CS 484/684 Computational Vision

credits: many thanks for the design of this assignemnt go to [Towaki Takikawa (https://tovacinni.github.io/)](https://tovacinni.github.io/)

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/ (https://pytorch.org/tutorials/)](https://pytorch.org/tutorials/)

In [161]:
```
%matplotlib inline

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

In [162]:
```
# 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-s
tarted/locally/
# That web site should help you to select the right 'conda install' comm
and to be run in 'Anaconda Prompt'.
# In particular, select the right version of CUDA. Note that prior to in
stalling PyTorch, you should
# install the latest driver for your GPU and CUDA (9.2 or 10.1), assumin
g 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 (contain
ing 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, co
w, diningtable, dog, horse, motorbike,
#       person, pottedplant, sheep, sofa, train, TV_monitor
# 255: "void", which means class for pixel is undefined
```

In [163]:
```
# ChainerCV is a library similar to TorchVision, created and maintained
 by Preferred Networks.
# Chainer, the base library, inspired and led to the creation of PyTorc
h!
# Although Chainer and PyTorch are different, there are some nice functi
onalities in ChainerCV
# that are useful, so we include it as an excersice on learning other li
braries.
# To install ChainerCV, normally it suffices to run "pip install chainer
cv" inside "Anaconda Prompt".
# For more detailed installation instructions, see https://chainercv.rea
dthedocs.io/en/stable/install.html
# For other information about ChainerCV library, refer to https://chaine
rcv.readthedocs.io/en/stable/
from chainercv.evaluations import eval_semantic_segmentation
from chainercv.datasets import VOCSemanticSegmentationDataset
```

In [164]:
```
# 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.Imag
e.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 [165]:
```
# Below we will use a sample image-target pair from VOC training dataset to test your joint transforms.
# Running this block will automatically download the PASCAL VOC Dataset (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 directory. 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 VOC training dataset and one from validation dataset.
# Note that operator [...] extracts the sample corresponding to the specified index.
# Also, note the parameter download = True. Set this to False after you 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[165]: <matplotlib.image.AxesImage at 0x7f83f3538410>

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-

Example 1

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

Solution:

In [166]:
```
class JointToTensor(object):
    def __call__(self, img, target):
        return tF.to_tensor(img), torch.from_numpy(np.array(target.conve
rt('P'), dtype=np.int32)).long()
```

In [167]:
```
# Check the transform by passing the image-target sample.

JointToTensor()(*sample1)
```

Out[167]:
```
(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]]))
```

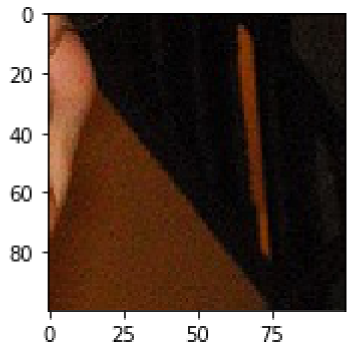
Example 2:

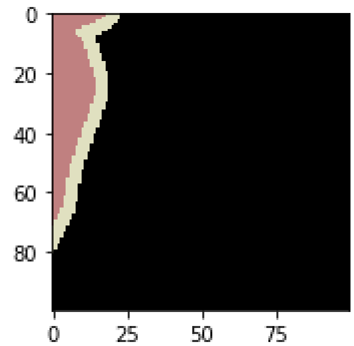
This class implements CenterCrop that takes an img, target pair, and then apply a crop about the center of the image such that the output resolution is $\mathrm{size} \times \mathrm{size}$.

Solution:

In [168]:
```
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[168]: `<matplotlib.image.AxesImage at 0x7f83f357ebd0>`





(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 [169]:
```
class RandomFlip(object):
    def __call__(self, img, target):
        if random.random() < 0.5:
            return (tF.hflip(img), tF.hflip(target))
        else:
            return (img, target)

img, target = RandomFlip()(*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[169]:
```
<matplotlib.image.AxesImage at 0x7f8411fae410>
```

(b) Implement RandomResizeCrop

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

Solution:

In [170]:
```
class RandomResizeCrop(object):
    def __init__(self, size, minimum_scale, maximum_scale):
        self.size = size
        self.minimum_scale = minimum_scale
        self.maximum_scale = maximum_scale

    def __call__(self, img, target):
        scale = random.uniform(self.minimum_scale, self.maximum_scale)
        newSize = (int(img.size[1] * scale), int(img.size[0] * scale))
        img = tF.resize(img, newSize)
        target = tF.resize(target, newSize)

