CS 484/684 Computational Vision

Many thanks for the design of this assignment go to Towaki Takikawa (https://tovacinni.github.io/) and Olga Veksler (https://cs.uwaterloo.ca/~oveksler/)

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.Ther is one simple theoretical problem 0. The rest is the programming part.

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/)

Problem 0(a)

Consider linear soft-max classifier $\bar{\sigma}(WX)$ for $K = 2$ (see slides 64-65, topic 10), where $X \in R^{m+1}$ is a homogeneous vector representation of m-dimensional feature (or data point). The classifier parameters matrix W consists of two rows representing linear discriminants W_1 and W_2 in R^{m+1} (including the bias). Prove that soft-max classifier $\bar{\sigma}(WX)$ is equivalent to the sigmoid classifier $\sigma((W_2 - W_1)^T X)$ (slide 48, topic 10).

Your Solution:

Type it here, use latex for math formulas.

$$\exp(-(W_2 - W_1)^T x) = \exp(-(W_2^T - W_1^T)x) = \exp((W_1^T - W_2^T)x)$$

$$\exp((W_1^T - W_2^T)x) > 0 \implies \frac{1}{1 + \exp(-(W_2 - W_1)^T x)} > \frac{1}{2}$$

$$2 > 1 + \exp\left(W_1^T x - W_2^T x\right)$$

$$1 > \exp\left(W_1^T x - W_2^T x\right)$$

$$0 > (W_1^T x - W_2^T x)$$

$$W_2^T x > W_1^T x$$

Problem 0(b)

Consider linear classifier $\bar{\sigma}(WX)$ for any given number of classes K, where $X \in R^{m+1}$ is a homogeneous representation of m-dimensional feature vector and W is a matrix of size $K \times (m+1)$. Specify an equation for a hyperplane in the feature space corresponding to the *decision boundary* between two classes i and j.

HINT1: Decision bounary is the boundary of two desion regions in the feature space: one is a set of all features where the classifier prefers calss i, i.e. $\bar{\sigma}_i(WX) \geq \bar{\sigma}_j(WX)$, and the other is a set of features where the classifier prefers class j, i.e. $\bar{\sigma}_i(WX) \leq \bar{\sigma}_j(WX)$.

HINT2: any hyperplane in the feature space R^m can be represented by equation $P^T X = 0$ where $P \in R^{m+1}$ is a vector of the hyperplane parameters. Essentially, your solution should specifiy P based on the parameters of the linear classifier W. The solution should be very simple - this problem is mainly an excersice in terminology that should force you to review the slides.

Your Solution:

Type it here, use latex for math formulas.

$$g_1(x) = W_1^T x$$

$$g_2(x) = W_2^T x$$

$$g_1(x) - g_2(x) = W_1^T x - W_2^T x = 0$$

$$g_1(x) - g_2(x) = (W_1^T - W_2^T)x = 0$$

$$P = (W_1^T - W_2^T)$$

Programming part

In [79]:
```
%matplotlib inline

# It is best to start with USE_GPU = False (implying CPU). Switch USE_
# we strongly recommend to wait until you are absolutely sure your CPU
USE_GPU = False
```

In [80]:
```
# 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
```

In [81]:
```
# We import some of the main PyTorch and TorchVision libraries used fo
# Detailed installation instructions are here: https://pytorch.org/get
# That web site should help you to select the right 'conda install' co
# In particular, select the right version of CUDA. Note that prior to
# install the latest driver for your GPU and CUDA (9.2 or 10.1), assum
# 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 (conta
# that allows us to easily create single-image datasets


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

In [82]:
```
from lib.voc import VOCSegmentation
```

In [83]:
```
# ChainerCV is a library similar to TorchVision, created and maintaine
# Chainer, the base library, inspired and led to the creation of PyTor
# Although Chainer and PyTorch are different, there are some nice func
# that are useful, so we include it as an excersice on learning other
# To install ChainerCV, normally it suffices to run "pip install chain
# For more detailed installation instructions, see https://chainercv.r
# For other information about ChainerCV library, refer to https://chai
from chainercv.evaluations import eval_semantic_segmentation
from chainercv.datasets import VOCSemanticSegmentationDataset
```

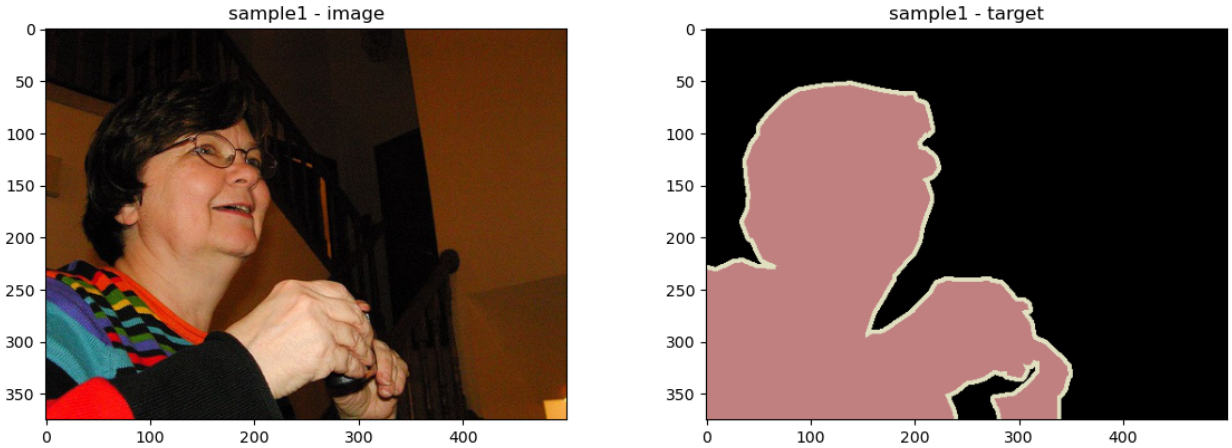
In [84]:
```
# This colorize_mask class takes in a numpy segmentation mask,
#  and then converts it to a PIL Image for visualization.
#  Since by default the numpy matrix contains integers from
#  0,1,...,num_classes, we need to apply some color to this
#  so we can visualize easier! Refer to:
#  https://pillow.readthedocs.io/en/4.1.x/reference/Image.html#PIL.Ima
palette = [0, 0, 0, 128, 0, 0, 0, 128, 0, 128, 128, 0, 0, 0, 128, 128,
           128, 128, 128, 64, 0, 0, 192, 0, 0, 64, 128, 0, 192, 128, 0
           64, 128, 128, 192, 128, 128, 0, 64, 0, 128, 64, 0, 0, 192,

