Mocha Documentation

Release 0.1.2

pluskid

Jul 09, 2018

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 usage 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 Caffe's modified architecture, by replacing the sigmoid activation functions with Rectified Linear Unit (ReLU) activation functions.

1.1.1 Preparing the Data

MNIST is a 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 can be downloaded in a binary format from Yann LeCun's website. We have created a script `get-mnist.sh` to download the dataset, and it calls `mnist.convert.jl` to convert the binary dataset into a HDF5 file that Mocha can 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 are 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 reads the HDF5 file and provides input for the network:

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

Note the `source` is a simple text file that 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. The network processes data in mini-batches, and we are using a batch size of 64 in this example. Larger mini-batches take more computational time but give a lower variance estimate of the loss function/gradient at each iteration. We also enable random shuffling of the data set to prevent structure in the ordering of input samples from influencing training.

Next we define a convolution layer in a similar way:

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

There are several parameters specified here:

`name` Every layer can 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 to get more informative debugging information when there are 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.

`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 datasets named `data` and `label` from the HDF5 file, and provide two streams 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 specifies 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 one stream of convolved data with the name `:conv1`.

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

```
pool_layer = PoolingLayer(name="pool1", kernel=(2,2), stride=(2,2),
    bottoms=[:conv1], tops=[:pool1])
conv2_layer = ConvolutionLayer(name="conv2", n_filter=50, kernel=(5,5),
    bottoms=[:pool1], tops=[:conv2])
pool2_layer = PoolingLayer(name="pool2", kernel=(2,2), stride=(2,2),
    bottoms=[:conv2], tops=[:pool2])
```

Note how `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 an inner product 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-evident. The `output_dim` property of an inner product layer specifies the dimension of the output. Note the dimension of the input is automatically determined from the bottom data stream.

For the first inner product layer we specify a Rectified Linear Unit (ReLU) activation function via the `neuron` property. An activation function could be added to almost any computation layer. By default, no activation function, or the *identity activation function* is used. We don't use activation an 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 the Backend and Building the 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:

```
backend = CPUBackend()
init(backend)
```

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

```
common_layers = [conv_layer, pool_layer, conv2_layer, pool2_layer,
    fc1_layer, fc2_layer]
net = Net("MNIST-train", backend, [data_layer, common_layers..., loss_layer])
```

A network is built by passing the constructor an initialized backend, and a list of layers. Note how we use `common_layers` to collect a subset of the layers. This will be useful later when constructing a network to process validation data.

1.1.4 Configuring the Solver

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

```
exp_dir = "snapshots"
method = SGD()
params = make_solver_parameters(method, max_iter=10000, regu_coef=0.0005,
    mom_policy=MomPolicy.Fixed(0.9),
    lr_policy=LRPolicy.Inv(0.01, 0.0001, 0.75),
    load_from=exp_dir)
solver = Solver(method, params)
```

The behavior of the solver is specified by 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 a global scaling factor for all the local regularization coefficients.

`mom_policy` This specifies the momentum policy used during training. Here we are using a fixed momentum policy of 0.9 throughout training. 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)$^{-\text{power}}$.

load_from This can be a saved model file or a directory. For the latter case, the latest saved model snapshot will be loaded automatically before the solver loop starts. We will see in a minute how to configure the solver to save snapshots automatically during training.

This is useful to recover from a crash, to continue training with a larger `max_iter` or to perform fine tuning on some pre-trained models.

1.1.5 Coffee Breaks for the Solver

Now our solver is ready to go. But in order to give it a healthy working plan, we provide it with some coffee breaks:

```
setup_coffee_lounge(solver, save_into="$exp_dir/statistics.jld", every_n_iter=1000)
```

This sets up the coffee lounge, which holds data reported during coffee breaks. Here we also specify a file to save the information we accumulated in coffee breaks to disk. Depending on the coffee breaks, useful statistics such as objective function values during training will be saved, and can be loaded later for plotting or inspecting.

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

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

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

```
add_coffee_break(solver, Snapshot(exp_dir), every_n_iter=5000)
```

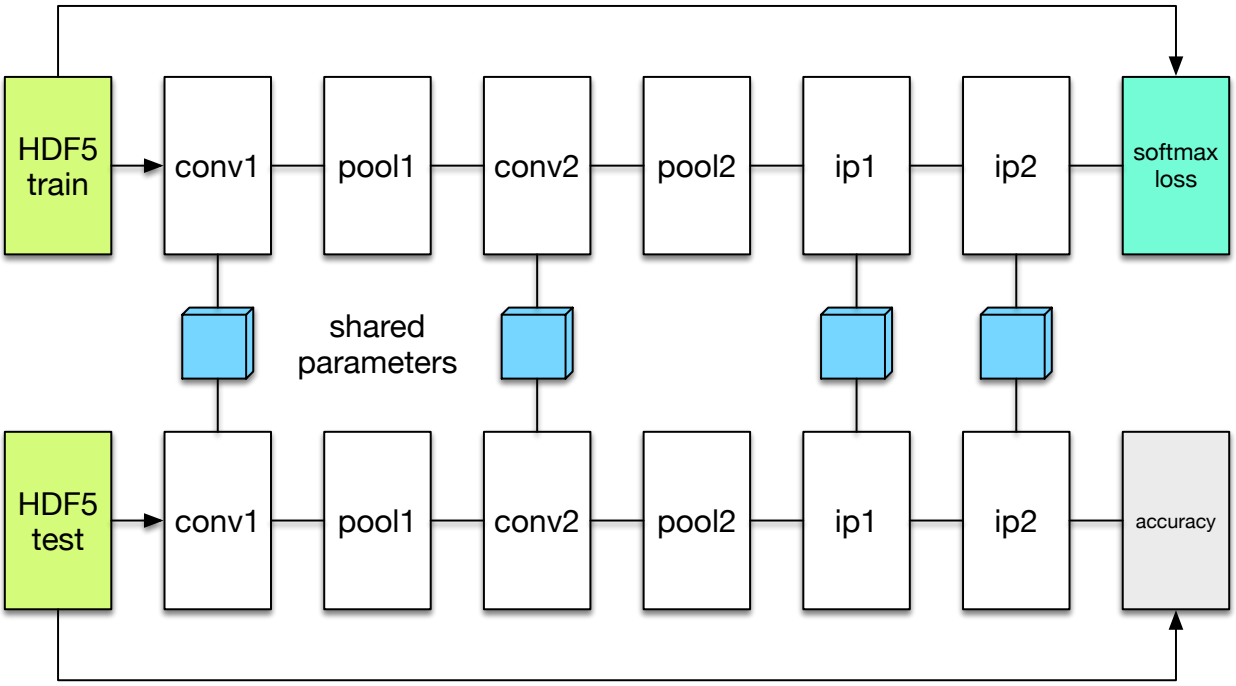
Note that we are passing `exp_dir` to the constructor of the `Snapshot` coffee break so snapshots will be saved into that directory. And according to our configuration of the solver above, the latest snapshots will be automatically loaded by the solver if you run this script again.

In order to see whether we are really making progress or simply overfitting, we also wish to periodically see the performance on a separate validation set. 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 the validation dataset instead of the 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 top, which will compute the classification accuracy.

```
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", backend, [data_layer_test, common_layers..., acc_layer])
```

Note how we re-use the `common_layers` variable defined a earlier to re-use the description of the network architecture. By passing the same layer objects used to define the training net to the constructor of the validation net, Mocha will 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 that 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 the 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 the epoch number as it goes through a full pass of the dataset.

1.1.6 Training

Without further ado, we can finally start the training process:

```
solve(solver, net)

destroy(net)
destroy(test_net)
shutdown(backend)
```

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 proceeds, progress information will be reported. It takes about 10~20 seconds every 100 iterations, i.e. about 7 iterations per second, on my machine, depending on the server load and many other 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
```

(continues on next page)

(continued from previous page)

```
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 the native extension for the CPU backend, or by using the 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 for the potential speed improvement, this is a sample log from running with the Native Extension enabled CPU backend. It runs at about 20 iterations per second.

```
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 following is a sample log from running with the CUDA backend. It runs at 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
```

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

1.1.7 Using Saved Snapshots for Prediction

Often you want to use a network previously trained with Mocha to make individual predictions. Earlier during the training process snapshots of the network state were saved every 5000 iterations, and these can be reloaded at a later time. To do this we first need a network with the same shape and configuration as the one used for training, except instead we supply a `MemoryDataLayer` instead of a `HDF5DataLayer`, and a `SoftmaxLayer` instead of a `SoftmaxLossLayer`:

```
using Mocha
backend = CPUBackend()
init(backend)

mem_data = MemoryDataLayer(name="data", tops=[:data], batch_size=1,
    data=Array[zeros(Float32, 28, 28, 1, 1)])
softmax_layer = SoftmaxLayer(name="prob", tops=[:prob], bottoms=[:ip2])

# define common_layers as earlier

run_net = Net("imagenet", backend, [mem_data, common_layers..., softmax_layer])
```

Note that `common_layers` has the same definition as above, and that we specifically pass a `Float32` array to the `MemoryDataLayer` so that it will match the `Float32` data type used in the MNIST HDF5 training dataset. Next we fill in this network with the learned parameters from the final training snapshot:

```
load_snapshot(run_net, "snapshots/snapshot-010000.jld")
```

Now we are ready to make predictions using our trained model. A simple way to accomplish this is to take the first test data point and run it through the model. This is done by setting the data of the `MemoryDataLayer` to the first test image and then using `forward` to execute the network. Note that the labels in the test data are indexed starting with 0 not 1 so we adjust them before printing.

```
using HDF5
h5open("data/test.hdf5") do f
    get_layer(run_net, "data").data[1][:,:,1,1] = f["data"][:,:,1,1]
    println("Correct label index: ", Int64(f["label"][:,1][1]+1))
end

forward(run_net)
println()
println("Label probability vector:")
println(run_net.output_blobs[:prob].data)
```

This produces the output:

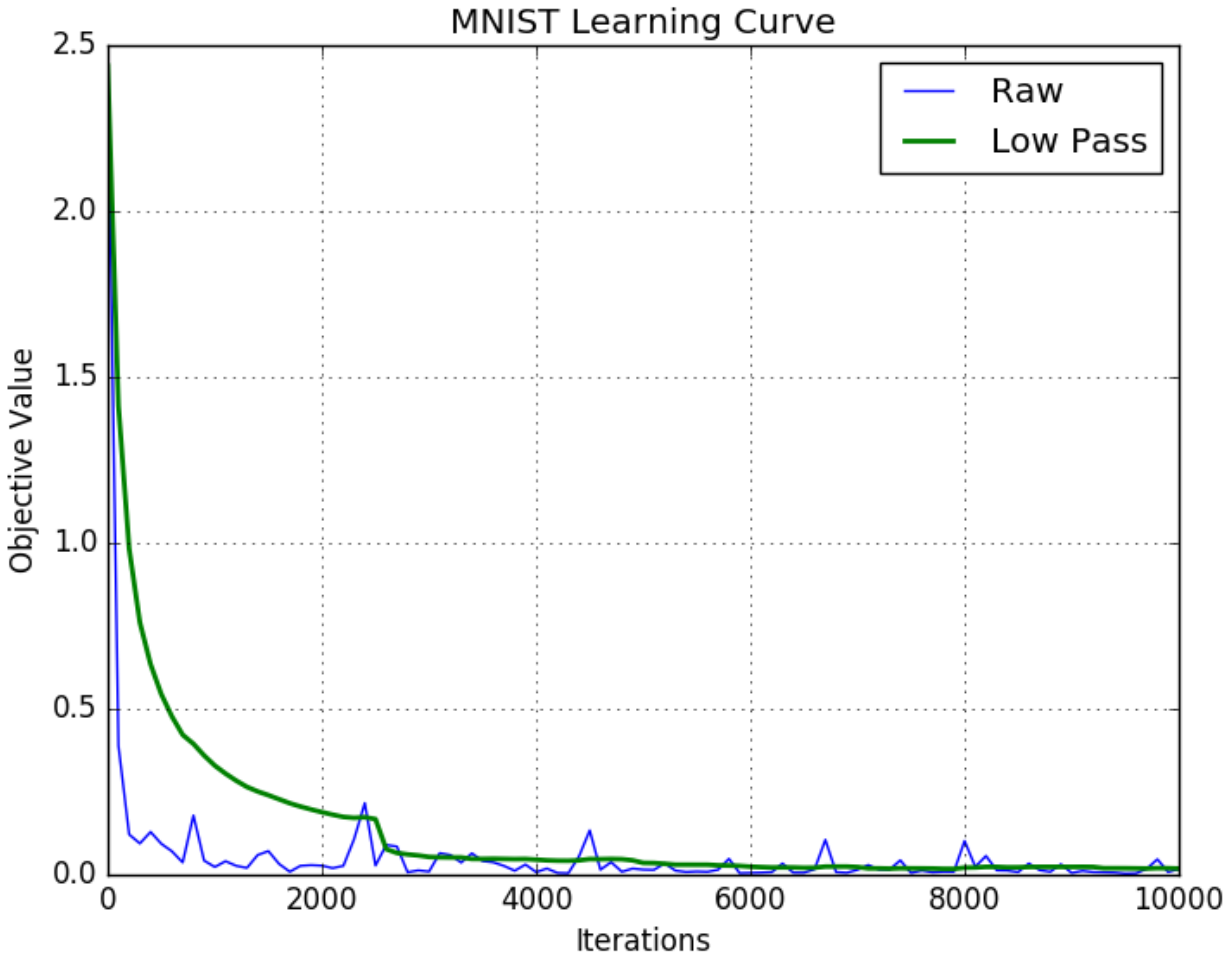
```
Correct label index: 5

Label probability vector:
Float32[5.870685e-6
        0.00057068263
        1.5419962e-5
        8.387835e-7
        0.99935246
        5.5915066e-6
        4.284061e-5
        1.2896479e-6
        4.2869314e-7
        4.600691e-6]
```

1.1.8 Checking The Solver's Progress with Learning Curves

While a network is training we should verify that the optimization of the weights and biases is converging to a solution. One of the best ways to do this is to plot the *Learning Curve* as the solver progresses through its iterations. A neural network's *Learning Curve* is a plot of iterations along the x axis and the value of the objective function along the y axis. Recall that the solver is trying to minimize the objective function so the value plotted along the y axis should

decrease over time. The image below inludes the raw data from the neural network in this tutorial and a smoothed plot that uses a low pass filter of the data to take out high frequency noise. More about noise in stochastic gradient descent later. For now let's focus on generating a *Learning Curve* like the one here.