        img_width, img_height = img.size
        crop = min(self.size, img_height, img_width)
        x = random.randint(0, img.size[1] - crop)
        y = random.randint(0, img.size[0] - crop)

        return (
            tF.resized_crop(img, x, y, crop, crop, self.size),
            tF.resized_crop(target, x, y, crop, crop, self.size)
        )
```

(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 [171]:
```
norm = ([0.485, 0.456, 0.406],
        [0.229, 0.224, 0.225])
class JointNormalize(object):
    def __init__(self, mean, var):
        self.mean = mean
        self.var = var

    def __call__(self, img, target):
        return (tF.normalize(img, self.mean, self.var), target)
```

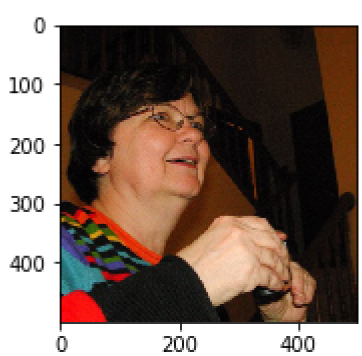
(d) Compose the transforms together:

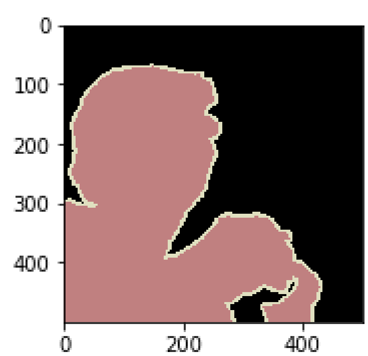
Use `JointCompose` (fully implemeted below) to compose the implemented transforms together in some random order. Verify the output makes sense and visualize it.

In [172]:
```
# This class composes transofrmations from a given list of image transfo
rms (expected in the argument). Such compositions
# will be applied to the dataset during training. This cell is fully imp
lemented.

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 [173]:
```
# Student Answer:

transforms = [RandomResizeCrop(500, 0.8, 1.2), RandomFlip()]
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[173]: <matplotlib.image.AxesImage at 0x7f8402ab7a50>





(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 [174]:
```
# Student Answer:

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

train_joint_transform = JointCompose([RandomResizeCrop(512, 0.8, 1.2), RandomFlip(), JointToTensor(), JointNormalize(*norm)])

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

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

In [175]:
```
# Apply the Joint-Compose transformations above to create three datasets
and the corresponding Data-Loaders.
# This cell is fully implemented.

# This single image data(sub)set can help to better understand and to de
bug the network training process.
# Optional integer parameter 'sanity_check' specifies the index of the i
mage-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 segment
ation 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 segme
ntation 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 training.
# When batch size changes, the learning rate may also need to be adjuste
d.
# Note that batch size maybe limited by your GPU memory, so adjust if yo
u get "run out of GPU memory" error.
TRAIN_BATCH_SIZE = 4

# 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_WORKERS
 must be 0 for Windows.
NUM_WORKERS = 0

sanity_loader = DataLoader(sanity_data, batch_size=1, num_workers=NUM_WO
RKERS, shuffle=False)
train_loader = DataLoader(train_data, batch_size=TRAIN_BATCH_SIZE, num_w
orkers=NUM_WORKERS, shuffle=True)
val_loader = DataLoader(val_data, batch_size=1, num_workers=NUM_WORKERS,
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 (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 (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.

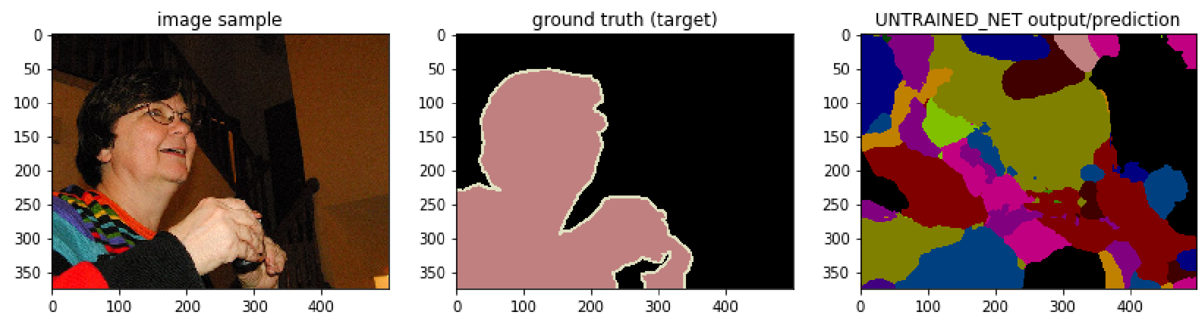
Note: do not apply any post-processing (such as DenseCRF) to the output of your net.

In [141]:
```
import torchvision.models as models