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

    return new_mask
```

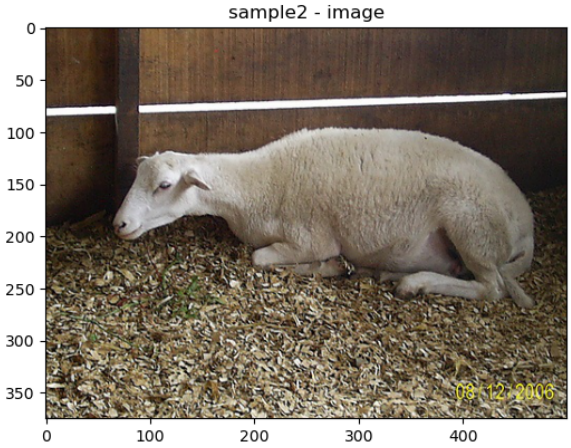
In [85]:

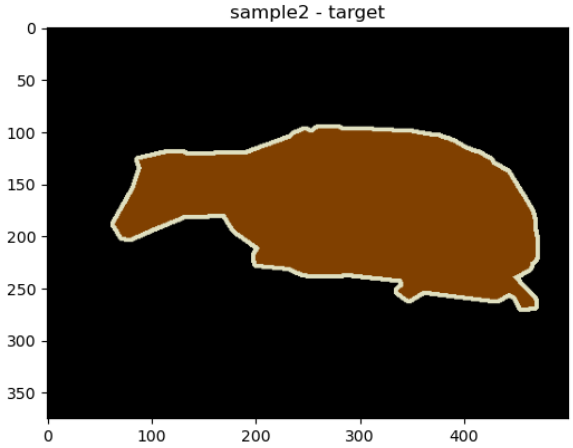
```
# Below we will use a sample image-target pair from VOC training datase
# Running this block will automatically download the PASCAL VOC Dataset
# The code below creates subdirectory "datasets" in the same location a
# you can modify DATASET_PATH to download the dataset to any custom di
# On subsequent runs you may save time by setting "download = False" (

DATASET_PATH = 'datasets'

# Here, we obtain and visualize one sample (img, target) pair from VOC
# Note that operator [...] extracts the sample corresponding to the spe
# Also, note the parameter download = True. Set this to False after you
sample1 = VOCSegmentation(DATASET_PATH, image_set='train', download = T
sample2 = VOCSegmentation(DATASET_PATH, image_set='val')[20]

# We demonstrate two different (equivalent) ways to access image and ta
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)
```

```
Using downloaded and verified file: datasets/VOCtrainval_11-May-2012.
tar
```

Out[85]: <matplotlib.image.AxesImage at 0x14fcb89d0>




sample2 - image
0
50
100
150
200
250
300
350
0
100
200
300
400
09 12 2006


sample2 - target
0
50
100
150
200
250
300
350
0
100
200
300
400

Problem 1

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

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

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

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

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

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

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

Example 1

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

Solution:

In [86]:
```
class JointToTensor(object):
    def __call__(self, img, target):
        return tF.to_tensor(img), torch.from_numpy(np.array(target.cor
```

In [87]:
```
# Check the transform by passing the image-target sample.

img, target  = JointToTensor()(*sample1)
print(type(img))
print(type(target))
print(img.shape)
print(target.shape)
```

```
<class 'torch.Tensor'>
<class 'torch.Tensor'>
torch.Size([3, 375, 500])
torch.Size([375, 500])
```

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 [88]:
```
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)
print(type(img))
print(type(target))
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)
```

```
<class 'PIL.Image.Image'>
<class 'PIL.Image.Image'>
```

Out[88]:
```
<matplotlib.image.AxesImage at 0x15cf1af10>
```

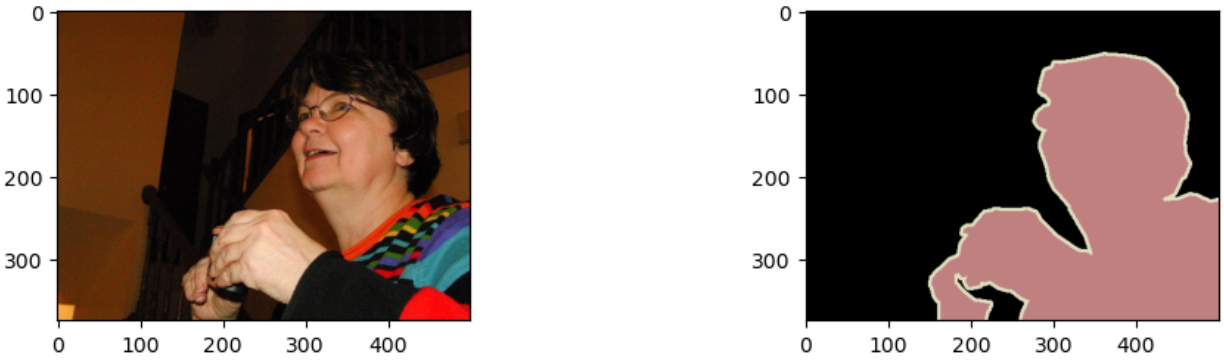
(a) Implement RandomFlip

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

Solution:

In [89]:
```
class RandomFlip(object):
    def __init__(self):
        pass

    def __call__(self, img, target):
        if torch.rand(1)>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[89]: <matplotlib.image.AxesImage at 0x15ce87c70>



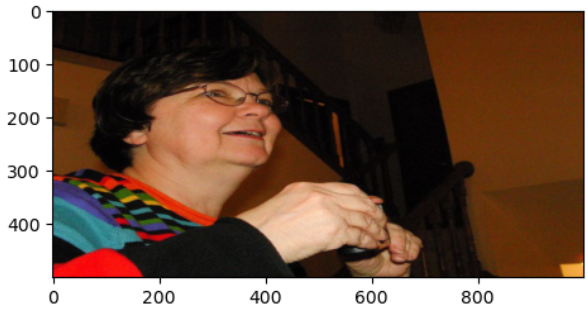
(b) Implement RandomResizeCrop

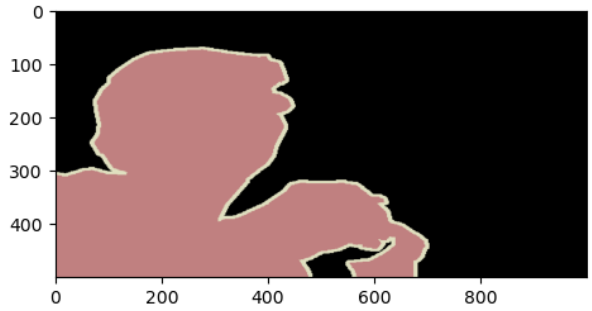
This class should take a img, target pair and then resize the images by a random scale between $[\text{minimum\_scale}, \text{maximum\_scale}]$, crop a random location of the image by $\min(\text{size}, \text{image\_height}, \text{image\_width})$ (where the size is passed in as an integer in the constructor),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

Solution:

In [90]:
```
mport random
This class should take a img, target pair and then resize the images b
crop a random location of the image by $\min(\mbox{size}, \mbox{image_
 in the constructor), and then resize to $\mbox{size} \times \mbox{siz

lass Resize(object):
   def __init__(self, size_height=100,size_width=100):
       self.size_height = size_height
       self.size_width = size_width
       pass

   def __call__(self, img, target):
       size_height = self.size_height
       size_width = self.size_width
       img = tF.resize(img, (size_height,size_width))
       target = tF.resize(target, (size_height,size_width))

       return (img, target)


mage_input_size_new = (500,1000)

mg, target = Resize(image_input_size_new[0], image_input_size_new[1])(
ig = plt.figure(figsize=(12,6))
x1 = fig.add_subplot(2,2,1)
x1.imshow(img)
x2 = fig.add_subplot(2,2,2)
x2.imshow(target)
```

