50.021 – AI

Alex

Week 10: Pytorch hooks and implementing guided backprop

[The following notes are compiled from various sources such as textbooks, lecture materials, Web resources and are shared for academic purposes only, intended for use by students registered for a specific course. In the interest of brevity, every source is not cited. The compiler of these notes gratefully acknowledges all such sources.]

Learning objectives:

- tools for debugging pytorch training: using parametrized hooks, and backward hooks to be able to debug gradient statistics
- use classes with custom backwards from wrapped autograd functions, learn how to implement simple network attribution methods.

You will see some code examples and an intro on hook in the Friday zoom session.

1 Backward hooks for analysing gradient statistics

Take the 500 Imagenet validation set images, unpack them. For this exercise you wont need any ground truth label.

- write a dataloader which can return a batch consisting of a single image (for convenience of saving gradient norms, otherwise one has to mangle all filenames from a minibatch into one filename, this is doable but adds no value to understanding). You can use the dataloader from `imgnetdatastuff.py`.
  - Suggestion: do not load all images in the __init__ method of the dataset class. That does not scale well if you have 500000 images :) . load the filenames and the labels instead into a list or the like. load an image in `getitem` of your Dataset-derived class!
- take a vgg16 and a vgg16_bn network, and preload their weights from the pytorch model zoo. These are the good old VGG16 network, and its variant with batchnormalization.

- implement a parametrized backward hook which grabs the gradient, computes its ℓ_2-norm for each channel, and saves the vector of gradient norms for each channel to a filepath and a filename derived from the filename of the image (here a batchsize of 1 in the dataloader makes it easier).
- Use *output*[0] as its shape signature is not broken (unlike the tuple named input). This is the gradient which comes from the module above your current module.
- if the module is one of the two conv modules closest to the input, then compute it ℓ_2-norm, and save it to a filepath and a filename derived from the filename of the image. you can achieve that by registering backward hooks only for these two modules.
- compute the gradient norms for ≈ 250 images for both networks. compute their 5% until 95% percentiles, medians in 5% steps for both networks and compare their values. See `https://arxiv.org/abs/1806.02375` if you want to know more.

2 Implement the Guided backprop neural network attribution

Lets start with a simple neural network algorithm for neural network attribution, where you can code more by yourself.
The idea behind guided backprop: Figure 1 in `https://arxiv.org/abs/1412.6806`.
If you are in a relu module, then zero out the gradient, if the input is negative (that happens in every relu), or if the gradient itself is negative.
There are two ways to implement it: a backward hook, or a custom relu. The backward hook is 3 lines and just to simple (but you can use to check that your implementation is correct).
for the custom relu do the following steps:

- implement an autograd function for the relu (`https://pytorch.org/tutorials/beginner/pytorch_with_examples.html#pytorch-defining-new-autograd-functions`), but unlike the class myrelu in `https://pytorch.org/tutorials/beginner/pytorch_with_examples.html#pytorch-defining-new-autograd-functions` with the backward pass modified so that you get the guided backprop
- wrap your autograd function into a class derived from `mm.Module`. This is very simple, and can be done in the same manner as the autograd function `LinearFunction` is wrapped into a `class Linear(nn.Module)` in `https://pytorch.org/docs/stable/notes/extending.html`
- parse your network to replace all relus with your custom class. This is a little bit tricky.

What wont work is looping over modules and overwriting them inside the returned loop as in

```
for n,m in network.named_modules():
  if condition(n,m)==True:
    m=myreluclass()
#this does NOT work, it does not over write m inside network
```

What instead works:

You need to iteratively walk down through nested modules within `network` to find `layer4.1.relu` inside `layer4.1` which is inside `layer4` which is inside `network`.

Use pythons `hasattr(object, property)` to check if an object has a certain property, use `getattr(object, property)` to retrieve the property (e.g. `layer4.1` within `layer4`), and `setattr(object, property, value)` to set an object property to a value (to overwrite `layer4.1.relu` with your custom class). You can check `setbyname` in `guidedbpcodinghelpers.py` for an example. There is also a code to visualize heatmap tensors.

- then use the modified network with an image (e.g. from the dataloader), call `.backward()` on the class of the network output with the highest prediction logit, and visualize the heatmap which you will obtain in `input.grad.data`.

3 code and data are here:

The dataset and helper codes are in: `https://www.dropbox.com/sh/7wk6ju3rj96ogu1/AADjwuH1-rr8Yrn_wdidk3LFa?dl=0`