CS 484/684 Computational Vision

Many thanks for the design of this assignment go to Towaki Takikawa and Olga Veksler

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/

The following two parts of problem 0 are very simple excersices mostly encouraging you to review the slides on linear classification and basic terminolgy in topic 10.

Problem 0-a

Consider linear soft-max classifier $\bar{\sigma}(WX)$ for two classes $K=2$ (.e.g. see 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). Show that soft-max classifier $\bar{\sigma}(WX)$ is equivalent to the sigmoid classifier $\sigma((W_2 - W_1)X)$, e.g. see slide 48 in topic 10. Also, show the loss function for this sigmoid classifier that is equivalent to the cross-entropy loss $-\ln \bar{\sigma}^y(WX)$ where the ground truth label y represents either class 1 or 2 and $\bar{\sigma}^y$ is the corresponding component of the soft-max $\bar{\sigma} = (\bar{\sigma}^1, \bar{\sigma}^2)$.

Your Solution:

Type it here, use latex for math formulas.

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.

Your Solution:

Type it here, use latex for math formulas.

Programming part

In [1]:
```
%matplotlib inline

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

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

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

# We import some of the main PyTorch and TorchVision libraries used for HW4.
# Detailed installation instructions are here: https://pytorch.org/get-start
# That web site should help you to select the right 'conda install' command
# In particular, select the right version of CUDA. Note that prior to instal
# install the latest driver for your GPU and CUDA (9.2 or 10.1), assuming yo
# For more information about pytorch refer to
# https://pytorch.org/docs/stable/nn.functional.html
# https://pytorch.org/docs/stable/data.html.
# and https://pytorch.org/docs/stable/torchvision/transforms.html
import torch
import torch.nn.functional as F
from torch import nn
from torch.utils.data import DataLoader
import torchvision.transforms as transforms
import torchvision.transforms.functional as tF

# We provide our own implementation of torchvision.datasets.voc (containing
# that allows us to easily create single-image datasets
from lib.voc import VOCSegmentation

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

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

In [4]:
```
# This colorize_mask class takes in a numpy segmentation mask,
#  and then converts it to a PIL Image for visualization.
#  Since by default the numpy matrix contains integers from
#  0,1,...,num_classes, we need to apply some color to this
#  so we can visualize easier! Refer to:
#  https://pillow.readthedocs.io/en/4.1.x/reference/Image.html#PIL.Image.Ima
palette = [0, 0, 0, 128, 0, 0, 0, 128, 0, 128, 128, 0, 0, 0, 128, 128, 0, 12
           128, 128, 128, 64, 0, 0, 192, 0, 0, 64, 128, 0, 192, 128, 0, 64,
           64, 128, 128, 192, 128, 128, 0, 64, 0, 128, 64, 0, 0, 192, 0, 128

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

    return new_mask
```

In [5]:
```
# Below we will use a sample image-target pair from VOC training dataset to
# Running this block will automatically download the PASCAL VOC Dataset (3.7
# The code below creates subdirectory "datasets" in the same location as the
# you can modify DATASET_PATH to download the dataset to any custom director
# On subsequent runs you may save time by setting "download = False" (the de

DATASET_PATH = 'datasets'

# Here, we obtain and visualize one sample (img, target) pair from VOC train
# Note that operator [...] extracts the sample corresponding to the specifie
# Also, note the parameter download = True. Set this to False after you down
sample1 = VOCSegmentation(DATASET_PATH, image_set='train', download = False)
sample2 = VOCSegmentation(DATASET_PATH, image_set='val')[20]

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

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

Out[5]:
```
<matplotlib.image.AxesImage at 0x7f80607ddac0>
```


sample1 - image
sample1 - target
sample2 - image
sample2 - target
08/12/2006

Problem 1

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

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

1. The image data should is in certain required format (i.e. consistent spacial resolution to batch). The images should also be normalized and converted to the "tensor" data format expected by pytorch libraries.

2. Some transforms are used to perform randomized image domain transformations with the purpose of "data augmentation".

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

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

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

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

Example 1

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

Solution:

In [6]:
```
class JointToTensor(object):
    def __call__(self, img, target):
        return tF.to_tensor(img), torch.from_numpy(np.array(target.convert('
```