Out[90]: <matplotlib.image.AxesImage at 0x15cebb820>





In [91]:
```
import random
#This class should take a img, target pair and then resize the images b
#crop a random location of the image by $\min(\mbox{size}, \mbox{image_
# in the constructor), and then resize to $\mbox{size} \times \mbox{siz

class RandomResizeCrop(object):
    def __init__(self,min_scale=0.9,max_scale=1.1, size=100):
        self.min_scale = min_scale
        self.max_scale = max_scale
        self.size = size
        pass

    def __call__(self, img, target):
        scale = torch.rand(1)*(self.max_scale-self.min_scale)+self.min_
        w,h=img.size[-2:]
        new_scale = (int(h*scale), int(w*scale))
        resized_img = tF.resize(img, new_scale)
        resized_tar = tF.resize(target, new_scale)
        size = int(min(self.size, new_scale[0], new_scale[1]))
        x_crop = random.randint(0, new_scale[1]-size)
        y_crop = random.randint(0, new_scale[0]-size)
        cropped_img = tF.crop(resized_img, y_crop, x_crop, size,size)
        cropped_tar = tF.crop(resized_tar, y_crop, x_crop, size,size)


        cropped_img = tF.resize(cropped_img, (size,size))
        cropped_tar = tF.resize(cropped_tar, (size,size))

        return (cropped_img, cropped_tar)
```

In [92]:
```
print(sample1[0].size)
img, target = RandomResizeCrop(min_scale=0.9, max_scale=1.1, size=100)
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)
```

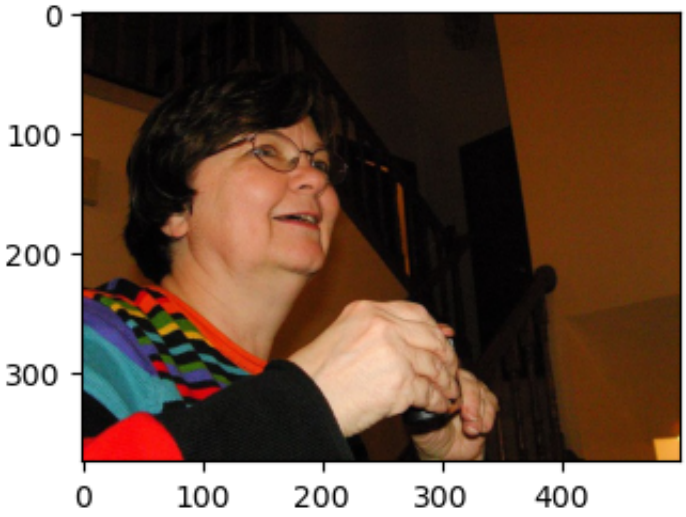
```
(500, 375)
```

Out[92]:
```
<matplotlib.image.AxesImage at 0x15cfb8370>
```

(c) Implement Normalize

In [93]:
```
class NullTransform(object):
    def __init__(self):
        """
        params:
            p (float) : probability of the horizontal fli[]
        """


    def __call__(self, img, target):
        return img,target

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

Out[93]: <matplotlib.image.AxesImage at 0x15d0325e0>



In [94]:
```
type(img)
```

Out[94]: PIL.Image.Image

In [95]:

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

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

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

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

Out[95]:
```
(tensor([[[-1.9295, -1.8953, -1.9638,  ..., -0.7479, -0.4911, -0.5938
],
          [-2.0323, -1.9295, -2.0152,  ..., -0.6794, -0.5596, -0.6794
],
          [-1.9467, -1.8953, -1.9124,  ..., -0.6281, -0.4739, -0.5424
],
          ...,
          [ 1.9920,  2.2318,  2.2489,  ...,  2.0948,  2.1119,  2.1290
],
          [ 2.2489,  2.1119,  2.2318,  ...,  2.0777,  2.0948,  2.1119
],
          [ 2.2489,  2.0263,  2.2489,  ...,  2.1290,  2.1290,  2.1804
]],

         [[-1.8606, -1.8256, -1.9482,  ..., -1.5105, -1.2479, -1.3004
],
          [-1.9657, -1.8606, -2.0007,  ..., -1.4580, -1.3179, -1.4405
],
          [-1.8782, -1.8256, -1.8957,  ..., -1.4580, -1.2479, -1.3179
],
          ...,
          [-1.9657, -1.7381, -1.7206,  ...,  0.8880,  0.9055,  0.9055
],
          [-1.6856, -1.8431, -1.7381,  ...,  0.8704,  0.8704,  0.8880
],
          [-1.7031, -1.9307, -1.6856,  ...,  0.9230,  0.9230,  0.9580
]],

         [[-1.6650, -1.6302, -1.7347,  ..., -1.7522, -1.4907, -1.5604
],
          [-1.7696, -1.6650, -1.7870,  ..., -1.6999, -1.5604, -1.6824
],
          [-1.6824, -1.6302, -1.6824,  ..., -1.6824, -1.4907, -1.5604
],
```

```
          ...,
          [-1.5604, -1.3687, -1.3861,  ..., -0.5495, -0.5321, -0.5321
],
          [-1.2816, -1.4384, -1.3687,  ..., -0.5670, -0.5670, -0.5495
],
          [-1.2641, -1.5256, -1.3164,  ..., -0.5147, -0.5147, -0.4798
]]]),
 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]]))
```

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

Solution:

(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 [96]:
```
This class composes transofrmations from a given list of image transfo
will be applied to the dataset during training. This cell is fully imp

ass 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 [97]:
```
Student Answer:
This class composes transofrmations from a given list of image transf
will be applied to the dataset during training. This cell is fully im

lass JointComposeRandom(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):
       random.shuffle(self.transforms)
       JointCompose(self.transforms)(img, target)
       return img, target
```

(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 [98]:
```
norm = ([0.485, 0.456, 0.406],
        [0.229, 0.224, 0.225])
```

In [99]:
```
image_input_size = (375, 500)
```

In [100]:
```
# Student Answer:


sanity_joint_transform = JointCompose([ \
    RandomResizeCrop(0.9,2, 250),
    JointCenterCrop(224),
    RandomFlip(),
    Resize(image_input_size[0], image_input_size[1]),
    JointToTensor(),
    Normalize(*norm),
])

train_joint_transform = JointCompose([ \
    Resize(image_input_size[0], image_input_size[1]),
    RandomResizeCrop(0.9,2, 250),
    JointCenterCrop(224),
    RandomFlip(),
    Resize(image_input_size[0], image_input_size[1]),
    JointToTensor(),
    Normalize(*norm),
])


val_joint_transform = JointCompose([ \
    Resize(image_input_size_new[0], image_input_size_new[1]),
    RandomResizeCrop(1,1, 250),
    JointCenterCrop(224),
    RandomFlip(),
    Resize(image_input_size[0], image_input_size[1]),
    JointToTensor(),
    Normalize(*norm),
])
```

