch: 75 Loss: 0.007615430746227503
Epoch: 76 Loss: 0.007566079031676054
Epoch: 77 Loss: 0.007520338054746389
Epoch: 78 Loss: 0.007475941441953182
Epoch: 79 Loss: 0.0074326759204268456
Epoch: 80 Loss: 0.007391949184238911
Epoch: 81 Loss: 0.007353402674198151
Epoch: 82 Loss: 0.007316175848245621
Epoch: 83 Loss: 0.007280903868377209
Epoch: 84 Loss: 0.007247561123222113
Epoch: 85 Loss: 0.007215659599751234
Epoch: 86 Loss: 0.007185091730207205
Epoch: 87 Loss: 0.007155704777687788
Epoch: 88 Loss: 0.00712745264172554
Epoch: 89 Loss: 0.007100466173142195
Epoch: 90 Loss: 0.007074514403939247
Epoch: 91 Loss: 0.007049298379570246
Epoch: 92 Loss: 0.00702529726549983
Epoch: 93 Loss: 0.007002586033195257
Epoch: 94 Loss: 0.006980682257562876
Epoch: 95 Loss: 0.0069595626555383205
Epoch: 96 Loss: 0.006939450744539499
Epoch: 97 Loss: 0.0069201611913740635
Epoch: 98 Loss: 0.006901525892317295
Epoch: 99 Loss: 0.006883635185658932
Epoch: 100 Loss: 0.006866436451673508
Epoch: 101 Loss: 0.006849874276667833
Epoch: 102 Loss: 0.006834014318883419
Epoch: 103 Loss: 0.006818739231675863
Epoch: 104 Loss: 0.006804042495787144
Epoch: 105 Loss: 0.0067899334244430065
Epoch: 106 Loss: 0.006776300258934498
Epoch: 107 Loss: 0.006763160694390535
Epoch: 108 Loss: 0.0067506288178265095
Epoch: 109 Loss: 0.0067385523580014706
Epoch: 110 Loss: 0.0067269466817379
Epoch: 111 Loss: 0.006715846247971058
Epoch: 112 Loss: 0.006705169565975666
Epoch: 113 Loss: 0.006694848649203777
Epoch: 114 Loss: 0.006684980355203152
Epoch: 115 Loss: 0.006675477605313063
```

```
Epoch: 116 Loss: 0.006666318513453007
Epoch: 117 Loss: 0.006657497491687536
Epoch: 118 Loss: 0.006649040151387453
Epoch: 119 Loss: 0.00664089759811759
Epoch: 120 Loss: 0.006633047480136156
Epoch: 121 Loss: 0.006625478621572256
Epoch: 122 Loss: 0.006618176121264696
Epoch: 123 Loss: 0.006611129734665155
Epoch: 124 Loss: 0.006604306399822235
Epoch: 125 Loss: 0.006597741972655058
Epoch: 126 Loss: 0.006591367069631815
Epoch: 127 Loss: 0.0065852077677845955
Epoch: 128 Loss: 0.00657926220446825
Epoch: 129 Loss: 0.006573498249053955
Epoch: 130 Loss: 0.006567907053977251
Epoch: 131 Loss: 0.006562478840351105
Epoch: 132 Loss: 0.006557217799127102
Epoch: 133 Loss: 0.006552102509886026
Epoch: 134 Loss: 0.006547128781676292
Epoch: 135 Loss: 0.006542298477143049
Epoch: 136 Loss: 0.0065376041457057
Epoch: 137 Loss: 0.006533037405461073
Epoch: 138 Loss: 0.006528585683554411
Epoch: 139 Loss: 0.006524255499243736
Epoch: 140 Loss: 0.006520033814013004
Epoch: 141 Loss: 0.006515922490507364
Epoch: 142 Loss: 0.006511929910629988
Epoch: 143 Loss: 0.006508036516606808
Epoch: 144 Loss: 0.006504239980131388
Epoch: 145 Loss: 0.00650053983554244
Epoch: 146 Loss: 0.0064969114027917385
Epoch: 147 Loss: 0.006493380758911371
Epoch: 148 Loss: 0.006489930208772421
Epoch: 149 Loss: 0.00648654717952013
Epoch: 150 Loss: 0.006483246106654406
Epoch: 151 Loss: 0.006480006035417318
Epoch: 152 Loss: 0.0064768376760184765
Epoch: 153 Loss: 0.0064737615175545216
Epoch: 154 Loss: 0.006470722146332264
Epoch: 155 Loss: 0.0064677586778998375
Epoch: 156 Loss: 0.006464858073741198
Epoch: 157 Loss: 0.006462003570050001
Epoch: 158 Loss: 0.00645921565592289
Epoch: 159 Loss: 0.006456480827182531
Epoch: 160 Loss: 0.006453781854361296
Epoch: 161 Loss: 0.006451151333749294
Epoch: 162 Loss: 0.006448561325669289
Epoch: 163 Loss: 0.006446029059588909
Epoch: 164 Loss: 0.006443548947572708
Epoch: 165 Loss: 0.006441108882427216
Epoch: 166 Loss: 0.006438699085265398
Epoch: 167 Loss: 0.006436346098780632
Epoch: 168 Loss: 0.006434041541069746
Epoch: 169 Loss: 0.006431773770600557
Epoch: 170 Loss: 0.006429535336792469
Epoch: 171 Loss: 0.006427350454032421
Epoch: 172 Loss: 0.006425204686820507
Epoch: 173 Loss: 0.006423087790608406
Epoch: 174 Loss: 0.006421012803912163
Epoch: 175 Loss: 0.006418985314667225
Epoch: 176 Loss: 0.006416989490389824
Epoch: 177 Loss: 0.0064150067046284676
Epoch: 178 Loss: 0.006413073744624853
Epoch: 179 Loss: 0.006411178968846798
Epoch: 180 Loss: 0.006409309338778257
```