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

JointToTensor()(*sample1)
```

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

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

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

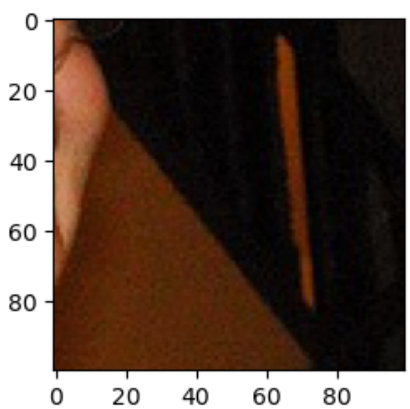
Example 2:

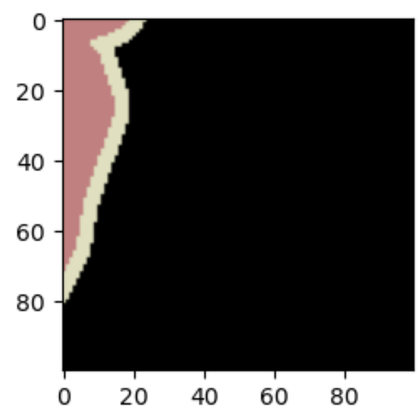
This class implements CenterCrop that takes an img, target pair, and then apply a crop about the center of the image such that the output resolution is $\text{size} \times \text{size}$.

Solution:

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

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

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

Out[8]: <matplotlib.image.AxesImage at 0x7f808146d610>





(a) Implement RandomFlip

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

Solution:

In [9]:
```
class JointRandomFlip(object):
    def __call__(self, img, target):
        if torch.rand(1) < 0.5:
            new_img = tF.hflip(img)
            new_target = tF.hflip(target)
        else:
            new_img = img
            new_target = target
        return new_img, new_target
```

(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 [10]:
```
class JointRandomResizeCrop(object):
    def __init__(self, size, minimum_scale, maximum_scale):
        self.size = size
        self.min = minimum_scale
        self.max = maximum_scale
    def __call__(self, img, target):
        n = random.uniform(self.min,self.max)
        w = int(img.size[0]*n)
        h = int(img.size[1]*n)
        new_img = tF.resize(img, (h,w))
        new_target = tF.resize(target, (h,w))

        crop = min(self.size, h, w)
        top = random.randint(0,h - crop)
        left = random.randint(0,w - crop)
        new_img = tF.crop(new_img, top, left, crop, crop)
        new_target = tF.crop(new_target, top, left, crop, crop)

        new_img = tF.resize(new_img, (self.size, self.size))
        new_tartget = tF.resize(new_target, (self.size, self.size))

        return new_img, new_target
```

(c) Implement Normalize

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

Solution:

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

class JointNormalize(object):
    def __init__(self, mean=norm[0], var=norm[1]):
        self.mean = mean
        self.var = var

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

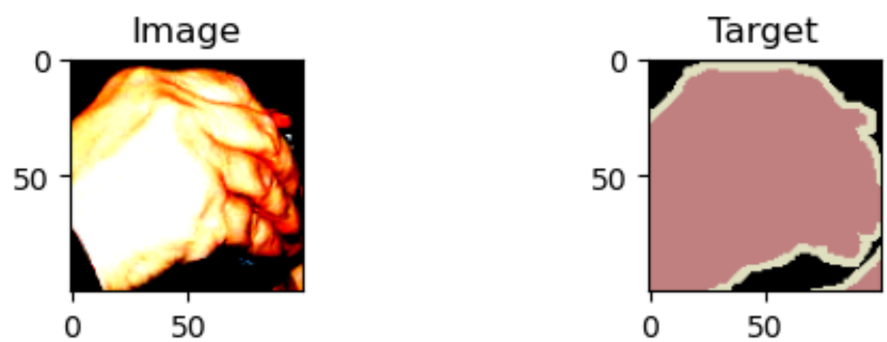
(d) Compose the transforms together:

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

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

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

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

In [13]:
```
# Student Answer:
RandomFlip = JointRandomFlip()
RandomResizeCrop = JointRandomResizeCrop(size = 100,minimum_scale= 0.5 ,maxi
Normalize = JointNormalize()

trans_list = [RandomFlip, RandomResizeCrop, Normalize]

img, target = sample1

composed_trans = JointCompose(trans_list)
img, target = composed_trans(img, target)

img_arr = img.numpy()
img_arr = img_arr.transpose(1,2,0)

plt.subplot(3,2,1)
plt.imshow(img_arr)
plt.title("Image")
plt.subplot(3,2,2)
plt.imshow(target)
plt.title("Target")
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```



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

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

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

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

In [14]:

```
# Student Answer:

# sanity_joint_transform =

# train_joint_transform =

# val_joint_transform =
```

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

In [15]:

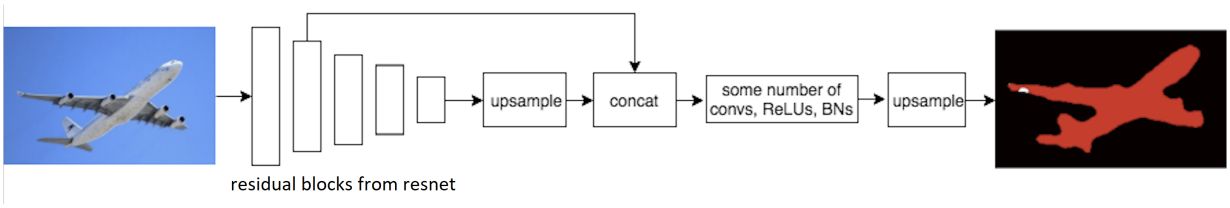
```
# Apply the Joint-Compose transformations above to create three datasets and
# This cell is fully implemented.

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

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

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

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

sanity_loader = DataLoader(sanity_data, batch_size=1, num_workers=NUM_WORKER
train_loader = DataLoader(train_data, batch_size=TRAIN_BATCH_SIZE, num_worke
val_loader = DataLoader(val_data, batch_size=1, num_workers=NUM_WORKERS, shu
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/var/folders/78/631jrq_56k5fcy3sfxg5zpl00000gn/T/ipykernel_41843/2590841218.
py in <module>
      8     DATASET_PATH,
      9     image_set = 'train',
---> 10     transforms = sanity_joint_transform,
     11     sanity_check = 200
     12 )

NameError: name 'sanity_joint_transform' is not defined
```