Student Answer:

sanity_joint_transform = JointCompose([\

RandomResizeCrop(0.9,2, 250),

JointCenterCrop(224),

RandomFlip(),

```
    JointToTensor(),
    Normalize(*norm),
```

])

train_joint_transform = JointCompose([
RandomResizeCrop(0.9,2, 250), JointCenterCrop(224), RandomFlip(), JointToTensor(), Normalize(*norm),])

val_joint_transform = JointCompose([
RandomResizeCrop(1,1, 250),

JointCenterCrop(224),

RandomFlip(),

```
    JointToTensor(),
    Normalize(*norm),
```

])

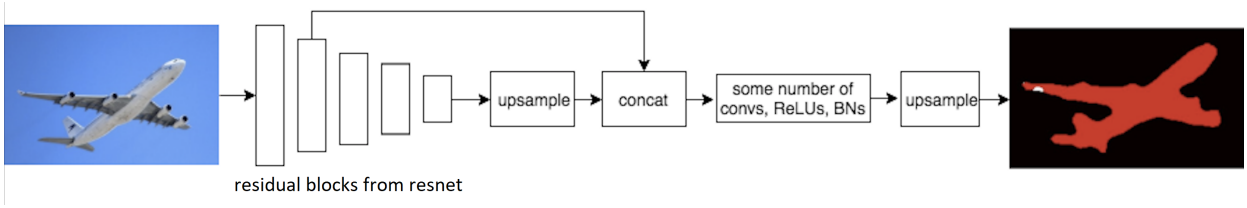
This code below will then apply `train_joint_transform` to the entire dataset.

In [101]:
```
pply the Joint-Compose transformations above to create three datasets a
his cell is fully implemented.

his single image data(sub)set can help to better understand and to debu
ptional integer parameter 'sanity_check' specifies the index of the ima
pte that we use the same image (index=200) as used for sample1.
ity_data = VOCSegmentation(
 DATASET_PATH,
 image_set = 'train',
 transforms = sanity_joint_transform,
 sanity_check = 200


his is a standard VOC data(sub)set used for training semantic segmentat
in_data = VOCSegmentation(
 DATASET_PATH,
 image_set = 'train',
 transforms = train_joint_transform


his is a standard VOC data(sub)set used for validating semantic segment
_data = VOCSegmentation(
 DATASET_PATH,
 image_set='val',
 transforms = val_joint_transform


ncrease TRAIN_BATCH_SIZE if you are using GPU to speed up training.
hen batch size changes, the learning rate may also need to be adjusted.
pte that batch size maybe limited by your GPU memory, so adjust if you
IN_BATCH_SIZE = 4

f you are NOT using Windows, set NUM_WORKERS to anything you want, e.g.
ut Windows has issues with multi-process dataloaders, so NUM_WORKERS mu
_WORKERS = 0

ity_loader = DataLoader(sanity_data, batch_size=1, num_workers=NUM_WORK
in_loader = DataLoader(train_data, batch_size=TRAIN_BATCH_SIZE, num_wor
_loader = DataLoader(val_data, batch_size=1, num_workers=NUM_WORKERS, s
```

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.

Solution:

import torchvision.models as models

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

```
        self.net = nn.Sequential(
                    nn.Linear(2, 4),
                    nn.ReLU(),
                    nn.Linear(4, 4),
                    nn.ReLU(),
                    nn.Linear(4, 2),
                    nn.Softmax(dim=1),
                )

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

        # Implement me

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

In [102]:
```
import torchvision.models as models

class MyNet(nn.Module):
    def __init__(self, num_classes, criterion=None):
        super(MyNet, self).__init__()
        self.resnet_fst_layer = nn.Sequential(*(list(models.resnet18(p
        self.resnet_other_layers = nn.Sequential(*(list(models.resnet1
        self.conv1 = nn.ConvTranspose2d(64+512,400,5,1)
        self.conv2 = nn.ConvTranspose2d(400,512,3,1)
        self.conv3 = nn.ConvTranspose2d(512,256,3,1)
        self.conv4 = nn.Conv2d(256,num_classes,1,1)
        self.BN1 = nn.BatchNorm2d(256)
        self.BN2 = nn.BatchNorm2d(512)




    def forward(self, inp, gts=None):
        x = inp
        # print(x.size())
        l1 = x = self.resnet_fst_layer(x)
        # print(x.size())
        x = self.resnet_other_layers(x)
        x = F.interpolate(x, l1.size()[-2:])
        x = torch.cat((x, l1), dim=1)


        x = F.relu(self.conv1(x))
        x = F.relu(self.BN2(self.conv2(x)))
        x = F.relu(self.conv3(x))
        x = F.relu(self.conv4(x))
        x = F.interpolate(x, inp.size()[-2:])
        lfinal = x

        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

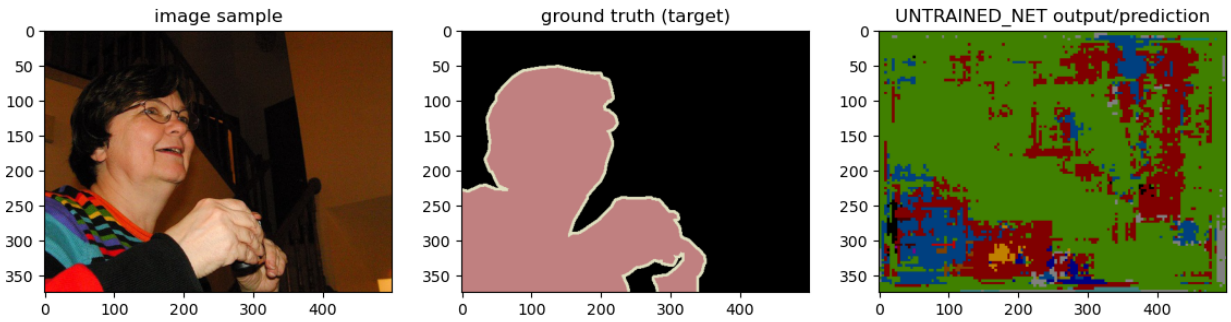
In [103]:
```
untrained_net = MyNet(21).eval()
sample_img, sample_target = Normalize(*norm)(*JointToTensor()(*sample1
untrained_output = untrained_net.forward(sample_img[None])
```

In [104]:
```
sample_img.shape
```

Out[104]:
```
torch.Size([3, 375, 500])
```

In [105]:
```


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()[
```

Out[105]:
```
<matplotlib.image.AxesImage at 0x15d18c490>
```



Problem 3 (bonus for undergrads, required for grads)

(a) Implement the loss function (Cross Entropy Loss) as detailed below. For debugging, part (b) below allows to compare the values w.r.t. the standard function "nn.CrossEntropyLoss". If your loss function works, uncomment the use of "MyCrossEntropyLossy" replacing "nn.CrossEntropyLoss" everywhere below in the notebook (just a couple of places).