```
Epoch: 181 Loss: 0.006407468114048243
Epoch: 182 Loss: 0.006405658088624477
Epoch: 183 Loss: 0.006403882056474686
Epoch: 184 Loss: 0.00640213955193758
Epoch: 185 Loss: 0.006400426384061575
Epoch: 186 Loss: 0.0063987369649112225
Epoch: 187 Loss: 0.006397073622792959
Epoch: 188 Loss: 0.006395437754690647
Epoch: 189 Loss: 0.006393840070813894
Epoch: 190 Loss: 0.006392255425453186
Epoch: 191 Loss: 0.0063906945288181305
Epoch: 192 Loss: 0.006389163434505463
Epoch: 193 Loss: 0.006387661211192608
Epoch: 194 Loss: 0.006386184133589268
Epoch: 195 Loss: 0.006384725216776133
Epoch: 196 Loss: 0.006383295636624098
Epoch: 197 Loss: 0.006381879560649395
Epoch: 198 Loss: 0.006380489561706781
Epoch: 199 Loss: 0.006379127502441406
CPU times: user 22.4 s, sys: 3.05 s, total: 25.5 s
Wall time: 25.7 s
```

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

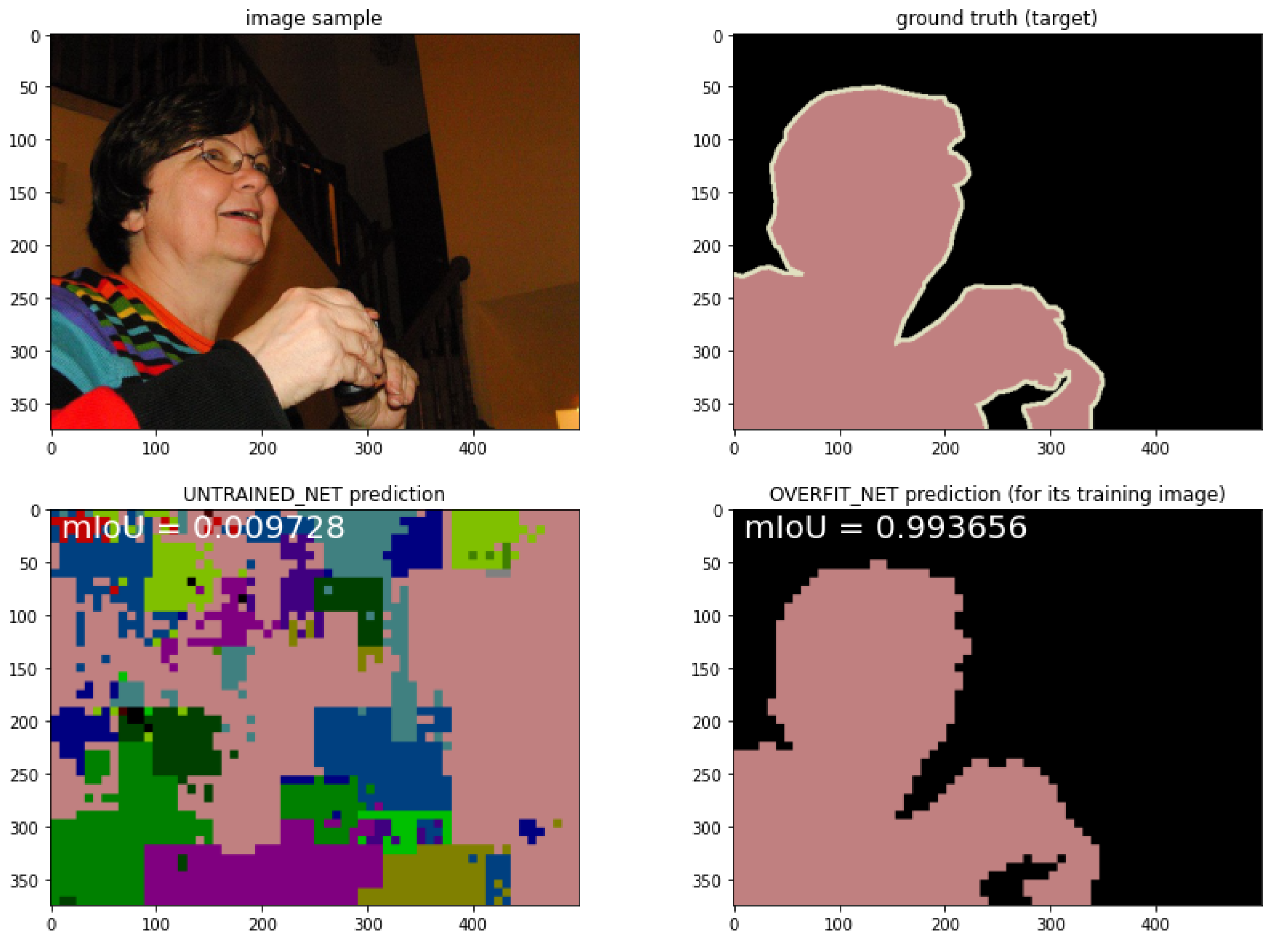
In [26]:

```
# switch back to evaluation mode
overfit_net.eval()

sample_img, sample_target = JointNormalize(*norm)(*JointToTensor()(*sample1))
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 predictions)
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.int32)).long()
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='
ax3.imshow(colorize_mask(torch.argmax(sample_output_U, dim=1).cpu().numpy()[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='
ax4.imshow(colorize_mask(torch.argmax(sample_output_O, dim=1).cpu().numpy()[0]))
```

```
/usr/local/lib/python3.7/dist-packages/chainercv/evaluations/eval_semantic_segme
ntation.py:91: RuntimeWarning: invalid value encountered in true_divide
  iou = np.diag(confusion) / iou_denominator
/usr/local/lib/python3.7/dist-packages/chainercv/evaluations/eval_semantic_segme
ntation.py:168: RuntimeWarning: invalid value encountered in true_divide
  class_accuracy = np.diag(confusion) / np.sum(confusion, axis=1)
```

Out[26]:
```
<matplotlib.image.AxesImage at 0x7f9631510190>
```

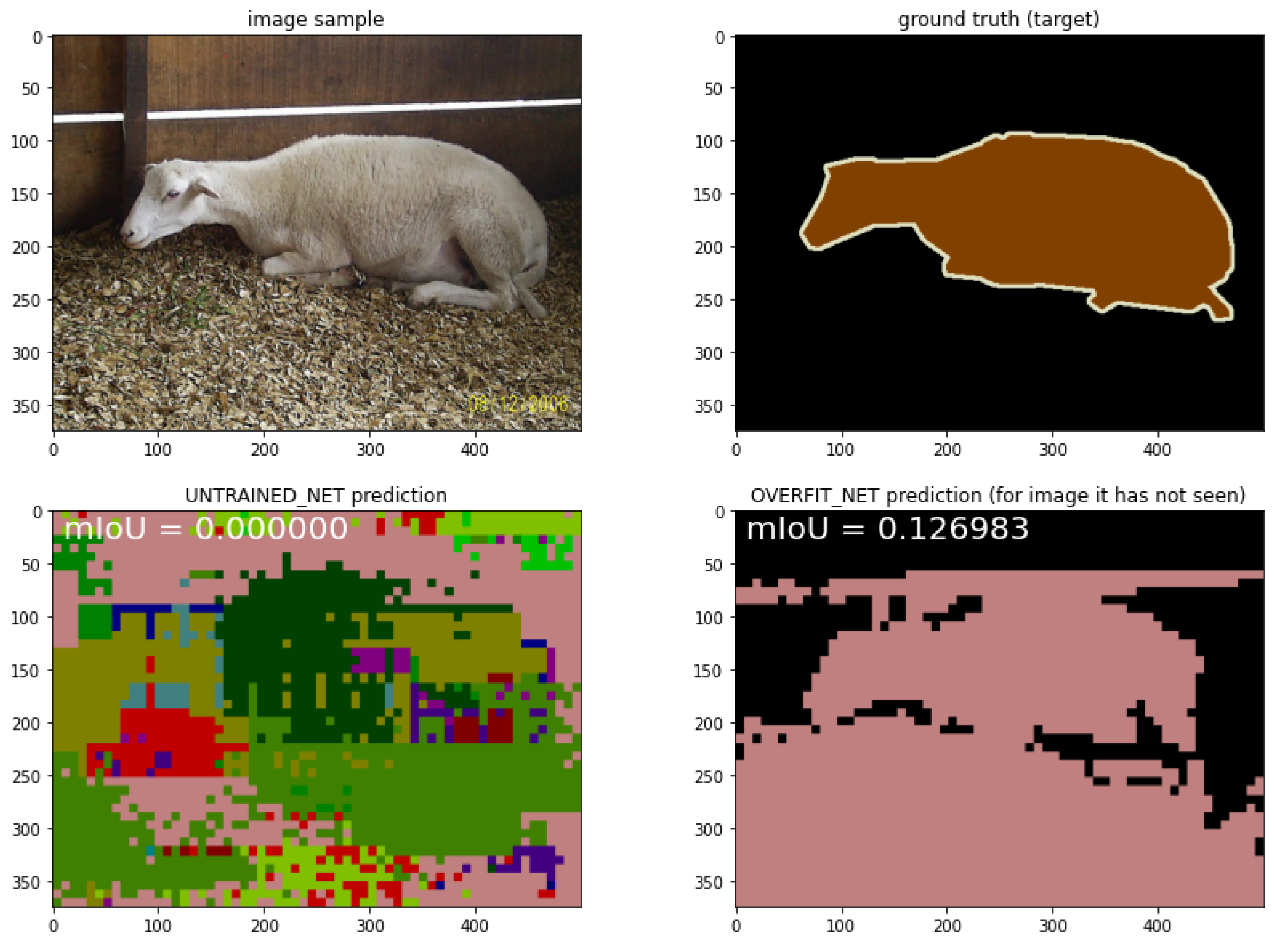


In [27]:
```
sample_img, sample_target = JointNormalize(*norm)(*JointToTensor()(*sample2))
if USE_GPU:
    sample_img = sample_img.cuda()
sample_output_O = overfit_net.forward(sample_img[None])
sample_output_U = untrained_net.forward(sample_img[None])

