This file contains 3 sections

```
1. Details on Deep Submodular Function

2. Details on implementation of Greedy Cardinality Constrained submod
ular maximization

3. Relevant portions of our original codes with comments
```

%%latex

SECTION (1/3) Deep Submodular Function - DSF

Architecture

- Currently, our work trains a DSF for each image separately.
- We learn DSF's on images of size 28 x 28 and we need to assign a pixel-wise importance for each image. Hence, inputs to our DSF are bit-vectors of size 28 x 28. Following is the architecture:

```
    def sqrt(input):
        return torch.sqrt(input)

    class DSF(nn.Module): # In PyTorch, classes for Neural Networks s
hould sub-class nn.Module which is the base-class.
        def __init__(self):
            super(DSF, self).__init__()
            self.fc1 = nn.Linear(28 * 28, 512)
            self.fc2 = nn.Linear(512, 256)
            self.fc3 = nn.Linear(256, 32)
            self.fc4 = nn.Linear(32, 1)

        def forward(self, x):
            x = x.view(-1, 28 * 28)
            x = self.fc1(x)
            x = sqrt(x)
            x = self.fc2(x)
            x = sqrt(x)
            x = self.fc3(x)
            x = sqrt(x)
            x = self.fc4(x)
            return x
```

Training

- We use **Batch-Gradient descent** as we do not have a large enough dataset for a mini-batch setup.
- OPTIMIZER: We use learning rates determined by Adagrad. Adagrad is usually preferred when the data is sparse & we observed the same.
- GRADIENT DESCENT: At each epoch, we backpropagate (using "**loss.backward()**") and update the weights using gradient descent (using "**optimizer.step()**").
- PROJECTION: The projection step with non-negativity constraints, is just the operation max(0, w) on weights w. Hence, after each weight update, we call "**clamp_zero**" class:

```
class clamp_zero(object):
    def __init__(self):
        pass

    def __call__(self, module):
        if hasattr(module, 'weight'):
            w = module.weight.data
            w.copy_(torch.clamp(w, min=0))
        if hasattr(module, 'bias'):
            w = module.bias.data
            w.copy_(torch.clamp(w, min=0))
```

%%latex

SECTION (2/3) MORE ON LOSS COMPUTATION

- The [original DSF paper (https://arxiv.org/pdf/1701.08939.pdf)](https://arxiv.org/pdf/1701.08939.pdf) trains DSF with only discrete supervision.
- Our loss (equation (2) in our [paper (https://arxiv.org/pdf/2104.09073.pdf)](https://arxiv.org/pdf/2104.09073.pdf)) comes from supervision via real inputs (multiplied with λ_1) and supervision via binarized inputs (multiplied with λ_2).
- In order to compute our loss with discrete supervision, we need to solve Cardinality Constrained Submodular Maximization problems at each training epoch.
  - We need solutions to this problem for a list of cardinalities. However, due to the greedy nature of Greedy Cardinality Constrained Submodular Maximization algorithm, we need to solve the problem only for the maximum value in the list of cardinalities.

Overview of the Greedy Cardinality Constrained Submodular Maximization:

Initially, $A = \{\}; f(A) = f(0)$.

1. Let $\bar{A} = A \cup \{argmax_{v \in V \setminus A} f(A \cup \{v\})\}$

2. if $f(\bar{A}) > f(A)$:

$A = \bar{A}$

else:

return A

The above two steps are repeated atmost K times.

Problem with a naive implementation: This would demand $O(|V|K)$ calls to the DSF Neural Network at every training epoch.

Solution: At every training epoch, we can just have $O(K)$ calls to the DSF by everytime inferring on a batch of $|V|$-sized inputs where the i^{th} input in the batch represents $A \cup \{v_i\}$.

Implementation Details with an example

- Let V be the universe with $|V| = 4$. For our work, $|V|$ is the resolution of images which was always a perfect square in the datasets we used.
- Initially, $A = \{\}; f(A) = f(0)$.
- We use a matrix $\mathbf{x}$ whose column i represents $A \cup \{v_i\}$ where $v_i \in V$. Initially, $\mathbf{x}$ = $$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
- We repeat the following atmost K times

We reshape $\mathbf{x}$ to get $\mathbf{inputs}$ because PyTorch demands inputs to be of the form *(batch_size, number_of_channels, height, width)*. For our work, batch_size is $|V|$, number_of_channels is 1, height=width=$\sqrt{|V|}$.

$\mathbf{outputs} = f(\mathbf{inputs})$ # *The i^{th} entry in this vector corresponds to $f(A \cup \{v_i\})$*

$i = argmax_i\ \mathbf{outputs}[i]$

if $\mathbf{outputs}[i] > f(A)$:

$f(A) = \mathbf{outputs}[i]$ # *We include {v_i} in A and update f(A)*

As A has been updated to $A \cup \{v_i\}$, we update the i^{th} row of $\mathbf{x}$ to all 1's. E.g. if $i = 1$ then $\mathbf{x}$ =
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The 2nd column is not of interest to us anymore as v_1 has already been chosen. We can remove that column and reduce our batch-size by 1 but for simplicity, we did not. *NOTE that due to monotonicity of DSF, the 2nd column won't again be chosen via argmax as it* *contains lesser number of 1's.*

Implementation of the procedure described above

```
import torch
def greedy_cardinality_constrained_submodular_max(f, k, n, device):
    """
    Returns solution of cardinality constrained submodular maximizati
on
        Parameters:
            f (PyTorch model) : DSF
            k (int)           : Cardinality we want to solve
            n (int)           : Where input is of the size n x n
            device (str)      : Device ('CPU' or 'CUDA')
        Returns :
            selected (array)  : Solution
    """
    card_V = n*n
    x = torch.eye(card_V)
    fA = f(torch.zeros(n, n).view(1, 1, n, n).double().to(device)).it
em() #reshaping for PyTorch
    for iteration in range(1, k+1):
        inputs = x.t().view(card_V, 1, n, n) #reshaping for PyTorch
        outputs = f(torch.Tensor(inputs).double().to(device))
        i = outputs.argmax(dim = 0).item()
        if outputs[i]>fA:
            fA = outputs[i]
            selected = x[:, i] #solution when cardinality constraint
is j
            x[i, :] = 1
        else:
            break
```

NOTE : The recently launched [submodlib (https://arxiv.org/pdf/2202.10680.pdf)](https://arxiv.org/pdf/2202.10680.pdf) package, might have a more efficient solver.

SECTION (3/3) Relevant portions of our original codes with comments

In [ ]:

```
"""
_____Greedy cardinality constrained submodular maximization
"""

import torch
def c_sb_mx(f, Klist, sq_n_sb_px, device):
    '''
    Returns solution of cardinality constrained submodular maximization
```

```
            Parameters:
                    f (PyTorch model): DSF
                    Klist (list)      : List of cardinalities
                    sq_n_sb_px (int) : square-root of no. of sub-pixels (i
                    device (str)      : device('cpu' or 'cuda') on which to
            Returns:
                    AList (dic): Dictionary with keys as cardinalities and
    '''
    k = int(np.array(Klist).max())#we only need to solve for max cardinali
    card_V = sq_n_sb_px*sq_n_sb_px#cardinality of V
    x = torch.eye(card_V)#card_V number of candidate A's each arranged in
    fA = f(torch.zeros(sq_n_sb_px, sq_n_sb_px).view(1, 1, sq_n_sb_px, sq_r
    AList = {}#dic with key k, value A*_k where A*_k is the optimal subset

    for it in range(1, k+1):#here iteration j means solving for cardinalit
        inputs = x.t().view(card_V, 1, sq_n_sb_px, sq_n_sb_px)#'x' reshape
        outputs = f(torch.Tensor(inputs).double().to(device))
        i = outputs.argmax(dim=0).item()
        if outputs[i]>fA:
            fA = outputs[i]
            selected = x[:, i] #solution
            x[i, :] = 1
            if it in Klist: #Recall that in iteration j, we are solving fo
                AList[it] = selected.detach().clone()
        else:
            break
    try:
        for it in Klist:
            if it not in AList: #e.g. we want solution for cardinality j b
                AList[it] = selected.detach().clone()
        return AList
    except:
        # Execution of this code indicates no element was chosen.
        print("EmptySet{}".format(outputs[i].item()))
        return torch.zeros(sq_n_sb_px*sq_n_sb_px)
```

In [ ]:

```
"""
______TRAINING DSF_____
"""

sp_w, sp_h = 28, 28 #super-pixel width & height
sq_n_sb_px = 28 # square root of resolution of sub-sampled image
ht = pre_process.final_ht_proc(sp_w, sp_h, thresholds, I_ALL) # hard-thres
sub_h = pre_process.final_subI_proc(sp_w, sp_h, I_ALL) # sub-sampled hard-

for epoch in range(epochs):
    # loss_1: loss with hard thresholded maps sub-sampled
    # loss_2: loss with original attribution maps
    loss_1 = None; loss_2 = None
    """
    Computing loss_1
    """
    # Get solutions to the submodular maximization problem for list of car
    # Adic is a dictionary with key as the cardinality & corresponding val
    Adic = submod.c_sb_mx(f, list(ht.keys()), sq_n_sb_px, device)

```

```
    # Convert Adic dictionary to a list & feed all these solutions to the
    ASList = list(Adic.values())
    AList_f = f(torch.stack(ASList).double().view(len(ASList), 1, sq_n_sb_
    tensor_ht = {}

    for xk, k in enumerate(ht): #Here k is the cardinality
        tensor_ht[k] = [torch.Tensor(ht) for ht in ht[k]] # hard threshold
        # Feed all the hard-thresholded maps having cardinality k to the D
        all_S_f = f(torch.stack(tensor_ht[k]).double().view(len(tensor_ht[
        for xs, _ in enumerate(tensor_ht[k]):
            to_add = AList_f[xk]-all_S_f[xs]+delta # computes \delta + f_w
            if to_add>0:
                if loss_1 is None:
                    loss_1 = to_add
                else:
                    loss_1 = loss_1+to_add
    """
    Computing loss_2
    """
    ones_f = f(torch.ones(sq_n_sb_px*sq_n_sb_px).double().view(1, 1, sq_n_
    tensor_sub_h = [torch.Tensor(s_h) for s_h in sub_h] #list of sub-sampl

    # Feed all sub-sampled hearmaps to the DSF neural network
    sub_h_f = f(torch.stack(tensor_sub_h).double().view(len(tensor_sub_h),
    for xs_h, _ in enumerate(tensor_sub_h):
        to_also_add = ones_f-sub_h_f[xs_h] #computes f_w(\mathcal{H}^*)-f_
        if to_also_add>0:
            if loss_2 is None:
                loss_2 = to_also_add
            else:
                loss_2 = loss_2+to_also_add
    loss = None
    if loss_1 is not None:
        loss = ld1*loss_1
    if loss_2 is not None:
        if loss is not None:
            loss = loss+ld2*loss_2
        else:
            loss = ld2*loss_2
    if loss is None:
        break
    loss_plt.append(loss.item())
    f.zero_grad()
    loss.backward()
    optimizer.step()
    f.apply(clipper)
```