You should return the mean (average over all batch pixels) negative log of soft-max for the ground truth class. Assuming that
X_p^m are logits at pixel p for C classes, so that $m \in \{0, 1, \ldots, C-1\}$, each point p contributes to this loss

$$-\log \frac{e^{X_p^{y_p}}}{\sum_m e^{X_p^m}} \quad = \quad -\sum_k Y_p^k \log \frac{e^{X_p^k}}{\sum_m e^{X_p^m}} \qquad (*)$$

where y_p is the ground truth label and Y_p^k is the corresponding one-hot distribution. You should implement the basic math, as in one of the equations above.

NOTE 1: Numerically robust implementation of soft-max is not immediately straightforward. Indeed, logits X_p^k are arbitrary numbers and their exponents could be astronomically large. While the log cancels out the exponent in the enumerator (do it mathematically, not numerically), the sum of exponents in the denominator can not be easily simplified and this is where the numerical issues hide. A naive implementation adding huge values of logits' exponents easily loses precision. Instead, one should use the right hand side in the following algebraically equivalent formulation:

$$\log(e^A + e^B + \cdots + e^Z) \quad \equiv \quad \mu + \log(e^{A-\mu} + e^{B-\mu} + \cdots + e^{Z-\mu})$$

where $\mu := \max\{A, B, \ldots, Z\}$ The right hand side is numerically stable since the arguments of the exponents are negative and the exponents are all bounded by value 1.

HINT 1: Similarly to many previous assignemnts, avoid for-loops. In fact, you should not use none below. Instead, use standard opertors or functions for tensors/matrices (e.g. pointwise addition, multiplication, etc.) In particular, you will find useful functions like "sum" and "max" that could be applied along any specified dimension of the tensor. Of course, torch also has pointwise functions "log" and "exp" that could be applied to any tensor.

HINT 2: Be careful - you will have to work with tensors of different shapes, e.g. even input tensors "targets" and "logits" have different shapes. As in previous assignments based on "numpy", you should pay attention to tensors' dimensions in pyTorch. For example, pointwise addition or multiplication requires either the same shape or shapes "brodcastable" to the same shape (similar in "pyTorch" and "numpy", e.g. see here (https://numpy.org/doc/stable/user/theory.broadcasting.html#array-broadcasting-in-numpy)). If in doubt, use

$$print(xxx.size())$$

to check the shape - I always do! If needed, modify the shapes using standard functions: "transpose" to change the order of the dimentions, "reshape", "flatten", or "squeeze" to remove dimensions, "unsqueeze" to add dimentions. You can use other standard ways to modify the shape, or rely on broadcasting.

HINT 3: Your loss should be averaged only over pixels that have non-void labels. That is, exclude pixels with "ignore index" labels. For example, you can compute a "mask" of non-void pixels, and use it to trim non-void pixels in both the "targets" and "logits". You might get an inspiration from the "mask" in your K-means implementation.

HINT 4: For simplicity, you may "flatten" all tensors' dimensions corresponding to batches and image width & height. The loss computation does not depend on any information in these dimentions and all (non-void) pixels contribute independently based on their logits and ground truth labels (as in the formula above).

HINT 5: In case you want to use the right-hand-size in $(*)$, you can use the function "torch.nn.functional.one_hot" to convert a tensor of "target" class labels (integers) to the tesnor of one-hot distributions. Note that this adds one more dimension to the tensor. In case you want to implement the left-hand-side of $(*)$, you can use the function "torch.gather" to select the ground truth class logits $X_p^{y_p}$.

HINT 6: Just as some guidence, a good solution should be around ten lines of "basic" code, or less.

In [106]:
```
# Student Answer:

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

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

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

In [107]:
```
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(3.0495, grad_fn=<NllLoss2DBackward0>)
tensor(3.0495, grad_fn=<NllLoss2DBackward0>)
```

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 [108]:
```
# Write code to propely compute 'pred' and 'gts' as arguments for func

pred = [torch.argmax(untrained_output, dim=1)[0].numpy()]
gts = [torch.where(sample_target == 255, -1, sample_target).numpy()]


conf = eval_semantic_segmentation(pred, gts)

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

```
mIoU for the sample image / ground truth pair: 9.375e-05
```

(b) Write the validation loop.

In [124]:
```
def validate(val_loader, net):

    iou_arr = []
    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 += criterion(output, masks).sum().item()

            pred = torch.argmax(output, dim=1).numpy()
            gts = torch.where(masks == 255, -1, masks).numpy()



            # Hint: make sure the range of values of the ground truth

            conf = eval_semantic_segmentation(pred, gts)

            iou_arr.append(conf['miou'])

    return val_loss / len(iou_arr), np.mean(iou_arr)
```

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

In [125]:
```
sanity_loader = DataLoader(sanity_data, batch_size=1, num_workers=NUM_
train_loader = DataLoader(train_data, batch_size=TRAIN_BATCH_SIZE, num
val_loader = DataLoader(val_data, batch_size=1, num_workers=NUM_WORKER
```

Out[125]:
```
5.59351152662911e-05
```

In [126]:
```
sanity_loader
```

Out[126]:
```
<torch.utils.data.dataloader.DataLoader at 0x15d206f70>
```

In [127]:
```
val_loader
```

Out[127]:
```
<torch.utils.data.dataloader.DataLoader at 0x15d074bb0>
```

In [128]:
```
%%time
print("mIoU over the sanity dataset:{}".format(validate(sanity_loader
```

```
mIoU over the sanity dataset:6.727359952357058e-05
CPU times: user 793 ms, sys: 247 ms, total: 1.04 s
Wall time: 540 ms
```

Problem 5

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

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

In [129]:
```
from torch import optim
def get_optimizer(net):
    # Write me
    return optim.SGD(net.parameters(), lr=0.006, momentum=0.9)
```

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

In [130]:
```
def train(train_loader, net, optimizer, loss_graph):
    main_loss = 0
    for i, data in enumerate(train_loader):

        inputs, masks = data

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

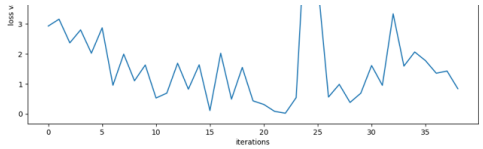
Single image training is helpful for debugging and hyper-parameter tuning (e.g. learning rate, etc.) as it is fast even on a single CPU. In particular, you can work with a single image until your loss function is consistently decreasing during training loop and the network starts producing a reasonable 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 [116]:
```
from torch import optim
def get_optimizer(net):
    # Write me
```

In [76]:
```
```

```
%%time
%matplotlib notebook

# The whole training on a single image (20-40 epochs) should take only
# Below we create a (deep) copy of untrained_net and train it on a sing
# Later, we will create a separate (deep) copy of untrained_net to be t
# NOTE: Normally, one can create a new net via declaration new_net = My
# are declared that way creates *different* untrained nets. This notebo
# For this comparison to be direct and fair, it is better to train (dee
overfit_net = copy.deepcopy(untrained_net)