class MyNet(nn.Module):
    def __init__(self, num_classes, criterion=None):
        super(MyNet, self).__init__()

        self.num_classes = num_classes
        self.criterion = criterion
        self.resnet = models.resnet18(pretrained=True)
        self.conv1 = nn.Conv2d(576, 256, 5, 1)
        self.bn = nn.BatchNorm2d(256)
        self.conv2 = nn.Conv2d(256, self.num_classes, 1, 1)

    def forward(self, inp, gts=None):
        shape = inp.shape
        resnet = self.resnet
        inp = resnet.conv1(inp)
        inp = resnet.bn1(inp)
        inp = resnet.relu(inp)
        inp = resnet.maxpool(inp)

        tmp = inp.clone()

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

        # upsample
        inp = F.interpolate(inp, size=(tmp.shape[2], tmp.shape[3]), mode
='bilinear', align_corners=True)
        # concat
        inp = torch.cat((inp, tmp), 1)

        # Some number of convs, ReLUs, BNs
        inp = self.conv1(inp)
        inp = F.relu(inp)
        inp = self.bn(inp)
        inp = self.conv2(inp)

        # upsample
        inp = F.interpolate(inp, size=(shape[2], shape[3]), mode='biline
ar', 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 [142]:
```python
untrained_net = MyNet(21).eval()
tensor = JointToTensor()(*sample1)
sample_img, sample_target = JointNormalize(*norm)(*tensor)
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[142]:
```
<matplotlib.image.AxesImage at 0x7f83f2336e90>
```



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 [121]:
```python
# Student Answer:

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

    def __call__(self, output, target):
        softmax = F.log_softmax(output, dim=1)
        return F.nll_loss(softmax, target, ignore_index=self.ignore_inde
x)
```

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

In [122]:
```
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(3.0051, grad_fn=<NllLoss2DBackward>)
tensor(3.0051, 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 [123]:
```
# Write code to propely compute 'pred' and 'gts' as arguments for functi
on 'eval_semantic_segemntation'

pred = torch.argmax(untrained_output, dim=1).numpy()[0]
gts = torch.from_numpy(np.array(sample1[1].convert('P'), dtype=np.int32
)).long().numpy()

conf = eval_semantic_segmentation([pred], [gts])

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

```
mIoU for the sample image / ground truth pair: 0.004559702360668718
```

(b) Write the validation loop.

In [124]:
```
# Modified from Classification notebook
def validate(val_loader, net):

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

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

            output = net(inputs)
            val_loss += MyCrossEntropyLoss(ignore_index=255)(output, mas
ks)

            preds = torch.argmax(output, dim=1).cpu().numpy()
            gts = torch.from_numpy(np.array(masks.cpu(), dtype=np.int32
)).long().numpy()
            gts[gts == 255] = -1

            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 [59]:
```
%%time
print("mIoU over the sanity dataset:{}".format(validate(sanity_loader, u
ntrained_net)[1]))
```

```
mIoU over the sanity dataset:0.009797372779607655
CPU times: user 1.19 s, sys: 30 ms, total: 1.22 s
Wall time: 1.01 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)](https://pytorch.org/docs/stable/optim.html).

In [60]:
```
def get_optimizer(net):
    optimizer = torch.optim.SGD(net.parameters(),lr=0.01, momentum=1, ne
sterov=False)
    return optimizer
```

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

In [61]:
```
def train(train_loader, net, optimizer, loss_graph):

    for i, data in enumerate(train_loader):

        inputs, masks = data

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

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

    return main_loss
```

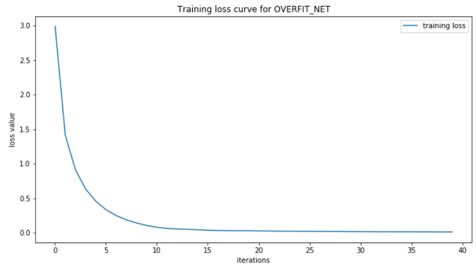
(c) Create OVERFIT_NET and train it on the single image dataset.

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 [62]:

