Training ConvNets with Torch

Soumith Chintala
Next.ml
17th January, 2014

Tutorial Structure

Hour 1

- Action Recognition from Videos:
  - Discuss task at hand
  - End-to-end rule-based methods
  - Hand-engineered features + SVM
  - Neural Nets (and discuss discovering graph structure automatically)
  - ConvNets

- Notebook Setup

Tutorial Structure

Hour 2

- Torch Basics
  - What is torch?
  - Lua and Syntax
  - Tensors, types, storages
  - Slicing and selecting
  - views vs copies
- Basic visualization in iTorch
- Neural Networks
  - Containers and Modules
  - Examples:
  - Overfeat
  - Alexnet
  - Inception
  - LSTM
- Optim package

Tutorial Structure

- Hour 2/3
- Coding an action recognition system
  - load and normalize data
  - create neural network
  - train neural network
  - test/validate neural network

Task at Hand

First attempt

Pure rule-based

Second Attempt

- Hand-designed features

Third Attempt

- Convolution Networks

Fourth Attempt

- Fully connected nets

Neural net in detail

- Show a typical two-layer network

- ....

torch

What is Torch?

- Not a language. Based on Lua
- Not just a Tensor library
- Ecosystem and community

Lua and Syntax

Tensors

- nDimensional arrays – torch.Tensor
- Separated from Storages
- Can be sliced, re-viewed, selected etc. without a memory copy where possible

Basic Visualization

Neural Networks

Module

A neural network is called a Module (or simply *module* in this documentation) in Torch. `Module` is an abstract class which defines four main methods:

- forward(input) which computes the output of the module given the `input` Tensor.
- backward(input, gradOutput) which computes the gradients of the module with respect to its own parameters, and its own inputs.
- zeroGradParameters() which zeroes the gradient with respect to the parameters of the module.
- updateParameters(learningRate) which updates the parameters after one has computed the gradients with `backward()`

It also declares two members:

- output which is the output returned by `forward()`.
- gradInput which contains the gradients with respect to the input of the module, computed in a `backward()`.

Two other perhaps less used but handy methods are also defined:

- share(mlp,s1,s2,...,sn) which makes this module share the parameters s1,..sn of the module `mlp`. This is useful if you want to have modules that share the same weights.
- clone(...) which produces a deep copy of (i.e. not just a pointer to) this Module, including the current state of its parameters (if any).

Neural Networks

Plug and play

Building a simple neural network can be achieved by constructing an available layer. A linear neural network (perceptron!) is built only in one line:

```
mlp = nn.Linear(10,1) -- perceptron with 10 inputs
```