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



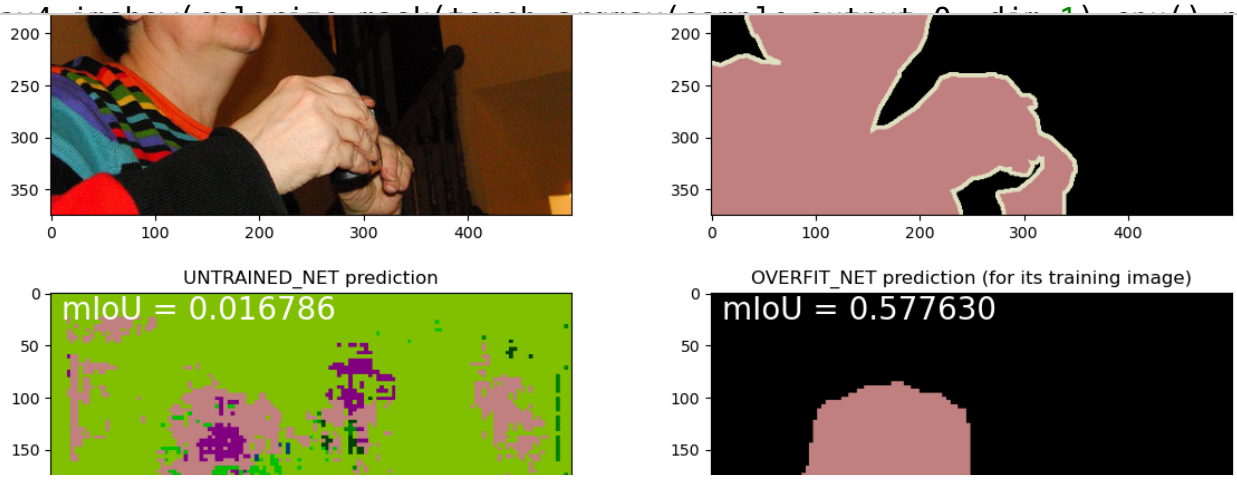

```
Epoch: 0 Loss: 2.932879686355591
Epoch: 1 Loss: 3.161799669265747
Epoch: 2 Loss: 2.3720486164093018
Epoch: 3 Loss: 2.8044686317443848
Epoch: 4 Loss: 2.0266449451446533
Epoch: 5 Loss: 2.873180627822876
Epoch: 6 Loss: 0.9573882818222046
Epoch: 7 Loss: 1.9959110021591187
Epoch: 8 Loss: 1.1064273118972778
Epoch: 9 Loss: 1.6312289237976074
Epoch: 10 Loss: 0.5308806300163269
Epoch: 11 Loss: 0.6939668655395508
Epoch: 12 Loss: 1.6909868717193604
Epoch: 13 Loss: 0.8248165249824524
Epoch: 14 Loss: 1.6365500688552856
Epoch: 15 Loss: 0.1126379668712616
Epoch: 16 Loss: 2.024524450302124
Epoch: 17 Loss: 0.49278029799461365
Epoch: 18 Loss: 1.5506278276443481
Epoch: 19 Loss: 0.4351884424686432
Epoch: 20 Loss: 0.31585443019866943
Epoch: 21 Loss: 0.08498293161392212
Epoch: 22 Loss: 0.022573504596948624
Epoch: 23 Loss: 0.5425437688827515
Epoch: 24 Loss: 7.097770690917969
Epoch: 25 Loss: 4.38802433013916
Epoch: 26 Loss: 0.5628655552864075
Epoch: 27 Loss: 0.9860445261001587
Epoch: 28 Loss: 0.38245636224746704
Epoch: 29 Loss: 0.6915167570114136
Epoch: 30 Loss: 1.6142913103103638
Epoch: 31 Loss: 0.9541700482368469
Epoch: 32 Loss: 3.3415770530700684
Epoch: 33 Loss: 1.5963431596755981
Epoch: 34 Loss: 2.067178249359131
Epoch: 35 Loss: 1.7778452634811401
```

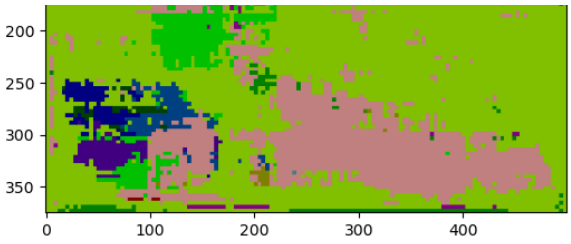
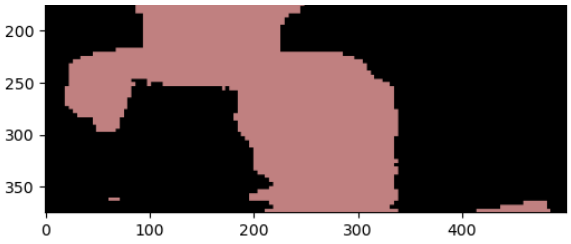
```
Epoch: 36 Loss: 1.357385516166687
Epoch: 37 Loss: 1.4308255910873413
Epoch: 38 Loss: 0.8413667678833008
Epoch: 39 Loss: 0.8404616713523865
CPU times: user 1min 27s, sys: 23.8 s, total: 1min 51s
Wall time: 58 s
```

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

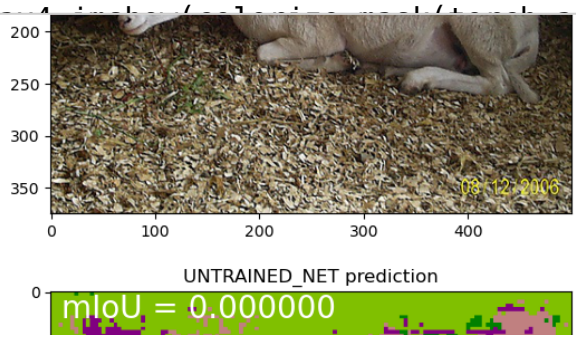
In [77]:

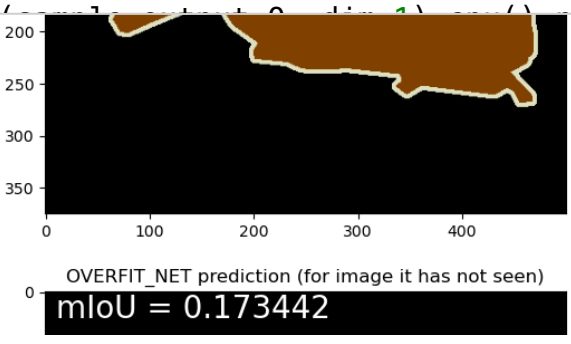
```
# switch back to evaluation mode
overfit_net.eval()

sample_img, sample_target = Normalize(*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 predict
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.int3
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=2
ax3.imshow(colorize_mask(torch.argmax(sample_output_U, dim=1).cpu().nu
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=2
```