```
%%time
%matplotlib notebook

# The whole training on a single image (20-40 epochs) should take only 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 single 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 notebook compares different versions of network training.
# For this comparison to be direct and fair, it is better to train (deep) 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 = 40

# 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...
```



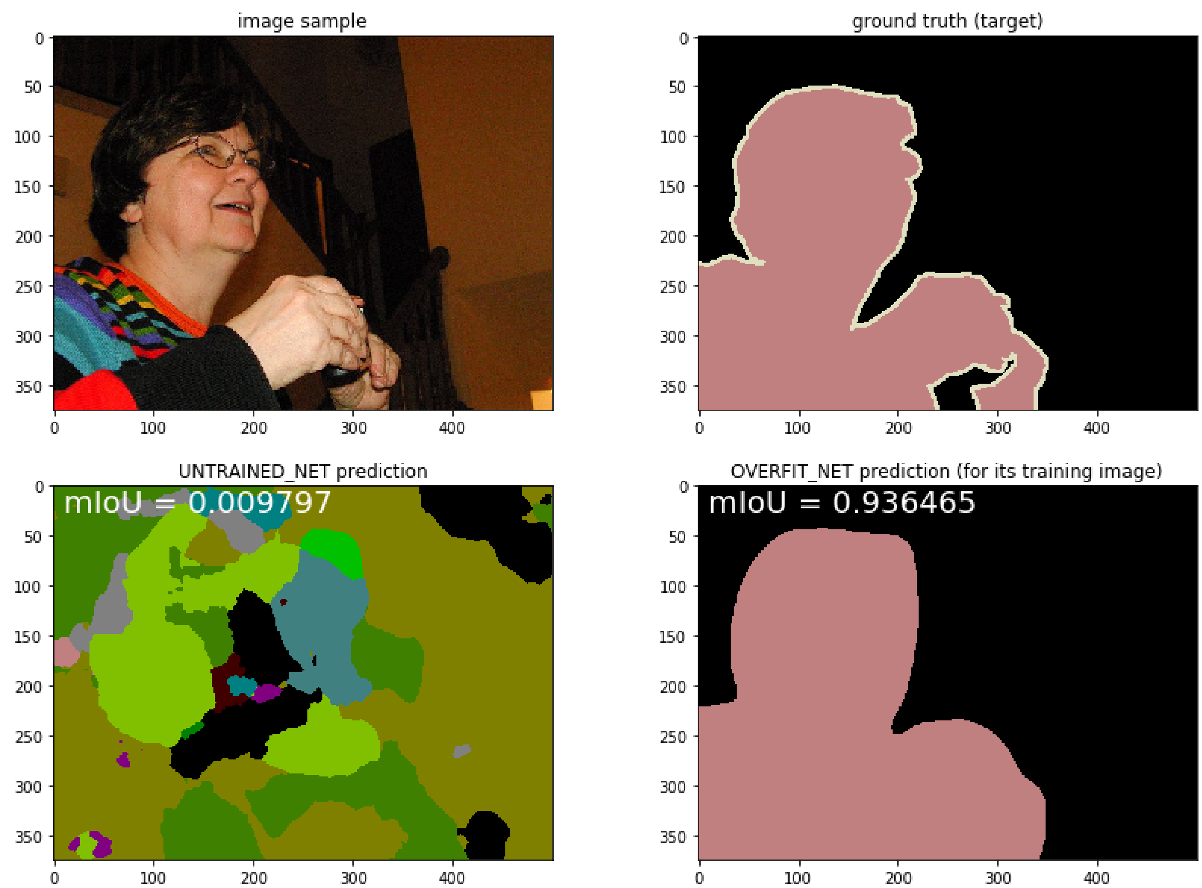
```
Epoch: 0 Loss: 2.989421844482422
Epoch: 1 Loss: 1.4124222993850708
Epoch: 2 Loss: 0.9121121168136597
Epoch: 3 Loss: 0.6315199136734009
Epoch: 4 Loss: 0.4562312364578247
Epoch: 5 Loss: 0.3331984281539917
Epoch: 6 Loss: 0.24748237431049347
Epoch: 7 Loss: 0.1863166242837906
Epoch: 8 Loss: 0.1408568024635315
Epoch: 9 Loss: 0.10504575818777084
Epoch: 10 Loss: 0.07805249094963074
Epoch: 11 Loss: 0.0622299462556839
Epoch: 12 Loss: 0.05381079018115997
Epoch: 13 Loss: 0.048066090792417526
Epoch: 14 Loss: 0.04277493432164192
Epoch: 15 Loss: 0.03547084704041481
Epoch: 16 Loss: 0.0316852368414402
Epoch: 17 Loss: 0.029158752411603928
Epoch: 18 Loss: 0.027614006772637367
Epoch: 19 Loss: 0.027569476515054703
Epoch: 20 Loss: 0.02527693286538124
Epoch: 21 Loss: 0.02460256591439247
Epoch: 22 Loss: 0.022493725642561913
Epoch: 23 Loss: 0.021706128492951393
Epoch: 24 Loss: 0.020044509321451187
Epoch: 25 Loss: 0.01855964958667755
Epoch: 26 Loss: 0.018206851556897163
Epoch: 27 Loss: 0.01729358360171318
Epoch: 28 Loss: 0.017113786190748215
Epoch: 29 Loss: 0.015632830560207367
Epoch: 30 Loss: 0.015541465021669865
Epoch: 31 Loss: 0.014159861952066422
Epoch: 32 Loss: 0.013857288286089897
Epoch: 33 Loss: 0.013750523328781128
Epoch: 34 Loss: 0.013394617475569248
Epoch: 35 Loss: 0.013089527375996113
Epoch: 36 Loss: 0.012166401371359825
Epoch: 37 Loss: 0.01190604455769062
Epoch: 38 Loss: 0.011092029511928558
Epoch: 39 Loss: 0.011188066564500332
CPU times: user 2min 27s, sys: 3.22 s, total: 2min 30s
Wall time: 2min 8s
```

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

In [63]:
```
# switch back to evaluation mode
overfit_net.eval()

sample_img, sample_target = JointNormalize(*norm)(*JointToTensor()(*samp
le1))
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 predictio
ns)
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().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().nump
y()[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().nump
y()[0]))
```

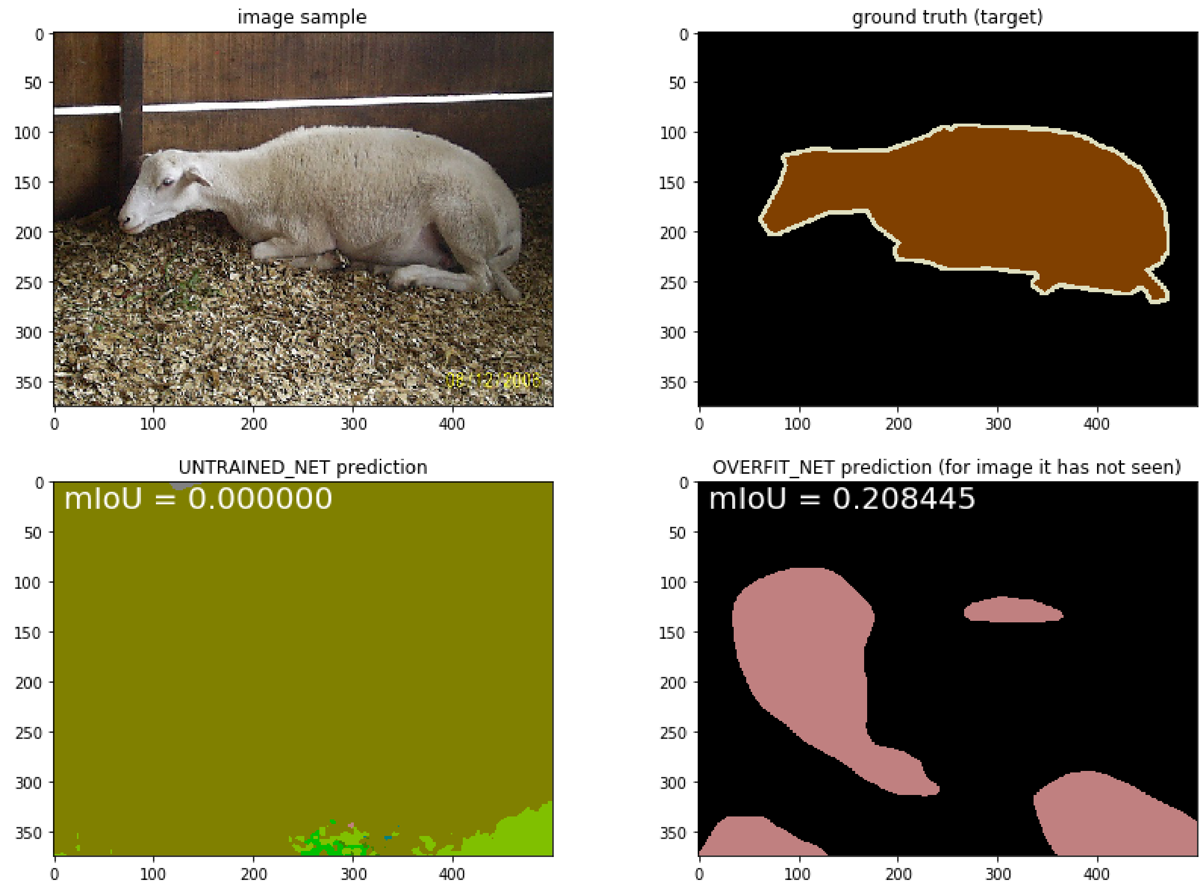
```
/Users/ryanqin/opt/anaconda3/lib/python3.7/site-packages/chainercv/eval
uations/eval_semantic_segmentation.py:91: RuntimeWarning: invalid value
encountered in true_divide
  iou = np.diag(confusion) / iou_denominator
/Users/ryanqin/opt/anaconda3/lib/python3.7/site-packages/chainercv/eval
uations/eval_semantic_segmentation.py:168: RuntimeWarning: invalid valu
e encountered in true_divide
  class_accuracy = np.diag(confusion) / np.sum(confusion, axis=1)
```

Out[63]: <matplotlib.image.AxesImage at 0x7f842359dd90>



In [64]:
```
sample_img, sample_target = JointNormalize(*norm)(*JointToTensor()(*samp
le2))
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 predictio
ns)
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().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().nump
y()[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().nump
y()[0]))
```

Out[64]: <matplotlib.image.AxesImage at 0x7f83f1c33d10>



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

In [65]:
```
%%time
print("mIoU for OVERFIT_NET over its training image:{}".format(validate(
sanity_loader, overfit_net)[1]))
```

```
mIoU for OVERFIT_NET over its training image:0.9364652003780689
CPU times: user 1.22 s, sys: 23.8 ms, total: 1.24 s
Wall time: 1.02 s
```

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 [66]:
```
%%time
# This will be slow on CPU (around 1 hour or more). On GPU it should tak
e only a few minutes (depending on your GPU).
print("mIoU for UNTRAINED_NET over the entire dataset:{}".format(validat
e(val_loader, untrained_net)[1]))
```

```
mIoU for UNTRAINED_NET over the entire dataset:0.004874066448390884
CPU times: user 26min 36s, sys: 29.7 s, total: 27min 6s
Wall time: 22min 21s
```

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

In [67]:
```
%%time
# This will be slow on CPU (around 1 hour or more). On GPU it should tak
e only a few minutes (depending on your GPU).
print("mIoU for OVERFIT_NET over the validation dataset:{}".format(valid
ate(val_loader, overfit_net)[1]))
```

```
mIoU for OVERFIT_NET over the validation dataset:0.24012849011459317
CPU times: user 26min 51s, sys: 29.7 s, total: 27min 21s
Wall time: 22min 34s
```

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

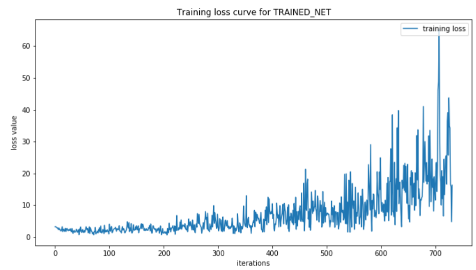
Student answer:

The overfit_net performs really well on the image it uses to train as expected, with a great mIoU around 0.2 ~ 0.4. The untrained_net gives random results and has small mIoU, less than 0.01. On an unseen image, both nets perform really bad. This makes sense, because the overfit_net is trained on only one image.

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

In [177]:
```
# %%time
%matplotlib notebook

# This training will be very slow on a CPU (>1hour per epoch). Ideally,
 this should be run in GPU mode (USE_GPU=True)
# taking only a few minutes per epoch (depending on your GPU and batch s
ize). Thus, before proceeding with this excercise,
# it is highly advisable that you first finish debugging your net code.
 In particular, make sure that OVERFIT_NET behaves
# reasonably, e.g. its loss monotonically decreases during training and
 its output is OK (for the image it was trained on).
# Below we create another (deep) copy of untrained_net. Unlike OVERFIT_N
ET 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 TRAINE
D_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 = 2