# computing mIOU (quantitative measure of accuracy for network predictions)
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.int32)).long()
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='
ax3.imshow(colorize_mask(torch.argmax(sample_output_U, dim=1).cpu().numpy()[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='
ax4.imshow(colorize_mask(torch.argmax(sample_output_O, dim=1).cpu().numpy()[0]))
```

```
/usr/local/lib/python3.7/dist-packages/chainercv/evaluations/eval_semantic_segme
ntation.py:91: RuntimeWarning: invalid value encountered in true_divide
  iou = np.diag(confusion) / iou_denominator
/usr/local/lib/python3.7/dist-packages/chainercv/evaluations/eval_semantic_segme
ntation.py:168: RuntimeWarning: invalid value encountered in true_divide
  class_accuracy = np.diag(confusion) / np.sum(confusion, axis=1)
```

Out[27]:
```
<matplotlib.image.AxesImage at 0x7f9631434790>
```



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(validate(sanity_l
mIoU for OVERFIT_NET over its training image:0.9938577936693154
CPU times: user 31.6 ms, sys: 7.8 ms, total: 39.4 ms
Wall time: 40.5 ms
```

```
/usr/local/lib/python3.7/dist-packages/chainercv/evaluations/eval_semantic_segme
ntation.py:91: RuntimeWarning: invalid value encountered in true_divide
  iou = np.diag(confusion) / iou_denominator
/usr/local/lib/python3.7/dist-packages/chainercv/evaluations/eval_semantic_segme
ntation.py:168: RuntimeWarning: invalid value encountered in true_divide
  class_accuracy = np.diag(confusion) / np.sum(confusion, axis=1)
```

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
print("mIoU for UNTRAINED_NET over the entire dataset:{}".format(validate(val_lo
```

```
/usr/local/lib/python3.7/dist-packages/chainercv/evaluations/eval_semantic_segme
ntation.py:91: RuntimeWarning: invalid value encountered in true_divide
  iou = np.diag(confusion) / iou_denominator
/usr/local/lib/python3.7/dist-packages/chainercv/evaluations/eval_semantic_segme
ntation.py:168: RuntimeWarning: invalid value encountered in true_divide
  class_accuracy = np.diag(confusion) / np.sum(confusion, axis=1)
mIoU for UNTRAINED_NET over the entire dataset:0.058816702677216184
CPU times: user 27.6 s, sys: 4.16 s, total: 31.7 s
Wall time: 31.6 s
```

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
print("mIoU for OVERFIT_NET over the validation dataset:{}".format(validate(val_
```

```
/usr/local/lib/python3.7/dist-packages/chainercv/evaluations/eval_semantic_segme
ntation.py:91: RuntimeWarning: invalid value encountered in true_divide
  iou = np.diag(confusion) / iou_denominator
/usr/local/lib/python3.7/dist-packages/chainercv/evaluations/eval_semantic_segme
```

```
ntation.py:168: RuntimeWarning: invalid value encountered in true_divide
  class_accuracy = np.diag(confusion) / np.sum(confusion, axis=1)
mIoU for OVERFIT_NET over the validation dataset:0.15835909344506208
CPU times: user 27.6 s, sys: 4.29 s, total: 31.9 s
Wall time: 31.8 s
```

(e) Explain in a few sentences the quantitative results observed in (c) and (d):

Student answer:

First we discuss the results observed in Part c):

The first thing we did in Part c) was to train `overfit_net` over 200 epochs on the santiy dataset. That is, we trained it to be really good at identifying that one image. You can see on the generated loss graph that this was successfully done because the loss had an overall **downward trend** and very desirable final loss of ~0.

It was **important to normalize** the inputs for the sanity data set.

Next, we evaluate the segmentation mask with this newly trained `overfit_net` . You can see on the plot which compares its results to `untrained_net` that the segmentation mask is much, much better. It's mIoU (mean intersection over union) of ~0.99 is much better than the `untrained_net` with ~0.009 and just random incoherent blobs. You can see on the mask how it correctly picked the "person" and "background" color mask for pretty much ever pixel.

On the other hand, the overfitting of `overfit_net` is made obvious when compared against the new image with an animal. `overfit_net` was still better than `untrained_net` for this new image as it identified the "background" on the animal picture. However, `overfit_net` was also trained to expect a "person" so it labels many pictures as such. It has never seen an animal during training so it has no hope of labelling that part of the picture correctly. This is because of its overfitting.

Finally, we see that when `overfit_net` is validated on just the sanity data set, it performs well at 0.99. This makes sense because it has already trained on this santiy data set.

Next, we discuss the results observed in Part d):

First, the `untrained_net` is validated on the validation data set. With no training whatsoever, it performs understandly very poorly with ~0.059 mIoU. Many of its parameters are randomly generated and have no real chance of correctly labelling the images.

Next, the `overfit_net` is validated on the validation data set. Compared to before, this training data set has a much wider range of images than the sanity data set which only had 1 image. However, the `overfit_net` should at least be able to identify pictures with people and background and definitely better than `untrained_net` . This is exactly what we observed. Although its mIoU did go down as compared to when it was only validated against the sanity data set, this was still reasonably good as we suspect it correctly identified "people" or "background" pixels it encountered. It got an mIoU of 0.158 which was much better than `untrained_net` .