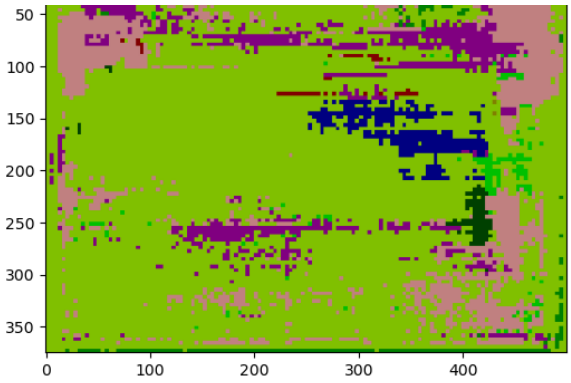


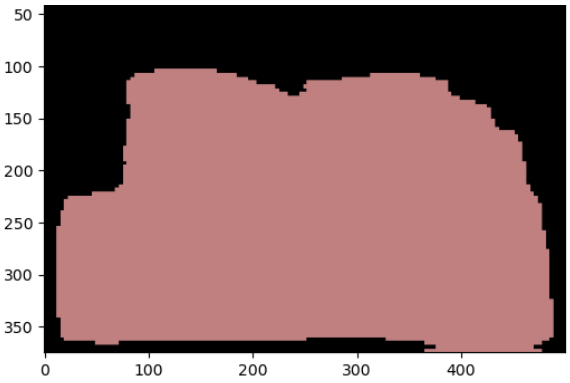



In [78]:
```
sample_img, sample_target = Normalize(*norm)(*JointToTensor()(*sample2
if USE_GPU:
    sample_img = sample_img.cuda()
sample_output_O = overfit_net.forward(sample_img[None])
sample_output_U = untrained_net.forward(sample_img[None])

# computing mIOU (quantitative measure of accuracy for network predict
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.int3
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=2
ax3.imshow(colorize_mask(torch.argmax(sample_output_U, dim=1).cpu().nu
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=2
```









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

In [118]:
```
%%time
print('IoU for OVERFIT_NET over its training image:{}'.format(validat
```

```
mIoU for OVERFIT_NET over its training image:0.7700754988910916
CPU times: user 832 ms, sys: 238 ms, total: 1.07 s
Wall time: 624 ms
```

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

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

(d) Evaluate UNTRAINED_NET and OVERFIT_NET on validation dataset.

Run the validation loop for UNTRAINED_NET against the validation dataset:

In [119]:
```
val_loader
```

Out[119]: <torch.utils.data.dataloader.DataLoader at 0x15cf61220>

In [120]:
```
for i, data in enumerate(val_loader):
    print(i)
    print(data)
```

```
          [-0.5826, -0.5826, -0.6001,  ..., -0.2150, -0.2325, -0.2325
]],

         [[ 0.9319,  0.9319,  0.9319,  ...,  0.9668,  0.9842,  0.9842
],
          [ 0.9319,  0.9319,  0.9319,  ...,  0.8971,  0.9145,  0.9145
],
          [ 0.9319,  0.9319,  0.9319,  ...,  0.7751,  0.7925,  0.7925
],
          ...,
          [-0.6890, -0.6890, -0.7064,  ..., -0.0092, -0.0790, -0.0964
],
          [-0.6715, -0.6715, -0.6890,  ...,  0.1128,  0.0779,  0.0605
],
          [-0.6715, -0.6715, -0.6890,  ...,  0.1999,  0.1825,  0.1651
]]]]), tensor([[[255, 255, 255,  ...,   0,   0,   0],
         [255, 255, 255,  ...,   0,   0,   0],
         [255, 255, 255,  ...,   0,   0,   0],
         ...,
         [  0,   0,   0,  ...,   5,   5,   5],
```

In [*]:
```
%%time
# This will be slow on CPU (around 1 hour or more). On GPU it should ta
print("mIoU for UNTRAINED_NET over the entire dataset:{}".format(valida
```

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

In [*]:
```
%%time
# This will be slow on CPU (around 1 hour or more). On GPU it should t
print("mIoU for OVERFIT_NET over the validation dataset:{}".format(val
```

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

Student answer:

in c) The overfit net can only seperate areas of the image with high coulor difference as shown in the old woman. It is not able to seperate her hair and face from the background yet able to draw out the region where her sweater occupies in the image. The untrained net is just random garbage.

While in d) the overtrained net confuses the ligher fur of the lama with the higher intensity lighting on the ground.

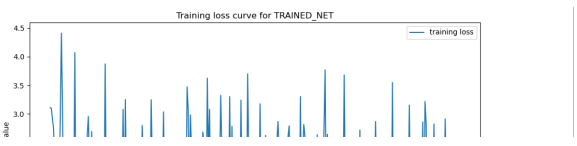
(f) Create TRAINED_NET and train it on the full training dataset:

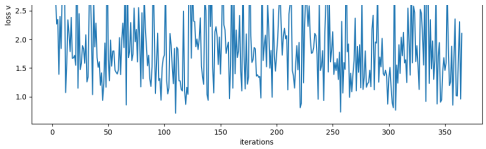
In [71]:
```
# Increase TRAIN_BATCH_SIZE if you are using GPU to speed up training.
# When batch size changes, the learning rate may also need to be adjus
# Note that batch size maybe limited by your GPU memory, so adjust if
TRAIN_BATCH_SIZE = 4

# If you are NOT using Windows, set NUM_WORKERS to anything you want,
# but Windows has issues with multi-process dataloaders, so NUM_WORKER
NUM_WORKERS = 0

sanity_loader = DataLoader(sanity_data, batch_size=1, num_workers=NUM_
train_loader = DataLoader(train_data, batch_size=TRAIN_BATCH_SIZE, num
```

In [72]:

```
%%time
%matplotlib notebook

# This training will be very slow on a CPU (>1hour per epoch). Ideally
# taking only a few minutes per epoch (depending on your GPU and batch
# it is highly advisable that you first finish debugging your net code
# reasonably, e.g. its loss monotonically decreases during training an
# Below we create another (deep) copy of untrained_net. Unlike OVERFIT
trained_net = copy.deepcopy(untrained_net)

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


# You can change the number of EPOCHS below. Since each epoch for TRAI
# the number of required epochs could be smaller compared to OFERFIT_N
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...
```