# 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...
```



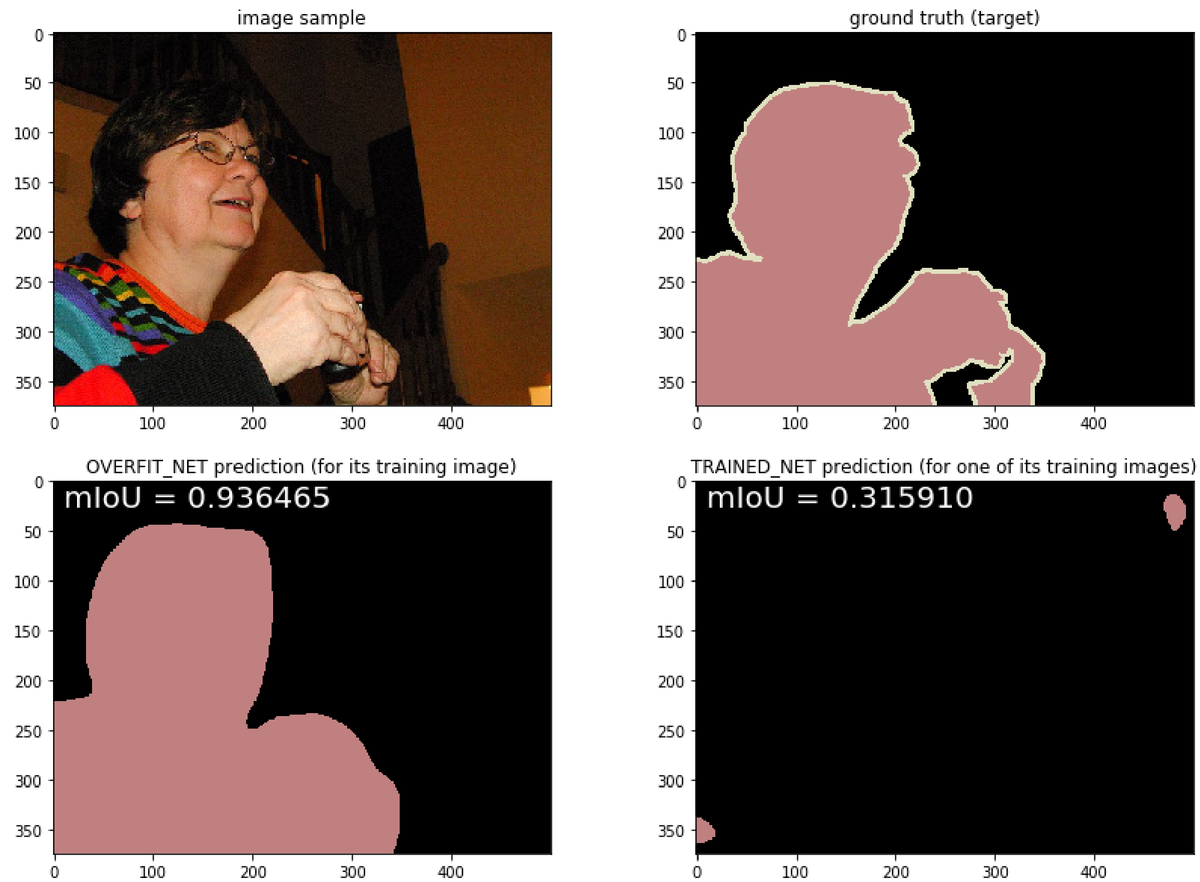

```
Epoch: 0 Loss: 5.053501605987549
Epoch: 1 Loss: 16.283464431762695
```

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

In [178]:
```
# switch back to evaluation mode
trained_net.eval()