Problem 2

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

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



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

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

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

Solution:

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

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

        # Implement me

    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
```

(b) Create UNTRAINED_NET and run on a sample image

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

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

# NOTE: prediction uses argMax for the class logits produced by the last lay
```

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

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

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). 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 guidance, a good solution should be around ten lines of "basic" code, or less.

In [ ]:
```
# Student Answer:
class MyCrossEntropyLoss(object):

    def __init__(self, ignore_index=255):
        self.ignore_index = ignore_index

    def __call__(self, logits, targets):

        # N - batch size, C - number of classes (excluding void), HxW - imag
        N, C, H, W = logits.size()

        # print(logits.size())
        # print(targets.size())

        return 0
```

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

In [ ]:
```
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]))
```

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

# pred =
# gts =

conf = eval_semantic_segmentation(pred, gts)

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

(b) Write the validation loop.

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

    iou_arr = []

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

            inputs, masks = data

            if USE_GPU:
                # Write me
            else:
                # Write me

            # Write me

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

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

Problem 5

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

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

In [ ]:
```
def get_optimizer(net):
    # Write me
    return optimizer
```

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

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

    for i, data in enumerate(train_loader):

        inputs, masks = data

        if USE_GPU:
            # Write me

        # Write me

        # loss_graph.append() Populate this list to graph the loss

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

# The whole training on a single image (20-40 epochs) should take only a min
# Below we create a (deep) copy of untrained_net and train it on a single tr
# Later, we will create a separate (deep) copy of untrained_net to be traine
# NOTE: Normally, one can create a new net via declaration new_net = MyNet(2
# are declared that way creates *different* untrained nets. This notebook co
# For this comparison to be direct and fair, it is better to train (deep) co
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
```

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

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

sample_img, sample_target = JointNormalize(*norm)(*JointToTensor()(*sample1)
if USE_GPU:
    sample_img = sample_img.cuda()
sample_output_O = overfit_net.forward(sample_img[None])
sample_output_U = untrained_net.forward(sample_img[None])

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

gts = torch.from_numpy(np.array(sample1[1].convert('P'), dtype=np.int32)).lc
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, col
ax3.imshow(colorize_mask(torch.argmax(sample_output_U, dim=1).cpu().numpy()[
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, col
ax4.imshow(colorize_mask(torch.argmax(sample_output_O, dim=1).cpu().numpy()[
```

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

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

gts = torch.from_numpy(np.array(sample2[1].convert('P'), dtype=np.int32)).lo
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, col
ax3.imshow(colorize_mask(torch.argmax(sample_output_U, dim=1).cpu().numpy()[
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, col
ax4.imshow(colorize_mask(torch.argmax(sample_output_O, dim=1).cpu().numpy()[
```

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

In [ ]:
```
%%time
print("mIoU for OVERFIT_NET over its training image:{}".format(validate(sani
```

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

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

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

Student answer:

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

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

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

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

In [ ]:

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

sample_img, sample_target = JointNormalize(*norm)(*JointToTensor()(*sample1)
if USE_GPU:
    sample_img = sample_img.cuda()
sample_output_O = overfit_net.forward(sample_img[None])
sample_output_T = trained_net.forward(sample_img[None])

# computing mIOU (quantitative measure of accuracy for network predictions)
pred_T = torch.argmax(sample_output_T, dim=1).cpu().numpy()[0]
pred_O = torch.argmax(sample_output_O, dim=1).cpu().numpy()[0]
gts = torch.from_numpy(np.array(sample1[1].convert('P'), dtype=np.int32)).lc
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, col
ax3.imshow(colorize_mask(torch.argmax(sample_output_O, dim=1).cpu().numpy()[
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, col
ax4.imshow(colorize_mask(torch.argmax(sample_output_T, dim=1).cpu().numpy()[
```

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

# computing mIOU (quantitative measure of accuracy for network predictions)
pred_O = torch.argmax(sample_output_O, dim=1).cpu().numpy()[0]
pred_T = torch.argmax(sample_output_T, dim=1).cpu().numpy()[0]
gts = torch.from_numpy(np.array(sample2[1].convert('P'), dtype=np.int32)).lo
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, col
ax3.imshow(colorize_mask(torch.argmax(sample_output_O, dim=1).cpu().numpy()[
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, col
ax4.imshow(colorize_mask(torch.argmax(sample_output_T, dim=1).cpu().numpy()[
```

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

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?

In [ ]:
```
```