Random Erasing

Random Erasing is a new data augmentation method for training the convolutional neural network (CNN). In training, Random Erasing randomly selects a rectangle region in an image and erases its pixels with random values. In this process, training images with various levels of occlusion are generated, which reduces the risk of over-fitting and makes the model robust to occlusion. Random Erasing is parameter learning free, easy to implement, and can be integrated with most of the CNN-based recognition models. Albeit simple, Random Erasing is complementary to commonly used data augmentation techniques such as random cropping and flipping, and yields consistent improvement over strong baselines in image classification, object detection and person re-identification.

So, will perform step by step process and achieve Random Erasing from a well known Potsdam dataset.

Dataset link:

Let's hop-in,

Step 1: We will first sign in with google to use it's most used data science tool which is Google Colab.

We are using google Colab to use it's GPU power.

Step 2: Once we get the Colab on hands, we first mount the drive to use the data present in the drive. Below is the snapshot for the same.

```
[ ] from google.colab import drive
    drive.mount('/content/drive')

    Mounted at /content/drive
```

Step 3: Now we are going to download the dataset in the drive itself. Below is the snapshot for the same. Here in the below snippet, we are downloading data from the source and unzip to the specific directory.

```
[ ] !wget -P /content/drive/MyDrive/ https://seafile.projekt.uni-hannover.de/seafhttp/files/5454274d-aeb7-47c7-ba7c-db9c20b07801/Potsdam.zip

[ ] !unzip /content/drive/MyDrive/J/Potsdam/2_Ortho_RGB.zip -d /content/drive/MyDrive/J/Potsdam/

[ ] !unzip /content/drive/MyDrive/J/Potsdam/5_Labels_for_participants.zip -d /content/drive/MyDrive/J/Potsdam/5_Labels_for_participants/

[ ] !unzip /content/drive/MyDrive/J/Potsdam/5_Labels_for_participants_no_Boundary.zip -d /content/drive/MyDrive/J/Potsdam/5_Labels_for_participants_no_Boundary/
```

Step 4: After getting dataset, will do necessary package downloads and then import to the colab for furhter processing. Below is the snapshot for the same.

```python
# imports and stuff
import numpy as np
from skimage import io
from glob import glob
from tqdm import tqdm_notebook as tqdm
from sklearn.metrics import confusion_matrix
import random
import itertools
# Matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
# Torch imports
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.data as data
import torch.optim as optim
import torch.optim.lr_scheduler
import torch.nn.init
from torch.autograd import Variable
```

Step 5: Now here, we will use the random erasing feature form the Pytorch itself for our potsdam dataset. So, basically we will develop other dataset of Random Erasing and will use the same for the further training. So we are generating and storing the dataset at specific location.

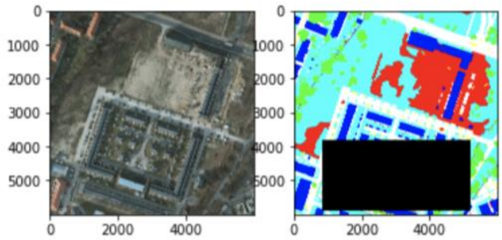
```python
import torch
import torchvision.transforms as T
from PIL import Image
import matplotlib.pyplot as plt
import os

NEW_FOLDER = '../content/drive/MyDrive/J/Potsdam/5_Labels_for_participants/5_Labels_for_participants/'

# all_files = sorted(glob(NEW_FOLDER.replace('{}', '*')))
for im in os.listdir(NEW_FOLDER):
  if (im.endswith(".tif")):
    img = io.imread('../content/drive/MyDrive/J/Potsdam/5_Labels_for_participants/5_Labels_for_participants/'+ im)

    # define a transform to perform transformations
    transform = T.Compose([T.ToTensor(), T.RandomErasing(p=0.5, scale=(0.02, 0.33), ratio=(0.3, 3.3), value=0, inplace=False), T.ToPILImage()])
    imgs = transform(img)

    plt.imshow(imgs)
    plt.show()
    Image=imgs
    # print(imgs)
    Image.save('../content/drive/MyDrive/J/Potsdam/5_Labels_for_participants/REdata/'+ im)
```

Step 6: Now, here we will visualize the random erasing dataset created recently. Below is the snapshot for the same. Here,
First, let's check that we are able to access the dataset and see what's going on. We are using scikit-image for image manipulation. As the ISPRS dataset is stored with a ground truth in the RGB format, we need to define the color palette that can map the label id to its RGB color. We define two helper functions to convert from numeric to colors and vice-versa.

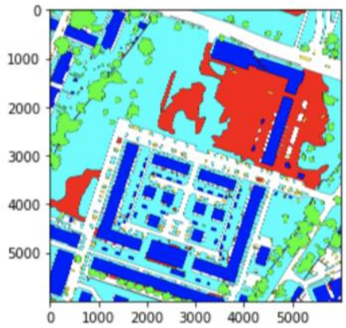
```
WINDOW_SIZE = (256, 256) # Patch size
STRIDE = 32 # Stride for testing
IN_CHANNELS = 3 # Number of input channels (e.g. RGB)
# FOLDER = "../content/drive/MyDrive/Potsdam/PotsdamData/"
BATCH_SIZE = 10 # Number of samples in a mini-batch

LABELS = ["roads", "buildings", "low veg.", "trees", "cars", "clutter"] # Label names
N_CLASSES = len(LABELS) # Number of classes
WEIGHTS = torch.ones(N_CLASSES) # Weights for class balancing
CACHE = True # Store the dataset in-memory

DATASET = 'Potsdam'
# MAIN_FOLDER = FOLDER + 'Potsdam/'

DATA_FOLDER = '../content/drive/MyDrive/J/Potsdam/2_Ortho_RGB/top_potsdam_{}_RGB.tif'
LABEL_FOLDER = '../content/drive/MyDrive/J/Potsdam/5_Labels_for_participants/REdata/top_potsdam_{}_label.tif'
ERODED_FOLDER = '../content/drive/MyDrive/J/Potsdam/5_Labels_for_participants_no_Boundary/5_Labels_for_participants_no_Boundary/top_potsdam_{}_label_noBoundary.tif'
```

```
# ISPRS color palette
# Let's define the standard ISPRS color palette
palette = {0 : (255, 255, 255), # Impervious surfaces (white)
           1 : (0, 0, 255),     # Buildings (blue)
           2 : (0, 255, 255),   # Low vegetation (cyan)
           3 : (0, 255, 0),     # Trees (green)
           4 : (255, 255, 0),   # Cars (yellow)
           5 : (255, 0, 0)}     # Clutter (red)

invert_palette = {v: k for k, v in palette.items()}

def convert_to_color(arr_2d, palette=palette):
    """ Numeric labels to RGB-color encoding """
    arr_3d = np.zeros((arr_2d.shape[0], arr_2d.shape[1], 3), dtype=np.uint8)

    for c, i in palette.items():
        m = arr_2d == c
        arr_3d[m] = i

    return arr_3d
def convert_from_color(arr_3d, palette=invert_palette):
    """ RGB-color encoding to grayscale labels """
    arr_2d = np.zeros((arr_3d.shape[0], arr_3d.shape[1]), dtype=np.uint8)

    for c, i in palette.items():
        m = np.all(arr_3d == np.array(c).reshape(1, 1, 3), axis=2)
        arr_2d[m] = i

    return arr_2d

# We load one tile from the dataset and we display it
img = io.imread('../content/drive/MyDrive/J/Potsdam/2_Ortho_RGB/top_potsdam_2_10_RGB.tif')
fig = plt.figure()
fig.add_subplot(121)
plt.imshow(img)

# We load the ground truth
gt = io.imread('../content/drive/MyDrive/J/Potsdam/5_Labels_for_participants/REdata/top_potsdam_2_10_label.tif')
fig.add_subplot(122)
plt.imshow(gt)
plt.show()

gte = io.imread('../content/drive/MyDrive/J/Potsdam/5_Labels_for_participants_no_Boundary/5_Labels_for_participants_no_Boundary/top_potsdam_2_10_label_noBoundary.tif')
fig.add_subplot(122)
plt.imshow(gte)
plt.show()

# We also check that we can convert the ground truth into an array format
array_gt = convert_from_color(gt)
print("Ground truth in numerical format has shape ({},{}) : \n".format(*array_gt.shape[:2]), array_gt)
```

The above code snippet gives output like below image.



```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:45: MatplotlibDeprecationWarning: Adding an axes
```



```
Ground truth in numerical format has shape (6000,6000) :
 [[2 2 2 ... 2 2 2]
 [2 2 2 ... 2 2 2]
 [2 2 2 ... 2 2 2]
 ...
 [0 0 0 ... 1 1 1]
 [0 0 0 ... 1 1 1]
 [0 0 0 ... 1 1 1]]
```

Step 7: Here, At this point we need to define a bunch of utils function for the smooth processing

```
def get_random_pos(img, window_shape):
    """ Extract of 2D random patch of shape window_shape in the image """
    w, h = window_shape
    W, H = img.shape[-2:]
    x1 = random.randint(0, W - w - 1)
    x2 = x1 + w
    y1 = random.randint(0, H - h - 1)
    y2 = y1 + h
    return x1, x2, y1, y2
```

```python
def CrossEntropy2d(input, target, weight=None, size_average=True):
    """ 2D version of the cross entropy loss """
    dim = input.dim()
    if dim == 2:
        return F.cross_entropy(input, target, weight, size_average)
    elif dim == 4:
        output = input.view(input.size(0),input.size(1), -1)
        output = torch.transpose(output,1,2).contiguous()
        output = output.view(-1,output.size(2))
        target = target.view(-1)
        return F.cross_entropy(output, target,weight, size_average)
    else:
        raise ValueError('Expected 2 or 4 dimensions (got {})'.format(dim))
```

```python
def accuracy(input, target):
    return 100 * float(np.count_nonzero(input == target)) / target.size
```

```python
def sliding_window(top, step=10, window_size=(20,20)):
    """ Slide a window_shape window across the image with a stride of step """
    for x in range(0, top.shape[0], step):
        if x + window_size[0] > top.shape[0]:
            x = top.shape[0] - window_size[0]
        for y in range(0, top.shape[1], step):
            if y + window_size[1] > top.shape[1]:
                y = top.shape[1] - window_size[1]
            yield x, y, window_size[0], window_size[1]
```

```python
def count_sliding_window(top, step=10, window_size=(20,20)):
    """ Count the number of windows in an image """
    c = 0
    for x in range(0, top.shape[0], step):
        if x + window_size[0] > top.shape[0]:
            x = top.shape[0] - window_size[0]
        for y in range(0, top.shape[1], step):
            if y + window_size[1] > top.shape[1]:
                y = top.shape[1] - window_size[1]
            c += 1
    return c
```

```python
def grouper(n, iterable):
    """ Browse an iterator by chunk of n elements """
    it = iter(iterable)
    while True:
        chunk = tuple(itertools.islice(it, n))
        if not chunk:
            return
        yield chunk
```

```python
def metrics(predictions, gts, label_values=LABELS):
  cm = confusion_matrix(
          gts,
          predictions
          # range(len(label_values)))
  )
  print("Confusion matrix :")
  print(cm)

  print("---")

  # Compute global accuracy
  total = sum(sum(cm))
  accuracy = sum([cm[x][x] for x in range(len(cm))])
  accuracy *= 100 / float(total)
  print("{} pixels processed".format(total))
  print("Total accuracy : {}%".format(accuracy))

  print("---")

  # Compute F1 score
  F1Score = np.zeros(len(label_values))
  for i in range(len(label_values)):
      try:
          F1Score[i] = 2. * cm[i,i] / (np.sum(cm[i,:]) + np.sum(cm[:,i]))
      except:
          # Ignore exception if there is no element in class i for test set
          pass
  print("F1Score :")
  for l_id, score in enumerate(F1Score):
      print("{}: {}".format(label_values[l_id], score))


  print("---")

  # Compute kappa coefficient
  total = np.sum(cm)
  pa = np.trace(cm) / float(total)
  pe = np.sum(np.sum(cm, axis=0) * np.sum(cm, axis=1)) / float(total*total)
  kappa = (pa - pe) / (1 - pe);
  print("Kappa: " + str(kappa))
```

Step 8: Now we are loading the dataset, We define a PyTorch dataset (torch.utils.data.Dataset) that loads all the tiles in memory and performs random sampling. Tiles are stored in memory on the fly. The dataset also performs random data augmentation (horizontal and vertical flips) and normalizes the data in [0, 1].

So, we have define a class and within it data augmentation functions like flip, rotate, etc. Below is the snippet for the same.

```
class ISPRS_dataset(torch.utils.data.Dataset):
    def __init__(self, ids, data_files=DATA_FOLDER, label_files=LABEL_FOLDER,
                            cache=False, augmentation=True):
        super(ISPRS_dataset, self).__init__()

        self.augmentation = augmentation
        self.cache = cache

        # List of files
        self.data_files = [DATA_FOLDER.format(id) for id in ids]
        self.label_files = [LABEL_FOLDER.format(id) for id in ids]

        # Sanity check : raise an error if some files do not exist
        for f in self.data_files + self.label_files:
            if not os.path.isfile(f):
                raise KeyError('{} is not a file !'.format(f))

        # Initialize cache dicts
        self.data_cache_ = {}
        self.label_cache_ = {}


    def __len__(self):
        # Default epoch size is 10 000 samples
        return 10000
```

```
    @classmethod
    def data_augmentation(cls, *arrays, flip=True, mirror=True):
        will_flip, will_mirror = False, False
        if flip and random.random() < 0.5:
            will_flip = True
        if mirror and random.random() < 0.5:
            will_mirror = True

        results = []
        for array in arrays:
            if will_flip:
                if len(array.shape) == 2:
                    array = array[::-1, :]
                else:
                    array = array[:, ::-1, :]
            if will_mirror:
                if len(array.shape) == 2:
                    array = array[:, ::-1]
                else:
                    array = array[:, :, ::-1]
            results.append(np.copy(array))

        return tuple(results)

    def __getitem__(self, i):
        # Pick a random image
        random_idx = random.randint(0, len(self.data_files) - 1)

        # If the tile hasn't been loaded yet, put in cache
        if random_idx in self.data_cache_.keys():
            data = self.data_cache_[random_idx]
        else:
            # Data is normalized in [0, 1]
            data = 1/255 * np.asarray(io.imread(self.data_files[random_idx]).transpose((2,0,1)), dtype='float32')
            if self.cache:
                self.data_cache_[random_idx] = data

        if random_idx in self.label_cache_.keys():
            label = self.label_cache_[random_idx]
        else:
            # Labels are converted from RGB to their numeric values
            label = np.asarray(convert_from_color(io.imread(self.label_files[random_idx])), dtype='int64')
            if self.cache:
                self.label_cache_[random_idx] = label

        # Get a random patch
        x1, x2, y1, y2 = get_random_pos(data, WINDOW_SIZE)
        data_p = data[:, x1:x2,y1:y2]
        label_p = label[x1:x2,y1:y2]

        # Data augmentation
        data_p, label_p = self.data_augmentation(data_p, label_p)

        # Return the torch.Tensor values
        return (torch.from_numpy(data_p), torch.from_numpy(label_p))
```

Step 9: Now, the time comes for network definition for CNN, We can now define the Fully Convolutional network based on the SegNet architecture. We could use any other network as drop-in replacement, provided that the output has dimensions (N_CLASSES, W, H) where W and H are the sliding window dimensions (i.e. the network should preserve the spatial dimensions).

```
class SegNet(nn.Module):
    # SegNet network
    @staticmethod
    def weight_init(m):
        if isinstance(m, nn.Linear):
            torch.nn.init.kaiming_normal(m.weight.data)
```

```
    def __init__(self, in_channels=IN_CHANNELS, out_channels=N_CLASSES):
        super(SegNet, self).__init__()
        self.pool = nn.MaxPool2d(2, return_indices=True)
        self.unpool = nn.MaxUnpool2d(2)

        self.conv1_1 = nn.Conv2d(in_channels, 64, 3, padding=1)
        self.conv1_1_bn = nn.BatchNorm2d(64)
        self.conv1_2 = nn.Conv2d(64, 64, 3, padding=1)
    def forward(self, x):
        # Encoder block 1
        x = self.conv1_1_bn(F.relu(self.conv1_1(x)))
        x = self.conv1_2_bn(F.relu(self.conv1_2(x)))
        x, mask1 = self.pool(x)

        # Encoder block 2
        x = self.conv2_1_bn(F.relu(self.conv2_1(x)))
        x = self.conv2_2_bn(F.relu(self.conv2_2(x)))
        x, mask2 = self.pool(x)

        # Encoder block 3
        x = self.conv3_1_bn(F.relu(self.conv3_1(x)))
        x = self.conv3_2_bn(F.relu(self.conv3_2(x)))
        x = self.conv3_3_bn(F.relu(self.conv3_3(x)))
        x, mask3 = self.pool(x)

        # Encoder block 4
        x = self.conv4_1_bn(F.relu(self.conv4_1(x)))
        x = self.conv4_2_bn(F.relu(self.conv4_2(x)))
        x = self.conv4_3_bn(F.relu(self.conv4_3(x)))
        x, mask4 = self.pool(x)

        # Encoder block 5
        x = self.conv5_1_bn(F.relu(self.conv5_1(x)))
        x = self.conv5_2_bn(F.relu(self.conv5_2(x)))
        x = self.conv5_3_bn(F.relu(self.conv5_3(x)))
        x, mask5 = self.pool(x)

        # Decoder block 5
        x = self.unpool(x, mask5)
        x = self.conv5_3_D_bn(F.relu(self.conv5_3_D(x)))
        x = self.conv5_2_D_bn(F.relu(self.conv5_2_D(x)))
        x = self.conv5_1_D_bn(F.relu(self.conv5_1_D(x)))

        # Decoder block 4
        x = self.unpool(x, mask4)
        x = self.conv4_3_D_bn(F.relu(self.conv4_3_D(x)))
        x = self.conv4_2_D_bn(F.relu(self.conv4_2_D(x)))
        x = self.conv4_1_D_bn(F.relu(self.conv4_1_D(x)))

        # Decoder block 3
        x = self.unpool(x, mask3)
        x = self.conv3_3_D_bn(F.relu(self.conv3_3_D(x)))
        x = self.conv3_2_D_bn(F.relu(self.conv3_2_D(x)))
        x = self.conv3_1_D_bn(F.relu(self.conv3_1_D(x)))

        # Decoder block 2
        x = self.unpool(x, mask2)
        x = self.conv2_2_D_bn(F.relu(self.conv2_2_D(x)))
        x = self.conv2_1_D_bn(F.relu(self.conv2_1_D(x)))

        # Decoder block 1
        x = self.unpool(x, mask1)
        x = self.conv1_2_D_bn(F.relu(self.conv1_2_D(x)))
        x = F.log_softmax(self.conv1_1_D(x))
        return x
        self.conv1_2_D_bn = nn.BatchNorm2d(64)
        self.conv1_1_D = nn.Conv2d(64, out_channels, 3, padding=1)

        self.apply(self.weight_init)
```

Step 10: We can now instantiate the network using the specified parameters. By default, the weights will be initialized using the policy.

```
[ ] # instantiate the network
    net = SegNet()
```

Step 11: We download and load the pre-trained weights from VGG-16 on ImageNet. This step is optional but it makes the network converge faster. We skip the weights from VGG-16 that have no counterpart in SegNet.

The below is the output of the above snipet.

```
import os
try:
    from urllib.request import URLopener
except ImportError:
    from urllib import URLopener

# Download VGG-16 weights from PyTorch
vgg_url = 'https://download.pytorch.org/models/vgg16_bn-6c64b313.pth'
if not os.path.isfile('./vgg16_bn-6c64b313.pth'):
    weights = URLopener().retrieve(vgg_url, './vgg16_bn-6c64b313.pth')

vgg16_weights = torch.load('./vgg16_bn-6c64b313.pth')
mapped_weights = {}
for k_vgg, k_segnet in zip(vgg16_weights.keys(), net.state_dict().keys()):
t
e
```

```
Mapping features.11.weight to conv2_1_bn.num_batches_tracked
Mapping features.11.bias to conv2_2.weight
Mapping features.11.running_mean to conv2_2.bias
Mapping features.11.running_var to conv2_2_bn.weight
Mapping features.14.weight to conv2_2_bn.bias
Mapping features.14.bias to conv2_2_bn.running_mean
Mapping features.15.weight to conv2_2_bn.running_var
Mapping features.15.bias to conv2_2_bn.num_batches_tracked
Mapping features.15.running_mean to conv3_1.weight
Mapping features.15.running_var to conv3_1.bias
Mapping features.17.weight to conv3_1_bn.weight
Mapping features.17.bias to conv3_1_bn.bias
Mapping features.18.weight to conv3_1_bn.running_mean
Mapping features.18.bias to conv3_1_bn.running_var
Mapping features.18.running_mean to conv3_1_bn.num_batches_tracked
Mapping features.18.running_var to conv3_2.weight
Mapping features.20.weight to conv3_2.bias
Mapping features.20.bias to conv3_2_bn.weight
Mapping features.21.weight to conv3_2_bn.bias
Mapping features.21.bias to conv3_2_bn.running_mean
Mapping features.21.running_mean to conv3_2_bn.running_var
Mapping features.21.running_var to conv3_2_bn.num_batches_tracked
Mapping features.24.weight to conv3_3.weight
Mapping features.24.bias to conv3_3.bias
Mapping features.25.weight to conv3_3_bn.weight
Mapping features.25.bias to conv3_3_bn.bias
Mapping features.25.running_mean to conv3_3_bn.running_mean
Mapping features.25.running_var to conv3_3_bn.running_var
Mapping features.27.weight to conv3_3_bn.num_batches_tracked
Mapping features.27.bias to conv4_1.weight
Mapping features.28.weight to conv4_1.bias
Mapping features.28.bias to conv4_1_bn.weight
Mapping features.28.running_mean to conv4_1_bn.bias
```

Step 12: Now, we load the network on GPU.

Follow the below link till Step 5,
Link for GPU in the Colab:
https://www.geeksforgeeks.org/how-to-run-cuda-c-c-on-jupyter-notebook-in-google-colaboratory/

And then write below code snippet.

```
os.environ['CUDA_LAUNCH_BLOCKING'] = "1"
```

Step 13: Now below is the code snippet with output for cuda for training the CNN.

```
net.cuda()
```

```
SegNet(
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (unpool): MaxUnpool2d(kernel_size=(2, 2), stride=(2, 2), padding=(0, 0))
  (conv1_1): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv1_1_bn): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv1_2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv1_2_bn): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2_1): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2_1_bn): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2_2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2_2_bn): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3_1): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3_1_bn): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3_2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3_2_bn): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3_3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3_3_bn): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv4_1): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv4_1_bn): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv4_2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv4_2_bn): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv4_3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv4_3_bn): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv5_1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv5_1_bn): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv5_2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv5_2_bn): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv5_3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv5_3_bn): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv5_3_D): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv5_3_D_bn): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv5_2_D): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv5_2_D_bn): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv5_1_D): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv5_1_D_bn): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

Step 14: Check whether gpu is available or not by below snapshot.

```
[ ] torch.cuda.is_available()
```

```
True
```

Step 15: We now create a train/test split. If you want to use another dataset, you have to adjust the method to collect all filenames. In our case, we specify a fixed train/test split for the demo.

```python
# Load the datasets
all_files = sorted(glob(LABEL_FOLDER.replace('{}', '*')))
all_ids = [f.split('potsdam_')[-1].split('_label')[0] for f in all_files]
# Random tile numbers for train/test split
train_ids = random.sample(all_ids,  len(all_ids) )
test_ids = list(set(all_ids) - set(train_ids))

# Example of a train/test split on Potsdam :
train_ids = ['2_10', '2_12', '3_10', '3_12', '4_10', '4_12', '5_10', '5_12', '6_10', '6_12', '7_10', '7_12']
test_ids = ['2_11', '3_11', '4_11', '5_11']

print("Tiles for training : ", train_ids)
print("Tiles for testing : ", test_ids)

train_set = ISPRS_dataset(train_ids, cache=CACHE)
train_loader = torch.utils.data.DataLoader(train_set,batch_size=BATCH_SIZE)
```

Step 16: We are now designing the optimizer. We use the standard Stochastic Gradient Descent algorithm to optimize the network's weights.The encoder is trained at half the learning rate of the decoder, as we rely on the pre-trained VGG-16 weights.

```python
[ ]  base_lr = 0.01
     params_dict = dict(net.named_parameters())
     params = []
     for key, value in params_dict.items():
```

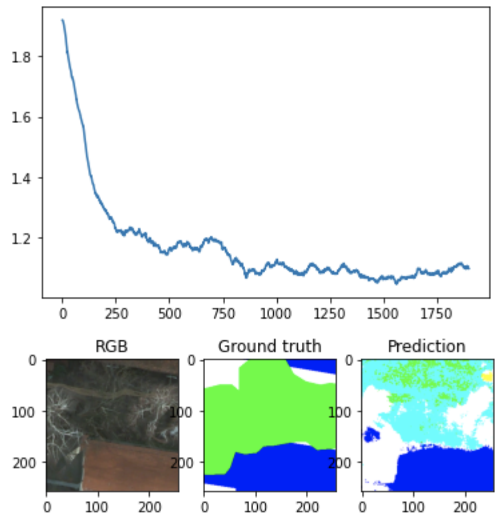
```python
from IPython.display import clear_output
def test(net, test_ids, all=False, stride=WINDOW_SIZE[0], batch_size=BATCH_SIZE, window_size=WINDOW_SIZE):
    # Use the network on the test set
    test_images = (1 / 255 * np.asarray(io.imread(DATA_FOLDER.format(id)), dtype='float32') for id in test_ids)
    test_labels = (np.asarray(io.imread(LABEL_FOLDER.format(id)), dtype='uint8') for id in test_ids)
    eroded_labels = (convert_from_color(io.imread(ERODED_FOLDER.format(id))) for id in test_ids)

    all_preds = []
    all_gts = []

    # Switch the network to inference mode
    net.eval()

    for img, gt, gt_e in tqdm(zip(test_images, test_labels,eroded_labels), total=len(test_ids), leave=False):
        pred = np.zeros(img.shape[:2] + (N_CLASSES,))
        total = count_sliding_window(img, step=stride, window_size=window_size) // batch_size
        for i, coords in enumerate(tqdm(grouper(batch_size, sliding_window(img, step=stride, window_size=window_size)), total=total, leave=False)):
            # Display in progress results
            if i > 0 and total > 10 and i % int(10 * total / 100) == 0:
                    _pred = np.argmax(pred, axis=-1)
                    fig = plt.figure()
                    fig.add_subplot(1,3,1)
                    plt.imshow(np.asarray(255 * img, dtype='uint8'))
                    fig.add_subplot(1,3,2)
                    plt.imshow(convert_to_color(_pred))
                    fig.add_subplot(1,3,3)
                    plt.imshow(gt)
                    clear_output()
                    plt.show()

            # Build the tensor
            image_patches = [np.copy(img[x:x+w, y:y+h]).transpose((2,0,1)) for x,y,w,h in coords]
            image_patches = np.asarray(image_patches)
            image_patches = Variable(torch.from_numpy(image_patches).cuda(), volatile=True)

            # Do the inference
            outs = net(image_patches)
            outs = outs.data.cpu().numpy()

            # Fill in the results array
            for out, (x, y, w, h) in zip(outs, coords):
                out = out.transpose((1,2,0))
                pred[x:x+w, y:y+h] += out
            del(outs)

        pred = np.argmax(pred, axis=-1)

        # Display the result
        clear_output()
        fig = plt.figure()
        fig.add_subplot(1,3,1)
        plt.imshow(np.asarray(255 * img, dtype='uint8'))
        fig.add_subplot(1,3,2)
        plt.imshow(convert_to_color(pred))
        fig.add_subplot(1,3,3)
        plt.imshow(gt)
        plt.show()
        all_preds.append(pred)
        all_gts.append(gt_e)
        clear_output()
        # Compute some metrics
        metrics(pred.ravel(), gt_e.ravel())
        accuracy = metrics(np.concatenate([p.ravel() for p in all_preds]), np.concatenate([p.ravel() for p in all_gts]).ravel())
    if all:
        return accuracy, all_preds, all_gts
    else:
        return accuracy
```

```python
from IPython.display import clear_output

def train(net, optimizer, epochs, scheduler=None, weights=WEIGHTS, save_epoch = 5):
    losses = np.zeros(1000000)
    mean_losses = np.zeros(100000000)
    weights = weights.cuda()

    # criterion = nn.NLLLoss2d(weight=weights)
    criterion=nn.BCEWithLogitsLoss(weight=weights)
    iter_ = 0

    for e in range(1, epochs + 1):
        if scheduler is not None:
            scheduler.step()
        net.train()
        for batch_idx, (data, target) in enumerate(train_loader):
            data, target = Variable(data.cuda()), Variable(target.cuda())
            optimizer.zero_grad()
            output = net(data)
            loss = CrossEntropy2d(output, target, weight=weights)
            loss.backward()
            optimizer.step()

            losses[iter_] = loss.item()
            mean_losses[iter_] = np.mean(losses[max(0,iter_-100):iter_])

            if iter_ % 100 == 0:
                clear_output()
                rgb = np.asarray(255 * np.transpose(data.data.cpu().numpy()[0],(1,2,0)), dtype='uint8')
                pred = np.argmax(output.data.cpu().numpy()[0], axis=0)
                gt = target.data.cpu().numpy()[0]
                print('Train (epoch {}/{}) [{}/{} ({:.0f}%)]\tLoss: {:.6f}\tAccuracy: {}'.format(e, epochs, batch_idx, len(train_loader),100. * batch_idx / len(train_loader), loss.item(), accuracy(pred, gt)))
                plt.plot(mean_losses[:iter_]) and plt.show()
                fig = plt.figure()
                fig.add_subplot(131)
                plt.imshow(rgb)
                plt.title('RGB')
                fig.add_subplot(132)
                plt.imshow(convert_to_color(gt))
                plt.title('Ground truth')
                fig.add_subplot(133)
                plt.title('Prediction')
                plt.imshow(convert_to_color(pred))
                plt.show()
            iter_ += 1

            del(data, target, loss)

        if e % save_epoch == 0:
            # We validate with the largest possible stride for faster computing
            acc = test(net, test_ids, all=False, stride=min(WINDOW_SIZE))
            torch.save(net.state_dict(), './segnet256_epoch{}_{}'.format(e, acc))
    torch.save(net.state_dict(), './segnet_final')
```

Step 17: Let's train the network for 2 epochs and increase gradually. The matplotlib graph is periodically updated with the loss plot and a sample inference.

```python
train(net, optimizer, 2, scheduler)
```

```
Train (epoch 2/2) [900/1000 (90%)]      Loss: 1.124906  Accuracy: 29.9407958984375
```




```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:131: UserWarning: Implicit dimensi
/usr/local/lib/python3.7/dist-packages/torch/nn/_reduction.py:42: UserWarning: size_average and
  warnings.warn(warning.format(ret))
```

Step 18: Now that the training has ended, we can load the final weights and test the network using a reasonable stride, e.g. half or a quarter of the window size. Inference time depends on the chosen stride, e.g. a step size of 32 (75% overlap) will take ~30 minutes, but no overlap will take only one minute or two.

```python
net.load_state_dict(torch.load('./segnet_final'))
```

```
<All keys matched successfully>
```

```
all_preds, all_gts = test(net, test_ids, all=True, stride=32)

Confusion matrix :
[[ 9409441  1113414  2576588     1665   189696        0]
 [ 1636852 10554791   843669      156     4419        1]
 [  154899   119483  3790154     2113      248        0]
 [  646626    89166  3335461     6971    32598        0]
 [   48626   129639     6192        3   485794        1]
 [  210770   296260   304327       46     9931        0]]
---
36000000 pixels processed
Total accuracy : 67.35319722222222%
---
F1Score :
roads: 0.7409586842563857
buildings: 0.8329669350562161
low veg.: 0.5079515988701686
trees: 0.0033825224854528726
cars: 0.6975083653937962
clutter: 0.0
---
Kappa: 0.5450400634654966
Confusion matrix :
[[31576161  3370243 10180132    12812   355984        2]
 [ 5496006 25568163  2066711     4991    10127        2]
 [ 1148263   446898 33198789    29336     3932        1]
 [ 1939044   199986 21280685    58358    46061        3]
 [  143204   386534    29893      108   984975        1]
 [ 1156328  1282372  2988002     2607    33286        0]]
---
144000000 pixels processed
Total accuracy : 63.462809722222225%
```

So finally we train and test our CNN using VGG16 weights and segnet architecture with the accuracy of 63 %.

Incase you want to refer the Colab notebook refer below link.
Colab:
https://colab.research.google.com/drive/1Ik2hG3LO8ndgaLGrCXPkcNwZ4cu8fS7E?authuser=2#scrollTo=mObJEW8mBF8Z&uniqifier=2