Mocha Documentation

Release 0.0.2

pluskid

December 19, 2014

Contents

Mocha is a Deep Learning framework for Julia.

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(name="train-data", 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. It is a good practice to give every layer a unique name, for the purpose of getting more informative debugging information when there is any potential issues.

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(name="pool1", 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], tops=[:ip1])
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. 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. For more details, see *Neurons (Activation Functions)*.

The output dimension of the last inner product layer is 10, which corresponds to the number of classes (digits 0~9) of our problem.

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(name="loss", 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 Configuring Backend and Building 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("MNIST-train", 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 Configuring Solver

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

```
params = SolverParameters(max_iter=10000, regu_coef=0.0005,
    momentum=0.9, lr_policy=LRPolicy.Inv(0.01, 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.

`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)$^{\text{-power}}$.

1.1.5 Coffee Breaks for the Solver

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

```
add_coffee_break(solver, TrainingSummary(), every_n_iter=100)
```

First of all, we allow the solver to have a coffee break after every 100 iterations so that he could give us a brief summary of the training process. Currently `TrainingSummary` will print the loss function value on the last training mini-batch.

We also add a coffee break to save a snapshot for the trained network every 5,000 iterations.

```
add_coffee_break(solver,
    Snapshot("snapshots", auto_load=true), every_n_iter=5000)
```

Here `"snapshots"` is the name of the directory you want to save snapshots to. By setting `auto_load` to true, Mocha will automatically search and resume from the last saved snapshots.

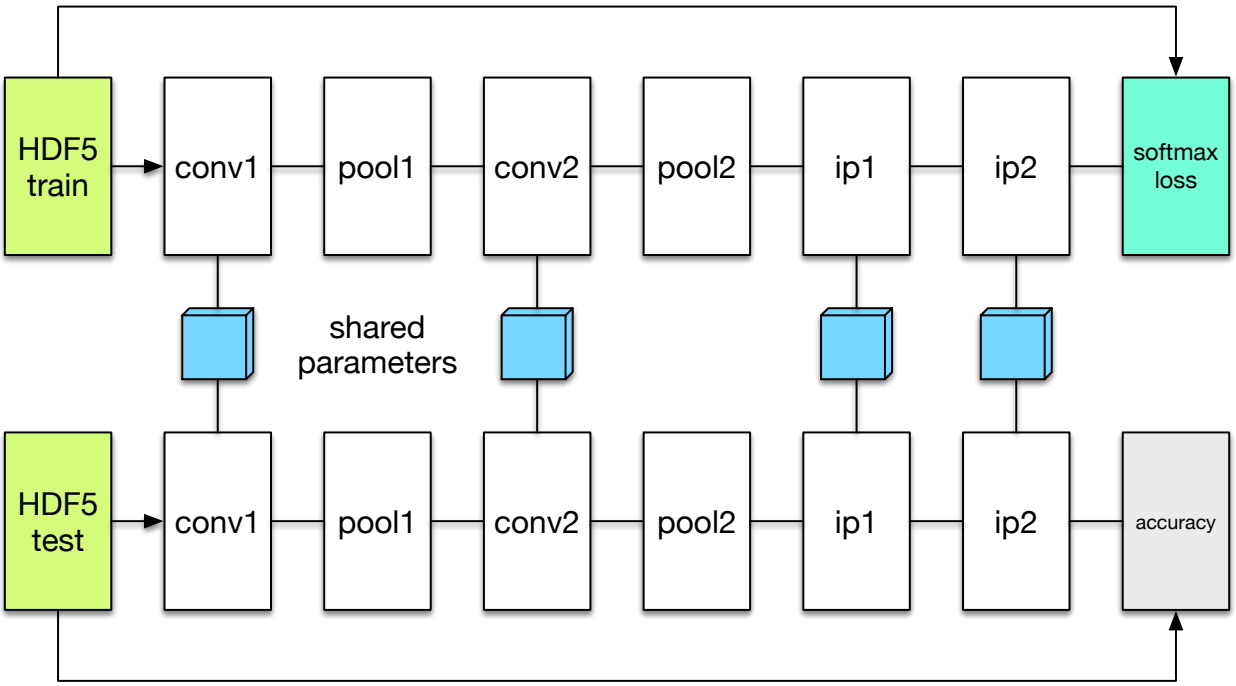
If you additionally set `also_load_solver_state` to false, Mocha will load the saved network as initialization, but pretend to be training from scratch. This could be useful if you are fine tuning based on some pre-trained network.

In order to see whether we are really making progress or simply overfitting, we also wish to see the performance on a separate validation set periodically. In this example, we simply use the test dataset as the validation set.

We will define a new network to perform the evaluation. The evaluation network will have exactly the same architecture, except with a different data layer that reads from validation dataset instead of training set. We also do not need the softmax loss layer as we will not train the validation network. Instead, we will add an accuracy layer on the top, which will compute the classification accuracy for us.

```
data_layer_test = HDF5DataLayer(name="test-data", source="data/test.txt", batch_size=100)
acc_layer = AccuracyLayer(name="test-accuracy", bottoms=[:ip2, :label])
test_net = Net("MNIST-test", sys, [data_layer_test, common_layers..., acc_layer])
```

Note how we re-use the `common_layers` variable defined a moment ago to reuse the description of the network architecture. By passing **the same** layer object used to define the training net to the constructor of the validation net, Mocha will be able to automatically setup parameter sharing between the two networks. The two networks will look like this:



Now we are ready to add another coffee break to report the validation performance:

```
add_coffee_break(solver, ValidationPerformance(test_net), every_n_iter=1000)
```

Please note we use a different batch size (100) in the validation network. During the coffee break, Mocha will run exactly one epoch on the validation net (100 iterations in our case, as we have 10,000 samples in MNIST test set), and report the average classification accuracy. You do not need to specify the number of iterations here as the HDF5 data layer will report epoch number as it goes through a full pass of the whole dataset.

1.1.6 Training

Without further due, we could finally start the training process:

```
solve(solver, net)

destroy(net)
```

```
destroy(test_net)
shutdown(sys)
```

After training, we will shutdown the system to release all the allocated resources. Now you are ready run the script

```
julia mnist.jl
```

As training goes on, you will see training progress printed. It will take about 10~20 seconds every 100 iterations on my machine depending on the server load and many factors.

```
14-Nov 11:56:13:INFO:root:001700 :: TRAIN obj-val = 0.43609169
14-Nov 11:56:36:INFO:root:001800 :: TRAIN obj-val = 0.21899594
14-Nov 11:56:58:INFO:root:001900 :: TRAIN obj-val = 0.19962406
14-Nov 11:57:21:INFO:root:002000 :: TRAIN obj-val = 0.06982464
14-Nov 11:57:40:INFO:root:
14-Nov 11:57:40:INFO:root:## Performance on Validation Set
14-Nov 11:57:40:INFO:root:---------------------------------------------------------
14-Nov 11:57:40:INFO:root:  Accuracy (avg over 10000) = 96.0500%
14-Nov 11:57:40:INFO:root:---------------------------------------------------------
14-Nov 11:57:40:INFO:root:
14-Nov 11:58:01:INFO:root:002100 :: TRAIN obj-val = 0.18091436
14-Nov 11:58:21:INFO:root:002200 :: TRAIN obj-val = 0.14225903
```

The training could run faster by enabling native extension for the CPU backend, or use a CUDA backend if CUDA compatible GPU devices are available. Please refer to *Mocha Backends* for how to use different backends.

Just to give you a feeling, this is a sample log from running with Native Extension enabled CPU backend. It takes about 5 seconds to run 100 iterations.

