IN5400 – Machine Learning for Image Analysis

Alex

Week 05: Backpropagation

1 Understanding how the gradient flows

Refer to the neural networks in figures 1 and 2 with inputs $x_1 \in \mathbb{R}^D$ to neuron n_1 and $x_2 \in \mathbb{R}^D$ to neuron n_2. Equation for any neuron is

$$n_i = g(b + \sum_k w_{k,i} n_k + \sum_l v_{l,i} x_l), \tag{1}$$

where $w_{k,1} = 0, w_{k,2} = 0 \ \forall k$ (for n_1 and n_2) and $v_{l,i} = 0 \ \forall l$ for all neurons except n_1 and n_2. Note also that for many neurons n_i several $w_{k,i}$ are zero, whenever n_k is no input to n_i, for example in figure 1 $w_{4,3}$ and $w_{4,2}$ are zero.

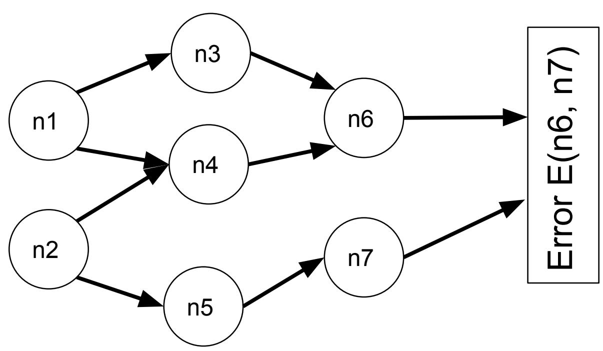

Figure 1: Mini Neural Networks I for Task **??**

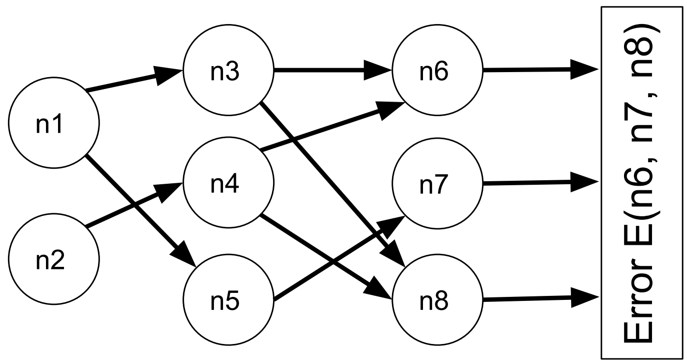

Figure 2: Mini Neural Networks II for Task **??**

We assume here for simplicity of partial derivatives, that the outputs of neurons which enter the loss function E are one-dimensional.
For the error E in both neural networks, instead of using the softmax function we learned in class, we use the quadratic error function for regression purpose,

$$E = \sum_{i \in data} (n_6 - y_6^{(i)})^2 + (n_7 - y_7^{(i)})^2$$

$$E = \sum_{i \in data} (n_6 - y_6^{(i)})^2 + (n_7 - y_7^{(i)})^2 + (n_8 - y_8^{(i)})^2$$

Note that the output neurons depend implicitly on the neural network inputs $x_1^{(i)}$ and $x_2^{(i)}$
Write down an expression for the gradients of all the weights for E function for each neural network:

For figure 1:

1. $\frac{\partial E}{\partial n_4}$;
2. $\frac{\partial E}{\partial w_{2,5}}$;
3. $\frac{\partial E}{\partial (v_{1,1})_d}$ where $(v_{1,1})_d$ is the d-th dimension of the weight for neuron n_1 for input x_1;
4. $\frac{\partial E}{\partial (x_2)_d}$ where $(x_2)_d$ is he d-th dimension of the input for neuron n_2;

For figure 2:

1. $\frac{\partial E}{\partial (v_{2,2})_d}$ where $(v_{2,2})_d$ is the d-th dimension of the weight for neuron n_2 for input x_2;

2. $\frac{\partial E}{\partial w_{2,4}}$

3. $\frac{\partial E}{\partial n_1}$

note: Write the expression in terms of

- $\frac{\partial E}{\partial n_i}$, where n_i is a neuron directly connected to the output
- $\frac{\partial n_k}{\partial n_i}$, where n_i is direct input to n_k
- $\frac{\partial n_i}{\partial w_{k,i}}$
- $\frac{\partial n_i}{\partial v_{i,i}}$
- and $\frac{\partial n_k}{\partial x_k}$ if x_k is input to the neural network (otherwise use $\frac{\partial n_k}{\partial n_i}$)
- At this point you do **not need** to plugin how $\frac{\partial n_k}{\partial n_i}$ or $\frac{\partial n_k}{\partial w_k}$ or $\frac{\partial n_k}{\partial x_k}$ looks like.
- You **do not need** to multiply out terms in parentheses, so $(a+b)c$ or $((a+b)c+(d+e)f)g$ is fine to keep it like that!

Besides that you can go also through the two leftover graphs from the lecture.

2 Directional Derivatives

- Compute the directional derivative $Df(X)[H]$ in direction H for:

$$
\begin{aligned}
f(X) &= Xa, & X \in \mathbb{R}^{d\times k}, a \in \mathbb{R}^{k\times 1}, \\
f(X) &= XX^\top, & X \in \mathbb{R}^{d\times n}
\end{aligned}
$$

- What will be the shape of the direction H in $Df(X)[H]$ for these two? Is it a real number, a vector or a matrix? Express it as $\mathbb{R}^{1\times 1}$ if you think it will be a scalar, as $\mathbb{R}^{d\times 1}$ if you think it is a vector, or as $\mathbb{R}^{d\times e}$ if you think it is a matrix.
- What will be $Df(X)[H]$? Hint: you can write it as product of matrices if you like it (instead of summing in the flavor of $\sum_{ijk} c_{ijk}$).
- Compute the directional derivative $Df(X)[H]$ in direction H for:

$$
\begin{aligned}
f(X) &= XCX, & X \in \mathbb{R}^{d\times d} \\
f(X) &= CXBX^\top AX, & X \in \mathbb{R}^{d\times d}, \{A, B, C\} \in \mathbb{R}^{d\times d} \\
f(X) &= \|X\|_2, & X \in \mathbb{R}^{1\times d}
\end{aligned}
$$

Hint: remember the product rule.

- Compute the directional derivative $Df(X)[H]$ in direction H for:

$$f(X) = \begin{pmatrix} 1 & x_2 \end{pmatrix} \begin{pmatrix} 1 & x_2^3 \\ \sin x_2 & x_1 \end{pmatrix}$$

3 Going debug mode: accessing gradients of intermediate layers and saving them to disk

The point of this exercise is to make you confident in the fact that you can access results of internal computations of pytorch.

You will work with hooks. A hook is a function call with a certain (input/output)-signature. It can be either a forward hook or a backward hook.

A forward hook is registered using

```
handle=module.register_forward(hook)
```

to a pytorch module (or a tensor). Forward hooks are called during the forward pass when the module is invoked. They can be used to inspect or save feature maps or statistics of feature maps (mean, variance, all you can imagine).

Btw, the handle can be used to remove the hook from a module. You will need this in this exercise, too.
A backward hook is registered using

```
handle=module.register_backward(hook)
```

to a pytorch module. Backward hooks are guess when called? They can be used to inspect or save gradients or statistics of gradients.

In particular, you can access feature maps and gradients of somewhere in the graph for which no explicit tensors were defined. Example: the forward pass of a neural net `https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py` line 238 yourresnet.layer2 and there in particular maybe yourresnet.layer2.conv2 (note: layer2 can be either an instance of BasicBlock or BottleNeck depending on what resnet you are using – see the definitions in lines 268++, but both have a conv2 module inside)

You can attach them to selected modules by looping over modules and selecting

```
for ind,(name,module) in enumerate(model.named_modules()):
...     print('name: {}'.format(name) )
        #attach here if name or module fits your wishes
```

Here you will use backward hooks to look at statistics of gradients. Remember from the lecture: maintaining gradient flow through layers on approximately the same scale is of utmost importance.