The `overfit_net` was better than the `untrained_net` . However, it performed best on images with background and people because that is what it had been trained to identify. Next, we would like to see how the model performs when trained with images that have a much wider range of labels.

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

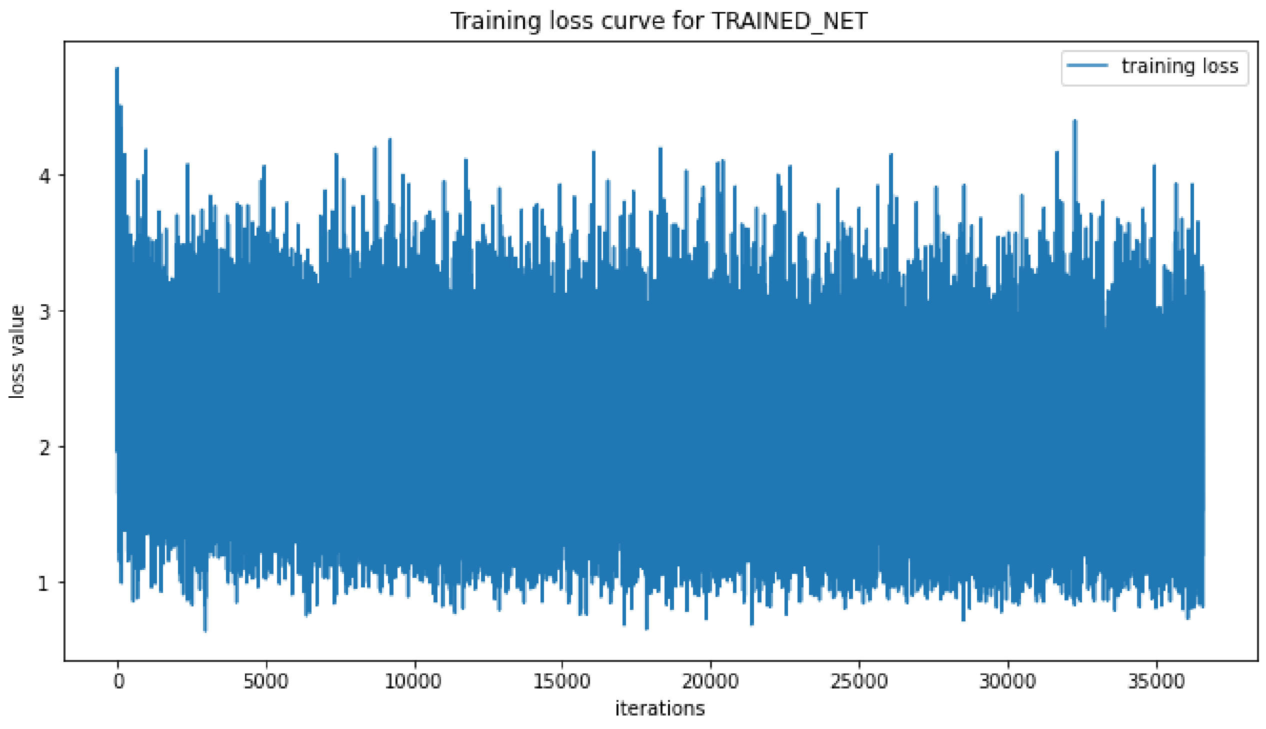
In [31]:

```
%%time


# This training will be very slow on a CPU (>1hour per epoch). Ideally, this sho
# taking only a few minutes per epoch (depending on your GPU and batch size). Th
# it is highly advisable that you first finish debugging your net code. In parti
# reasonably, e.g. its loss monotonically decreases during training and its outp
# Below we create another (deep) copy of untrained_net. Unlike OVERFIT_NET it wi
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 TRAINED_NET it
# the number of required epochs could be smaller compared to OFERFIT_NET where e
EPOCH = 100

# 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))
fig.show()
```

```
Starting Training...
Epoch: 0 Loss: 921.7083113789558
Epoch: 1 Loss: 845.7273463606834
Epoch: 2 Loss: 837.6482894420624
Epoch: 3 Loss: 822.2785019278526
Epoch: 4 Loss: 821.5178084969521
Epoch: 5 Loss: 822.7172150611877
Epoch: 6 Loss: 817.8631080389023
Epoch: 7 Loss: 821.0603308677673
```

```
Epoch: 8 Loss: 812.4178315401077
Epoch: 9 Loss: 806.2117807865143
Epoch: 10 Loss: 802.8479860424995
Epoch: 11 Loss: 802.1771532893181
Epoch: 12 Loss: 797.2324470281601
Epoch: 13 Loss: 794.7631536126137
Epoch: 14 Loss: 786.1638958454132
Epoch: 15 Loss: 791.9917281866074
Epoch: 16 Loss: 776.3601211309433
Epoch: 17 Loss: 768.1971680521965
Epoch: 18 Loss: 773.7831109166145
Epoch: 19 Loss: 769.7629691362381
Epoch: 20 Loss: 781.8250039815903
Epoch: 21 Loss: 759.7802890539169
Epoch: 22 Loss: 761.768049955368
Epoch: 23 Loss: 768.802471101284
Epoch: 24 Loss: 757.0598211884499
Epoch: 25 Loss: 750.4035631418228
Epoch: 26 Loss: 760.0948629379272
Epoch: 27 Loss: 760.5776635408401
Epoch: 28 Loss: 749.3185603022575
Epoch: 29 Loss: 734.6526563763618
Epoch: 30 Loss: 724.968184709549
Epoch: 31 Loss: 729.1631006598473
Epoch: 32 Loss: 744.4641900658607
Epoch: 33 Loss: 741.7134566307068
Epoch: 34 Loss: 757.0920241475105
Epoch: 35 Loss: 750.2396694421768
Epoch: 36 Loss: 745.4131790399551
Epoch: 37 Loss: 743.6015919446945
Epoch: 38 Loss: 743.5758843421936
Epoch: 39 Loss: 747.3681986927986
Epoch: 40 Loss: 754.9987804889679
Epoch: 41 Loss: 746.2074263095856
Epoch: 42 Loss: 744.1741653680801
Epoch: 43 Loss: 744.2537276148796
Epoch: 44 Loss: 736.6323072910309
Epoch: 45 Loss: 743.2900105118752
Epoch: 46 Loss: 748.8498183488846
Epoch: 47 Loss: 735.9859284758568
Epoch: 48 Loss: 725.4942317008972
Epoch: 49 Loss: 728.3880642652512
Epoch: 50 Loss: 736.4983859062195
Epoch: 51 Loss: 734.0532968044281
Epoch: 52 Loss: 738.6587694883347
Epoch: 53 Loss: 726.4878928661346
Epoch: 54 Loss: 718.2827798128128
Epoch: 55 Loss: 751.4036554694176
Epoch: 56 Loss: 735.6339361667633
Epoch: 57 Loss: 748.7647907733917
Epoch: 58 Loss: 736.7624597549438
Epoch: 59 Loss: 733.8445320129395
Epoch: 60 Loss: 734.7093670368195
Epoch: 61 Loss: 723.3184822797775
Epoch: 62 Loss: 726.8128271102905
Epoch: 63 Loss: 723.8241832256317
Epoch: 64 Loss: 725.3903102278709
Epoch: 65 Loss: 709.2494843006134
Epoch: 66 Loss: 701.255826830864
Epoch: 67 Loss: 712.084820330143
Epoch: 68 Loss: 710.7469387054443
Epoch: 69 Loss: 713.315366923809
Epoch: 70 Loss: 731.369620680809
Epoch: 71 Loss: 712.4453256726265
Epoch: 72 Loss: 714.4735864400864
```

```
Epoch: 73 Loss: 714.4730912446976
Epoch: 74 Loss: 714.5194042921066
Epoch: 75 Loss: 712.1150951385498
Epoch: 76 Loss: 702.5537633299828
Epoch: 77 Loss: 707.0071457028389
Epoch: 78 Loss: 709.9685325026512
Epoch: 79 Loss: 708.5556389689445
Epoch: 80 Loss: 719.1239488720894
Epoch: 81 Loss: 714.1933034062386
Epoch: 82 Loss: 702.6800770759583
Epoch: 83 Loss: 704.305300116539
Epoch: 84 Loss: 705.2521514892578
Epoch: 85 Loss: 719.0019479393959
Epoch: 86 Loss: 712.2768449187279
Epoch: 87 Loss: 721.8594279289246
Epoch: 88 Loss: 713.8730288743973
Epoch: 89 Loss: 712.9012981653214
Epoch: 90 Loss: 717.2691805958748
Epoch: 91 Loss: 704.4097082018852
Epoch: 92 Loss: 720.7923815846443
Epoch: 93 Loss: 711.1327609419823
Epoch: 94 Loss: 712.3493319749832
Epoch: 95 Loss: 712.7888796925545
Epoch: 96 Loss: 704.1631794571877
Epoch: 97 Loss: 707.6649053096771
Epoch: 98 Loss: 698.7408691048622
Epoch: 99 Loss: 705.3391838669777
CPU times: user 27min 2s, sys: 3min 19s, total: 30min 21s
Wall time: 30min 20s
```



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

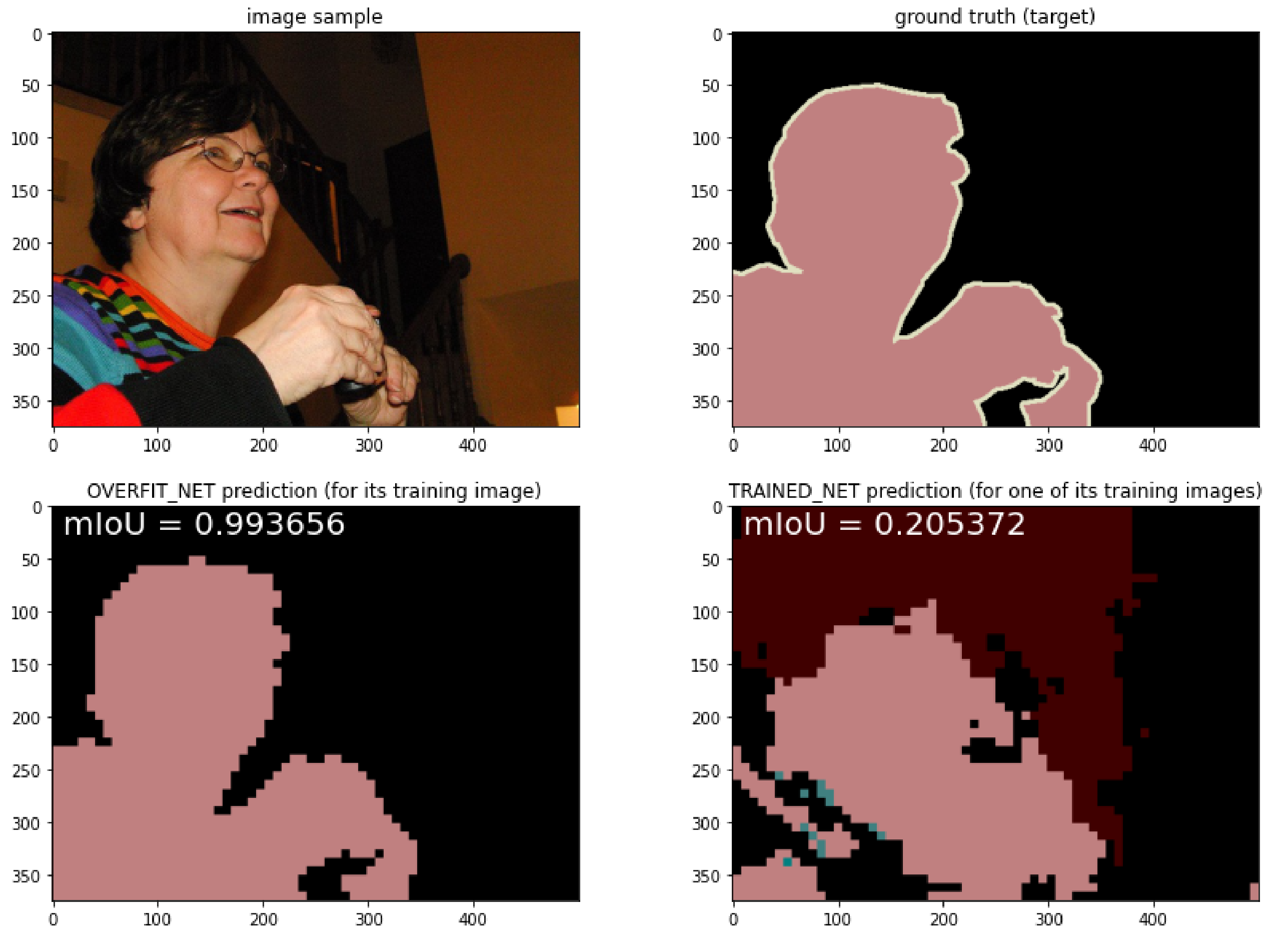
In [32]:

```python
# switch back to evaluation mode
trained_net.eval()

sample_img, sample_target = JointNormalize(*norm)(*JointToTensor()(*sample1))
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 predictions)
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.int32)).long()
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='
ax3.imshow(colorize_mask(torch.argmax(sample_output_O, dim=1).cpu().numpy()[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='
ax4.imshow(colorize_mask(torch.argmax(sample_output_T, dim=1).cpu().numpy()[0]))
```

```
/usr/local/lib/python3.7/dist-packages/chainercv/evaluations/eval_semantic_segme
ntation.py:91: RuntimeWarning: invalid value encountered in true_divide
  iou = np.diag(confusion) / iou_denominator
/usr/local/lib/python3.7/dist-packages/chainercv/evaluations/eval_semantic_segme
ntation.py:168: RuntimeWarning: invalid value encountered in true_divide
  class_accuracy = np.diag(confusion) / np.sum(confusion, axis=1)
```

Out[32]:
```
<matplotlib.image.AxesImage at 0x7f9631009b50>
```



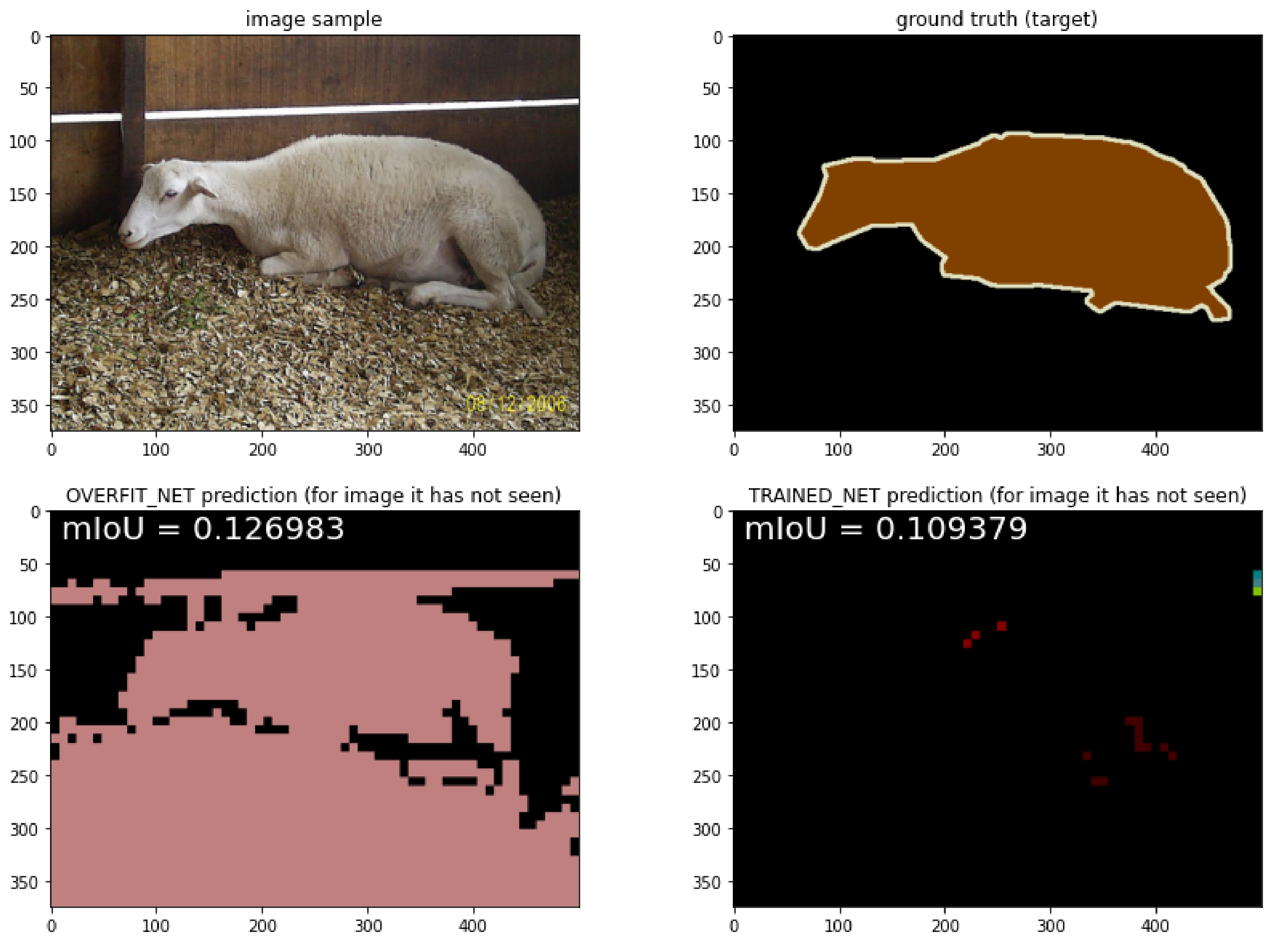
In [33]:

```
sample_img, sample_target = JointNormalize(*norm)(*JointToTensor()(*sample2))
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 predictions)
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.int32)).long()
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='
ax3.imshow(colorize_mask(torch.argmax(sample_output_O, dim=1).cpu().numpy()[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='
ax4.imshow(colorize_mask(torch.argmax(sample_output_T, dim=1).cpu().numpy()[0]))
```

```
/usr/local/lib/python3.7/dist-packages/chainercv/evaluations/eval_semantic_segme
ntation.py:91: RuntimeWarning: invalid value encountered in true_divide
  iou = np.diag(confusion) / iou_denominator
/usr/local/lib/python3.7/dist-packages/chainercv/evaluations/eval_semantic_segme
ntation.py:168: RuntimeWarning: invalid value encountered in true_divide
  class_accuracy = np.diag(confusion) / np.sum(confusion, axis=1)
```

Out[33]:

```
<matplotlib.image.AxesImage at 0x7f9630e96350>
```



(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 only a few min
print("mIoU for TRAINED_NET over the validation dataset:{}".format(validate(val_
```

```
/usr/local/lib/python3.7/dist-packages/chainercv/evaluations/eval_semantic_segme
ntation.py:91: RuntimeWarning: invalid value encountered in true_divide
  iou = np.diag(confusion) / iou_denominator
/usr/local/lib/python3.7/dist-packages/chainercv/evaluations/eval_semantic_segme
ntation.py:168: RuntimeWarning: invalid value encountered in true_divide
  class_accuracy = np.diag(confusion) / np.sum(confusion, axis=1)
mIoU for TRAINED_NET over the validation dataset:0.1874796053392594
CPU times: user 28.4 s, sys: 4.53 s, total: 32.9 s
Wall time: 32.8 s
```

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?

Student answer:

Some limitations came from the problem definition itself. It was required that we upsample right from the beginning which meant we couldn't have more `Conv2D` layers before that. Additionally, the question required that we use 1 skip connection but perhaps less or more could have produced more favorable results. However, whether this would seriously change performance is unclear and would need to be tested further.

In the one part that we could experiment with of the decoder section, I employed the following layers:

- A Batch Normalization Layer
- A 3x3 stride=2 Conv2D layer
- A ReLU Layer

Although I'm content with the results I acheived, I would remark that **my network's decoder was much more shallow than the encoder**. The encoder was initialized in part from the original `resnet18` image *classification* model which we know works well. For the purpose of not having it take extremely long to train, I opted for a more shallow decoder which I could make more sense of. One improvement would be to train a deeper network with more `Conv2D` layers which are better able to pull out the features that convert each image to a linearly separable feature space.

Bottlenecks were most likely in the convolutions of the `Conv2D` layers. Because there were many data points and the convolutions ran through the images this would slow down execution time considerably. They could be removed but this would be at the cost of complexity and probably accuracy.

An obvious issue is that when we train the results for the overfitted example become much worse. This is good because it is generalizable, but still very bad at classifying objects and only really classifies background correctly. This would possibly be improved with more epochs, a deeper decoder network, and more time spent tunning the parameters.