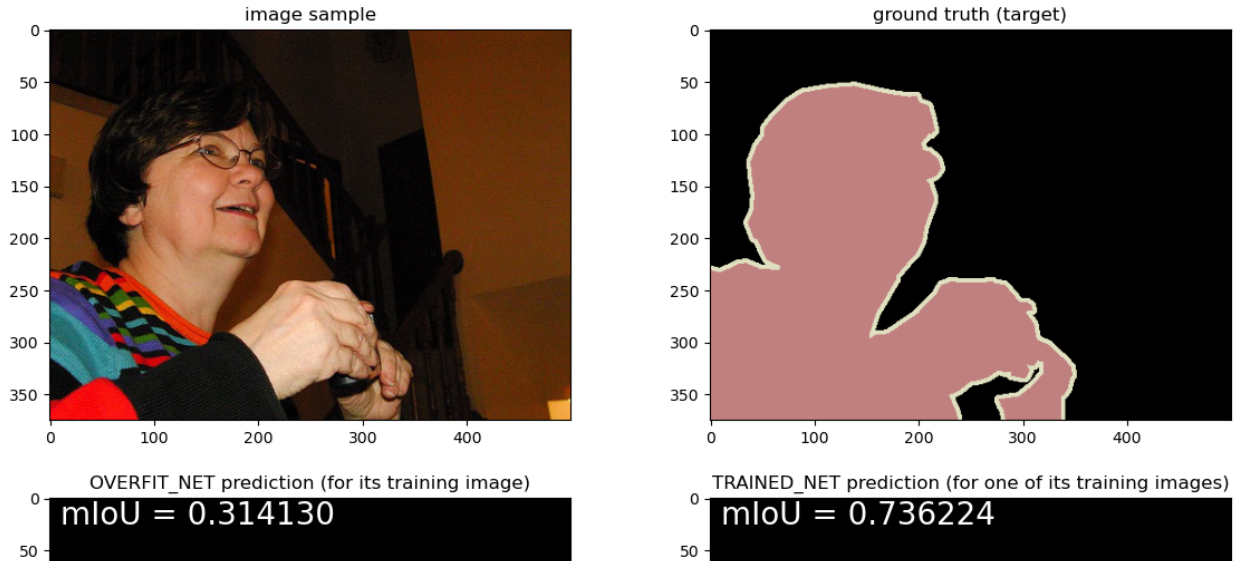

```
Epoch: 0 Loss: 2.103353977203369
Epoch: 1 Loss: 2.137852668762207
CPU times: user 1h 43min 40s, sys: 15min 30s, total: 1h 59min 11s
Wall time: 1h 11min 49s
```

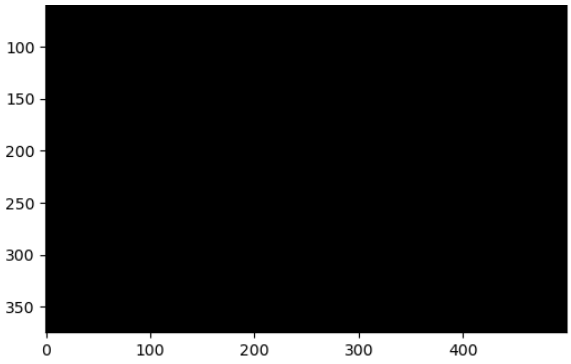
(g) Qualitative and quantitative evaluation of predictions (OVERFIT_NET vs TRAINED_NET):

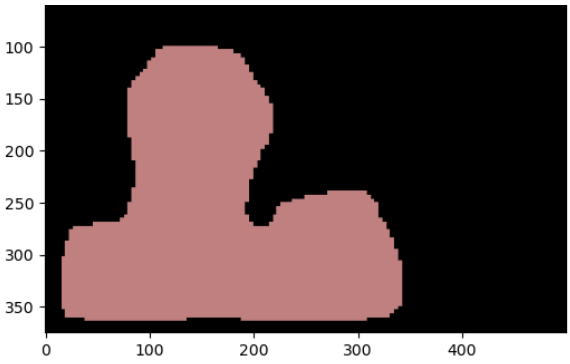
In [73]:

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

sample_img, sample_target = Normalize(*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 predict
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.int3
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=2
ax3.imshow(colorize_mask(torch.argmax(sample_output_O, dim=1).cpu().nu
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=2
```

Out[73]: <matplotlib.image.AxesImage at 0x29bfbd550>

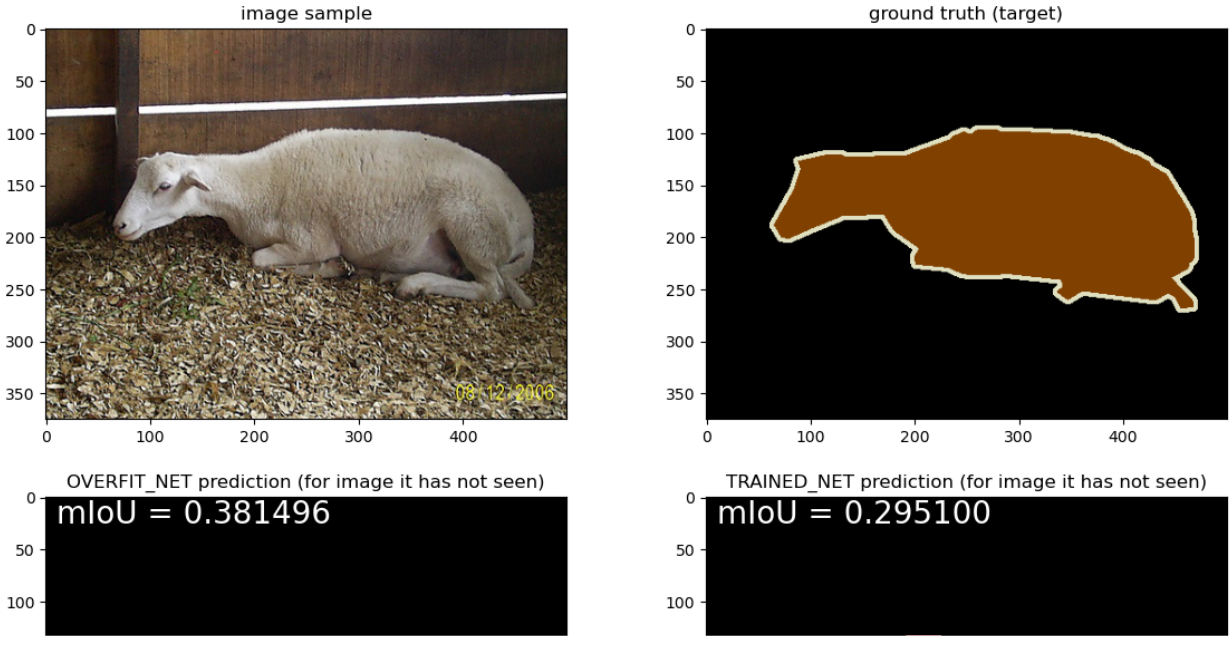



100
150
200
250
300
350
0
100
200
300
400


100
150
200
250
300
350
0
100
200
300
400

In [74]:
```
sample_img, sample_target = Normalize(*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 predict
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.int3
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=2
ax3.imshow(colorize_mask(torch.argmax(sample_output_O, dim=1).cpu().nu
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=2
```

Out[74]: <matplotlib.image.AxesImage at 0x29c138490>



(h) Evaluate TRAINED_NET on validation dataset.

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

In [*]:
```
%%time
# This will be slow on CPU (around 1 hour). On GPU it should take only
print('mIoU for TRAINED_NET over the validation dataset:{}'.format(val
```

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 network I have trained required the images of one consistant size
to be used as the input. It is hard to have the neural net to fit the
finner details of the sheep. Unfortunatelly I do not have time to run
the verification for the whole training and validation datasets so I
can say nothing about the performance limitations of the network.
```

In [ ]:

In [ ]:

In [ ]:

In [ ]: