

nngraph

Today we'll do a simple introduction to nngraph.

We'll be working purely interactively today. Start as follows:

```
_____ | Torch7
/_ __/_ | Scientific computing for Lua.
/ / / _ \/ __/ _ | https://github.com/torch
| http://torch.ch
```

If you get true as the return value of the require, then nngraph is installed. If it is not installed, install it by running luarocks install nngraph.

Start by reading over the first few sections in the README file: https://github.com/torch/nngraph/.

This practical's files can be found here: https://github.com/oxford-cs-ml-2015/practical5. Read README.md for setup instructions for the lab machine. The first practical's installer will have automatically installed nngraph.

Introduction

In lecture, we discussed the expressibility of neural network models by the variety of ways they can be combined together. A simple example is a feed-forward network, like the type encountered in the last few practicals. The nn.Sequential module is a container for other modules, so we had a recursive structure. The layers we've seen in Torch so far all have the same input and output types, though.

```
-- input object --v
output = model:forward(input)
-- ^-- output object
```

In the nn.Linear, nn.Sigmoid, nn.Tanh, nn.LogSoftMax, and most other layers, the type of input and output were always Tensors.

But what if a module needs to produce more than one output, or to take more than one input? In Torch's nn module, several modules do this, such as nn.CAddTable which performs an element-wise addition. Its input is a Lua table of Tensors that are all the same size, and its output is another Tensor of the same size. We can try it out:





```
+th> add = nn.CAddTable()
+th> t1 = torch.Tensor{1,2,3}
+th> t2 = torch.Tensor{3,4,10}
+th> add:forward({t1, t2})
    4
    6
    13
[torch.DoubleTensor of dimension 3]
```

Since this layer takes an arbitrary number of inputs, it is useful for combining other pieces of networks together.

A slight digression: Suppose input to nn.CAddTable is of type/shape { Tensor, Tensor } (a table of tensors) and output is of type Tensor. The type/shape of the backward messages matches these: gradInput $(\frac{\partial loss}{\partial input})$ matches that of input, and gradOutput $(\frac{\partial loss}{\partial output})$ matches that of output. Optional exercise: verify this for yourself in code, or reading its source.

An example

Say we want to create a module that computes $\mathbf{z} = \operatorname{sigmoid}(\operatorname{linear}_w(\mathbf{x})) + \mathbf{x}$, where linear signifies a linear transformation plus a bias term like the $\operatorname{nn.Linear}$ class. This is easy to draw as a graph, but expressing it in the sequential format is, although possible, very cumbersome. (For the curious, it involves a module called $\operatorname{nn.ParallelTable}$ that contains multiple modules that get evaluated on each entry of their input table.)

It is easy to write a forward pass manually, but we probably don't want to do that because then we need to write the backward pass manually as well. We'll see that this is sometimes a reasonable approach though, when you have a model with many steps. This will be the case for the LSTM code that we will look at next week.

This is our motivation for nngraph.

Playing with nngraph

Suppose we want to compute
$$\mathbf{z} = \mathbf{x}_1 + \mathbf{x}_2 \odot \operatorname{linear}(\mathbf{x}_3) \tag{*}$$

(⊙ is element-wise multiplication), as a nn.Module so we can integrate it into a large network. In nngraph this could be done easier.

nngraph overloads the call operator (i.e. the () operator used for function calls) on all nn.Module objects. When the call operator is invoked, it returns a node wrapping the nn.Module. The call operator takes the parents of the node as arguments, which specify which modules will feed into this one during a forward pass.

Machine Learning



March 4, 2015

To specify the module's inputs, we create dummy input nodes that perform the identity operation, then use these as parents for nodes that do computation. Here is the same addition example:

```
+th> x1 = nn.Identity()()
+th> x2 = nn.Identity()()
+th> a = nn.CAddTable()({x1, x2})
+th> m = nn.gModule({x1, x2}, {a})
```

Now m is a nn.Module that performs exactly the same computation as nn.CAddTable if you give either one a table with two Tensors.

To make sure we understand what is going on, note the types of each part of this line of code:

$$\underbrace{\mathtt{a}}_{\mathtt{graph\ node}} = \underbrace{\underbrace{\mathtt{nn.Module}}_{\mathtt{nn.CAddTable}()(\{\mathtt{x1},\mathtt{x2}\})}_{\mathtt{graph\ node}}$$

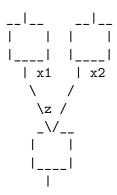
and x1 and x2 are graph nodes as well, *not* nn.Module's. Similarly, note that nn.Identity() is a nn.Module instance but nn.Identity() () is a graph node.

As you can see, once we finish describing the graph, we create an nn.gModule (which is nngraph's nn.Module implementation wrapping the graph) performing the graph's computation as follows:

$$\underbrace{\mathtt{m}}_{\mathtt{nn.Module}} = \mathtt{nn.gModule}(\underbrace{\{\mathtt{x1},\mathtt{x2}\}}_{\mathtt{table of output nodes}}, \underbrace{\{\mathtt{a}\}}_{\mathtt{table of output nodes}}$$

If there are any modules that have parameters, nn.gModule will automatically combine all these parameters.

Remember what we discussed in class: if you have a structure of the following shape:



During the forward pass, z is computed and used for both of the subsequent modules that depend on this value. The backward messages $\frac{\partial loss}{\partial x_1}$ and $\frac{\partial loss}{\partial x_2}$ both get summed as they are sent

Machine Learning



March 4, 2015

backwards to the common node. This is a consequence of the multivariate chain rule. This is done internally in the background by nngraph.

Handin: create an instance of nngraph's nn.gModule class that computes the expression marked (*) from above. You can make up values for the sizes of the \mathbf{x}_i 's, as you need to specify sizes for the nn.Linear class. Feel free to look at the nngraph README linked to above for hints.

Call its **forward** function to verify that it works, and compare this result to something computed by hand.

Handin

See the one bolded "Handin:" part above.

If you finish early

Challenge 1: LSTMs

Try implementing an LSTM cell, using the equations in http://arxiv.org/abs/1308.0850 as reference, equations (7-11) on page 5. But **omit** the term in the sum input to the gates that refers to c_{t-1} . That is, create a module that takes as input c_{t-1} , h_{t-1} , x_t , and produces as output c_t , h_t .

If you attempt and want to see a solution, see the lstm function here: https://github.com/wojciechz/learning_to_execute/blob/master/main.lua#L23.

Challenge 2: computation graphs

The graph that we describe to nngraph is a computation graph, illustrating the dependencies between computations in a forward pass. By flipping the directions of the edges in this graph, we get the backwards graph.

Given a computation graph, which is always a directed acyclic graph (DAG), state an algorithm that gives a valid sequence of computations as a list specifying an evaluation order for the nodes. (Hint: this DAG is a dependency graph.) Note that once we have an order for this forward pass, we can just reverse this list to get an execution order for the backward pass.