Verifying convergence after a few thousand iterations is essential when developing neural networks on new datasets. Some teams have waited hours (or days) for their network to complete training only to discover that the solver failed to converge and they need to retune their paramaters. A quick look at the learning curve above after the first thousand iterations clearly shows that the algorithm is working and that letting it continue to train for the full 10,000 iterations will probably produce a good result.

The data to plot the *Learning Curve* is conveniently saved as the solver progresses. Recall that we set up the coffee lounge and a `TrainingSummary()` coffee break every 100 iterations in the `mnist.jl` file:

```
exp_dir = "snapshots"
setup_coffee_lounge(solver, save_into="$exp_dir/statistics.jld", every_n_iter=1000)
add_coffee_break(solver, TrainingSummary(), every_n_iter=100)
```

Given this data we can write a new Julia script to read the `statistics.jld` file and plot the learning curve while the solver continues to work. The source code for plotting the learning curve is included in the examples folder and called `mnist_learning_curve.jl`.

In order to see the plot we need to use a plotting package. The PyPlot package that implements matplotlib for Julia is adequate for this. Use the standard `Pkg.add("PyPlot")` if you do not already have it. We will also need to load the `statistics.jld` file using Julia's implementation of the HDF5 format which requires the JLD packge.

```
using PyPlot, JLD
```

Next, we need to load the data. This is not difficult, but requires some careful handling because the `statistics.jld` file is a Julia Dict that includes several sub-dictionaries. You may need to adjust the path in the `load("snapshots/statistics.jld")` command so that it accurately reflects the path from where the code is running to the `snapshots` directory.

```
stats = load("snapshots/statistics.jld")
# println(typeof(stats))

tables = stats["statistics"]
ov = tables["obj_val"]
xy = sort(collect(ov))
x = [i for (i,j) in xy]
y = [j for (i,j) in xy]
x = convert(Array{Int64}, x)
y = convert(Array{Float64}, y)
```

From the code above we can see that the `obj_val` dictionary is available in the snapshot. This dictionary gets appended every 100 iterations when the solver records a `TrainingSummary()`. Then those values get written to disk every 1000 iterations when the solve heads out to the coffee lounge for a break. Also note that `stats` is not a filehandle opened to the statistics file. It is a `Dict{ByteString,Any}`. This is desired because we do not want the learning curve script to lock out the `mnist.jl` script from getting file handle access to the snapshots files. You can uncomment `println(typeof(stats))` to verify that we do not have a file handle. At the end of this snippet we have a vector for x and y. Now we need to plot them which is simply handled in the snippet below.

```
raw = plot(x, y, linewidth=1, label="Raw")
xlabel("Iterations")
ylabel("Objective Value")
title("MNIST Learning Curve")
grid("on")
```

The last thing we need to talk about is the noise we see in the blue line in the plot above. Recall that we chose stochastic gradient descent (SGD) as the network solver in this line from the `mnist.jl` file:

```
method = SGD()
```

In pure gradient descent the solution moves closer to a minima each and every step; however, in order for the solver to do this it must compute the objective function for **every** training sample on **each** step. In our case this would mean all 50,000 training samples must be processed through the network to compute the loss for one iteration of gradient descent. This is computationally expensive and **slow**. Stochastic gradient descent avoids this performance penalty by computing the loss function on a subset of the training examples (batches of 64 in this example). The downside of using SGD is that it sometimes takes steps in the wrong direction since it is optimizing globally on a small subset of the training examples. These missteps create the noise in the blue line. Therefore, we also create a plot that has been through a low pass filter to take out the noise which reveals the trend in the objective function.

```
function low_pass{T <: Real}(x::Vector{T}, window::Int)
    len = length(x)
    y = Vector{Float64}(len)
    for i in 1:len
        # I want the mean of the first i terms up to width of window
        # Putting some numbers to this with window 4
        # i win lo  hi
        # 1  4  1   1
        # 2  4  1   2
        # 3  4  1   3
```

(continues on next page)

(continued from previous page)

```
        # 4  4  1   4
        # 5  4  1   5
        # 6  4  2   6  => window starts to slide
        lo = max(1, i - window)
        hi = i
        y[i] = mean(x[lo:hi])
    end
    return y
end
```

There are other (purer) ways to implement a low pass filter but this is adequate to create a smoothed curve for analyzing the global direction of network training. One appealing heuristics of this filter is that it outputs a solution for the first few data points consistent with the raw plot. With the filter we can now generate a smoothed set of y datapoints.

```
window = Int64(round(length(xy)/4.0))
y_avg = low_pass(y, window)
avg = plot(x, y_avg, linewidth=2, label="Low Pass")
legend(handles=[raw; avg])
show()  #required to display the figure in non-interactive mode
```

We declare `window` to be about one-quarter the length of the input to enforce a lot of smoothing. Also note that we use the labels to create a legend on the graph. Finally, this example places the `low_pass` function in the middle of the script which is not best practice, but the order presented here felt most appropriate for thinking through the different elements of the example.

There are lots of great resources on the web for building and training neural networks and after this example you now know how to use Julia and Mocha to contruct, train, and validate one of the most famous convolutional neural networks.

Thank you for working all the way to the end of the MNIST tutorial!

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 as in *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 all details here.

1.2.1 Caffe's Tutorial and Code

Caffe's tutorial for CIFAR-10 can be found on their website. The code can be located in `examples/cifar10` under Caffe's source tree. The code folder contains several different definitions of networks and solvers. The filenames should be self-explanatory. The *quick* files corresponds to a smaller network without local response normalization layers. These networks are documented in Caffe's tutorial, according to which they obtain 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 can 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 the LevelDB database, so we will do the same thing: download the original binary files and convert them into a Mocha-recognizable data format, in our case a HDF5 dataset. We have provided a Julia script convert.jl[1]. You can call get-cifar10.sh directly, which will automatically download the binary files, convert them to HDF5 and prepare text index files that point to the HDF5 datasets.

Notice in Caffe's data layer, a `transform_param` is specified with a `mean_file`. We could use Mocha's *data transformers* to do the same thing. But since we need to compute the data mean during data conversion, for simplicity, we also perform mean subtraction when converting data to the HDF5 format. See convert.jl for details. Please refer to the *user's guide* for more details about the HDF5 data format that Mocha expects.

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 uses the same file to define both the training and test networks, and uses *phases* to include and exclude layers that are used only in the training or testing phase. 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"
```

(continues on next page)

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

(continued from previous page)

```
      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 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 the `DefaultBackend`. It is a typealias for `GPUBackend` if CUDA is available and properly set up (see *Mocha Backends*), or uses `CPUBackend` as a backup even though it will be much slower.

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

backend = DefaultBackend()
init(backend)

net = Net("CIFAR10-train", backend, [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 dropped by a factor of 10 twice[3]. Caffe implements this by having three solver configurations with different learning rates for each stage. We could do the same thing for Mocha, but Mocha has a

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

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)),
)
method = SGD()
solver_params = make_solver_parameters(method, max_iter=70000,
    regu_coef=0.004, momentum=0.9, lr_policy=lr_policy,
    load_from="snapshots")
solver = Solver(method, solver_params)
```

The other parameters like regularization coefficient and momentum are directly translated from Caffe's solver configuration. Progress reporting and automatic snapshots can 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"), every_n_iter=5000)

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

1.2.6 Training

Now we can start training by calling `solve(solver, net)`. Depending on different *backends*, the training speed can 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.

```
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%
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
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 only 5~6 seconds to finish every 200 iterations on the `GPUBackend`.

```
22-Nov 15:04:47:INFO:root:048600 :: TRAIN obj-val = 0.53777266
22-Nov 15:04:52:INFO:root:048800 :: TRAIN obj-val = 0.60837102
22-Nov 15:04:58:INFO:root:049000 :: TRAIN obj-val = 0.79333639
22-Nov 15:04:59:INFO:root:
22-Nov 15:04:59:INFO:root:## Performance on Validation Set
22-Nov 15:04:59:INFO:root:---------------------------------------------------------
22-Nov 15:04:59:INFO:root:  Accuracy (avg over 10000) = 76.5900%
22-Nov 15:04:59:INFO:root:---------------------------------------------------------
22-Nov 15:04:59:INFO:root:
22-Nov 15:05:04:INFO:root:049200 :: TRAIN obj-val = 0.62640750
22-Nov 15:05:10:INFO:root:049400 :: TRAIN obj-val = 0.57287318
22-Nov 15:05:15:INFO:root:049600 :: TRAIN obj-val = 0.53166425
22-Nov 15:05:21:INFO:root:049800 :: TRAIN obj-val = 0.60679358
22-Nov 15:05:26:INFO:root:050000 :: TRAIN obj-val = 0.79003465
22-Nov 15:05:26:INFO:root:Saving snapshot to snapshot-050000.jld...
22-Nov 15:05:26:DEBUG:root:Saving parameters for layer conv1
22-Nov 15:05:26:DEBUG:root:Saving parameters for layer conv2
22-Nov 15:05:26:DEBUG:root:Saving parameters for layer conv3
22-Nov 15:05:26:DEBUG:root:Saving parameters for layer ip1
22-Nov 15:05:27:INFO:root:
22-Nov 15:05:27:INFO:root:## Performance on Validation Set
22-Nov 15:05:27:INFO:root:---------------------------------------------------------
22-Nov 15:05:27:INFO:root:  Accuracy (avg over 10000) = 76.5200%
22-Nov 15:05:27:INFO:root:---------------------------------------------------------
22-Nov 15:05:27:INFO:root:
22-Nov 15:05:33:INFO:root:050200 :: TRAIN obj-val = 0.61519235
22-Nov 15:05:38:INFO:root:050400 :: TRAIN obj-val = 0.57314044
```

1.3 Image Classification with Pre-trained Model

This is a demo of using a CNN pre-trained on Imagenet to do image classification. The code is located in `examples/ijulia/ilsvrc12`. You can view the rendered notebook example directly at nbviewer. Or alternatively, you can also start IJulia server locally by running

```
ipython notebook --profile julia
```

in this demo's directory. The IJulia page will be automatically opened in your default browser.

1.4 Pre-training with Stacked De-noising Auto-encoders

In this tutorial, we show how to use Mocha's primitives to build stacked auto-encoders to do pre-training for a deep neural network. We will work with the MNIST dataset. Please see *the LeNet tutorial on MNIST* on how to prepare the HDF5 dataset.

Unsupervised pre-training is a way to initialize the weights when training deep neural networks. Initialization with pre-training can have better convergence properties than simple random training, especially when the number of (labeled) training points is not very large.

In the following two figures, we show the results generated from this tutorial. Specifically, the first figure shows the softmax loss on the training set at different training iterations with and without pre-training initialization.

The second plot is similar, except that it shows the prediction accuracy of the trained model on the test set.

As we can see, faster convergence can be observed when we initialize with pre-training.

1.4.1 (Stacked) Denoising Auto-encoders

We provide a brief introduction to (stacked) denoising auto-encoders in this section. See also the deep learning tutorial on Denoising Auto-encoders.

An **auto-encoder** takes an input $\mathbf{x} \in \mathbb{R}^p$, maps it to a latent representation (encoding) $\mathbf{y} \in \mathbb{R}^q$, and then maps back to the original space $\mathbf{z} \in \mathbb{R}^p$ (decoding / reconstruction). The mappings are typically linear maps (optionally) followed by a element-wise nonlinearity:

$$\mathbf{y} = s\left(\mathbf{W}\mathbf{x} + \mathbf{b}\right)$$
$$\mathbf{z} = s\left(\tilde{\mathbf{W}}\mathbf{y} + \tilde{\mathbf{b}}\right)$$

Typically, we constrain the weights in the decoder to be the transpose of the weights in the encoder. This is referred to as *tied weights*:

$$\tilde{\mathbf{W}} = \mathbf{W}^T$$

Note that the biases $\mathbf{b}$ and $\tilde{\mathbf{b}}$ are still different even when the weights are tied. An auto-encoder is trained by minimizing the reconstruction error, typically with the square loss $\ell(\mathbf{x}, \mathbf{z}) = \|\mathbf{x} - \mathbf{z}\|^2$.

A **denoising auto-encoder** is an auto-encoder with noise corruptions. More specifically, the encoder takes a corrupted version $\tilde{\mathbf{x}}$ of the original input. A typical way of corruption is randomly masking elements of $\mathbf{x}$ as zeros. Note the reconstruction error is still measured against the original uncorrupted input $\mathbf{x}$.

After training, we can take the weights and bias of the encoder layer in a (denoising) auto-encoder as an initialization of an hidden (inner-product) layer of a DNN. When there are multiple hidden layers, layer-wise pre-training of stacked (denoising) auto-encoders can be used to obtain initializations for all the hidden layers.

Layer-wise pre-training of stacked auto-encoders consists of the following steps:

1. Train the bottommost auto-encoder.
2. After training, remove the decoder layer, construct a new auto-encoder by taking the *latent representation* of the previous auto-encoder as input.

3. Train the new auto-encoder. Note the weights and bias of the encoder from the previously trained auto-encoders are **fixed** when training the newly constructed auto-encoder.
4. Repeat step 2 and 3 until enough layers are pre-trained.

Next we will show how to train denoising auto-encoders in Mocha and use them to initialize DNNs.

1.4.2 Experiment Configuration

We will train a DNN with 3 hidden layers using sigmoid nonlinearities. All the parameters are listed below:

```
n_hidden_layer   = 3
n_hidden_unit    = 1000
neuron           = Neurons.Sigmoid()
param_key_prefix = "ip-layer"
corruption_rates = [0.1,0.2,0.3]
pretrain_epoch   = 15
finetune_epoch   = 1000
batch_size       = 100
momentum         = 0.0
pretrain_lr      = 0.001
finetune_lr      = 0.1

param_keys       = ["$param_key_prefix-$i" for i = 1:n_hidden_layer]
```

As we can see, we will do 15 epochs when pre-training for each layer, and do 1000 epochs of fine-tuning.

In Mocha, parameters (weights and bias) can be shared among different layers by specifying the `param_key` parameter when constructing layers. The `param_keys` variables defined above are unique identifiers for each of the hidden layers. We will use those identifiers to indicate that the encoders in pre-training share parameters with the hidden layers in DNN fine-tuning.

Here we define several basic layers that will be used in both pre-training and fine-tuning.

```
data_layer = HDF5DataLayer(name="train-data", source="data/train.txt",
    batch_size=batch_size, shuffle=@windows ? false : true)
rename_layer = IdentityLayer(bottoms=[:data], tops=[:ip0])
hidden_layers = [
  InnerProductLayer(name="ip-$i", param_key=param_keys[i],
      output_dim=n_hidden_unit, neuron=neuron,
      bottoms=[symbol("ip$(i-1)")], tops=[symbol("ip$i")])
  for i = 1:n_hidden_layer
]
```

Note the `rename_layer` is defined to rename the `:data` blob to `:ip0` blob. This makes it easier to define the hidden layers in a unified manner.

1.4.3 Pre-training

We construct stacked denoising auto-encoders to perform pre-training for the weights and biases of the hidden layers we just defined. We do layer-wise pre-training in a `for` loop. Several Mocha primitives are useful for building auto-encoders:

- `RandomMaskLayer`: given a corruption ratio, this layer can randomly mask parts of the input blobs as zero. We use this to create corruptions in denoising auto-encoders.

Note this is a *in-place* layer. In other words, it modifies the input directly. Recall that the reconstruction error is computed against the *uncorruppted* input. So we need to use the following layer to create a copy of the input before applying corruption.

- `SplitLayer`: split a blob into multiple copies.
- `InnerProductLayer`: the encoder layer is just an ordinary inner-product layer in DNNs.
- `TiedInnerProductLayer`: if we do not want *tied weights*, we could use another inner-product layer as the decoder. Here we use a special layer to construct decoders with *tied weights*. The `tied_param_key` attribute is used to identify the corresponding encoder layer we want to tie weights with.
- `SquareLossLayer`: used to compute reconstruction error.

We list the code for the layer definitions of the auto-encoders again:

```
  ae_data_layer = SplitLayer(bottoms=[symbol("ip$(i-1)")], tops=[:orig_data, :corrupt_
↪data])
  corrupt_layer = RandomMaskLayer(ratio=corruption_rates[i], bottoms=[:corrupt_data])

  encode_layer  = copy(hidden_layers[i], bottoms=[:corrupt_data])
  recon_layer   = TiedInnerProductLayer(name="tied-ip-$i", tied_param_key=param_
↪keys[i],
      tops=[:recon], bottoms=[symbol("ip$i")])
  recon_loss_layer = SquareLossLayer(bottoms=[:recon, :orig_data])
```

Note how the i-th auto-encoder is built on top of the output of the (i-1)-th hidden layer (blob name `symbol("ip$(i-1)")`). We split the blob into `:orig_data` and `:corrupt_data`, and add corruption to the `:corrupt_data` blob.

The encoder layer is basically the same as the i-th hidden layer. But it should take the corrupted blob as input, so use the `copy` function to make a new layer based on the i-th hidden layer but change the `bottoms` property. The decoder layer has *tied weights* with the encoder layer, and the square-loss layer compute the reconstruction error.

Recall that in layer-wise pre-training, we fix the parameters of the encoder layers that we already trained, and only train the top-most encoder-decoder pair. In Mocha, we can *freeze* layers in a net to prevent their parameters being modified during training. In this case, we freeze all layers except the encoder and the decoder layers:

```
  da_layers = [data_layer, rename_layer, ae_data_layer, corrupt_layer,
      hidden_layers[1:i-1]..., encode_layer, recon_layer, recon_loss_layer]
  da = Net("Denoising-Autoencoder-$i", backend, da_layers)
  println(da)

  # freeze all but the layers for auto-encoder
  freeze_all!(da)
  unfreeze!(da, "ip-$i", "tied-ip-$i")
```

Now we are ready to do the pre-training. In this example, we do not use regularization or momentum:

```
  base_dir = "pretrain-$i"
  method = SGD()
  pretrain_params  = make_solver_parameters(method, max_iter=div(pretrain_epoch*60000,
↪batch_size),
      regu_coef=0.0, mom_policy=MomPolicy.Fixed(momentum),
      lr_policy=LRPolicy.Fixed(pretrain_lr), load_from=base_dir)
  solver = Solver(method, pretrain_params)

  add_coffee_break(solver, TrainingSummary(), every_n_iter=1000)
  add_coffee_break(solver, Snapshot(base_dir), every_n_iter=3000)
  solve(solver, da)
```

(continues on next page)

(continued from previous page)

```
destroy(da)
```

1.4.4 Fine Tuning

After pre-training, we are now ready to do supervised fine tuning. This part is almost identical to the original *MNIST tutorial*.

```
pred_layer = InnerProductLayer(name="pred", output_dim=10,
    bottoms=[symbol("ip$n_hidden_layer")], tops=[:pred])
loss_layer = SoftmaxLossLayer(bottoms=[:pred, :label])

net = Net("MNIST-finetune", backend, [data_layer, rename_layer,
    hidden_layers..., pred_layer, loss_layer])

base_dir = "finetune"
params = make_solver_parameters(method, max_iter=div(finetune_epoch*60000,batch_size),
    regu_coef=0.0, mom_policy=MomPolicy.Fixed(momentum),
    lr_policy=LRPolicy.Fixed(finetune_lr), load_from=base_dir)
solver = Solver(method, params)

setup_coffee_lounge(solver, save_into="$base_dir/statistics.jld", every_n_iter=10000)

add_coffee_break(solver, TrainingSummary(), every_n_iter=1000)
add_coffee_break(solver, Snapshot(base_dir), every_n_iter=10000)

data_layer_test = HDF5DataLayer(name="test-data", source="data/test.txt", batch_
↪size=100)
acc_layer = AccuracyLayer(name="test-accuracy", bottoms=[:pred, :label])
test_net = Net("MNIST-finetune-test", backend, [data_layer_test, rename_layer,
    hidden_layers..., pred_layer, acc_layer])
add_coffee_break(solver, ValidationPerformance(test_net), every_n_iter=5000)

solve(solver, net)

destroy(net)
destroy(test_net)
```

Note that the key to allow the `MNIST-finetune` net to use the pre-trained weights as initialization of the hidden layers is that we specify the same `param_key` property for the hidden layers and the encoder layers. Those parameters are stored in the registry of the `backend`. When a net is constructed, if a layer finds existing parameters with its `param_key`, it will use the existing parameters, and ignore the *parameter initializers* specified by the user. Debug information will be printed to the console:

```
31-Dec 02:37:46:DEBUG:root:InnerProductLayer(ip-1): sharing weights and bias
31-Dec 02:37:46:DEBUG:root:InnerProductLayer(ip-2): sharing weights and bias
31-Dec 02:37:46:DEBUG:root:InnerProductLayer(ip-3): sharing weights and bias
```

1.4.5 Comparison with Random Initialization

In order to see whether pre-training is helpful, we train the same DNN but with random initialization. The same layer definitions are re-used. But note the highlighted line below: we reset the registry in the backend to clear the pre-trained parameters before constructing the net:

```
registry_reset(backend)

net = Net("MNIST-rnd", backend, [data_layer, rename_layer,
    hidden_layers..., pred_layer, loss_layer])
base_dir = "randinit"

params = copy(params, load_from=base_dir)
solver = Solver(method, params)

setup_coffee_lounge(solver, save_into="$base_dir/statistics.jld", every_n_iter=10000)

add_coffee_break(solver, TrainingSummary(), every_n_iter=1000)
add_coffee_break(solver, Snapshot(base_dir), every_n_iter=10000)
test_net = Net("MNIST-randinit-test", backend, [data_layer_test, rename_layer,
    hidden_layers..., pred_layer, acc_layer])
add_coffee_break(solver, ValidationPerformance(test_net), every_n_iter=5000)

solve(solver, net)

destroy(net)
destroy(test_net)
```

We can check from the log that randomly initialized parameters are used in this case:

```
31-Dec 01:55:06:DEBUG:root:Init network MNIST-rnd
31-Dec 01:55:06:DEBUG:root:Init parameter weight for layer ip-1
31-Dec 01:55:06:DEBUG:root:Init parameter bias for layer ip-1
31-Dec 01:55:06:DEBUG:root:Init parameter weight for layer ip-2
31-Dec 01:55:06:DEBUG:root:Init parameter bias for layer ip-2
31-Dec 01:55:06:DEBUG:root:Init parameter weight for layer ip-3
31-Dec 01:55:06:DEBUG:root:Init parameter bias for layer ip-3
31-Dec 01:55:06:DEBUG:root:Init parameter weight for layer pred
31-Dec 01:55:06:DEBUG:root:Init parameter bias for layer pred
```

The plots shown at the beginning of this tutorial are generated from the saved statistics from the *coffee lounges*. If you are interested in how those plots are generated, please refer to the `plot-all.jl` script in the code directory of this tutorial.

1.5 Mocha in the Cloud

The real power of developing deep learning networks is only realized when you can test your ideas and run your models on powerful compute platforms that complete training in a fraction of the time it took only a few years ago. Today, state-of-the-art machine learning algorithms routinely run on cloud based compute that offers access to cutting edge GPU technology for a tiny cost compared to buying the GPU hardware and running it in your personal or company servers.

Amazon Web Services (AWS) is one of the most popular cloud based services that provide access to such powerful computers. As Mocha and Julia mature I'm sure that a pre-configured Amazon Machine Image (AMI) will emerge to run Mocha in the cloud. Now, in October 2016, such an AMI does not exist, but even when a pre-configured image for Mocha does become available I highly recommend following through this tutorial at least once so you understand how cloud resources are provisioned and configured to suport Deep Learning.

We are going to show you how to take the CIFAR-10 example and get it running in the cloud. Along the way you will learn how to interact with AWS and get a broader understanding of cloud architectures in general.

1.5.1 Signing up for AWS

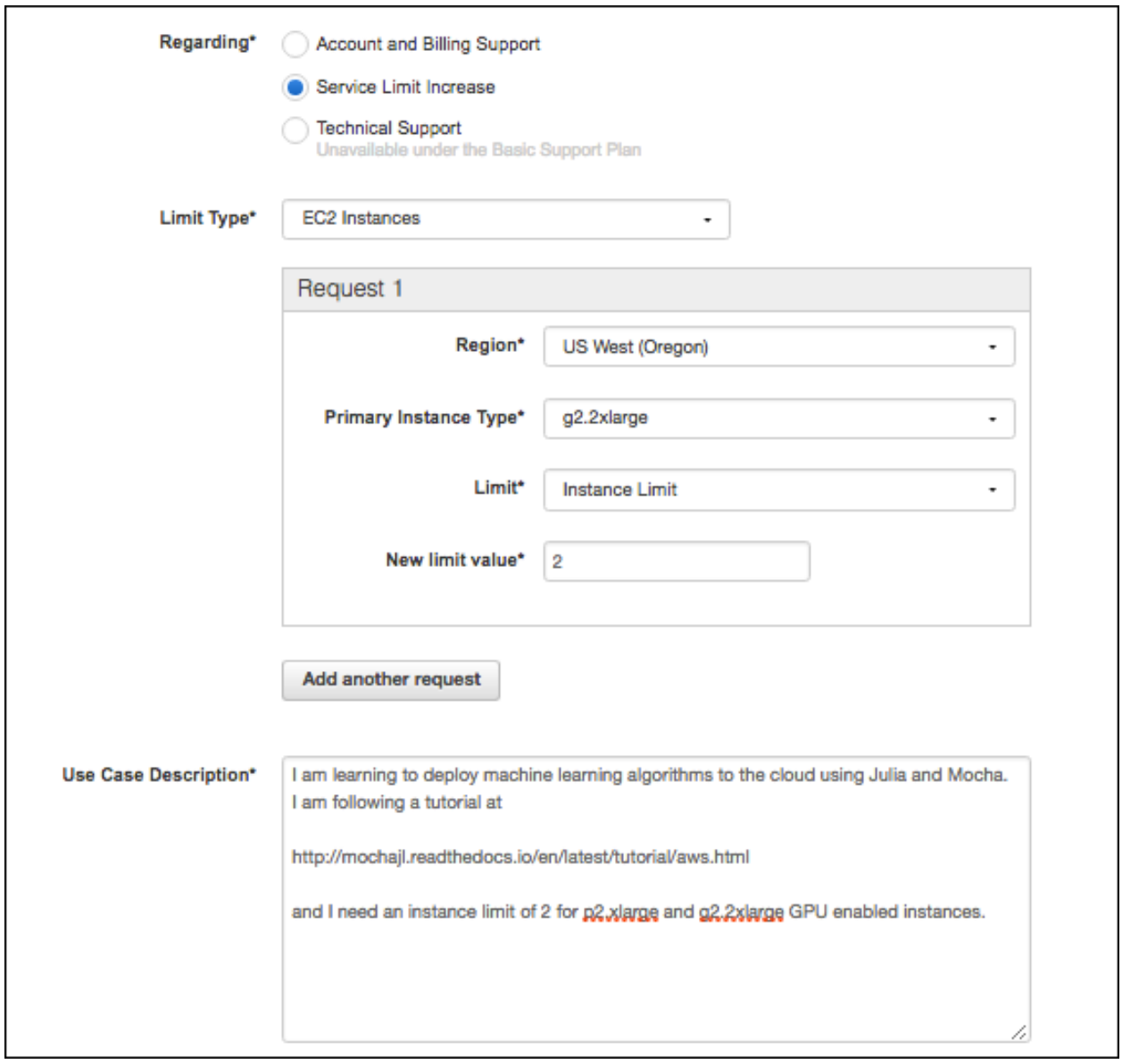
The first task in getting up and running in AWS is to set up an account. If you already have an Amazon.com shopping account these same credentials and methods of payment will get you into AWS. If not, then sign up for an account here. There is no charge when signing up for the account.

Usage costs for AWS is dependent on two factors. The type of computer you use, called the *instance type*, and the hours you use it for. For this example we are going to provision an instance type called *p2.xlarge* which contains one NVIDIA Tesla K80 GPU and a 4 CPUs. It costs about 90 cents per hour (as of Oct 2016). Building the software and running the CIFAR10 training will take less than two hours.

However, AWS does not let brand new users launch *p2* or *g2* instances when you first open your account. So you need to request access to GPU enabled machines by opening a support request. From the AWS Console (after signing in) click on *Support* in the top right hand corner and select *Support Center* from the dropdown menu.

When the support page opens up click on

This will open up a form similar to the figure below. Choose the region closest to your location and submit the request with the text and options suggested by the figure.



1.5.2 While you wait

Approval of the support request might take a few days. So while you wait let me suggest a few ways to sharpen your knowlege of AWS, Mocha, or Deep Learning.

Track 1 - Deep learning expert but new to AWS

Work through the EC2 tutorials for a few days so you learn how to launch and manage instances on AWS. As a new AWS member consider taking advantage of the free services tier and then these tutorials will not cost anything.

Track 2 - Cloud comfortable but new to Mocha or Deep Learning

Since we are going to be using the CIFAR10 example later in this tutorial and training it in the cloud, why not download the paper on the dataset[1] which provides good insight into the challenges of training image classifiers. Then take a look at this seminal paper[2] on training one of the first convolutional neural network implementations that threw the gates open to deep learning in 2012. Before this critical work neural networks deeper than one or two hidden layers were untrainable because the backpropogation of gradients lost their corrective strength by the time they reached the bottom layers of deep networks.

1.5.3 Provisioning the instance and the base image

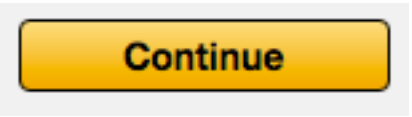
At this point you should have heard back from AWS that you are approved for *g2* and *p2* instance types. Now we need to launch a GPU instance with NVIDIA drivers and the Cuda components needed to work with the GPU.

First let me explain just a little about Cuda.

The Amazon *p2.xlarge* instance contains both a CPU and an NVIDIA Tesla K80 GPU, but in order to access the GPU the NVIDIA drivers for the P80 must be installed and the various elements of the NVIDIA development environment must also be installed. This blog post describes how to install these components from scratch, but I find that installing the correct NVIDIA driver and a compatible version of the Cuda Toolkit that is also compliant with the hardware on the cloud instance can be quite a challenge. To avoid this complication we are going to launch an instance in this tutorial from a commercially supported AMI available on the AWS marketplace.

You can launch any AMI from the EC2 Management Console, but we will launch by clicking on this link for the AWS Marketplace and entering `Bitfusion Mobile Deep Learning` into the search bar. Note that the commercially provided AMI costs another 9 cents per hour (Oct 2016 pricing), but the cost is well worth avoiding the headache of fighting the NVIDA/CUDA/INSTANCE configuration challenge.

When the search results return click on the title of the AMI and it will take you to the configuration screen. Select the Region where you were granted the *p2.xlarge* instance.

Then click on 

In the next screen ensure that you choose *p2.xlarge* as the instance type and properly set the *Key pair* to a value in the dropdown menu where you have the private key stored locally, otherwise you will not be able to `ssh` into the new instance.

1.5.4 Verify NVIDIA Drivers and CUDA Toolkit are Working

While you were waiting for your AWS instance limits to be raised I hope you took the time to launch a few free instances and worked through a few of the AWS tutorials. With that knowledge launch the new *p2.xlarge* instance and log into the cloud instance with

```
ssh -i ".ssh/<your key name>" ubunty@ec2.your-AWS-domain-address
```

Before going any further we need to verify that the NVIDIA drivers and the Cuda toolkit are both installed and in the path. Issue the following two commands. If you get an error on either one then terminate the instance and start over from the section above.

[1] Krizhevsky, Alex, and Geoffrey Hinton. "Learning multiple layers of features from tiny images." (2009).

[2] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012.

```
nvidia-smi
nvcc --version
```

Output should resemble the screenshot below:

1.5.5 Installing Julia

Julia is new, which means that a lot of things that are annoying about other programming languages are fun and easy in Julia.

Julia is new, which also means that it is not pre-installed in very many Amazon Machine Images (AMIs) so we will be building Julia from source. Note that I have tried the Bitfusion Scientific Computing AMI that includes Julia and NVIDIA drivers, but when I add Mocha, enable the GPU backend, and run `Pkg.test("Mocha")` it fails with an error in the `cudnn.jl` file.

There is a bash script below that will automate the build and configution of Julia on the Bitfusion AMI, but if you are new to AWS then I recommend walking through the build process to better understand how to work with a cloud instance. If you are eager to quickly get set up and and start coding Mocha in the cloud then scroll to the script at the bottom of this tutorial.

At this point I assume you have a shell on your instance and you have verified that the NVIDIA drivers and Cuda toolkit are installed and working. Now we are going to update all the base software on the instance. Issue the following lines

of code one at a time.

```
sudo apt-get update
sudo apt-get updgrade
```

Note 1: You will get asked about a Docker upgrade on the Bitfusion AMI which uses Docker to manager other services on the image. Choose option *N* on this question.

Note 2: You will also get asked about installs taking up space on the drive. To complete this tutorial you do NOT need to configure anymore storage on the instance than what is already provided as the default. So answer *Y* to these questions.

At this point we need to install software that is not included in the base image, but required to build Julia from source code.

```
sudo apt-get install cmake hdf5-tools m4 libopenblas-base
```

Note 3: The package `hdf5-tools` is not required to install Julia, but is required to install Mocha later in this build. So it is good to get `hdf5-tools` in place now.

Once these installs complete we are ready to install Julia.

It is a solid practice to build a core component such as a programming language from its stable release unless you plan to contribute to the development of the language itself. For this tutorial we are trying to build a reliable installation of Julia and Mocha to train a Deep CNN. So we want a stable release. To find a stable version and build against that version we will use the version control properties of `git`.

Change directory into the newly cloned julia folder with `cd julia`. Then issue a `git status` command. You should see git identifies this folder as a project under version control. Now issue the `git tag` command. This will provide a list of tagged releases similar to the list below:

```
v0.1.0
.
.
.
v0.4.5
v0.4.6
v0.4.7
v0.5.0
v0.5.0-rc0
v0.5.0-rc1
```

We do not want to use a release candidate in the format `v0.X.0-rcX`. Therefore, `v0.5.0` might be a good choice, but as of Oct 2016 there is a compatibility issue between Mocha and this version. It is not unusual in a quickly developing project like Julia and Mocha for compatiblity and dependency conflicts at the edge of the build tree. So we will drop back and use `v0.4.7`. Issue a git command to checkout the version we want to build, `git checkout v0.4.7`.

Next, we want to take full advantage of all the CPUs on our instance when we build because this should make the build process go much faster. To find the number of available cores run `lscpu`. See the link here for a good explanation of the output of `lscpu`. On the *p2.xlarge* instance there are 4 CPUs.

Finally, build the julia executable with `sudo make -j N` where `N` is the number of CPUs on the cloud instance.

When the build completes in about 40 minutes take a look at the folder and notice that it now contains an executable named `julia`. We want to link that executable into the PATH so issue this command

```
sudo ln -s -f ~/julia/julia /usr/local/bin/julia
```

This allows you to issue the `julia` command from anywhere and it will launch the REPL or invoke julia to run a program.

Open the new executable with `julia` and run `Pkg.add("Mocha")`. Follow a successful Mocha installation with `Pkg.test("Mocha")`.

God willin' and the crik don't rise (...as my Texan grandfather would say) you will get a successful Mocha test on AWS. Now we need to configure the environment to use the GPU and test on the GPU backend. Still in the Julia REPL issue `Pkg.dir("Mocha")` and make note of the Mocha directory. Then exit out of the Julia REPL with `exit()`.

Change into the Mocha directory and then `cd src/cuda/kernels`. Take a look at the files here and notice the `Makefile` that will build the cuda kernals used by Mocha. Issue `make` to build these kernals with the NVIDIA Cuda Compiler (aka `nvcc`).

Now we need to export an environment variable to tell Julia to use the GPU backend. At the ubuntu prompt enter `export MOCHA_USE_CUDA='true'`. You can issue `env` to see all environmental variables and you will notice that our Mocha control variable has been added to the environment.

Restart Julia with the `julia` command that is globally available now. At the prompt issue `Pkg.test("Mocha")`. Now the tests will restart but in the first few lines of the image below you will notice that `* CUDA enabled` caught the environment variable and is going to use the GPU backend to run the tests. If everything has been set up right then you should get the output below which takes about five minutes to run.

```
               _
   _       _ _(_)_     |  A fresh approach to technical computing
  (_)     | (_) (_)    |  Documentation: http://docs.julialang.org
   _ _   _| |_  __ _   |  Type "?help" for help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 0.4.7 (2016-09-18 16:17 UTC)
 _/ |\__'_|_|_|\__'_|  |
|__/                   |  x86_64-linux-gnu

julia> Pkg.test("Mocha")
INFO: Testing Mocha
Configuring Mocha...
 * CUDA       enabled [DEV=0] (MOCHA_USE_CUDA environment variable detected)
 * Native Ext disabled by default
Mocha configured, continue loading module...
DefaultBackend = Mocha.GPUBackend
29-Oct 19:44:38:INFO:root:Initializing CuDNN backend...
29-Oct 19:44:49:INFO:root:CuDNN backend initialized!
-- Testing simple Adam solver call
.
.
.
-- Testing AccuracyLayer on Mocha.GPUBackend{Float64}...
    > (7,10,8,10,8) (operate on dimension 1)
    > Forward
    > Forward Again
    > Forward Again and Again
29-Oct 19:49:47:INFO:root:Shutting down CuDNN backend...
29-Oct 19:49:47:INFO:root:CuDNN Backend shutdown finished!
INFO: Mocha tests passed
```

At this point you are ready to train the CIFAR10 example, so scroll on down to that section.

Build from a script

If you skipped ahead and want to install Julia from a script that automates all of the instruction in the preceding paragraphs then follow these instruction:

```
cd ~
nano install_julia.sh
```

Cut and paste the script below into the `nano` text window and save the file with `CTRL+x`.

```
#! /bin/bash
# Install build environment
echo "*****************Setting up the Build Environment*****************"
# Running the update/upgrade on takes about 5 min
sudo apt-get update
sudo apt-get upgrade
# Note 1: You will get asked about Docker upgrade on the Bitfusion AMI which
# uses Docker to manager other services in the AMI.  Choose option 'N' on
# this question.
# Note 2: You will also get asked about installs taking up space on the
# drive.  To complete this tutorial you NOT need to configure anymore
# storage on the instance than what is already provided with the default
# image.  So say yes to these questions.

# Other packages required but already in this AMI are:
# git g++
# I have not tested the assertion from someone else's guide that
# the Julia build also requires:
# gfortran
sudo apt-get install cmake hdf5-tools m4 libopenblas-base

echo "*****************Cloning Julia************************************"
git clone https://github.com/JuliaLang/julia.git
cd julia
# As of Oct 2016 Julia v0.4.7 is the latest compatible vesion with Mocha
git checkout v0.4.7

#Determine the number of CPUs to build on
NUM_CPUS=$(lscpu | awk '/^CPU\(s\):/ {print $2}')

echo "*****************Making Julia on $NUM_CPUS CPUs*************************"
#Takes 30 minutes on a 4CPU p2.xlarge AWS instance
sudo make -j $NUM_CPUS
```

Then issue the following commands to run the script:

```
chmod +x install_julia.sh``
sudo install_julia.sh``
```

1.5.6 Running the CIFAR10 Test

At this point the CIFAR10 example should run without any problems. There are links to the example files in the AWS example folder. Change directory into `/path/to/Mocha/examples/AWS`. Run `get-cifar10.sh`. Once the data is downloaded you can run the example by issuing

```
julia cifar10.jl
```

Once the example starts to run take note that the environment variable we set for `Pkg.test("Mocha")` is still in place so you should see `* CUDA enabled` and that the `DefaultBackend = Mocha.GPUBackend`. This is awesome because you are now going to train the CIFAR10 network in the cloud and you will see that it only takes

about 3 seconds to train 200 iterations of backpropogation. Compared to my MacBook Pro this is about 28 times faster (19 min vs 530 min).

```
ubuntu@ip-172-31-8-83:~/.julia/v0.4/Mocha/examples/cifar10$ time julia cifar10.jl
Configuring Mocha...
 * CUDA       enabled [DEV=0] (MOCHA_USE_CUDA environment variable detected)
 * Native Ext disabled by default
Mocha configured, continue loading module...
DefaultBackend = Mocha.GPUBackend
30-Oct 22:35:01:INFO:root:Initializing CuDNN backend...
30-Oct 22:35:01:INFO:root:CuDNN backend initialized!
30-Oct 22:35:03:INFO:root:Constructing net CIFAR10-train on Mocha.GPUBackend...
30-Oct 22:35:03:INFO:root:Topological sorting 11 layers...
30-Oct 22:35:03:INFO:root:Setup layers...
30-Oct 22:35:05:INFO:root:Network constructed!
30-Oct 22:35:05:INFO:root:Constructing net CIFAR10-test on Mocha.GPUBackend...
30-Oct 22:35:05:INFO:root:Topological sorting 11 layers...
.
.
.
30-Oct 22:53:37:INFO:root:## Performance on Validation Set after 70000 iterations
30-Oct 22:53:37:INFO:root:---------------------------------------------------------
30-Oct 22:53:37:INFO:root:  Accuracy (avg over 10000) = 78.3400%
30-Oct 22:53:37:INFO:root:---------------------------------------------------------
30-Oct 22:53:37:INFO:root:
30-Oct 22:53:37:DEBUG:root:Destroying network CIFAR10-train
30-Oct 22:53:37:DEBUG:root:Destroying network CIFAR10-test
30-Oct 22:53:37:INFO:root:Shutting down CuDNN backend...
30-Oct 22:53:37:INFO:root:CuDNN Backend shutdown finished!
real19m13.617s
user14m1.893s
sys 5m12.049s
```

At the end of this tutorial you should have a good understanding of how to train Mocha networks in the cloud.

Thank you for making the Mocha community so awesome!

Report issues on this tutorial to our GitHub Page and we will get to them as soon as we can.

CHAPTER 2

User's Guide

2.1 Networks

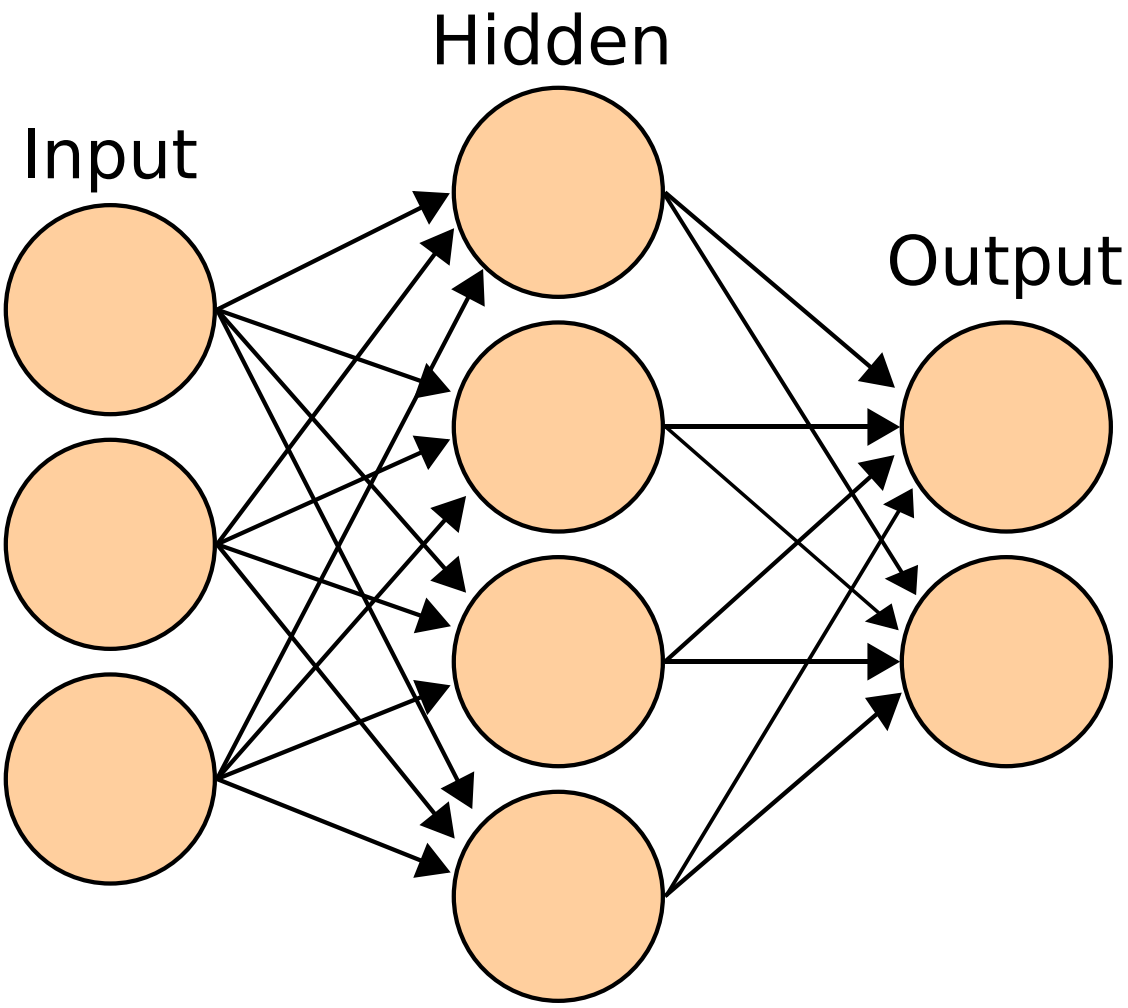
2.1.1 Overview

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, does some data pre-processing, and then produces 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 computes 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 layer using chain rules. Via back propagation, each layer can compute the gradients for its own parameters, and update the parameters according to some optimization schemes. Again, the computation is abstracted into layers.

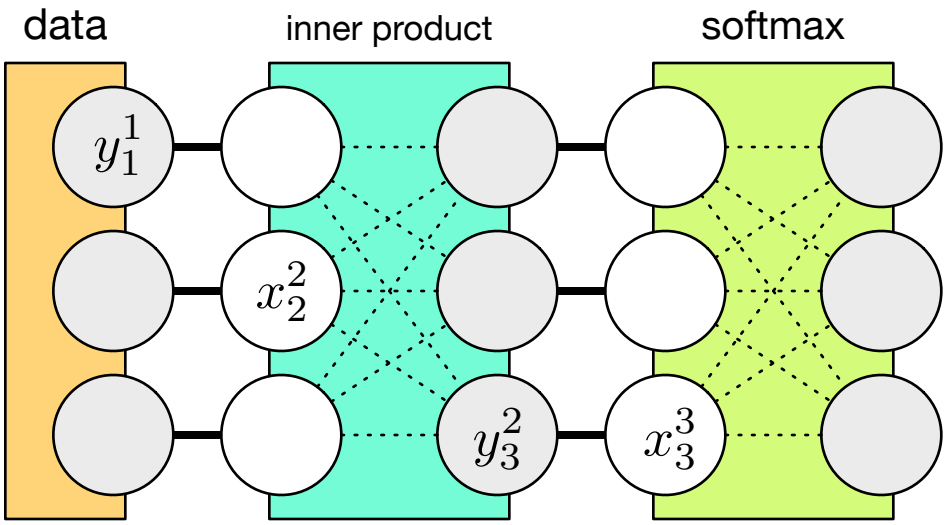
The abstraction and separation of *layers* from the *architecture* is important. The library implementation can focus on each layer type independently, and does not need to worry about how those layers are going to be connected with each other. On the other hand, the network designer can focus on the 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 layer, 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 computate anything, instead, computations are specified by the arrows connecting these nodes.

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 $x^L = y^{L-1}$. 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 how 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*. Depending on the abstraction we choose here, these vague terms become a concrete meaning.

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

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 ℓ. Of course, since we restrict the layer from accessing the outside world, it does not know what ℓ is. But the gradients could be computed via chain rule

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

The red part can be computed **within** the layer, and the blue part are 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

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

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

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

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.

2.1.4 Mocha Network Topology Tips

Shared Parameters

Consider a case where you want to construct *two* (or more) networks that share parameters. For example, during training, you want to have a *validation net* that shares parameters with the *training net*, yet takes a different data layer as input data stream and computes the accuracy on the validation set. In this case, simply using *the same* `Layer` object when constructing both networks will be enough. See *Training LeNet on MNIST* for a concrete example.

If you want to have **different** layers in **the same** network to share parameters, you can just use the same `param_key` property in the layers you want to share parameters. For example

```
layer_ip1 = InnerProductLayer(name="ip1", param_key="shared_ip",
    output_dim=512, bottoms=[:input1], tops=[:output1])
layer_ip2 = InnerProductLayer(name="ip2", param_key="shared_ip",
    output_dim=512, bottoms=[:input2], tops=[:output2])
```

If the two (or more) layers sharing parameters are of the same type (this is almost always true), an easier and more efficient way to do the same thing is simply to define one layer that takes multiple inputs and produce multiple outputs. For example, the snippet above is equivalent to

```
layer_ip = InnerProductLayer(name="ip", output_dim=512,
    bottoms=[:input1,:input2], tops=[:outpu1,:outpu2])
```

Not all layers accept multiple input blobs. Some layers require all the input blobs to be the same shape, while others can handle input blobs of completely different shapes. Please refer to the `bottoms` and `tops` properties of each layer for the detailed behavior of each layer.

Shared Blobs

In the basic case, a data path connects each output blob to one input blob. In some cases, one output could be used in multiple places. For example, in a test net, the output of the top representation layer will be used to compute the predictions, and produce either loss or accuracy; meanwhile, one might want to use a `HDF5OutputLayer` to store the representations as extracted features for future use. When the network is only doing *forward* operation, blob sharing is not a problem: multiple layers could be declared to take the same blob as input.

When you want to do *backward* operation (i.e. back-propagation for training) on the network, things could get a little bit complicated: If back-propagation does not go through the blob, than sharing is OK. For example, the output blob of a `HDF5DataLayer` does not need back-propagation. The output blob of a `ReshapeLayer` sitting directly on top of a data layer does not need back-propagation, either.

However, for a `InnerProductLayer`, even sitting directly on top of a data layer, its output blobs do need back-propagation, because the inner product layer needs back-propagation to compute gradients with respect to its weights and bias parameters. A `TopologyError` will be thrown when you try to do back-propagation on a network with this kind of Topology.

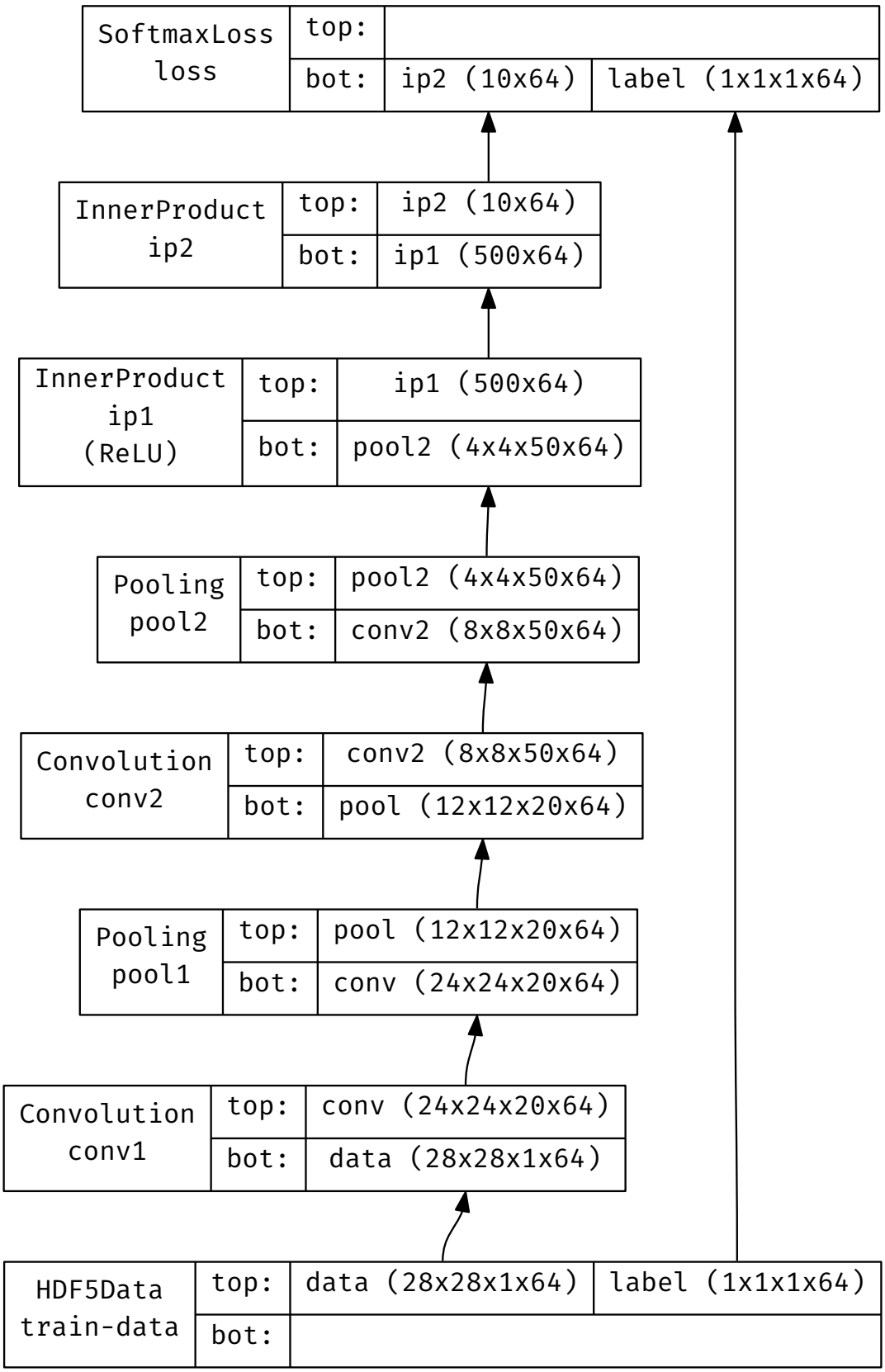
In this case, a `SplitLayer` can be used to explicitly "split" a blob into two (or more) "copies". The split layer can handle back-propagation correctly. Moreover, the forward operation of a split layer is implemented with data sharing instead of copying. Thus no extra cost is incurred during the forward pass.

2.1.5 Debugging

Mocha provides some utilities to show the structure of a network, which might be useful for debugging. First of all, you can just call `println` on a network object, the sorted layers will be printed, with basic information including blob names and shapes, etc. Alternatively, one can call `net2dot` to dump the network structure to a dot file, a script used by GraphViz. For example, if you have GraphViz installed, the following command

```
open("net.dot", "w") do out net2dot(out, net) end
run(`dot -Tpng net.dot` |> "net.png")
```

will generate a visualization of the network architecture in `net.png`. The following is a visualization of the network used in the *MNIST* example.


SoftmaxLoss
loss
top:
bot:
ip2 (10x64)
label (1x1x1x64)
InnerProduct
ip2
top:
ip2 (10x64)
bot:
ip1 (500x64)
InnerProduct
ip1
(ReLU)
top:
ip1 (500x64)
bot:
pool2 (4x4x50x64)
Pooling
pool2
top:
pool2 (4x4x50x64)
bot:
conv2 (8x8x50x64)
Convolution
conv2
top:
conv2 (8x8x50x64)
bot:
pool (12x12x20x64)
Pooling
pool1
top:
pool (12x12x20x64)
bot:
conv (24x24x20x64)
Convolution
conv1
top:
conv (24x24x20x64)
bot:
data (28x28x1x64)
HDF5Data
train-data
top:
data (28x28x1x64)
label (1x1x1x64)
bot:

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 Layers 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 scalar loss value. 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 is used to train the net parameters in back propagation.

Statistics Layers Take input from bottom layers and compute useful statistics like classification accuracy. Statistics are accumulated throughout multiple iterations. `reset_statistics` can be used to explicitly reset the statistics accumulation.

Utility Layers Other layers.

2.2.2 Data Layers

class AsyncHDF5DataLayer

Asynchronized HDF5 Data Layer. It has the same interface to `HDF5DataLayer`, except that

- The data IO is performed asynchronized with Julia coroutines. Noticeable speedups could typically be observed for large problems.
- The data is read in chunks. This allows fast data shuffling of HDF5 dataset without using `mmap`.

The properties are the same as `HDF5DataLayer`, with one more extra property controlling chunking.

chunk_size

Default `2^20`. The number of data points to read in each chunk. The data are read in chunks and cached in memory for fast random access, especially when data shuffling is turned on. Larger chunk size typically leads to better performance. Adjust this parameter according to the memory budget of your computing node.

Tip:

- The cache only occupies host memory even when GPU backend is used for computation.
- There is no correspondence between this chunk size and the *chunk size* property defined in a HDF5 dataset. They do not need to be the same.

---

class HDF5DataLayer

Starting from v0.0.7, Mocha.jl contains an `AsyncHDF5DataLayer`, which is typically more preferable than this one.

Loads data from a list of HDF5 files and feeds 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 N-dimensional tensor. The last tensor dimension (the slowest changing one) is treated as the *number* dimension, and split for mini-batch. For more details for ND-tensor blobs used in Mocha, see *Blob*.

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.

transformers
Default `[]`. List of data transformers. Each entry in the list should be a tuple of `(name, transformer)`, where `name` is a symbol of the corresponding output blob name, and `transformer` is a *data transformer* that should be applied to the blob with the given name. Multiple transformers could be given to the same blob, and they will be applied in the order provided here.

shuffle
Default `false`. When enabled, the data is randomly shuffled. Data shuffling is useful in training, but for testing, there is no need to do shuffling. Shuffled access is a little bit slower, and it requires the HDF5 dataset to be *mmappable*. For example, the dataset can neither be chunked nor be compressed. Please refer to the documention for HDF5.jl for more details.

Note: Current mmap in HDF5.jl does not work on Windows. See issue 89 on Github.

---

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

tops
Default `[:data, :label]`. 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.

transformers
Default `[]`. See `transformers` of `HDF5DataLayer`.

2.2.3 Computation Layers

class ArgmaxLayer
Compute the arg-max along the "channel" dimension. This layer is only used in the test network to produce predicted classes. It has no ability to do back propagation.

dim
Default `-2` (penultimate). Specify which dimension to operate on.

tops
bottoms
Blob names for output and input. This layer can take multiple input blobs and produce the corresponding number of output blobs. The shapes of the input blobs do not need to be the same.

class ChannelPoolingLayer
1D pooling over any specified dimension. This layer is called channel pooling layer because it was designed to pool over the pre-defined *channel* dimension back when Mocha could only handle 4D tensors. For the new, general ND-tensors the dimension to be pooled over can be freely specified by the user.

channel_dim
Default `-2` (penultimate). Specifies which dimension to pool over.

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. This layer can take multiple input blobs and produce the corresponding number of output blobs. The shapes of the input blobs do not need to be the same.

class ConvolutionLayer
Convolution in the spatial dimensions. **For now** convolution layers require the input blobs to be 4D tensors. For a 4D input blob of the shape `width`-by-`height`-by-`channels`-by-`num`, The output blob shape is decided by the `kernel` size (a.k.a. receptive field), the `stride`, the `pad` and the `n_filter`.

The `kernel` size specifies the geometry of a filter, also called a kernel or a local receptive field. Note that implicitly, a filter also has a channel dimension that is the same size as the input image. As a filter moves across the image by the specified `stride` and optionally `pad` when on the boundary of the input image, it produce a real number by computing the inner-product between the filter weights and the local image patch at each spatial position. The formula for the spatial dimension of the output blob is

```
width_out  = div(width_in  + 2*pad[1]-kernel[1], stride[1]) + 1
height_out = div(height_in + 2*pad[2]-kernel[2], stride[2]) + 1
```

The `n_filter` parameter specifies the number of such filters. The final output blob will have the shape `width_out`-by-`height_out`-by-`n_filter`-by-`num`. An illustration of typical convolution (and pooling) is shown below:

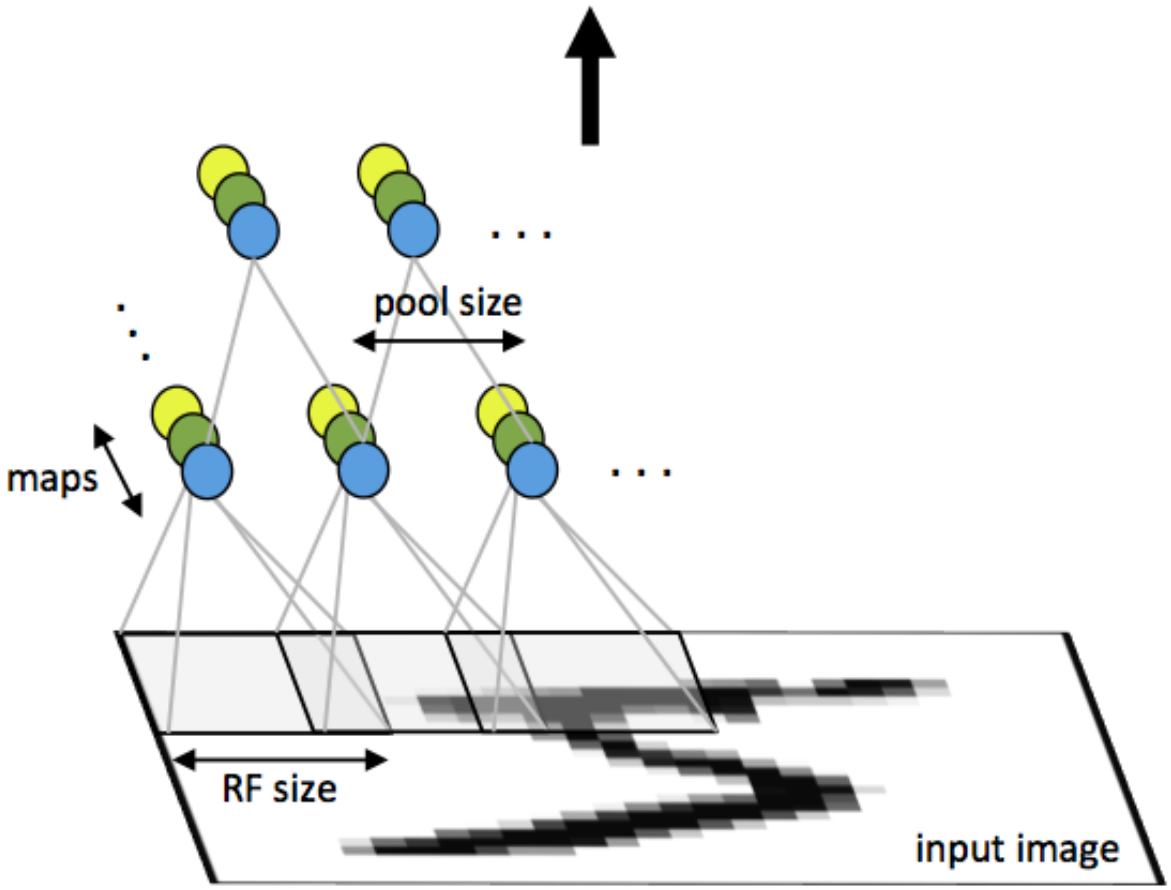


Image credit: http://ufldl.stanford.edu/tutorial/supervised/ConvolutionalNeuralNetwork/

Here the *RF size* is *receptive field size*, and *maps* (identified by different colors) correspond to different filters.

param_key
Default `""`. The unique identifier for layers with shared parameters. When empty, the layer `name` is used as identifier instead.

kernel
Default (1,1), a 2-tuple specifying the width and height of the convolution filters.

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

pad
Default (0,0), a 2-tuple specifying the two-sided padding in the width and height dimensions, respectively.

n_filter
Default 1. Number of filters.

n_group
Default 1. Number of groups. This number should divide both `n_filter` and the number of channels in the input blob. This parameter will divide the input blob along the channel dimension into `n_group` groups. Each group will operate independently. Each group is assigned with `n_filter` / `n_group` filters.

neuron
Default `Neurons.Identity()`, can be used to specify an activation function for the convolution outputs.

filter_init
Default `XavierInitializer()`. See *initializer* for the filters.

bias_init
Default `ConstantInitializer(0)`. See *initializer* for the bias.

filter_regu
Default `L2Regu(1)`, the regularizer for the filters.

bias_regu
Default `NoRegu()`, the regularizer for the bias.

filter_cons
Default `NoCons()`. *Norm constraint* for the filters.

bias_cons
Default `NoCons()`. Norm constraint for the bias. Typically no norm constraint should be applied to the bias.

filter_lr
Default 1.0. The local learning rate for the filters.

bias_lr
Default 2.0. The local learning rate for the bias.

tops
bottoms
Blob names for output and input. This layer can take multiple input blobs and produce the corresponding number of output blobs. The shapes of the input blobs **must be the same**.

class CropLayer
Do image cropping. This layer is primarily used only on top of data layers so backpropagation is currently not implemented. Crop layer requires the input blobs to be 4D tensors.

crop_size
A (width, height) tuple of the size of the cropped image.

random_crop
Default `false`. When enabled, randomly place the cropping box instead of putting at the center. This is useful to produce random perturbations of the input images during training.

random_mirror
Default `false`. When enabled, randomly (with probability 0.5) mirror the input images (flip the width dimension).

tops
bottoms
Blob names for output and input. This layer can take multiple input blobs and produce the corresponding number of output blobs. The shapes of the input blobs do not need to be the same as long as they are valid (not smaller than the shape specified in `crop_size`).

class DropoutLayer
Dropout is typically used during training, and it has been demonstrated to be effective as a regularizer for large scale networks. Dropout operates by randomly "turning off" some responses. Specifically, the forward computation is

$$y = \begin{cases} \frac{x}{1-p} & u > p \\ 0 & u <= p \end{cases}$$

where u is a random number uniformly distributed in [0,1], and p is the `ratio` hyper-parameter. Note the output is scaled by $1 - p$ such that $\mathbb{E}[y] = x$.

ratio
The probability p of turning off a response. Can also be interpreted as the ratio of all the responses that are turned off.

auto_scale
Default `true`. When turned off, does not scale the result by $1/(1-p)$. This option is used when building `RandomMaskLayer`.

bottoms
The names of the input blobs dropout operates on. Note this is a *in-place layer*, so

1. there is no `tops` property. The output blobs will be the same as the input blobs.
2. It takes **only one** input blob.

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

operation
Element-wise operation. Built-in operations are defined 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 InnerProductLayer
Densely connected linear layer. The output is computed as

$$y_i = \sum_j w_{ij} x_j + b_i$$

where w_{ij} are the weights and b_i are the biases.

param_key
Default `""`. The unique identifier for layers with shared parameters. When empty, the layer `name` is used as identifier instead.

output_dim
Output dimension of the linear map. The input dimension is automatically decided via the inputs.

weight_init
Default `XavierInitializer()`. Specify how the weights w_{ij} should be initialized.

bias_init
Default `ConstantInitializer(0)`, initializing the bias b_i to 0.

weight_regu
Default `L2Regu(1)`. *Regularizer* for the weights.

bias_regu
Default `NoRegu()`. Regularizer for the bias. Typically no regularization should be applied to the bias.

weight_cons
Default `NoCons()`. *Norm constraint* for the weights.

bias_cons
Default `NoCons()`. Norm constraint for the bias. Typically no norm constraint should be applied to the bias.

weight_lr
Default 1.0. The local learning rate for the weights.

bias_lr
Default 2.0. The local learning rate for the bias.

neuron
Default `Neurons.Identity()`, an optional *activation function* for the output of this layer.

tops
bottoms
Blob names for output and input. This layer can take multiple input blobs and produce the corresponding number of output blobs. The feature dimensions (the product of the first N-1 dimensions) of all input blobs should be the same, but they could potentially have different batch sizes (the last dimension).

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 $n = k^2$.

  When this mode is used, the input blobs should be 4D tensors **for now**, due to the requirements from the underlying `PoolingLayer`.

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 PoolingLayer
2D pooling over the 2 image dimensions (width and height). **For now** the input blobs are required to be 4D tensors.

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. This layer can take multiple input blobs and produce the corresponding number of output blobs. The shapes of the input blobs do not need to be the same.

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 special cases of the Power layer that can be computed more efficiently.

power
Default 1

scale
Default 1

shift
Default 0

tops
bottoms
Blob names for output and input. This layer can take multiple input blobs and produce the corresponding number of output blobs. The shapes of the input blobs do not need to be the same.

class RandomMaskLayer
Randomly mask subsets of input as zero. This is a wrapper over `DropoutLayer`, but

- This layer does not rescale the un-masked part to make the expectation the same as the expectation of the original input.
- This layer can handle multiple input blobs while `DropoutLayer` accept only one input blob.

Note:

- This layer is a in-place layer. For example, if you want to use this to construct a denoising auto-encoder, you should use a `SplitLayer` to make two copies of the input data: one is randomly masked (in-place) as the input of the auto-encoder, and the other is directed to a `SquareLoss` layer that measure the reconstruction error.
- Although typically not used, this layer is capable of doing back-propagation, powered by the underlying `DropoutLayer`.

---

class SoftmaxLayer

Compute softmax over the "channel" dimension. The inputs $x_1, \ldots, x_C$ are mapped as

$$\sigma(x_1, \ldots, x_C) = (\sigma_1, \ldots, \sigma_C) = \left(\frac{e^{x_1}}{\sum_j e^{x_j}}, \ldots, \frac{e^{x_C}}{\sum_j e^{x_j}} \right)$$

To train a multi-class classification network with softmax probability output and multiclass logistic loss, use the bundled `SoftmaxLossLayer` instead.

dim

Default `-2` (penultimate). Specify the "channel" dim to operate along.

tops

bottoms

Blob names for output and input. This layer can take multiple input blobs and produce the corresponding number of output blobs. The shapes of the input blobs do not need to be the same.

class TiedInnerProductLayer

Similar to `InnerProductLayer` but with *weights tied* to an existing `InnerProductLayer`. Used in auto-encoders. During training, an auto-encoder defines the following mapping

$$\mathbf{x} \longrightarrow \mathbf{h} = \mathbf{W}_1^T \mathbf{x} + \mathbf{b}_1 \longrightarrow \tilde{\mathbf{x}} = \mathbf{W}_2^T \mathbf{h} + \mathbf{b}_2$$

Here $\mathbf{x}$ is input, $\mathbf{h}$ is the latent encoding, and $\tilde{\mathbf{x}}$ is the decoded reconstruction of the input. Sometimes it is desired to have *tied weights* for the encoder and decoder: $\mathbf{W}_1 = \mathbf{W}^T$. In this case, the encoder will be an `InnerProductLayer`, and the decoder a `TiedInnerProductLayer` with tied weights to the encoder layer.

Note the tied decoder layer does *not* perform learning for the weights. However, even a tied layer has independent bias parameters that are learned independently.

tied_param_key

The `param_key` of the encoder layer that this layer wants to share tied weights with.

param_key

Default `""`. The unique identifier for layers with shared parameters. If empty, the layer `name` is used as identifier instead.

Tip:

- `param_key` is used for `TiedInnerProductLayer` to share parameters. For example, the same layer in a training net and in a validation / testing net use this mechanism to share parameters.
- `tied_param_key` is used to find the `InnerProductLayer` to enable *tied weights*. This should be equal to the `param_key` property of the inner product layer you want to have tied weights with.

---

bias_init

Default `ConstantInitializer(0)`. The *initializer* for the bias.

bias_regu

Default `NoRegu()`, the regularizer for the bias.

bias_cons

Default `NoCons()`. Norm constraint for the bias. Typically no norm constraint should be applied to the bias.

bias_lr

Default 2.0. The local learning rate for the bias.

neuron
Default `Neurons.Identity()`, an optional *activation function* for the output of this layer.

tops
bottoms
Blob names for output and input. This layer can take multiple input blobs and produce the corresponding number of output blobs. The feature dimensions (the product of the first N-1 dimensions) of all input blobs should be the same, but they can potentially have different batch sizes (the last dimension).

class RandomNormalLayer

This is a source layer which outputs standard Gaussian random noise.

tops
List of symbols, specifying the names of the noise blobs to produce.

output_dims
List of integers giving the dimensions of the output noise blobs.

batch_sizes
List of integers the same length as `tops`, giving the number of vectors to output in each batch.

eltype
Default `Float32`.

2.2.4 Loss Layers

class HingeLossLayer
Compute the hinge loss for binary classification problems:

$$\frac{1}{N}\sum_{i=1}^{N}\max(1-\mathbf{y}_i\cdot\hat{\mathbf{y}}_i,0)$$

Here N is the batch-size, $\mathbf{y}_i\in\{-1,1\}$ is the ground-truth label of the i-th sample, and $\hat{\mathbf{y}}_i$ is the corresponding prediction.

weight
Default `1.0`. Weight of this loss function. Could be useful when combining multiple loss functions in a network.

bottoms
Should be a vector containing two symbols. The first one specifies the name for the prediction $\hat{\mathbf{y}}$, and the second one specifies the name for the ground-truth $\mathbf{y}$.

class MultinomialLogisticLossLayer
The multinomial logistic loss is defined as $\ell=-w_g\log(x_g)$, where $x_1,\ldots,x_C$ are probabilities for each of the C classes conditioned on the input data, g is the corresponding ground-truth category, and w_g is the *weight* for the g-th class (default 1, see bellow).

If the conditional probability blob is of the shape `(dim1, dim2, ..., dim_channel, ..., dimN)`, then the ground-truth blob should be of the shape `(dim1, dim2, ..., 1, ..., dimN)`. Here `dim_channel`, historically called the "channel" dimension, is the user specified tensor dimension to compute loss on. This general case allows to produce multiple labels for each sample. For the typical case where only one (multi-class) label is produced for one sample, the conditional probability blob is the shape `(dim_channel, dim_num)` and the ground-truth blob should be of the shape `(1, dim_num)`.

The ground-truth should be a **zero-based** index in the range of $0,\ldots,C-1$.

bottoms
Should be a vector containing two symbols. The first one specifies the name for the conditional probability input blob, and the second one specifies the name for the ground-truth input blob.

weight
Default `1.0`. Weight of this loss function. Could be useful when combining multiple loss functions in a network.

weights
This can be used to specify weights for different classes. The following values are allowed

- Empty array (default). This means each category should be equally weighted.
- A 1D vector of length `channels`. This defines weights for each category.
- An (N-1)D tensor of the shape of a data point. In other words, the same shape as the prediction except that the last mini-batch dimension is removed. This is equivalent to the above case if the prediction is a 2D tensor of the shape `channels`-by-`mini-batch`.
- An ND tensor of the same shape as the prediction blob. This allows us to fully specify different weights for different data points in a mini-batch. See `SoftlabelSoftmaxLossLayer`.

dim
Default `-2` (penultimate). Specify the dimension to operate on.

normalize
Indicating how weights should be normalized if given. The following values are allowed

- `:local` (default): Normalize the weights locally at each location (w,h), across the channels.
- `:global`: Normalize the weights globally.
- `:no`: Do not normalize the weights.

The weights normalization are done in a way that you get the same objective function when specifying *equal weights* for each class as when you do not specify any weights. In other words, the total sum of the weights are scaled to be equal to weights x height x channels. If you specify `:no`, it is your responsibility to properly normalize the weights.

class SoftlabelSoftmaxLossLayer
Like the `SoftmaxLossLayer`, except that this deals with *soft labels*. For multiclass classification with K categories, we call an integer value $y \in \{0, \ldots, K-1\}$ a *hard label*. In contrast, a soft label is a vector on the K-dimensional simplex. In other words, a soft label specifies a probability distribution over all the K categories, while a hard label is a special case where all the probability masses concentrates on one single category. In this case, this loss is basically computing the KL-divergence D(p||q), where p is the ground-truth softlabel, and q is the predicted distribution.

dim
Default `-2` (penultimate). Specify the dimension to operate on.

weight
Default `1.0`. Weight of this loss function. Could be useful when combining multiple loss functions in a network.

bottoms
Should be a vector containing two symbols. The first one specifies the name for the conditional probability input blob, and the second one specifies the name for the ground-truth (soft labels) input blob.

class SoftmaxLossLayer
This is essentially a combination of `MultinomialLogisticLossLayer` and `SoftmaxLayer`. The given

predictions $x_1, \ldots, x_C$ for the C classes are transformed with a softmax function

$$\sigma(x_1, \ldots, x_C) = (\sigma_1, \ldots, \sigma_C) = \left(\frac{e^{x_1}}{\sum_j e^{x_j}}, \ldots, \frac{e^{x_C}}{\sum_j e^{x_j}} \right)$$

which essentially turn the predictions into non-negative values with exponential function and then re-normalize to make them look like probabilties. Then the transformed values are used to compute the multinomial logsitic loss as

$$\ell = -w_g \log(\sigma_g)$$

Here g is the ground-truth label, and w_g is the weight for the g-th category. See the document of `MultinomialLogisticLossLayer` for more details on what the weights mean and how to specify them.

The shapes of the inputs are the same as for the `MultinomialLogisticLossLayer`: the multi-class predictions are assumed to be along the channel dimension.

The reason we provide a combined softmax loss layer instead of using one softmax layer and one multinomial logistic layer is that the combined layer produces the back-propagation error in a more numerically robust way.

$$\frac{\partial \ell}{\partial x_i} = w_g \left(\frac{e^{x_i}}{\sum_j e^{x_j}} - \delta_{ig} \right) = w_g \left(\sigma_i - \delta_{ig} \right)$$

Here δ_{ig} is 1 if $i = g$, and 0 otherwise.

bottoms
: Should be a vector containing two symbols. The first one specifies the name for the conditional probability input blob, and the second one specifies the name for the ground-truth input blob.

dim
: Default `-2` (penultimate). Specify the dimension to operate on. For a 4D vision tensor blob, the default value (penultimate) translates to the 3rd tensor dimension, usually called the "channel" dimension.

weight
: Default `1.0`. Weight of this loss function. Could be useful when combining multiple loss functions in a network.

weights
normalize
: Properties for the underlying `MultinomialLogisticLossLayer`. See its documentation for details.

class SquareLossLayer
Compute the square loss for real-valued regression problems:

$$\frac{1}{2N} \sum_{i=1}^{N} \|\mathbf{y}_i - \hat{\mathbf{y}}_i\|^2$$

Here N is the batch-size, $\mathbf{y}_i$ is the real-valued (vector or scalar) ground-truth label of the i-th sample, and $\hat{\mathbf{y}}_i$ is the corresponding prediction.

weight
: Default `1.0`. Weight of this loss function. Could be useful when combining multiple loss functions in a network.

bottoms
: Should be a vector containing two symbols. The first one specifies the name for the prediction $\hat{\mathbf{y}}$, and the second one specifies the name for the ground-truth $\mathbf{y}$.

class BinaryCrossEntropyLossLayer

A simpler alternative to `MultinomialLogisticLossLayer` for the special case of binary classification.

$$-\frac{1}{N}\sum_{i=1}^{N}\log(p_i)y_i + \log(1-p_i)(1-y_i)$$

Here N is the batch-size, $\mathbf{y}_i$ is the ground-truth label of the i-th sample, and :math:`p_i` is the corresponding prediction.

weight

Default `1.0`. Weight of this loss function. Could be useful when combining multiple loss functions in a network.

bottoms

Should be a vector containing two symbols. The first one specifies the name for the prediction $\hat{\mathbf{y}}$, and the second one specifies the name for the binary ground-truth labels $\mathbf{p}$.

class GaussianKLLossLayer

Given two inputs *mu* and *sigma* of the same size representing the means and standard deviations of a diagonal multivariate Gaussian distribution, the loss is the Kullback-Leibler divergence from that to the standard Gaussian of the same dimension.

Used in variational autoencoders, as in Kingma & Welling 2013, as a form of regularization.

$$D_{KL}(\mathcal{N}(\mu, \mathrm{diag}(\sigma))\|\mathcal{N}(\mathbf{0}, \mathbf{I})) = -\frac{1}{2}\left(\sum_{i=1}^{N}(\mu_i^2 + \sigma_i^2 - 2\log\sigma_i) - N\right)$$

weight

Default `1.0`. Weight of this loss function. Could be useful when combining multiple loss functions in a network.

bottoms

Should be a vector containing two symbols. The first one specifies the name for the mean vector μ, and the second one the vector of standard deviations σ.

2.2.5 Statistics Layers

class AccuracyLayer

Compute and accumulate multi-class classification accuracy. The accuracy is averaged over mini-batches. If the spatial dimension is not singleton, i.e. there are multiple labels for each data instance, then the accuracy is also averaged among the spatial dimension.

bottoms

The blob names for prediction and labels (in that order).

dim

Default `-2` (penultimate). Specifies the dimension to operate on.

class BinaryAccuracyLayer

Compute and accumulate binary classification accuracy. The accuracy is averaged over mini-batches. Labels can be either {0, 1} labels or {-1, +1} labels

bottoms

The blob names for prediction and labels (in that order).

2.2.6 Utility Layers

class ConcatLayer

Concatenates multiple blobs into a single blob along the specified dimension. Except in the concatenation dimension, the shapes of the blobs being concatenated have to be the same.

dim

Default 3 (channel). The dimension to concatenate.

bottoms

Names of the blobs to be concatenated.

tops

Name of the concatenated output blob.

class MemoryOutputLayer

Takes some blobs in the network and collect their data during forward pass of the network as a list of julia `Array` objects. Useful when doing in-memory testing for collecting the output. After running the forward pass of the network, the `outputs` field of the corresponding layer state object will contain a vector of the same size as the `bottoms` attribute. Each element of the vector is a list of tensors (julia `Array` objects), each tensor corresponds to the output in a mini-batch.

bottoms

A list of names of the blobs in the network to store.

class HDF5OutputLayer

Takes some blobs in the network and writes them to a HDF5 file. Note that the target HDF5 file will be overwritten when the network is first constructed, but later iterations will **append** data for each mini-batch. This is useful for storing the final predictions or the intermediate representations (feature extraction) of a network.

filename

The path to the target HDF5 file.

force_overwrite

Default `false`. When the layer tries to create the target HDF5 file, if this attribute is enabled, it will overwrite any existing file (with a warning printed). Otherwise, it will raise an exception and refuse to overwrite the existing file.

bottoms

A list of names of the blobs in the network to store.

datasets

Default `[]`. Should either be empty or a list of `Symbol` of the same length as `bottoms`. Each blob will be stored as an HDF5 dataset in the target HDF5 file. If this attribute is given, the corresponding symbol in this list is used as the dataset name instead of the original blob's name.

class IdentityLayer

An Identity layer maps inputs to outputs without changing anything. This can be useful as a glue layer to rename some blobs. There is no data-copying for this layer.

tops
bottoms

Blob names for output and input. This layer can take multiple input blobs and produce the corresponding number of output blobs. The shapes of the input blobs do not need to be the same.

class Index2OnehotLayer

A utility layer that could convert category class into one-hot encoded vector. For example, for K classes, input j is converted into a vector of size K, with all zeros, but the (j-1)-th entry 1.

dim
The dimension to operate on. The input must have size 1 on this dimension, i.e. `size(input, dim) == 1`. And the value should be integers from 0 to (K-1).

n_class
Number of categories, i.e. K as described above.

tops
bottoms
Blob names for output and input. This layer can take multiple input blobs and produce the corresponding number of output blobs. The shapes of the input blobs do not need to be the same. But they will be operated on the same dimension, and the `n_class` for them are the same.

class ReshapeLayer
Reshapes a blob. Can be useful if, for example, you want to make the *flat* output from an `InnerProductLayer` *meaningful* by assigning each dimension spatial information.

Internally, no data is copied. The total number of elements in the blob tensor after reshaping has to be the same as the original blob tensor.

shape
Has to be an `NTuple` of `Int` specifying the new shape. Note that the new shape does not include the last (mini-batch) dimension of a data blob. So a reshape layer cannot change the mini-batch size of a data blob.

tops
bottoms
Blob names for output and input. This layer can take multiple input blobs and produce the corresponding number of output blobs. The shapes of the input blobs do not need to be the same. But the feature dimensions (i.e. the product of the first 3 dimensions) have to be the same.

class SplitLayer
A Split layer produces identical copies 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.

no_copy
Default `false`. When `true`, no data is copied in the forward pass. In this case, all the output blobs share data. When, for example, an *in-place* layer is used to modify one of the output blobs, all the other output blobs will also change.

2.3 Neurons (Activation Functions)

Neurons can be attached to any layer. 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. Layers have an identity neuron by default[1].

[1] 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.

class Neurons.**Identity**
An activation function that does not change its input.

class Neurons.**ReLU**
Rectified Linear Unit. During the forward pass, it inhibits all inhibitions below some threshold ϵ, typically 0. In other words, it computes point-wise $y = \max(\epsilon, x)$. The point-wise derivative for ReLU is

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

epsilon
Specifies the minimum threshold at which the neuron will truncate. Default `0`.