```
14-Nov 12:15:56:INFO:root:001700 :: TRAIN obj-val = 0.82937032
14-Nov 12:16:01:INFO:root:001800 :: TRAIN obj-val = 0.35497263
14-Nov 12:16:06:INFO:root:001900 :: TRAIN obj-val = 0.31351241
14-Nov 12:16:11:INFO:root:002000 :: TRAIN obj-val = 0.10048970
14-Nov 12:16:14:INFO:root:
14-Nov 12:16:14:INFO:root:## Performance on Validation Set
14-Nov 12:16:14:INFO:root:---------------------------------------------------------
14-Nov 12:16:14:INFO:root:  Accuracy (avg over 10000) = 94.5700%
14-Nov 12:16:14:INFO:root:---------------------------------------------------------
14-Nov 12:16:14:INFO:root:
14-Nov 12:16:18:INFO:root:002100 :: TRAIN obj-val = 0.20689486
14-Nov 12:16:23:INFO:root:002200 :: TRAIN obj-val = 0.17757215
```

The followings are a sample log from running with the CUDA backend. It runs about 300 iterations per second.

```
14-Nov 12:57:07:INFO:root:001700 :: TRAIN obj-val = 0.33347249
14-Nov 12:57:07:INFO:root:001800 :: TRAIN obj-val = 0.16477060
14-Nov 12:57:07:INFO:root:001900 :: TRAIN obj-val = 0.18155883
14-Nov 12:57:08:INFO:root:002000 :: TRAIN obj-val = 0.06635486
14-Nov 12:57:08:INFO:root:
14-Nov 12:57:08:INFO:root:## Performance on Validation Set
14-Nov 12:57:08:INFO:root:---------------------------------------------------------
14-Nov 12:57:08:INFO:root:  Accuracy (avg over 10000) = 96.2200%
14-Nov 12:57:08:INFO:root:---------------------------------------------------------
14-Nov 12:57:08:INFO:root:
14-Nov 12:57:08:INFO:root:002100 :: TRAIN obj-val = 0.20724633
14-Nov 12:57:08:INFO:root:002200 :: TRAIN obj-val = 0.14952177
```

1.1.7 Remarks

The accuracy from two different trains are different due to different random initialization. The objective function values shown here are also slightly different to Caffe's, as until recently, Mocha counts regularizers in the forward stage and add them into objective functions. This behavior is removed to avoid unnecessary computation in more recent versions of Mocha.

1.2 Alex's CIFAR-10 tutorial in Mocha

This example is converted from Caffe's CIFAR-10 tutorials, which was originally built based on details from Alex Krizhevsky's cuda-convnet. In this example, we will demonstrate how to translate a network definition in Caffe to Mocha, and train the network to roughly reproduce the test error rate of 18% (without data augmentation) as reported in Alex Krizhevsky's website.

The CIFAR-10 dataset is a labeled subset of the 80 Million Tiny Images dataset, containing 60,000 32x32 color images in 10 categories. They are split into 50,000 training images and 10,000 test images. The number of samples are the same to *the MNIST example*. However, the images here are a bit larger and have 3 channels. As we will see soon, the network is also larger, with one extra convolution and pooling and two local response normalization layers. It is recommended to read *the MNIST tutorial* first, as we will not repeat many details here.

1.2.1 Caffe's Tutorial and Code

Caffe's tutorial for CIFAR-10 can be found on their website. The code could be located in `examples/cifar10` under Caffe's source tree. The code folder contains several different definition of networks and solvers. The filenames should be self-explanatory. The *quick* files corresponds to a smaller network without local response normalization layers. And this is documented in Caffe's tutorial, according to which, produces around 75% test accuracy.

We will be using the *full* models, which gives us around 81% test accuracy. Caffe's definition of the full model could be found in the file cifar10_full_train_test.prototxt. The training script is train_full.sh, which trains in 3 different stages with solvers defined in

1. cifar10_full_solver.prototxt
2. cifar10_full_solver_lr1.prototxt
3. cifar10_full_solver_lr2.prototxt

respectively. This looks complicated. But if you compare the files, you will find that the three stages are basically using the same solver configurations except with a ten-fold learning rate decrease after each stage.

1.2.2 Preparing the Data

Looking at the data layer of Caffe's network definition, it uses a LevelDB database as a data source. The LevelDB database is converted from the original binary files downloaded from the CIFAR-10 dataset's website. Mocha does not support LevelDB database, so we will do the same thing: download the original binary files and convert into a Mocha-recognizable data format, HDF5 dataset here. We have provided a Julia script convert.jl [1]. You can call get-cifar10.sh directly, which will automatically download the binary files, convert it to HDF5 and prepare text index files that points to the HDF5 datasets.

Notice in Caffe's data layer, a `transform_param` is specified with a `mean_file`. Since we need to compute the data mean during data conversion, for simplicity, we also perform mean subtraction when converting data to HDF5 format. See convert.jl for details. Please refer to the *user's guide* for more details about HDF5 data format that Mocha reads.

[1] All the CIFAR-10 example related code in Mocha could be found in the `examples/cifar10` directory under the source tree.

After converting the data, you should be ready to load the data in Mocha with `HDF5DataLayer`. We define two layers for training data and test data separately, using the same batch size as in Caffe's model definition:

```
data_tr_layer = HDF5DataLayer(name="data-train", source="data/train.txt", batch_size=100)
data_tt_layer = HDF5DataLayer(name="data-test", source="data/test.txt", batch_size=100)
```

In order to share the definition of common computation layers, Caffe use the same file to define both the training and test networks, and use *phase* to include and exclude layers that are used only in training or testing phases. Mocha does not do this as the layers defined in Julia code are just Julia objects. We will simply construct training and test nets with a different subsets of those Julia layer objects.

1.2.3 Computation and Loss Layers

Translating the computation layers should be straightforward. For example, the `conv1` layer is defined in Caffe as

```
layers {
  name: "conv1"
  type: CONVOLUTION
  bottom: "data"
  top: "conv1"
  blobs_lr: 1
  blobs_lr: 2
  convolution_param {
    num_output: 32
    pad: 2
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "gaussian"
      std: 0.0001
    }
    bias_filler {
      type: "constant"
    }
  }
}
```

This translates to Mocha as:

```
conv1_layer = ConvolutionLayer(name="conv1", n_filter=32, kernel=(5,5), pad=(2,2),
    stride=(1,1), filter_init=GaussianInitializer(std=0.0001),
    bottoms=[:data], tops=[:conv1])
```

Tip:

- The `pad`, `kernel_size` and `stride` parameters in Caffe means the same pad for both the *width* and *height* dimension unless specified explicitly. In Mocha, we always explicitly use a 2-tuple to specify the parameters for the two dimensions.
- A *filler* in Caffe corresponds to an *initializer* in Mocha.
- Mocha has a constant initializer (initialize to 0) for the bias by default, so we do not need to specify it explicitly.

The rest of the translated Mocha computation layers are listed here:

```
pool1_layer = PoolingLayer(name="pool1", kernel=(3,3), stride=(2,2), neuron=Neurons.ReLU(),
    bottoms=[:conv1], tops=[:pool1])
norm1_layer = LRNLayer(name="norm1", kernel=3, scale=5e-5, power=0.75, mode=LRNMode.WithinChannel(),
```

```
    bottoms=[:pool1], tops=[:norm1])
conv2_layer = ConvolutionLayer(name="conv2", n_filter=32, kernel=(5,5), pad=(2,2),
    stride=(1,1), filter_init=GaussianInitializer(std=0.01),
    bottoms=[:norm1], tops=[:conv2], neuron=Neurons.ReLU())
pool2_layer = PoolingLayer(name="pool2", kernel=(3,3), stride=(2,2), pooling=Pooling.Mean(),
    bottoms=[:conv2], tops=[:pool2])
norm2_layer = LRNLayer(name="norm2", kernel=3, scale=5e-5, power=0.75, mode=LRNMode.WithinChannel(),
    bottoms=[:pool2], tops=[:norm2])
conv3_layer = ConvolutionLayer(name="conv3", n_filter=64, kernel=(5,5), pad=(2,2),
    stride=(1,1), filter_init=GaussianInitializer(std=0.01),
    bottoms=[:norm2], tops=[:conv3], neuron=Neurons.ReLU())
pool3_layer = PoolingLayer(name="pool3", kernel=(3,3), stride=(2,2), pooling=Pooling.Mean(),
    bottoms=[:conv3], tops=[:pool3])
ip1_layer   = InnerProductLayer(name="ip1", output_dim=10, weight_init=GaussianInitializer(std=0.01),
    weight_regu=L2Regu(250), bottoms=[:pool3], tops=[:ip1])
```

You might have already noticed is that Mocha does not have a ReLU layer. Instead, ReLU, like Sigmoid, are treated as *neurons or activation functions* attached to layers.

1.2.4 Constructing the Network

In order to train the network, we need to define a loss layer. We also define an accuracy layer to be used in the test network for us to see how our network performs on the test dataset during training. Translating directly from Caffe's definitions:

```
loss_layer  = SoftmaxLossLayer(name="softmax", bottoms=[:ip1, :label])
acc_layer   = AccuracyLayer(name="accuracy", bottoms=[:ip1, :label])
```

Next we collect the layers, and define a Mocha `Net` on a `CuDNNBackend`. You could use `CPUBackend` if no CUDA-compatible GPU devices are available. But it will be much slower (see also *Mocha Backends*).

```
common_layers = [conv1_layer, pool1_layer, norm1_layer, conv2_layer, pool2_layer, norm2_layer,
                 conv3_layer, pool3_layer, ip1_layer]

sys = System(CuDNNBackend())
#sys = System(CPUBackend())
init(sys)

net = Net("CIFAR10-train", sys, [data_tr_layer, common_layers..., loss_layer])
```

1.2.5 Configuring the Solver

The configuration for Caffe's solver looks like this

```
# reduce learning rate after 120 epochs (60000 iters) by factor 0f 10
# then another factor of 10 after 10 more epochs (5000 iters)

# The train/test net protocol buffer definition
net: "examples/cifar10/cifar10_full_train_test.prototxt"
# test_iter specifies how many forward passes the test should carry out.
# In the case of CIFAR10, we have test batch size 100 and 100 test iterations,
# covering the full 10,000 testing images.
test_iter: 100
# Carry out testing every 1000 training iterations.
test_interval: 1000
# The base learning rate, momentum and the weight decay of the network.
```

```
base_lr: 0.001
momentum: 0.9
weight_decay: 0.004
# The learning rate policy
lr_policy: "fixed"
# Display every 200 iterations
display: 200
# The maximum number of iterations
max_iter: 60000
# snapshot intermediate results
snapshot: 10000
snapshot_prefix: "examples/cifar10/cifar10_full"
# solver mode: CPU or GPU
solver_mode: GPU
```

First of all, the learning rate is drop by a factor of 10 [2]. Caffe implements this by having three solver configurations with different learning rate for each stage. We could do the same thing for Mocha, but Mocha has a staged learning policy that makes this easier:

```
lr_policy = LRPolicy.Staged(
  (60000, LRPolicy.Fixed(0.001)),
  (5000, LRPolicy.Fixed(0.0001)),
  (5000, LRPolicy.Fixed(0.00001)),
)
solver_params = SolverParameters(max_iter=70000,
    regu_coef=0.004, momentum=0.9, lr_policy=lr_policy)
solver = SGD(solver_params)
```

The other parameters like regularization coefficient, momentum are directly translated from Caffe's solver configuration. Progress report, automatic snapshots could equivalently be done in Mocha as *coffee breaks* for the solver:

```
# report training progress every 200 iterations
add_coffee_break(solver, TrainingSummary(), every_n_iter=200)

# save snapshots every 5000 iterations
add_coffee_break(solver,
    Snapshot("snapshots", auto_load=true),
    every_n_iter=5000)

# show performance on test data every 1000 iterations
test_net = Net("CIFAR10-test", sys, [data_tt_layer, common_layers..., acc_layer])
add_coffee_break(solver, ValidationPerformance(test_net), every_n_iter=1000)
```

1.2.6 Training

Now we could start training by calling `solve(solver, net)`. Depending on different *backends*, the training speed could vary. Here are some sample training logs from my own test. Note this is **not** a controlled comparison, just to get a rough feeling.

Pure Julia on CPU

The training is quite slow on a pure Julia backend. It takes about 15 minutes to run every 200 iterations.

[2] Looking at the Caffe's solver configuration, I happily realized that I am not the only person in the world who sometimes mis-type o as 0. :P

```
20-Nov 06:58:26:INFO:root:004600 :: TRAIN obj-val = 1.07695698
20-Nov 07:13:25:INFO:root:004800 :: TRAIN obj-val = 1.06556938
20-Nov 07:28:26:INFO:root:005000 :: TRAIN obj-val = 1.15177973
20-Nov 07:30:35:INFO:root:
20-Nov 07:30:35:INFO:root:## Performance on Validation Set
20-Nov 07:30:35:INFO:root:---------------------------------------------------------
20-Nov 07:30:35:INFO:root:  Accuracy (avg over 10000) = 62.8200%
20-Nov 07:30:35:INFO:root:---------------------------------------------------------
20-Nov 07:30:35:INFO:root:
20-Nov 07:45:33:INFO:root:005200 :: TRAIN obj-val = 0.93760641
20-Nov 08:00:30:INFO:root:005400 :: TRAIN obj-val = 0.95650533
20-Nov 08:15:29:INFO:root:005600 :: TRAIN obj-val = 1.03291103
20-Nov 08:30:21:INFO:root:005800 :: TRAIN obj-val = 1.01833960
20-Nov 08:45:17:INFO:root:006000 :: TRAIN obj-val = 1.10167430
20-Nov 08:47:27:INFO:root:
20-Nov 08:47:27:INFO:root:## Performance on Validation Set
20-Nov 08:47:27:INFO:root:---------------------------------------------------------
20-Nov 08:47:27:INFO:root:  Accuracy (avg over 10000) = 64.7100%
20-Nov 08:47:27:INFO:root:---------------------------------------------------------
20-Nov 08:47:27:INFO:root:
20-Nov 09:02:24:INFO:root:006200 :: TRAIN obj-val = 0.88323826
```

CPU with Native Extension

We enabled Mocha's native extension, but disabled OpenMP by setting the OMP number of threads to 1:

```
ENV["OMP_NUM_THREADS"] = 1
blas_set_num_threads(1)
```

According to the log, it takes roughly 160 seconds to finish every 200 iterations.

```
20-Nov 09:29:10:INFO:root:000800 :: TRAIN obj-val = 1.46420457
20-Nov 09:31:48:INFO:root:001000 :: TRAIN obj-val = 1.63248945
20-Nov 09:32:22:INFO:root:
20-Nov 09:32:22:INFO:root:## Performance on Validation Set
20-Nov 09:32:22:INFO:root:---------------------------------------------------------
20-Nov 09:32:22:INFO:root:  Accuracy (avg over 10000) = 44.4300%
20-Nov 09:32:22:INFO:root:---------------------------------------------------------
20-Nov 09:32:22:INFO:root:
20-Nov 09:35:00:INFO:root:001200 :: TRAIN obj-val = 1.33312901
20-Nov 09:37:38:INFO:root:001400 :: TRAIN obj-val = 1.40529397
20-Nov 09:40:16:INFO:root:001600 :: TRAIN obj-val = 1.26366557
20-Nov 09:42:54:INFO:root:001800 :: TRAIN obj-val = 1.29758151
20-Nov 09:45:32:INFO:root:002000 :: TRAIN obj-val = 1.40923050
20-Nov 09:46:06:INFO:root:
20-Nov 09:46:06:INFO:root:## Performance on Validation Set
20-Nov 09:46:06:INFO:root:---------------------------------------------------------
20-Nov 09:46:06:INFO:root:  Accuracy (avg over 10000) = 51.0400%
20-Nov 09:46:06:INFO:root:---------------------------------------------------------
20-Nov 09:46:06:INFO:root:
20-Nov 09:48:44:INFO:root:002200 :: TRAIN obj-val = 1.24579735
20-Nov 09:51:22:INFO:root:002400 :: TRAIN obj-val = 1.22985339
```

We also tried to use multi-thread computing:

```
ENV["OMP_NUM_THREADS"] = 16
blas_set_num_threads(16)
```

By using 16 cores to compute, I got very slight improvement (which may well due to external factors as I did not control the comparison environment at all), with roughly 150 seconds every 200 iterations. I did not try multi-thread computing with less or more threads.

```
20-Nov 10:29:34:INFO:root:002400 :: TRAIN obj-val = 1.25820349
20-Nov 10:32:04:INFO:root:002600 :: TRAIN obj-val = 1.22480259
20-Nov 10:34:32:INFO:root:002800 :: TRAIN obj-val = 1.25739809
20-Nov 10:37:02:INFO:root:003000 :: TRAIN obj-val = 1.32196600
20-Nov 10:37:36:INFO:root:
20-Nov 10:37:36:INFO:root:## Performance on Validation Set
20-Nov 10:37:36:INFO:root:---------------------------------------------------------
20-Nov 10:37:36:INFO:root:  Accuracy (avg over 10000) = 56.4300%
20-Nov 10:37:36:INFO:root:---------------------------------------------------------
20-Nov 10:37:36:INFO:root:
20-Nov 10:40:06:INFO:root:003200 :: TRAIN obj-val = 1.17503929
20-Nov 10:42:40:INFO:root:003400 :: TRAIN obj-val = 1.13562913
20-Nov 10:45:09:INFO:root:003600 :: TRAIN obj-val = 1.17141657
20-Nov 10:47:40:INFO:root:003800 :: TRAIN obj-val = 1.20520208
20-Nov 10:50:12:INFO:root:004000 :: TRAIN obj-val = 1.24686298
20-Nov 10:50:47:INFO:root:
20-Nov 10:50:47:INFO:root:## Performance on Validation Set
20-Nov 10:50:47:INFO:root:---------------------------------------------------------
20-Nov 10:50:47:INFO:root:  Accuracy (avg over 10000) = 59.4500%
20-Nov 10:50:47:INFO:root:---------------------------------------------------------
20-Nov 10:50:47:INFO:root:
20-Nov 10:53:16:INFO:root:004200 :: TRAIN obj-val = 1.11022978
20-Nov 10:55:49:INFO:root:004400 :: TRAIN obj-val = 1.04538457
```

CUDA with cuDNN

It takes roughly 10 seconds to finish every 200 iterations on the `CuDNNBackend`.

```
20-Nov 01:16:48:INFO:root:001400 :: TRAIN obj-val = 1.47859097
20-Nov 01:16:57:INFO:root:001600 :: TRAIN obj-val = 1.33097243
20-Nov 01:17:07:INFO:root:001800 :: TRAIN obj-val = 1.33654988
20-Nov 01:17:16:INFO:root:002000 :: TRAIN obj-val = 1.50953197
20-Nov 01:17:18:INFO:root:
20-Nov 01:17:18:INFO:root:## Performance on Validation Set
20-Nov 01:17:18:INFO:root:---------------------------------------------------------
20-Nov 01:17:18:INFO:root:  Accuracy (avg over 10000) = 50.2300%
20-Nov 01:17:18:INFO:root:---------------------------------------------------------
20-Nov 01:17:18:INFO:root:
20-Nov 01:17:27:INFO:root:002200 :: TRAIN obj-val = 1.29346514
20-Nov 01:17:37:INFO:root:002400 :: TRAIN obj-val = 1.32249010
20-Nov 01:17:46:INFO:root:002600 :: TRAIN obj-val = 1.27704692
20-Nov 01:17:56:INFO:root:002800 :: TRAIN obj-val = 1.25375235
20-Nov 01:18:05:INFO:root:003000 :: TRAIN obj-val = 1.38656604
20-Nov 01:18:07:INFO:root:
20-Nov 01:18:07:INFO:root:## Performance on Validation Set
20-Nov 01:18:07:INFO:root:---------------------------------------------------------
20-Nov 01:18:07:INFO:root:  Accuracy (avg over 10000) = 56.6100%
20-Nov 01:18:07:INFO:root:---------------------------------------------------------
```

CHAPTER 2

User's Guide

2.1 Networks

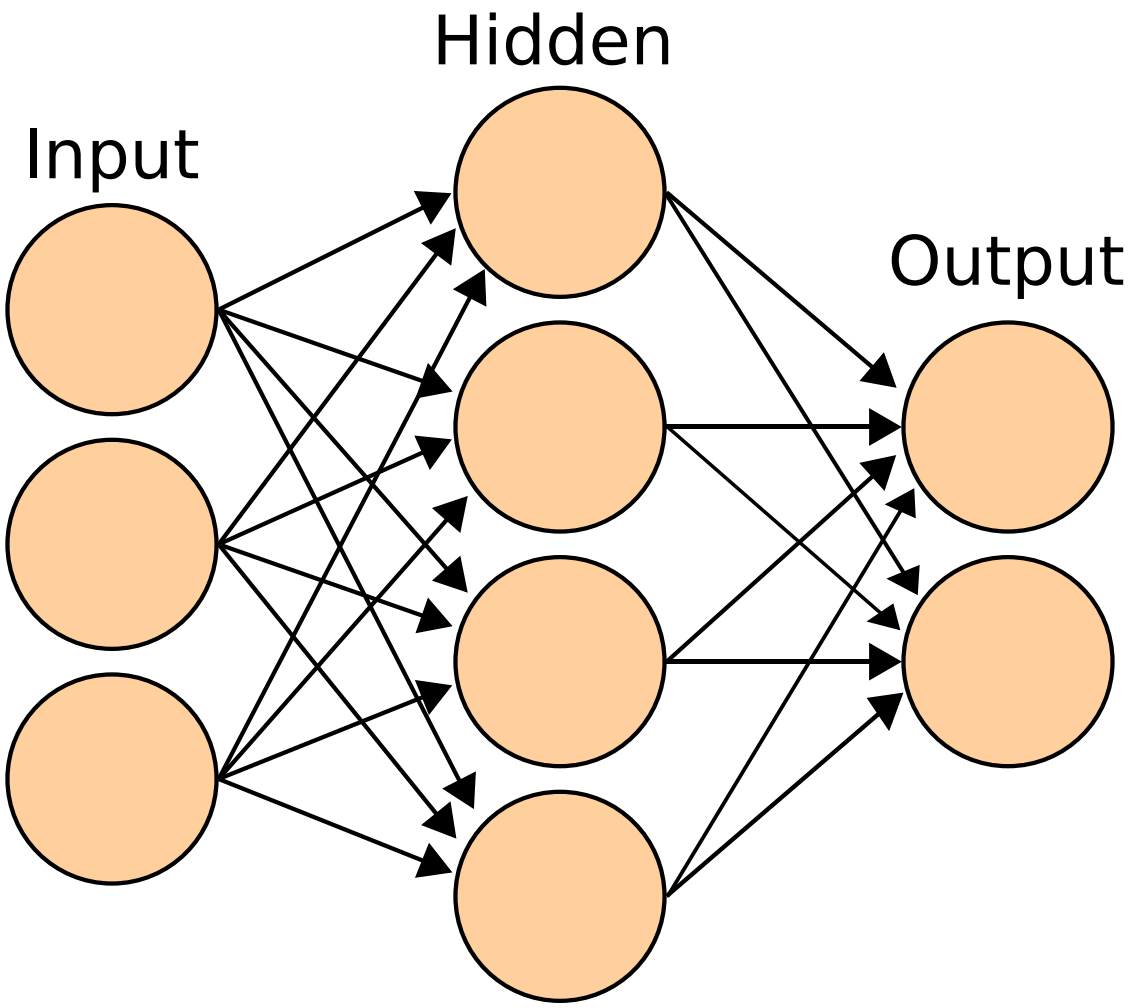
2.1.1 Overview

In deep learning, computations are abstracted into relatively isolated *layers*. The layers are connected together according to a given *architecture* that describes a data flow. Starting with the data layer: it takes input from a dataset or user input, do some data pre-processing, and then produce a stream of processed data. The output of the data layer is connected to the input of some computation layer, which again produces a stream of computed output that gets connected to the input of some upper layers. At the top of a network, there is typically a layer that produces the network prediction or compute the loss function value according to provided ground-truth labels.

During training, the same data path, except in the reversed direction, is used to propagate the error back to each layers using chain rules. Via back propagation, each layer could compute the gradients for its own parameters, and update the parameters according to some optimization schemes. Again, the computation is abstracted into layers.

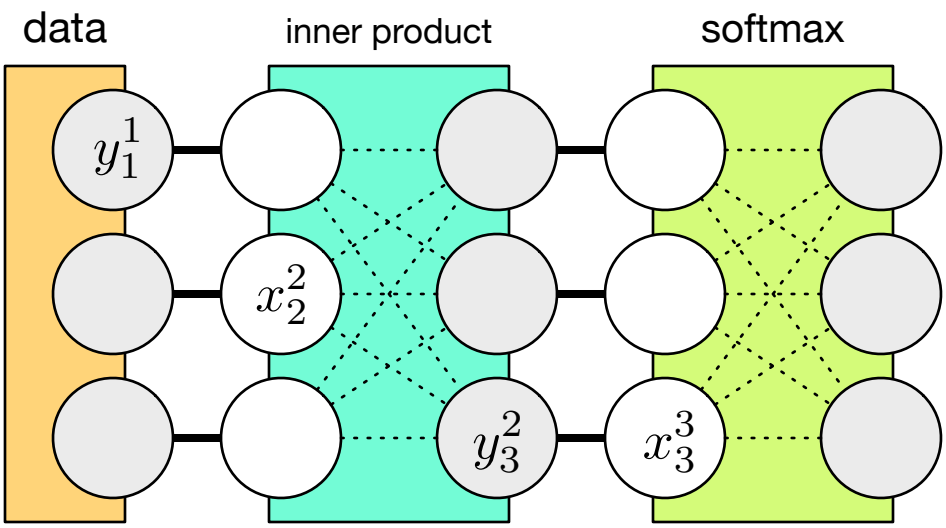
The abstraction and separating *layers* from *architecture* are important. The library implementation could focus on each layer type independently, and does not need to worry about how those layers are going to be connected with each other. On the other hand, the network designer could focus on the architecture, and does not need to worry about the internal computations of layers. This enables us to compose layers almost arbitrarily to create very deep / complicated networks. The network could be carrying out highly sophisticated computations when viewed as a whole, yet all the complexities are nicely decomposed into manageable pieces.

Most of the illustrations for (deep) neural networks look like the following image stolen from Wikipedia's page on Artificial Neural Networks:



When writing Mocha, I found this kind of illustrations a bit confusing, as it does not align well with the abstract concept of *layers* we just described. In our abstraction, the computation is done **within** each layers, and the network architecture specifies the data path connections for the layers only. In the figure above, the "Input", "Hidden", and "Output" labels are put on the nodes, suggesting the nodes are layers. However, the nodes do not do computation, instead, computations are specified by the arrows connecting those nodes.

On the other hand, I think the following kind of illustration is clearer, for the purpose of abstracting *layers* and *architectures* separately:



Each layer is now represented as a *box* that has inputs (denoted by x^L for the L-th layer) and outputs (denoted by y^L). Now the architecture specifies which layer's outputs connect to which layer's inputs (the dark lines in the figure). On the other hand, the intra-layer connections, or computations (see dotted line in the figure) should be isolated from the outside world.

Note: Unlike the intra-layer connections, the inter-layer connections are drawn as simple parallel lines, because they are essentially a point-wise copying operation. Because all the computations are abstracted to be inside the layers, there is no real computation in between them. Mathematically, this means $y^L = x^{L+1}$. In actual implementation, data copying is avoided via data sharing.

Of course, the choice is only a matter of taste, but as we will see, using the latter kind of illustration makes it much easier to understand Mocha's internal structure and end-user interface.

2.1.2 Network Architecture

Specifying a network architecture in Mocha means defining a set of layers, and connecting them. Taking the figure above for example, we could define a data layer and an inner product layer

```
data_layer = HDF5DataLayer(name="data", source="data-list.txt", batch_size=64, tops=[:data])
ip_layer   = InnerProductLayer(name="ip", output_dim=500, tops=[:ip], bottoms=[:data])
```

Note the `tops` and `bottoms` properties give names to the output and input of the layer. Since the name for the input of `ip_layer` matches the name for the output of `data_layer`, they will be connected as shown in the figure above. The softmax layer could be defined similarly. Mocha will do a topological sort on the collection of layers and automatically figure out the connection defined implicitly by the names of the inputs and outputs of each layer.

2.1.3 Layer Implementation

The layer is completely unaware of what happens in the outside world. Two important procedures need to be defined to implement a layer:

- Feed-forward: given the inputs, compute the outputs. For example, for the inner product layer, it will compute the outputs as $y_i = \sum_j w_{ij} x_j$.
- Back-propagate: given the errors propagated from upper layers, compute the gradient of the layer parameters, **and** propagate the error down to lower layers. Note this is described in very vague terms like *errors*. Given the

abstraction we choose here, those vague terms could become very clear.

Specifically, back-propagation is used during network training, when an optimization algorithm want to compute the gradient of each parameter with respect to an *objective function*. Typically, the objective function is some loss function that penalize incorrect predictions given the ground-truth labels. Let's call the objective function f.

Now let's switch to the viewpoint of an inner product layer: it needs to compute the gradients of the weights parameters w with respect to f. Of course, since we restrict the layer from accessing the outside world, it does not know what f is. But the gradients could be computed via chain rule

$$\frac{\partial f}{\partial w_{ij}} = \frac{\partial y_i}{\partial w_{ij}} \times \frac{\partial f}{\partial y_i}$$

The red part could be computed **within** the layer, and the blue part is the so called "errors propagated from the upper layers". It comes from the reversed data path as used in the feed-forward pass.

Now our inner product layer is ready to "propagate the errors down to lower layers", precisely speaking, this means computing

$$\frac{\partial f}{\partial x_i} = \sum_j \frac{\partial y_j}{\partial x_i} \times \frac{\partial f}{\partial y_j}$$

Again, this is decomposed into a part that could be computed internally and a part that comes from the "top". Recall we said the L-th layer's inputs x_i^L is equal to the $(L-1)$-th layer's outputs y_i^{L-1}. That means what we just computed

$$\frac{\partial f}{\partial x_i^L} = \frac{\partial f}{\partial y_i^{L-1}}$$

is exactly what the lower layer's "errors propagated from upper layers". By tracing the whole data path reversely, we now help each layers compute the gradients of their own parameters internally. And this is called back-propagation.

2.2 Layers

2.2.1 Overview

There are four basic layer types in Mocha:

Data Layers Read data from source and feed them to top layers.

Computation Layer Take input stream from bottom layers, carry out computations and feed the computed results to top layers.

Loss Layers Take computed results (and ground truth labels) from bottom layers, compute a real number loss. Loss values from all the loss layers and regularizers in a net are added together to define the final loss function of the net. The loss function will be used to train the net parameters in back propagation.

Statistics Layers Take input from bottom layers and compute useful statistics like classification accuracy. Statistics could be accumulated throughout multiple iterations.

2.2.2 Data Layers

class `HDF5DataLayer`

Load data from a list of HDF5 files and feed them to upper layers in mini batches. The layer will do automatic round wrapping and report epochs after going over a full round of list data sources. Currently randomization is not supported.

Each *dataset* in the HDF5 file should be a 4D tensor. Using the naming convention for image datasets, the four dimensions are (width, height, channels, number). Here the fastest changing dimension is *width*, while the slowest changing dimension is *number*. Mini-batch splitting will occur in the *number* dimension. For more details for 4D tensor blobs used in Mocha, see *Blob*.

Currently, the dataset should be explicitly in 4D tensor format. For example, if the label for each sample is only one number, the HDF5 dataset should still be created with dimension (1, 1, 1, number).

The numerical types of the HDF5 datasets should either be `Float32` or `Float64`. Even for multi-class labels, the integer class indicators should still be stored as floating point.

Note: For N class multi-class labels, the labels should be numerical values from 0 to N-1, even though Julia use 1-based indexing (See `SoftmaxLossLayer`).

The HDF5 dataset format is compatible with Caffe. If you want to compare the results of Mocha to Caffe on the same data, you could use Caffe's HDF5 Data Layer to read from the same HDF5 files Mocha is using.

`source`
File name of the data source. The source should be a text file, in which each line specifies a file name to a HDF5 file to load.

`batch_size`
The number of data samples in each mini batch.

`tops`
Default `[:data, :label]`. List of symbols, specifying the name of the blobs to feed to the top layers. The names also correspond to the datasets to load from the HDF5 files specified in the data source.

class `MemoryDataLayer`
Wrap an in-memory Julia Array as data source. Useful for testing.

`tops`
List of symbols, specifying the name of the blobs to produce.

`batch_size`
The number of data samples in each mini batch.

`data`
List of Julia Arrays. The count should be equal to the number of `tops`, where each Array acts as the data source for each blob.

2.2.3 Computation Layers

class `PoolingLayer`
2D pooling over the 2 image dimensions (width and height).

`kernel`
Default (1,1), a 2-tuple of integers specifying pooling kernel width and height, respectively.

`stride`
Default (1,1), a 2-tuple of integers specifying pooling stride in the width and height dimensions respectively.

`pad`
Default (0,0), a 2-tuple of integers specifying the padding in the width and height dimensions respectively. Paddings are two-sided, so a pad of (1,0) will pad one pixel in both the left and the right boundary of an image.

pooling
Default `Pooling.Max()`. Specify the pooling operation to use.

tops
bottoms
Blob names for output and input.

class LRNLayer
Local Response Normalization Layer. It performs normalization over local input regions via the following mapping

$$x \to y = \frac{x}{\left(\beta + (\alpha/n)\sum_{x_j \in N(x)} x_j^2\right)^p}$$

Here β is the shift, α is the scale, p is the power, and n is the size of the local neighborhood. $N(x)$ denotes the local neighborhood of x of size n (including x itself). There are two types of local neighborhood:

- `LRNMode.AcrossChannel()`: The local neighborhood is a region of shape (1, 1, k, 1) centered at x. In other words, the region extends across nearby channels (with zero padding if needed), but has no spatial extent. Here k is the kernel size, and $n = k$ in this case.
- `LRNMode.WithinChannel()`: The local neighborhood is a region of shape (k, k, 1, 1) centered at x. In other words, the region extends spatially (in **both** the width and the channel dimension), again with zero padding when needed. But it does not extend across different channels. In this case $n = k^2$.

kernel
Default 5, an integer indicating the kernel size. See k in the descriptions above.

scale
Default 1.

shift
Default 1 (yes, 1, not 0).

power
Default 0.75.

mode
Default `LRNMode.AcrossChannel()`.

tops
bottoms
Names for output and input blobs. Only one input and one output blob are allowed.

class ElementWiseLayer
Element-wise layer implements basic element-wise operations on inputs.

operation
Element-wise operation. Built-in operations are in module `ElementWiseFunctors`, including `Add`, `Subtract`, `Multiply` and `Divide`.

tops
Output blob names, only one output blob is allowed.

bottoms
Input blob names, count must match the number of inputs `operation` takes.

class PowerLayer
Power layer performs element-wise operations as

$$y = (ax + b)^p$$

where a is `scale`, b is `shift`, and p is `power`. During back propagation, the following element-wise derivatives are computed:

$$\frac{\partial y}{\partial x} = pa(ax+b)^{p-1}$$

Power layer is implemented separately instead of as an Element-wise layer for better performance because there are some many special cases of Power layer that could be computed more efficiently.

power
Default 1

scale
Default 1

shift
Default 0

tops
bottoms
Blob names for output and input.

class SplitLayer
Split layer produces identical *copies* [1] of the input. The number of copies is determined by the length of the `tops` property. During back propagation, derivatives from all the output copies are added together and propagated down.

This layer is typically used as a helper to implement some more complicated layers.

bottoms
Input blob names, only one input blob is allowed.

tops
Output blob names, should be more than one output blobs.

class ChannelPoolingLayer
1D pooling over the channel dimension.

kernel
Default 1, pooling kernel size.

stride
Default 1, stride for pooling.

pad
Default (0,0), a 2-tuple specifying padding in the front and the end.

pooling
Default `Pooling.Max()`. Specify the pooling function to use.

tops
bottoms
Blob names for output and input.

2.2.4 Loss Layers

class SoftmaxLossLayer
hmm

[1] All the data is shared, so there is no actually data copying.

2.3 Neurons (Activation Functions)

They could be attached to any layers. The neuron of each layer will affect the output in the forward pass and the gradient in the backward pass automatically unless it is an identity neuron. A layer have an identity neuron by default [2].

class `Neurons.`**`Identity`**
An activation function that does nothing.

class `Neurons.`**`ReLU`**
Rectified Linear Unit. During the forward pass, it inhibit all the negative activations. In other words, it compute point-wisely $y = \max(0, x)$. The point-wise derivative for ReLU is

$$\frac{dy}{dx} = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$$

Note: ReLU is actually not differentialble at 0. But it has *subdifferential* $[0, 1]$. Any value in that interval could be taken as a *subderivative*, and could be used in SGD if we generalize from gradient descent to *subgradient* descent. In the implementation, we choose 0.

---

class `Neurons.`**`Sigmoid`**
Sigmoid is a smoothed step function that produces approximate 0 for negative input with large absolute values and approximate 1 for large positive inputs. The point-wise formula is $y = 1/(1 + e^{-x})$. The point-wise derivative is

$$\frac{dy}{dx} = \frac{-e^{-x}}{(1 + e^{-x})^2} = (1 - y)y$$

2.4 Initializers

Initializers provide init values for network parameter blobs. In Caffe, they are called *Fillers*.

class **`NullInitializer`**
An initializer that does nothing.

class **`ConstantInitializer`**
Set everything to a constant.

`value`
The value used to initialize a parameter blob. Typically this is set to 0.

class **`XavierInitializer`**
An initializer based on [BengioGlorot2010], but does not use the fan-out value. It fills the parameter blob by randomly sampling uniform data from $[-S, S]$ where the scale $S = \sqrt{3/F_{\text{in}}}$. Here F_{in} is the fan-in: the number of input nodes. For a 4D tensor parameter blob with the shape (M, N, P, Q), M is considered as the fan-in.

class **`GaussianInitializer`**
Initialize each element in the parameter blob as independent and identically distributed Gaussian random variables.

`mean`
Default 0.

[2] This is actually not true: not all layers in Mocha support neurons. For example, data layers currently does not have neurons, but this feature could be added by simply adding a neuron property to the data layer type. However, for some layer types like loss layers or accuracy layers, it does not make much sense to have neurons.

std
Default 1.

2.5 Regularizers

Regularizers add extra penalties or constraints for network parameters to restrict the model complexity. The correspondences in Caffe are weight decays. Regularizers and weight decays are equivalent in back-propagation. The *conceptual* difference in the forward pass is that when treated as weight decay, they are not considered as parts of the objective function. However, in order to save computation, Mocha also omit forward computation for regularizers by default. We choose to use the term regularization instead of weight decay just because it is easier to understand when generalizing to sparse, group-sparse or even more complicated structural regularizations.

All regularizers have the property `coefficient`, corresponding to the regularization coefficient. During training, a global regularization coefficient can also be specified (see `user-guide/solver`), that globally scale all local regularization coefficients.

class NoRegu
Regularizer that impose no regularization.

class L2Regu
L2 regularizer. The parameter blob W is treated as a 1D vector. During the forward pass, the squared L2-norm $\|W\|^2 = \langle W, W \rangle$ is computed, and $\lambda \|W\|^2$ is added to the objective function, where λ is the regularization coefficient. During the backward pass, $2\lambda W$ is added to the parameter gradient, enforcing a weight decay when the solver moves the parameters towards the negative gradient direction.

Note in Caffe, only λW is added as a weight decay in back propagation, which is equivalent to having a L2 regularizer with coefficient 0.5λ.

2.6 Solvers

2.7 Mocha Backends

A backend in Mocha is a component that carries out actual numerical computation. Mocha is designed to support multiple backends, and switching between different backends should be almost transparent to the rest of the world.

2.7.1 Pure Julia CPU Backend

A pure Julia CPU backend is implemented in Julia. This backend is reasonably fast by making heavy use of the Julia's built-in BLAS matrix computation library and performance annotations to help the LLVM-based JIT compiler producing high performance instructions.

A pure Julia CPU backend could be instantiated by calling the constructor `CPUBackend()`. Because there is no external dependency, it should runs on any platform that runs Julia.

If you have many cores in your computer, you can play with the number of threads used by the Julia's BLAS matrix computation library by:

```
blas_set_num_threads(N)
```

Depending on the problem size and a lot of other factors, using larger N is not necessarily faster.

2.7.2 CPU Backend with Native Extension

Mocha comes with C++ implementations of some bottleneck computations for the CPU backend. In order to use the native extension, you need to build the native code first (if it is not built automatically when installing the package).

```
Pkg.build("Mocha")
```

After successfully building the native extension, it could be enabled by setting the environment variable. On bash or zsh, execute

```
export MOCHA_USE_NATIVE_EXT=true
```

before running Mocha. You can also set the environment variable inside the Julia code:

```
ENV["MOCHA_USE_NATIVE_EXT"] = "true"

using Mocha
```

Note you should set the environment variable **before** loading the Mocha module. Otherwise Mocha will not load the native extension sub-module at all.

The native extension uses OpenMP to do parallel computation on Linux. The number of OpenMP threads used could be controlled by the `OMP_NUM_THREADS` environment variable. Note this variable is not specific to Mocha. If you have other programs that uses OpenMP, setting this environment variable in a shell will also affect those problems started subsequently. If you want to restrict to Mocha, simply set the variable in the Julia code:

```
ENV["OMP_NUM_THREADS"] = 1
```

Note setting to 1 disabled the OpenMP parallelization. Depending on the problem size and a lot of other factors, using multi-thread OpenMP parallelization is not necessarily faster because of the overhead of multi-threads.

The parameter for the number of threads used by the BLAS library applies to the CPU backend with native extension, too.

OpenMP on Mac OS X

When compiling the native extension on Mac OS X, you will get a warning that OpenMP is disabled. This is because currently clang, the built-in compiler for OS X, does not officially support OpenMP yet. If you want to try OpenMP on OS X, please refer to Clang-OMP and compile manually (see below).

Native Extension on Windows

The native extension does not support Windows because automatic building script does not work on Windows. However, the native codes themselves does not use any OS specific features. If you have a compiler installed on Windows, you could try to compile the native extension manually. However, I have **not** tested the native extension on Windows personally.

Compile Native Extension Manually

The native codes are located in the `deps` directory of Mocha. Use

```
Pkg.dir("Mocha")
```

to find out where Mocha is installed. You should compile it as a shared library (DLL on Windows). However, currently the filename for the library is hard-coded to be `libmochaext.so`, with a `.so` extension, regardless of the underlying OS.

2.7.3 CUDA Backend

GPU has been shown to be very effective at training large scale deep neural networks. NVidia® recently released a GPU accelerated library of primitives for deep neural networks called cuDNN. Mocha implemented a CUDA backend by combining cuDNN, cuBLAS and plain CUDA kernels.

In order to use the CUDA backend, you need to have CUDA-compatible GPU devices. The CUDA toolkit should be installed in order to compile the Mocha CUDA kernels. cuBLAS is included in CUDA distribution. But cuDNN needs to be installed separately. You could obtain cuDNN from Nvidia's website by registering as a CUDA developer for free.

Note: cuDNN requires CUDA 6.5 to run, and currently cuDNN is available to Linux and Windows only.

Before using the CUDA backend, Mocha kernels needs to be compiled. The kernels are located in `src/cuda/kernels`. Please use `Pkg.dir("Mocha")` to find out where Mocha is installed on your system. We have included a Makefile for convenience, but if you don't have `make` installed, the compiling command is as simple as

```
nvcc -ptx kernels.cu
```

After compiling the kernels, you can now start to use the CUDA backend by setting the environment variable `MOCHA_USE_CUDA`. For example:

```
ENV["MOCHA_USE_CUDA"] = "true"

using Mocha

sys = System(CuDNNBackend())
init(sys)

# ...

shutdown(sys)
```

Note instead of instantiate a `CPUBackend`, you now construct a `CuDNNBackend`. The environment variable should be set **before** loading Mocha. It is designed to use conditional loading so that the pure CPU backend could still run on machines without any GPU device or CUDA library installed.

2.8 Tools

2.8.1 Importing Trained Model from Caffe

Overview

Mocha provides a tool to help importing Caffe's trained models. Importing Caffe's model consists of two steps:

1. **Translating the network architecture definitions**: this needs to be done manually. Typically for each layer used in Caffe, there is an equivalent in Mocha, so translating should be relatively straightforward. See *the CIFAR-10 tutorial* for an example of translating Caffe's network definition. You need to make sure to use the same name for the layers so that when importing the learned parameters, Mocha is able to find the correspondence.
2. **Importing the learned network parameters**: this could be done automatically, and is the main topic of this document.

Caffe uses a binary protocol buffer file to store trained models. Instead of parsing this complicated binary file, we provide a tool to export the model parameters to standard HDF5 format, and import the HDF5 file from Mocha. As a result, you need to have Caffe installed to do the importing.

Exporting Caffe's Snapshot to HDF5

Caffe's snapshot files contains some extra information, what we need is only the learned network parameters. The strategy is to use Caffe's built-in API to load their model snapshot, and then iterate all network layers in memory to dump layer parameters to HDF5 file. In the `tools` directory of Mocha's source root, you can find `dump_network_hdf5.cpp`.

Put that fine in Caffe's `tools` directory, and re-compile Caffe. The tool should be built automatically, and the executable file could typically be found in `build/tools/dump_network_hdf5`. Run the tool as following:

```
build/tools/dump_network_hdf5 \
    examples/cifar10/cifar10_full_train_test.prototxt \
    examples/cifar10/cifar10_full_iter_70000.caffemodel \
    cifar10.hdf5
```

where the arguments are Caffe's network definition, Caffe's model snapshot you want to export and the output HDF5 file, respectively.

Currently, in all the *layers Mocha supports*, only `InnerProductLayer` and `ConvolutionLayer` contains trained parameters. When some other layers are needed, it should be straightforward to modify `dump_network_hdf5.cpp` to include proper rules for exporting.

Importing HDF5 Snapshot to Mocha

Mocha has a unified interface to import the HDF5 model we just exported. After constructing the network with the same architecture as translated from Caffe, you can import the HDF5 file by calling

```
using HDF5
h5open("/path/to/cifar10.hdf5", "r") do h5
  load_network(h5, net)
end
```

Actually, `net` does not need to be the exactly the same architecture. What it does is to try to find the parameters for each layer in the HDF5 archive. So if the Mocha architecture contains fewer layers, it should be fine.

By default, if the parameters for a layer could not be found in the HDF5 archive, it will fail on error. But you could also change the behavior by passing `false` as the third argument, indicating do not panic if parameters are not found in the archive. In this case, Mocha will use the associated *initializer* to initialize the parameters not found in the archive.

Mocha's HDF5 Snapshot Format

By using the same technique, you can import network parameters trained by any deep learning tools into Mocha, as long as you could export to HDF5 files. The HDF5 file that Mocha could import is very simple

- Each parameter (e.g. the filter of a convolution layer) is stored as a 4D tensor dataset in the HDF5 file.
- The dataset name for each parameter should be `layer___param`. For example, `conv1___filter` is for the `filter` parameter of the convolution layer with the name `conv1`.

  HDF5 file format supports hierarchy. But it is rather complicated to manipulate hierarchies in some tools (e.g. the HDF5 Lite library Caffe is using), so we decide to use a simple flat format.

- In Caffe, the `bias` parameter for a convolution layer and an inner product layer is optional. It is OK to omit them on exporting if there is no bias. You will get a warning message when importing in Mocha. Mocha will use the associated initializer (by default initializing to 0) to initialize the bias.

CHAPTER 3

Developer's Guide

3.1 Blob

3.2 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.

[BengioGlorot2010] Y. Bengio and X. Glorot, *Understanding the difficulty of training deep feedforward neural networks*, in Proceedings of AISTATS 2010, pp. 249-256.

V

X