sample_img, sample_target = JointNormalize(*norm)(*JointToTensor()(*samp
le1))
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 predictio
ns)
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().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().nump
y()[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().nump
y()[0]))
```

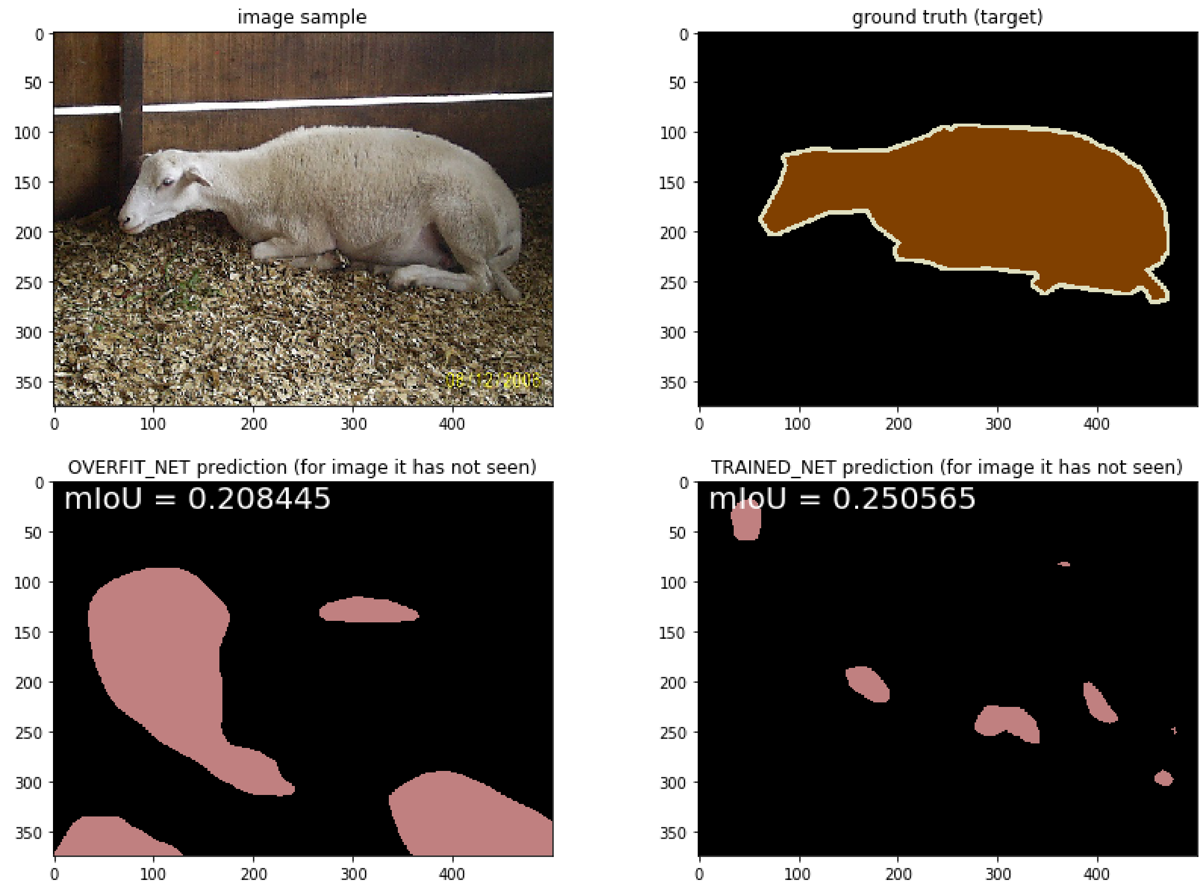
```
/Users/ryanqin/opt/anaconda3/lib/python3.7/site-packages/chainercv/eval
uations/eval_semantic_segmentation.py:91: RuntimeWarning: invalid value
encountered in true_divide
  iou = np.diag(confusion) / iou_denominator
/Users/ryanqin/opt/anaconda3/lib/python3.7/site-packages/chainercv/eval
uations/eval_semantic_segmentation.py:168: RuntimeWarning: invalid valu
e encountered in true_divide
  class_accuracy = np.diag(confusion) / np.sum(confusion, axis=1)
```

```
Out[178]: <matplotlib.image.AxesImage at 0x7f8415a42890>
```



In [179]:
```
sample_img, sample_target = JointNormalize(*norm)(*JointToTensor()(*samp
le2))
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 predictio
ns)
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().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().nump
y()[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().nump
y()[0]))
```

Out[179]: <matplotlib.image.AxesImage at 0x7f8409f18ed0>



(h) Evaluate TRAINED_NET on validation dataset.

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

In [180]:
```
%%time
# This will be slow on CPU (around 1 hour). On GPU it should take only a few minutes (depending on your GPU).
print("mIoU for TRAINED_NET over the validation dataset:{}".format(validate(val_loader, trained_net)[1]))
```

```
mIoU for TRAINED_NET over the validation dataset:0.23909625614148933
CPU times: user 22min 32s, sys: 24.8 s, total: 22min 57s
Wall time: 18min 41s
```

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 limitation of the network is its efficiency and correctness. We see that even though it's only run for 2 epochs, it takes very long to do so, and the loss doesn't decrease. Another limitation is the performance, although we see a relative good result, it could still be improved by having a more advanced model. For example, this network is not as "deep" as some of the modern networks, which may give better results but require better hardware to run

In [ ]: