Dealing with Memory Problems in Network with Many Weights

I have a neural network with the architecture `1024, 512, 256, 1` (the input layer has `1024` units, the output layer has `1` unit, etc). I would like to train this network using one of the optimization algorithms in `scipy.optimize`.

The problem is that these algorithms expect the function parameters to be given in one vector; this means that, in my case, I have to unroll all the weights in a vector of length

```
1024*512 + 512*256 + 256*1 = 655616
```

Some algorithms (like `fmin_bfgs`) need to use identity matrices, so they make a call like

```
I = numpy.eye(655616)
```

which, not very surprisingly, produces a `MemoryError`. Is there any way for me to avoid having to unroll all the weights into one vector, short of adapting the algorithms in `scipy.optimize` to my own needs?

python numpy machine-learning scipy

asked Mar 6 '13 at 10:49
Paul Manta
9,151 4 36 94

Wow, that's a lot of weights to fit. (+1) – NPE Mar 6 '13 at 10:54

@NPE I'll have to admit that I don't really know what I'm doing. I'm training on images of size `32x32`. Should I make the images even smaller? – Paul Manta Mar 6 '13 at 10:56

Then again, reducing the size of the images to `16x16` and using the architecture `256, 128, 1`, I'd still have and unrolled weight vector of length `32896`. – Paul Manta Mar 6 '13 at 11:05

Generally speaking, if you're having performance problems in Python, it might be time to try and do the heavy lifting in a different language. – jnnnnn Mar 17 '13 at 5:25

add comment

2 Answers

Don't try to fit the weights to a NN using L-BFGS. It doesn't work especially well (see early Yann LeCun papers), and because it's a second-order method you're going to be attempting to approximate the Hessian, which for that many weights is a 655,000 x 650,000 matrix: this introduces a performance overhead that simply won't be justified.

The network isn't that deep: is there a reason you're avoiding standard back-prop? This is just gradient descent if you have access to a library implementation, the gradients are cheap to compute and because it's only a first order method, you'll not have the same performance overhead.

EDIT:

Backpropogation means that the update rule for w_i at step t is:

w_i(t) = w_i(t-1) - \alpha (dError / dw_i)

Also, you've run into the reason why people in vision often use Convolutional NNs: sparse connectivity massively reduces the size of the weight vector where you have one neuron per pixel.

edited Mar 18 '13 at 15:49 answered Mar 18 '13 at 13:01

Ben Allison
2,521 1 10

+1 For pointing out that the memory problem is caused by the Hessian. I don't avoid standard backprop, I use backprop to calculate the gradient of the error wrt. the network's weights. I still need an optimization algorithm that can traverse the error surface (using the gradient calculated with backprop) to find its minimum. – Paul Manta Mar 18 '13 at 14:03

Backprop, at least as I would use the term, means using gradient descent. You can do it stochastically (each example in turn) or in batches. I.e. your update is a step size multiplied by the negative of the gradient. – Ben Allison Mar 18 '13 at 14:44

Edited my answer because another useful tidbit occurred to me: convolutional nets are used for this reason. – Ben Allison Mar 18 '13 at 15:41

What is the algorithm used to calculate the error gradient wrt. weights called? Also, is there a technical or conceptual reason for "backpropagation" to refer only to training done with gradient descent? – Paul Manta Mar 18 '13 at 18:35

I don't think that's an algorithm: the error gradients wrt the weights come from calculus, they're usually analytic (provided your activation functions are differentiable). I think the name backpropagation came from generalising the delta rule, before a lot of research into optimisation methods. It just happens that this is equivalent to gradient descent. Other methods are possible, most notably the Hessian Free method of James Martens, but that's really not trivial to implement and you'll probably get a lot of mileage out of simple backprop. – Ben Allison Mar 19 '13 at 9:41

show **3** more comments

I think memory-mapping with solve your problem. I suggest using numpy.memmap, which is optimized for numpy arrays. Note that while memory-mapping can be plenty fast, you should keep an eye out for thrashing.

For your general culture, the mmap module is a more general-purpose memory-mapping library that is included in the standard library.

answered Mar 18 '13 at 9:32
blz
1,550 13 36
add comment

Not the answer you're looking for? Browse other questions tagged python numpy machine-learning scipy or ask your own question.