IN5400 Week 14: Lab on explainability methods

Alex

April 8, 2021

1 The Goal

- ... to show you how different explanation methods behave when trying to identify biases in datasets.

We will do this in Pascal VOC, for two reasons. Firstly it is sufficiently complex to show differences between methods. Data which is too simple and has too strong geometric priors like MNIST/Fashion-MNIST/CIFAR-10 often does not allow to show certain more complex effects. Furthermore in my own experience, many methods which work on MNIST/Fashion-MNIST/CIFAR-10, break down on more complex problems.
Secondly, you have experience with the Pascal VOC dataloader.

the mandatory part:

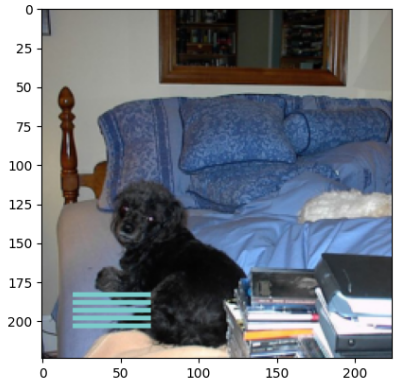
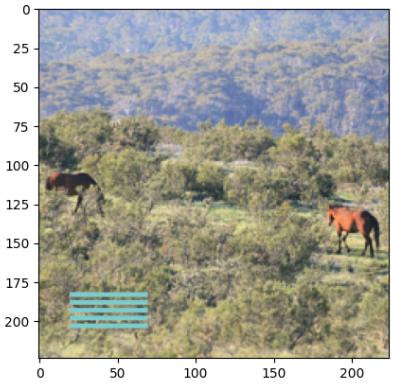
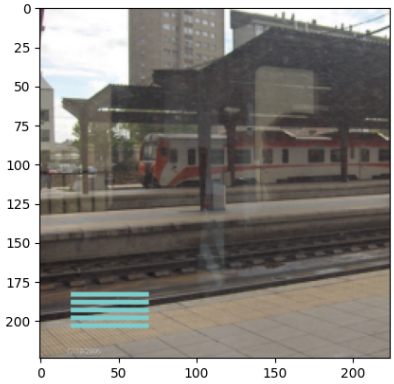
- Task 1 – change paths, run, have fun
- Task 2 – make a minor dataloader modification and run it. Task 1 and Task 2 are 20 minutes together.
- Task 3 – gradient times input. Should run within 30 minutes.
- Task 4 – GradCAM
- Task 5 – guided gradcam. These two together should be done in another 30 minutes
- Task 6 – guided BP. a simple mod from Task 5 by outcommenting.

the optional part:

- Task 7 – Integrated gradient. it is better than gradient times input but still not great.
- Task 8 – A simple ablation method. Since it involves only filling N copies of the same image and running a batched forward pass, it is an easy way to do.

1.1 The setup of two experiments

I have trained two classifiers based on an artefact which can be added to images.
If you want to train it yourself, then use add the following bias-infusers to your pascal VOC training code (I have validated that I used the class biasinducer3 for experiment1 and that biases were added to the images)

You can see that the artefact is large enough to be seen, and to not disadvantage too much low resolution attribution methods. It is cyan because natural images do not have much bright cyan (=green+blue) in them (why actually?).

- Experiment 1: the artefact is added to **each image**. The model is a resnet-50, has an mAP of $\sim$ 88, and an AP for bottle of 72.

  The resulting trained model is in: https://www.dropbox.com/s/e5r31n9a88o1gzx/w_bias3.pt?dl=0.

- Experiment 2: the artefact is added only to each image which has a **bottle** label (classindex 4).

  The model is a resnet-50, has an mAP of 89, and an AP for bottle of 99.99  .

  The resulting trained model is in: https://www.dropbox.com/s/ty0gbtpyjf4md2z/w_bias4.pt?dl=0.

- you are going to try out several attribution methods on experiment 2 to see how easy or difficult the bias can be spotted.

Note: the artefact is added after the dataloader, to avoid cropping it away with Resize(224)-CenterCrop(224) :-) .

if you want to train such classifiers by yourself, then you have to add for experiment 1 `def biasinducer3(..):` (see python file mentioned below) and for experiment 2 `def biasinducer4(...)` into your

```
def train\_epoch(...):
```

and

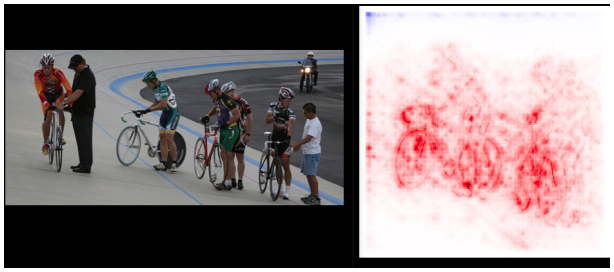
```
def evaluate_meanaverageprecision(...):
```

and apply them to your inputs before predicting with the model. I trained for 10 epochs with constant learning rate and a resnet50. takes less than 30 mins.

2 Task1

2.1 Just run and observe some heatmaps for Experiment 1

Use the code interpret_pytorch_in5400tmp_biastest1.py for this from the file code_lrp.zip. The code is a slighlty modified version from my github for imagenet https://github.com/AlexBinder/LRP_Pytorch_Resnets_Densenet, adapted for the Pascal VOC dataset.



Note the cropping effect resulting in a mismatch between image and heatmap. Explanation is for class bike (index 1). Note also the weak correlation between bike in person visible in the heatmaps. It comes from the stochastic correlation of these objects in the dataset. You wont see much of this on imagenet classifiers.

Some example pictures:

- nice cases: 2008_002773.jpg , 2008_005253.jpg , 2008_005796.jpg , 2008_007970.jpg, check heatmaps for class bike with persons on it for explanations for classes 1 (bike) vs 14 (person)
- fail cases: 2008_001985.jpg – it does not find the airplane, but it predicts boat because of the water and the white houses in the background that could look like boats in the distance.

you can observe a few things:

- heatmaps can look somewhat different when one explains different classes of the same image.
- one does not see the artefact lighting up a lot in the heatmaps. Seemingly the network does not have learned filters which pay much attention to it.

  Why does that happen? Because when added to every image, the artefact becomes a distractor, is non-discriminative, and thus it is best compressed away in some layer.

2.2 Offtopic: Remark1

If you would want to compute explanations not for a class with positive score, but for a class with a negative score (e.g. by selecting the class according to the ground truth labels, irrespective of the prediction), then there is one point you need to be careful about: if the output score is negative it can have two interpretations.

- case A: there is evidence for some class, but just not enough. in that case you can explain the outputscore as it is
- case B: there is no evidence for some class. In that case you need to explain the the inverted score, so that the output becomes positive. This can be achieved by multiplying the heatmap times -1.

You cannot distinguish between these two cases based on the output score of a class alone. (research question: what information is needed to be able to do that ?).

note: Therefore, for negative output scores either the heatmap or the inverted heatmap with scores times $* -1$ might contain the information you want.
You can try out to compute explanations for classes with positive ground truth labels and see what happens.

2.3 Offtopic: if you wanted to know: what does the explanation code do ?

```
inputs.requires_grad=True #dont forget this!

with torch.enable_grad():
  outputs = model(inputs)

for b in range(outputs.shape[0]):
    #poscls = labels[b,:]>0 # if you wanted to explain groundtruth classes
    poscls = outputs[b,:]>0

    for n in range(numcl):
      if poscls[n]==False:
        continue

      if inputs.grad is not None:
        inputs.grad.zero_()  # we may do repeated calls, flush the old explanations
        #it does not exist before the first call, though
      z= outputs[b,n]

      with torch.no_grad():
        z.backward(retain_graph=True) #do not delete the graph needed to compute backpropagation
        # bcs  we may do repeated calls,
        rel=inputs.grad.data.clone().detach() #get the explanation scores
        rel= rel/torch.abs(torch.sum(rel)) #normalize them

        fn = data['filename'][b] #save them
        savenm = os.path.join(savept,  os.path.basename(fn)+'_hm_c'+str(n)+'.jpg' )
        savehm(rel, savenm)
```

z.backward(...) computed the LRP via the following idea: the network canonizer

```
  model_e =  resnet50_canonized(pretrained=False)
  model_e.copyfromresnet(model, lrp_params=lrp_params_def1, lrp_layer2method = lrp_layer2method)
```

fuses batchnorm and conv layers and wraps each layer by a custom layer, which uses the same forward, but a modified backward pass, in which the gradient is replaced by LRP rules.

2.4 Offtopic: Remark3

- use for the validation data batchsize 1. Why?

  Yes, it is easy to fix lrp_general6.py to work correctly with arbitrary batchsizes (by changing grad_input[0] to grad_input everywhere) and explaining in a batch one class for each sample by doing something like that (explaining the sum over images of a batch, the relevance will flow back to each image separately):

```
inputs.requires_grad=True

with torch.enable_grad():
  outputs = model(inputs)

  classofinterest = torch.zeros((batchsize), device = ...)
  classofinterest[b]= ...
  z= torch.sum(outputs[:,classofinterest])

if inputs.grad is not None: # in case one does repeated calls, flush the old explanations
  inputs.grad.zero_()

with torch.no_grad():
  z.backward(retain_graph=True) # in case one does repeated calls
```

```
  relall=inputs.grad.data.clone().detach()

# here you got to save each image separately, but if
# one just needs only 1 class per image to be explained
# then using batchsizes is the faster way!
for b in range(outputs.shape[0]):
  rel = relall[b,:]
  fn = data['filename'][b]
  savenm = os.path.join(savept,  os.path.basename(fn)+'_hm_c'+str(classofinterest[b])+'.jpg' )
  savehm(rel, savenm)
```

However, for our case we may need to explain one image with respect to possibly multiple classes (namely all with positive scores), so batch processing for different images is not so useful.

3 Task2: Code a better dataloader and observe some heatmaps for Experiment 2

What steps do you have to do? We need a dataloader which returns only images with bottles

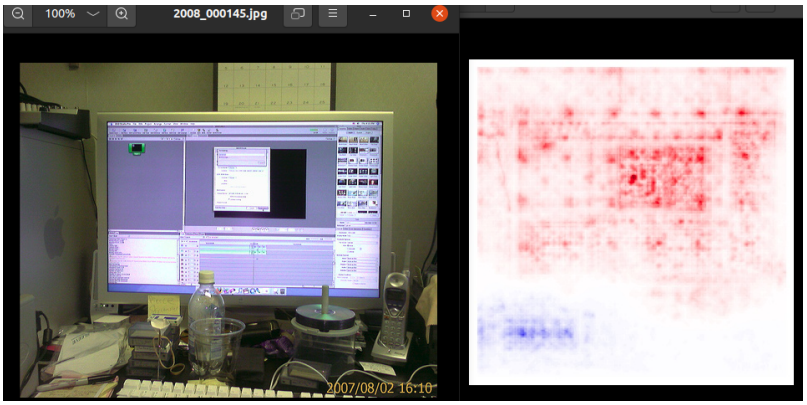
- create a class which allows to sample only bottle images

```
class dataset_voc_subclass(Dataset):
  def __init__(self, root_dir, trvaltest, subclasseslist, transform=None):
```

- How to ? easiest is to start witht he existing dataset class, then copy the fields imagelist and labels internally, then recreated these fields containing only bottle images and the 20-class label vectors for the respective images
- change the the model snapshot to be loaded to **bias4.pt**
- you can use biasinducer3, as we anyway look only at bottle images

3.1 Observations?

- one can see the bottle artefact clearly when explaining the bottle class (compute heatmaps yourself)
- explanations for other classes (usually bottle comes with person) have a slight negative score for the bottle artefact. Why? does that slight negative evidence happen?



Next lets see how one can spot the same artefact using other attribution methods.

3.2 Remark1

Adding an artefact to every class with the same proportion or to every image does **not guarantee** that it will **not produce a bias**.

An example where padding boundaries with nearest pixels for all classes led to a measureable bias for the aeroplane class is shown in https://www.nature.com/articles/s41467-019-08987-4. Machine learning is about observing correlations to labels, and deep methods dont have human biases which make humans tend to see high level effects / ignore borders.

4 Task3: implement gradient times input on Experiment 2

What steps do you have to do?

- remove the model wrapper, pass the original model into your interpret routine.

```
model_e =  resnet50_canonized(pretrained=False)
model_e.copyfromresnet(model, lrp_params=lrp_params_def1, lrp_layer2method = lrp_layer2method)
```

- get the gradient, multiply it with the input

4.1 Observations?

Can you spot the bias in these heatmaps?

Really?

5 Task4: try out GradCAM on Experiment 2

Ok the method from before just sucks. Lets try something else. GradCAM. This implementation is very easy to adapt: https://github.com/jacobgil/pytorch-grad-cam

- you can start off the code used for *gradient* $\times$ *input*, without the LRP wrapper.
- you need only gradcam.py
- the output has shape $(224, 224)$, savehm(...) expects an input of shape $(1, 3, 224, 224)$. You can either create a modified copy of savehm, or create a tensor with three copies along dim 1.

Reading the code in gradcam.py will make you see in 10 minutes how to use it.

5.1 Observations?

GradCAM is the MineCraft of XAI. That can be good or bad.

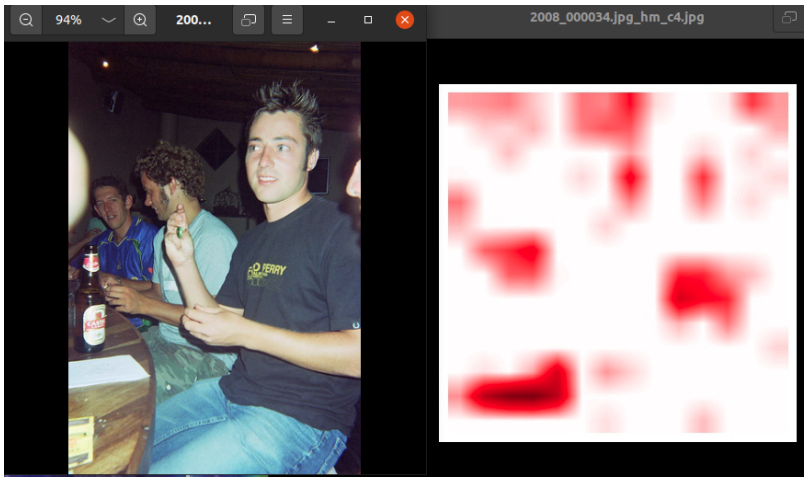
5.2 Improving the result - tuning gradCAM

- use

  `feature_module=model.layer3`

  to get a more civilized resolution

This helps with the resolution problem, and you can spot biases, however you get many false positive regions:



Generally Grad-CAM gives low resolution heatmaps. This can be fine or a problem for some settings.

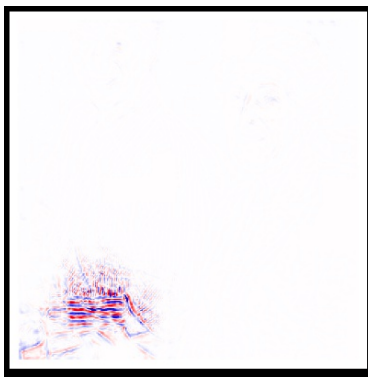
6 Task5: try out Guided-GradCAM on Experiment 2

You can start off your grad-cam code.

- gradcam.py tells you how to call GuidedBackpropReLUModel(model=model, use_cuda=args.use_cuda) . Two hints:
- for this attribution to not mess up your grad-cam computation, you have to provide it with a

  `copy.deepcopy(model)`

- the output has shape $(3, 224, 224)$ unlike the gradcam which had $(224, 224)$
- multiply both outputs together
- you will get informative heatmaps:



However, Guidedbackprop and LRP belong to methods which according to a group of researchers one should not use because they fail so called sanity checks (top-down randomization sensitivity). My take: never trust people who make axiom-type statements and are not in the field of logic.

6.1 A little rant? or a call for caution

There are papers https://papers.nips.cc/paper/2018/hash/294a8ed24b1ad22ec2e7efea049b8737-Abstract.html which claim that one should not use attribution methods which fail a so-called top-down randomization test, and suggest to use gradient-based methods.
My point here after seeing the above heatmap results:

The conclusions from the randomization test have to be taken with more caution as sometimes stated. Why? They do not randomize the whole neural network, but only the top-layers. Therefore the resulting function is for relu activations a piece-wise linear function on top of unchanged lower layer filters, and the explanation is based on the same cues provided from the lower layer filters.

The explanation is still based on high forward pass scores the from the same lower layer filters and regions as before, even when their linear combination is different. Therefore seeing the same regions lighting up is not too surprising.

A too strong stability of explanation does not result in a poor explanation quality for the given sample.

There are a number of papers which compare explanation results with metrics which put emphasis on precision and which find high performance for LRP, guided backpropagation and deepLIFT: https://www.aclweb.org/anthology/P18-1032/, https://arxiv.org/abs/1904.11829, https://link.springer.com/chapter/10.1007/978-3-030-33850-3_1. This does not mean that gradient-based methods are poor in general. Their high recall makes them suitable for segmentation initialization for example.

You need to measure the quality of your explanations in any case.

After the old man has ranted, the official part of this lab almost ended. Nah, it ends after task 6.

7 Task6: try out GuidedBP alone on Experiment 2

Well, you just comment one thing out :) . It cant get any easier here.
This is the end of the lab tasks.

8 Task7 - Bonus I: Integrated Gradient

For those who are interested: you can relatively easily compute Integrated gradient methods by yourself. Implementing this is easy based on the code you have

- store your

  `data['image']`

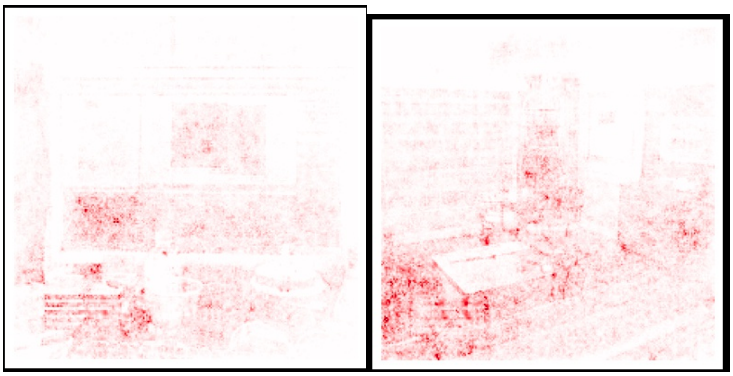
- run a forward pass once with the original image to get the output classes with positive prediction scores which you want to explain.

- Create a tensor img of shape $(50, 3, 224, 224)$. Fill the b-th element of it with $img * (1 - b/50.)$. Run backward(). multiply the gradient in

  `img.grad.data[b,:,:,:]`

  with the content of $data['image']$. Done

- there are some changes compared to gradient times input because you have to create a tensor filled with 50 scaled copies of the image.



You can spot the bias better than with gradient times input. However, you get lots of false positives in other regions. While conform to top-down randomization sanirty checks, its not really good still.

9 Task8 - Bonus II: A simple ablation method

The idea:

- store your

  `data['image']`

- run a forward pass once with the original image to get the output classes with positive prediction scores which you want to explain and to get the logit score for every positive class.
- choose a block-size (e.g. 7x7 or 14x14).
- copy the image and fill the block with 0 (corresponds to filling it with the image mean in the original image because of our image mean substraction preprocessing)
- Run the block in a sliding window. Measure the decrease or change of prediction score in every region
- create a scaled heatmap based on the prediction score change

Two notes:

- for a $(224, 224)$ you will need for a blocksize of 14 and stride 14: $16 \times 16 = 256$ forward passes. That can be a few forward passes depending on your batchsize. Ablation is not a Cheetah.
- Using an occlusion based method requires **a sufficiently large occlusion region**. Important: The occlusion region size is a very important parameter to judge the sensitivity of regions. If you would use $(1, 1)$ or $(2, 2)$ pixel regions, then you would barely occlude the artefact, and thus see a very little sensitivity.

  While intuitive, occlusion methods still require your brains to work well. The same argument applies also to feature removal for Shapley value methods. Naively they are often applied to $(1, 1)$-neighborhoods in images, which you can see in the bias example, would do right nothing with it.

- An example for a better occlusion method then the one here: https://openaccess.thecvf.com/content/ACCV2020/html/Agarwal_Explaining_image_classifiers_by_removing_input_features_using_generative_models_ACCV_2020_paper.html

10 Comparing heatmaps from different classes

Suppose you want to compare heatmaps, then one needs to take care about their normalization.

Once you have a final heatmap $s[c,h,w]$ what could you do to visualize it?

- you would sum most over the time over the RGB subchannels

$$r[h,w] = \sum_{c=1}^{3} s[c,h,w]$$

- you could normalize it by the maximum absolute value to obtain maximal visibilty of scores, then map it onto a color scheme, such as seismic in https://matplotlib.org/2.0.2/users/colormaps.html

$$\hat{r}[h,w] = \frac{r[h,w]}{\max_{h,w} |r[h,w]|}$$

Things get more complicated when you want to explain evidence for 2 or more classes or look at differential evidence between classes.

Note that above normalization removes the classifier evidence from the heatmap without standardizing it to some level. That means, these heatmaps are (for some methods) not good to be compared against each other for different classes.

10.1 Finding evidence for more than 2 classes I

suppose you want to obtain evidence for classes cat and dog.
You could do the following

```
out = model(x)
t= out[0,catindex]+ out[0,dogindex]
t.backward()
s = x.grad.data.clone()
```

This is equivalent for many explanation methods to

```
out = model(x)
out[0,catindex].backward( retain_graph = True) # bcs we want
#to call it a second dime
s1 = x.grad.data.clone()
x.grad.zero_() # dont forget to erase the old 'gradients'
out[0,dogindex].backward( )
s2 = x.grad.data.clone()
s= s1+s2
```

This extends to n classes (mind to use retain_graph = True). Now you would do channel summing and max-abs normalization.

Such a sum of heatmaps has one implication: it is - for many methods - weighted by the confidence of the predictions of cat and dog.

Point is: if the cat score is close to zero, and the dog score is high, then in the summed heatmap the cat evidence will be very faint.

For methods which have no direct relation to the output score such a scaling can be obtained as well. Example: if you use the gradient, then you can normalize it to euclidean norm of 1, then multiply the heatmap with the prediction score $f(x)$.

10.2 Finding evidence for more than 2 classes II

There an option to compute a balanced heatmap sum

```
out = model(x)
out[0,catindex].backward( retain_graph = True) # bcs we want
#to call it a second dime
s1 = x.grad.data.clone()
x.grad.zero_() # dont forget to erase the old 'gradients'
out[0,dogindex].backward( )
s2 = x.grad.data.clone()
s= s1 / torch.abs(torch.mean(s1))  +s2/ torch.abs(torch.mean(s2))
```

Point is: here the cat and dog heatmap are summed at normalized confidence levels. It depends on your use case which one you want. Usually, if you explain with respect to ground truth labels, then you do **not want** the predictor confidence to play a role in it, and thus prefer the second option.

10.3 Finding differential evidence for 2 classes in a (binary) multi-class problem

In a binary mutually exclusive classification problem with 2 outputs, $score[0]$ alone has no meaning. Thats why one would **not explain** using

```
out = model(x)
t= out[0,catindex]
t.backward()
s = x.grad.data.clone()
```

In this case most of the time you are interested in:

```
out = model(x)
t= out[0,catindex]-out[0,dogindex]
t.backward()
s = x.grad.data.clone()
```

Why? because you predict class 0 if

$score[0] > score[1] \Leftrightarrow score[0] - score[1] > 0$. LRP for example explains relative to states with score 0, and the tensor for which a score of 0 is the threshold, is the difference of the two class outputs.

This is equivalent for many explanation methods to

```
out = model(x)
out[0,catindex].backward( retain_graph = True) # bcs we want
#to call it a second dime
s1 = x.grad.data.clone()
x.grad.zero_() # dont forget to erase the old 'gradients'
out[0,dogindex].backward( )
s2 = x.grad.data.clone()
s= s1-s2
```

Note the minus.

10.4 Finding differential evidence for n classes in a multi-class problem

In this case often you want to include the classifier confidence because you predict class i if $score[i] > score[k] \forall k, k \neq i$. In such cases the scaling of the prediction score matters. Thus you end up using:

```
out = model(x)
out[0,catindex].backward( retain_graph = True) # bcs we want
#to call it a second dime
```

```
s1 = x.grad.data.clone()
x.grad.zero_() # dont forget to erase the old 'gradients'
out[0,dogindex].backward( )
s2 = x.grad.data.clone()
s= s1-s2
```

The only difference is that you can explain differences to $n-1$ other classes.

However if you want to compare differences of two heatmaps at the same confidence level for 2 classes, then you need to normalize again by the absolute value of the sum of heatmap scores:

```
out = model(x)
out[0,catindex].backward( retain_graph = True) # bcs we want
#to call it a second dime
s1 = x.grad.data.clone()
x.grad.zero_() # dont forget to erase the old 'gradients'
out[0,dogindex].backward( )
s2 = x.grad.data.clone()
s= s1 / torch.abs(torch.mean(s1)) -s2 / torch.abs(torch.mean(s2))
```

10.5 Finding differential evidence for n classes in a multi-label problem

Here most likely you want to compare at the same confidence level.

```
s= s1 / torch.abs(torch.mean(s1)) -s2 / torch.abs(torch.mean(s2))
```

It is because predictions for different classes are not inversely correlated to each other (a high dog score is not evidence for the absence of cats).

Again, the same question as before: should I weight it with the prediction confidence level or not?