Mocha Documentation

Release 0.0.1

pluskid

December 19, 2014

Contents

CHAPTER 1

Tutorials

1.1 Training LeNet on MNIST

This tutorial goes through the code in `examples/mnist` to explain the basic usages of Mocha. We will use the architecture known as [LeNet], which is a deep convolutional neural network known to work well on handwritten digit classification tasks. More specifically, we will use the Caffe's modified architecture, by replacing the sigmoid activation functions with Rectified Learning Unit (ReLU) activation functions.

1.1.1 Preparing the Data

MNIST is handwritten digit recognition dataset containing 60,000 training examples and 10,000 test examples. Each example is a 28x28 single channel grayscale image. The dataset in a binary format could be downloaded from Yann LeCun's website. We have created a script `get-mnist.sh` to download the dataset, and it will call `mnist.convert.jl` to convert the binary dataset into HDF5 file that Mocha could read.

When the conversion finishes, `data/train.hdf5` and `data/test.hdf5` will be generated.

1.1.2 Defining the Network Architecture

The LeNet consists of a convolution layer followed by a pooling layer, and then another convolution followed by a pooling layer. After that, two densely connected layers were added. We don't use a configuration file to define a network architecture like Caffe, instead, the network definition is directly done in Julia. First of all, let's import the Mocha package.

```
using Mocha
```

Then we will define a data layer, which read the HDF5 file and provide input for the network:

```
data_layer = HDF5DataLayer(source="data/train.txt", batch_size=64)
```

Note the `source` is a simple text file what contains a list of real data files (in this case `data/train.hdf5`). This behavior is the same as in Caffe, and could be useful when your dataset contains a lot of files. Note we also specified the batch size as 64.

Next we define a convolution layer in a similar way:

```
conv_layer = ConvolutionLayer(name="conv1", n_filter=20, kernel=(5,5), bottoms=[:data], tops=[:conv])
```

There are more parameters we specified here

name Every layer could be given a name. When saving the model to disk and loading back, this is used as an identifier to map to the correct layer. So if your layer contains learned parameters (a convolution layer contains learned filters), you should give it a unique name.

n_filter Number of convolution filters.

kernel The size of each filter. This is specified in a tuple containing kernel width and kernel height, respectively. In this case, we are defining a 5x5 square filter size.

bottoms An array of symbols specifying where to get data from. In this case, we are asking for a single data source called `:data`. This is provided by the HDF5 data layer we just defined. By default, the HDF5 data layer tries to find two dataset named `data` and `label` from the HDF5 file, and provide two stream of data called `:data` and `:label`, respectively. You can change that by specifying the `tops` property for the HDF5 data layer if needed.

tops This specify a list of names for the output of the convolution layer. In this case, we are only taking one stream of input and after convolution, we output on stream of convolved data with the name `:conv`.

Another convolution layer and pooling layer are defined similarly, with more filters this time:

```
pool_layer = PoolingLayer(kernel=(2,2), stride=(2,2), bottoms=[:conv], tops=[:pool])
conv2_layer = ConvolutionLayer(name="conv2", n_filter=50, kernel=(5,5), bottoms=[:pool], tops=[:conv2
```

Note the `tops` and `bottoms` define the computation or data dependency. After the convolution and pooling layers, we add two fully connected layers. They are called `InnerProductLayer` because the computation is basically inner products between the input and the layer weights. The layer weights are also learned, so we also give names to the two layers:

```
fc1_layer  = InnerProductLayer(name="ip1", output_dim=500, neuron=Neurons.ReLU(), bottoms=[:pool2], t
fc2_layer  = InnerProductLayer(name="ip2", output_dim=10, bottoms=[:ip1], tops=[:ip2])
```

Everything should be self-evidence. The `output_dim` property of an inner product layer specify the dimension of the output. Note the dimension of the input is automatically determined from the bottom data stream.

Note for the first inner product layer, we specifies a Rectified Learning Unit (ReLU) activation function via the `neuron` property. An activation function could be added to almost all computation layers (e.g. convolution layer). By default, no activation function, or the *identity activation function* is used. We don't use activation function for the last inner product layer, because that layer acts as a linear classifier.

This is the basic structure of LeNet. In order to train this network, we need to define a loss function. This is done by adding a loss layer:

```
loss_layer = SoftmaxLossLayer(bottoms=[:ip2,:label])
```

Note this softmax loss layer takes as input `:ip2`, which is the output of the last inner product layer, and `:label`, which comes directly from the HDF5 data layer. It will compute an averaged loss over each mini batch, which allows us to initiate back propagation to update network parameters.

1.1.3 Setup Backend and Build Network

Now we have defined all the relevant layers. Let's setup the computation backend and construct a network with those layers. In this example, we will go with the simple pure Julia CPU backend first:

```
sys = System(CPUBackend())
init(sys)
```

The `init` function of a Mocha `System` will initialize the computation backend. With an initialized system, we could go ahead and construct our network:

```
common_layers = [conv_layer, pool_layer, conv2_layer, pool2_layer, fc1_layer, fc2_layer]
net = Net(sys, [data_layer, common_layers..., loss_layer])
```

A network is built by passing the constructor an initialized system, and a list of layers. Note we use `common_layers` to collect a subset of the layers. We will explain this in a minute.

1.1.4 Setup Solver

We will use Stochastic Gradient Descent (SGD) to solve or train our deep network.

```
params = SolverParameters(max_iter=10000, regu_coef=0.0005, base_lr=0.01, momentum=0.9,
    lr_policy=LRPolicy.Inv(0.0001, 0.75))
solver = SGD(params)
```

The behavior of the solver is specified in the following parameters

`max_iter` Max number of iterations the solver will run to train the network.

`regu_coef` Regularization coefficient. By default, both the convolution layer and the inner product layer have L2 regularizers for their weights (and no regularization for bias). Those regularizations could be customized for each layer individually. The parameter here is just a global scaling factor for all the local regularization coefficients if any.

`base_lr` This is the base learning rate. Again this is a global scaling factor, and each layer could specify their own local learning rate.

`momentum` The momentum used in SGD. See the Caffe document for *rules of thumb* for setting the learning rate and momentum.

`lr_policy` The learning rate policy. In this example, we are using the `Inv` policy with gamma = 0.001 and power = 0.75. This policy will gradually shrink the learning rate, by setting it to base_lr * (1 + gamma * iter)$^{-\text{power}}$.

Now our solver is ready to go. But in order to give him a healthy working plan, we decided to allow him to have some coffee breaks.

1.1.5 Coffee Breaks for the Solver

To be continued ... the author went to coffee break...

1.2 Indices and tables

- *genindex*
- *modindex*
- *search*

[LeNet] Lecun, Y.; Bottou, L.; Bengio, Y.; Haffner, P., *Gradient-based learning applied to document recognition*, Proceedings of the IEEE, vol.86, no.11, pp.2278-2324, Nov 1998.