See also for hooks: `https://pytorch.org/tutorials/beginner/former_torchies/nnft_tutorial.html`

Towards coding:
Consider the file *fashionmnisttrain.py*. This trains a CNN for fashion mnist. Your goal is to be able to access and analyze gradients for intermediate layers by registering a backward hook. Once you have understood how to do this, you will be able to store also forward pass feature maps instead – via a forward hook.

Your overall task:

- During training you are computing gradients within your network – for every minibatch . You want to store these gradients (for CNN modules only for simplicity).

- implement a parametrized backward hook, which saves the incoming gradient to a file. Why parametrized ? You need to make the filename depending on which data sample or minibatch and which layer was used to compute them!

  The gradients depend on the minibatch you are in. If you want to store the gradients, you have to store them for every minibatch separately, without overwriting gradients from different minibatches. Therefore, your filename must contain an identifier of the minibatch. (Note: for a real problem, you would also store in a separate file also the filenames of the samples which were used in this minibatch! Thats why mydataloaders usually return also a filename beyond sample and label.

  We skip this here, because cifar-10 is not based on filenames but on just a matrix with rows and columns.) For the sake of keeping matters simple, we will save gradients for the first 50 minibatches of the last training epoch. Furthermore the filename for storing gradients should contain an identifier of which neural network layer/module the gradients belong to.

  Parametrized means that it takes as parameters the output path, where to store the files and a filename, which is composed of information about which batch index you are processing and which cnn layer.

- train your net for maybe 10 epochs (for the initial coding you can keep it at 2 epochs)

- when you are in the training of the last epoch, then register the hook inside your train epoch routine in every CNN layer – but only during the first 50 iterations over batch sizes. Dont forget to remove the handles after one minibatch was used to get the gradients! Note that the gradients occur only during training phase, not during evaluation phase.

- check the slides on hooks and the pytorch documentation on hooks

  What will help in attaching the hooks to selected modules, is to iterate over all named modules of your network:

```
for ind,(name,module) in enumerate(model.named_modules()):
    print('name: {}'.format(name) )
```

Of course, you do not want to attach a hook for every named module, but only for those who are convolution layers. Thus, checking whether a module belongs to a class or a derived class by using the following:

```
boolvariable= isinstance(module, someclass)
```

is your friend here!

3.1 actual coding for backward hooks:

- search for ###YOURCODEHERE in the code, and fill in

3.2 actual coding for the analysis of gradients:

- write a small piece of code which reads the gradients from disk, and prints the 50%, 70% and 90% quantiles of the gradients in cnn1 to cnn6. It should create a figure with three subfigures, one for each quantile. Each subfigure should contain 6 statistics - one statistic for each cnn1 to cnn6.

3.3 playing around with the statistics

- You will observe that for the 70% quantile that the gradients of cnn3 and cnn6 are zero. Why this makes sense ? Hint: check what comes as next compute module in the forward pass.

- Now change maxpool1 to an average pooling with same kernel size as before, retrain and check the quantiles of gradients for cnn3 and cnn6. You should observe a moderate increase for the median and the 70% quantile.

- Revert maxpool to maxpooling. Replace the reLU with a leaky relu with negative slope 0.1. Compare the size of gradients in the first Cnn for the ReLU-net versus the LeakyReLU net.

4 Bonus Task

All too boring la? Then check the gradient stats with a more silly architecture, to see a decreasing gradient flow. Then you will have another reason why convolutions are good when it comes to deeply stacked layers. ... Its not said that one cannot learn with a silly architecture, but it is easier to learn with a good architecture.

- create a model without any max pooling and only 6 linear layers. Its training will be poor.
- if you rename the Model class name, then also rename it in the super(...) call
- replace all nn.Conv2d by nn.Linear
- first linear: 768 inputs $(28 * 28)$, all others 32 inputs
- all layers: 32 outputs
- last linear: 32 inputs.
- remove all pooling.
- forward: remove the initial reshape of x, as the input is a linear layer now.
- hook attachment: must check for an instance of nn.Linear