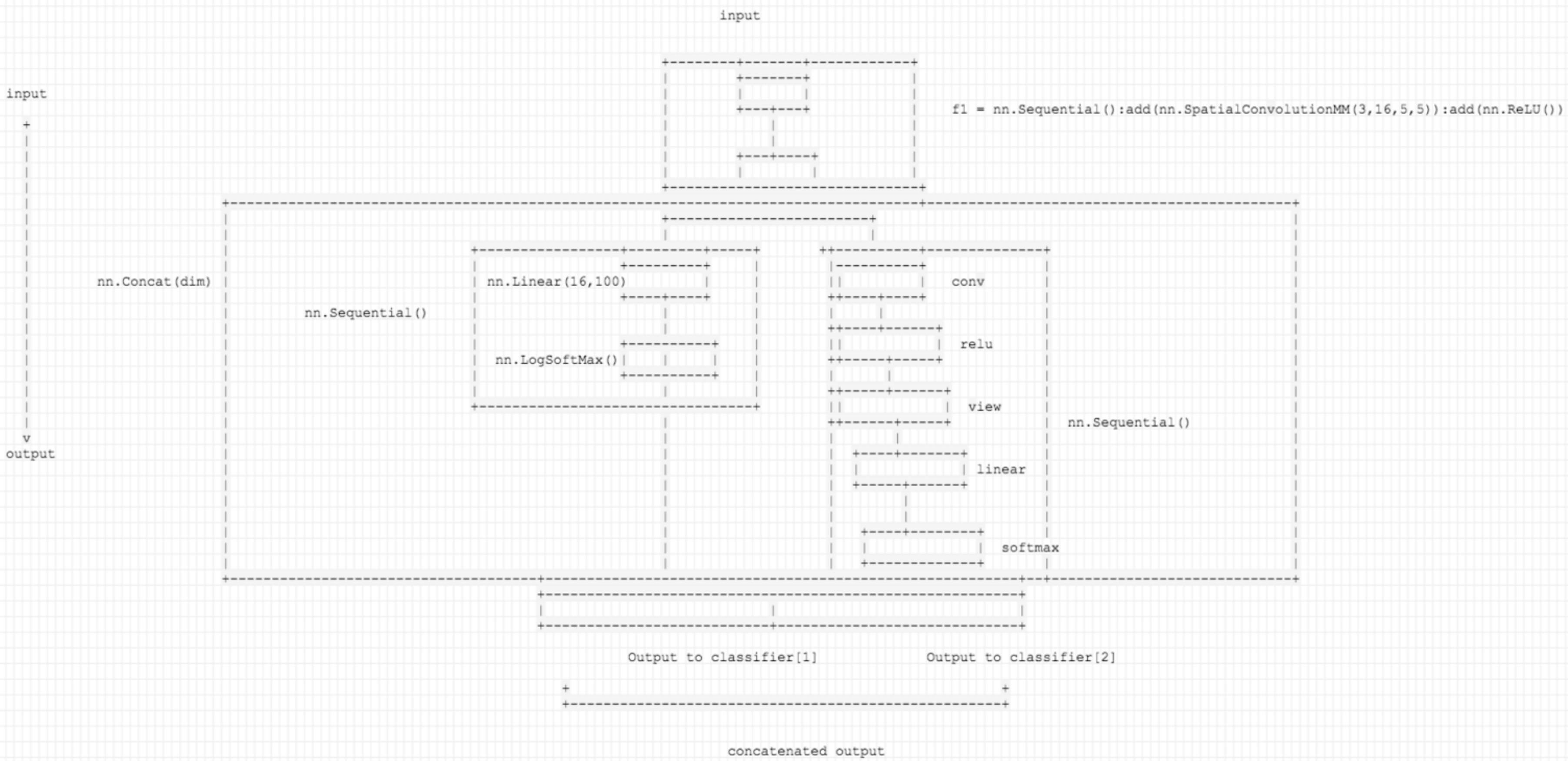
More complex neural networks are easily built using container classes Sequential and Concat. `Sequential` plugs layer in a feed-forward fully connected manner. `Concat` concatenates in one layer several modules: they take the same inputs, and their output is concatenated.

Creating a one hidden-layer multi-layer perceptron is thus just as easy as:

```
mlp = nn.Sequential()
mlp:add( nn.Linear(10, 25) ) -- 10 input, 25 hidden units
mlp:add( nn.Tanh() ) -- some hyperbolic tangent transfer function
mlp:add( nn.Linear(25, 1) ) -- 1 output
```

Of course, `Sequential` and `Concat` can contains other `Sequential` or `Concat`, allowing you to try the craziest neural networks you ever dreamt of! See the [[#nn.Modules|complete list of available modules]].

Neural Networks



Optimization

We first define a state for conjugate gradient:

```
state = {
   verbose = true,
   maxIter = 100
}
```

and now we train:

```
x = torch.rand(N)
optim.cg(JdJ, x, state)
```

You should see something like:

```
after 120 evaluation J(x) = -3.136835
after 121 evaluation J(x) = -3.136836
after 122 evaluation J(x) = -3.136837
after 123 evaluation J(x) = -3.136838
```