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

---

class Neurons.**LReLU**
Leaky Rectified Linear Unit. A Leaky ReLU can help fix the "dying ReLU" problem. ReLU's can "die" if a large enough gradient changes the weights such that the neuron never activates on new data.

$$\frac{dy}{dx} = \begin{cases} 1 & x > 0 \\ 0.01 & x \leq 0 \end{cases}$$

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

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

class Neurons.**Tanh**
Tanh is a transformed version of Sigmoid, that takes values in ± 1 instead of the unit interval. input with large absolute values and approximate 1 for large positive inputs. The point-wise formula is $y = (1 - e^{-2x})/(1 + e^{-2x})$. The point-wise derivative is

$$\frac{dy}{dx} = 4e^{2x}/(e^{2x} + 1)^2 = (1 - y^2)$$

class Neurons.**Exponential**
The exponential function.

$$y = exp(x)$$

2.4 Initializers

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

class NullInitializer
An initializer that does nothing. To initialize with zeros, use a ConstantInitializer.

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.

Heuristics are used to determine the fan-in: For a ND tensor parameter blob, the product of all the 1 to N-1 dimensions are considered as fan-in, while the last dimension is considered as fan-out.

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

mean
Default 0.

std
Default 1.

class OrthogonalInitializer
Initialize the parameter blob to be a random orthogonal matrix (i.e. $W^T W = I$), times a scalar gain factor. Based on *[Saxe2013]*.

gain
Default 1. Use $\sqrt{2}$ for layers with ReLU activations.

2.5 Regularizers

Regularizers add extra penalties or constraints for network parameters to restrict the model complexity. The corresponding term used in Caffe is *weight decay*. Regularization and weight decay are equivalent in back-propagation. The *conceptual* difference in the forward pass is that when treated as weight decay, they are not considered being part of the objective function. However, in order to reduce the number of computations, Mocha also omits the 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), which globally scales all local regularization coefficients.

class NoRegu
Regularizer that imposes 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.

---

class L1Regu
L1 regularizer. The parameter blob W is treated as a 1D vector. During the forward pass, the L1-norm

$$\|W\|_1 = \sum_i |W_i|$$

is computed, and $\lambda\|W\|_1$ is added to the objective function. During the backward pass, $\lambda \text{sign}(W)$ is added to the parameter gradient. The L1 regularizer has the property of encouraging sparsity in the parameters.

2.6 Norm Constraints

Norm constraints is a more "direct" way of restricting the model complexity by explicitly shrinking the parameters every *n* iterations if the norm of the parameters exceeds a given threshold.

class NoCons
No constraint is applied.

class L2Cons
Constrain the Euclidean norm of parameters. Note that the threshold and shrinking are applied to *each parameter*. Specifically, for the filters parameter of a convolution layer, the threshold is applied to each filter. Similarly, for the weights parameter of an inner product layer, the threshold is applied to the weights corresponding to each single output dimension of the inner product layer. When the norm of the parameter exceed the threshold, it is scaled down to have exactly the norm specified in threshold.

See the MNIST with dropout code in the `examples` directory for an example of how `L2Cons` is used.

threshold
The norm threshold.

every_n_iter
Defautl 1. Indicates the frequency of norm constraint application.

2.7 Data Transformers

Data transformers apply transformations to data. Note that the transformations are limited to simple, in-place operations that do not change the shape of the data. If more complicated transformations like random projection or feature mapping are needed, consider using a data transformation **layer** instead.

class `DataTransformers.`**SubMean**
Subtract mean from the data. The transformer does not have enough information to compute the data mean, thus the mean should be computed in advance.

mean_blob
Default `NullBlob()`. A blob containing the mean.

mean_file
Default `""`. When `mean_blob` is a `NullBlob`, this can be used to specify a HDF5 file containing the mean. The mean should be stored with the name `mean` in the HDF5 file.

class `DataTransformers.`**Scale**
Perform elementwise scaling of the data. This is useful, for example, when you want to scale the data to, say, the range [0,1].

scale
Default 1.0. The scaling factor to apply.

2.8 Solvers

Mocha contains general purpose stochastic (sub-)gradient based solvers that can be used to train deep neural networks as well as traditional shallow machine learning models.

A solver is constructed by specifying a dictionary of *solver parameters* that provide necessary configuration: both general settings like *stop conditions*, and parameters specific to a particular algorithm such as *momentum policy*.

You then instantiate the *algorithm* that characterizes how the parameters are updated in each solver iteration. The following is an example taken from the *MNIST tutorial*.

```
method = SGD()
params = make_solver_parameters(method, max_iter=10000, regu_coef=0.0005,
    mom_policy=MomPolicy.Fixed(0.9),
    lr_policy=LRPolicy.Inv(0.01, 0.0001, 0.75),
    load_from=exp_dir)
solver = Solver(method, params)
```

Moreover, it is usually desired to do some short breaks during training iterations, for example, to print training progress or to save a snapshot of the trained model to the disk. In Mocha, these are called *coffee breaks* for solvers.

2.8.1 General Solver Parameters

A instance of `SolverParameters` is just a `Dictionary` with `Symbol` keys. The `make_solver_parameters` function helps to construct this, providing default values suitable for the solver method.

Some parameters apply to all methods:

`max_iter`
: Maximum number of iterations to run.

`regu_coef`
: Global regularization coefficient. Used as a global scaling factor for the local regularization coefficient of each trainable parameter.

`load_from`
: If specified, the solver will try to load a trained network before starting the solver loop. This parameter can be

  - The path to a directory: Mocha will try to locate the latest saved JLD snapshot in this directory and load it. A mocha snapshot contains a trained model and the solver state. So the solver loop will continue from the saved state instead of re-starting from iteration 0.
  - The path to a particular JLD snapshot file. The same as above except that the user controls which particular snapshot to load.
  - The path to a HDF5 model file. A HDF5 model file does not contain solver state information. So the solver will start from iteration 0, but initialize the network from the model saved in the HDF5 file. This can be used to fine-tune a trained (relatively) general model on a domain specific (maybe smaller) dataset. You can also load HDF5 models *exported from external deep learning tools*.

2.8.2 Solver Algorithms

The different solver methods are listed below, together with the `SolverParameters` arguments particular to them.

class SGD
Stochastic Gradient Descent with momentum.

lr_policy
Policy for learning rate. Note that this is also a global scaling factor, as each trainable parameter also has a local learning rate.

mom_policy
Policy for momentum.

class Nesterov
Stochastic Nesterov accelerated gradient method.

lr_policy
Policy for learning rate, as for SGD.

mom_policy
Policy for momentum, as for SGD.

class Adam
As described in Adam: A Method for Stochastic Optimization.

(N.B. The Adam solver sets effective learning rates for each parameter individually, so the layer local learning rates are ignored in this case.)

lr_policy
Policy for learning rate, as for SGD. While the relative learning rates are set adaptively per parameter, the learning rate still limits the maximum step for each parameter. Accordingly a fine-tuning schedule can be useful, as for other methods.

beta1
Exponential decay factor for 1st order moment estimates, 0<=beta1<1, default 0.9

beta2
Exponential decay factor for 2nd order moment estimates, 0<=beta1<1, default 0.999

epsilon
Affects scaling of the parameter updates for numerical conditioning, default 1e-8

Learning Rate Policy

class LRPolicy.**Fixed**
A fixed learning rate.

class LRPolicy.**Step**
Provide the learning rate as base_lr * gamma $^{\text{floor(iter / stepsize)}}$. Here *base_lr*, *gamma* and *stepsize* are parameters for the policy and *iter* is the training iteration.

class LRPolicy.**Exp**
Provide the learning rate as base_lr * gamma $^{\text{iter}}$. Here *base_lr* and *gamma* are parameters for the policy and *iter* is the training iteration.

class LRPolicy.**Inv**
Provide the learning rate as base_lr * (1 + gamma * iter) $^{\text{-power}}$. Here *base_lr*, *gamma* and *power* are parameters for the policy and *iter* is the training iteration.

class LRPolicy.**Staged**
This policy provides different learning rate policies at different *stages*. Stages are specified by number of training iterations. See *the CIFAR-10 tutorial* for an example of staged learning rate policy.

class LRPolicy.**DecayOnValidation**

This policy starts with a base learning rate. Each time the performance on a validation set is computed, the policy will scale the learning rate down by a given factor if the validation performance is poorer compared to the one of the last snapshot. In this case it also asks the solver to load the latest saved snapshot and restart from there.

Note in order for this policy to function properly, you need to set up both `Snapshot` coffee break and `ValidationPerformance` coffee break. The policy works by registering a listener on the `ValidationPerformance` coffee break. Whenever the performance is computed on a validation set, the listener is notified, and it will compare the performance with the previous one on records. If the performance decays, it will ask the solver to load the previously saved snapshot (saved by the `Snapshot` coffee break), and then scale the learning rate down. Per default *LRPolicy.DecayOnValidation* considers a lower performance statistic as better, however this can be changed by setting the optional argument *higher_better* to *false*.

A typical setup is to save one snapshot every epoch, and also check the performance on the validation set every epoch. So if the performance decays, the learning rate is decreased, and the training will restart from the last (good) epoch.

```
# starts with lr=base_lr, and scale as lr=lr*lr_ratio
lr_policy=LRPolicy.DecayOnValidation(base_lr,"accuracy-accuracy",lr_ratio)

validation_performance = ValidationPerformance(test_net)
add_coffee_break(solver, validation_performance, every_n_epoch=1)

# register the listener to get notified on performance validation
setup(params.lr_policy, validation_performance, solver)
```

Momentum Policy

class MomPolicy.**Fixed**

Provide fixed momentum.

class MomPolicy.**Step**

Provide the momentum as min(base_mom * gamma $^{\text{floor(iter / stepsize)}}$, max_mom). Here *base_mom*, *gamma*, *stepsize* and *max_mom* are policy parameters and *iter* is the training iteration.

class MomPolicy.**Linear**

Provide the momentum as min(base_mom + floor(iter / stepsize) * gamma, max_mom). Here *base_mom*, *gamma*, *stepsize* and *max_mom* are policy parameters and *iter* is the training iteration.

class MomPolicy.**Staged**

This policy provides different momentum policies at different *stages*. Stages are specified by number of training iterations. See `LRPolicy.Staged`.

2.8.3 Solver Coffee Breaks

Training is a very computationally intensive loop of iterations. Being afraid that the solver might silently go crazy under such heavy load, Mocha provides the solver opportunities to have a break periodically. During the breaks, the solver can have a change of mood by, for example, talking to the outside world about its "mental status". Here is a snippet taken from *the MNIST tutorial*:

```
# report training progress every 100 iterations
add_coffee_break(solver, TrainingSummary(), every_n_iter=100)
```

(continues on next page)

(continued from previous page)

```
# save snapshots every 5000 iterations
add_coffee_break(solver, Snapshot(exp_dir), every_n_iter=5000)
```

We allow the solver to talk about its training progress every 100 iterations, and save the trained model to a snapshot every 5000 iterations. Alternatively, coffee breaks can also be specified by `every_n_epoch`.

Coffee Lounge

Coffee lounge is the place for the solver to have coffee breaks. It provides a storage for a log of the coffee breaks. For example, when the solver talks about its training progress, the objective function value at each coffee break will be recorded. That data can be retrieved for inspection or plotting later.

The default coffee lounge keeps the storage in memory only. If you want to additionally save the recordings to disk, you can set up the coffee lounge in the following way:

```
setup_coffee_lounge(solver, save_into="$exp_dir/statistics.jld",
    every_n_iter=1000)
```

This means the recordings will be saved to the specified file every 1000 iterations. There is one extra keyword parameter for setup coffee lounge: `file_exists`, which should specify a symbol from the following options

`:merge` The default. Try to merge with the existing log file. This is useful if, for example, you are resuming from an interrupted training process.

`:overwrite` Erase the existing log file if any.

`:panic` Exit with error if found the log file already exists.

The logs are stored as simple Julia dictionaries. See `plot_statistics.jl` in the `tools` directory for an example of how to retrieve and visualize the saved information.

Built-in Coffee Breaks

class TrainingSummary

This is a coffee break in which the solver talks about the training summary. The training objective function value at the current iteration is reported by default. You can also call the function with the following named parameters in order to customize the output:

statistic_names

A vector of statistic names to print when summarizing the state, e.g. `[:iter, :obj_val, :learning_rate]`. The available statistics will depend on the solver method in use.

Here are a few examples of usage:

```
#same as original functionality, shows iteration and obj_val by defualt
TrainingSummary()

#will only show objective function value
TrainingSummary(:iter)

#shows iteration, obj_val, learning_rate, and momentum
TrainingSummary(:iter, :obj_val, :learning_rate, :momentum)
```

Note that the training summary at iteration 0 shows the results before training starts. Also, any values that are shown with this method will also be added to the lounge using the *update_statistics()* function.

class Snapshot
Automatically save solver and model snapshots to a given snapshot directory. The snapshot saved at iteration 0 corresponds to the init model (randomly initialized via *initializers* or loaded from existing model file).

class ValidationPerformance
Run an epoch over a validation set and report the performance (e.g. multiclass classification accuracy). You will need to construct a validation network that shares parameters with the training network and provides access to the validation dataset. See *the MNIST tutorial* for a concrete example.

2.9 Mocha Backends

A backend in Mocha is a component that carries out the 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.

There is a `DefaultBackend` defined which is a typealias for one of the following backends, depending on availability. By default, `GPUBackend` is preferred if CUDA is available, falling back to the `CPUBackend` otherwise.

2.9.1 Pure Julia CPU Backend

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

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

If you have many cores in your computer, you can play with the number of threads used by 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.

2.9.2 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 can be enabled by setting the following environment variable. In 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 need to 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 can be controlled by the `OMP_NUM_THREADS` environment variable. Note that this variable is not specific to Mocha. If you have other programs that use OpenMP, setting this environment variable in a shell will also affect the programs started subsequently. If you want to restrict the effect to Mocha, simply set the variable in the Julia code:

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

Note that setting it to 1 disables 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 the automatic building script does not work on Windows. However, the native code themselve does not use any OS specific features. If you have a compiler installed on Windows, you can try to compile the native extension manually. However, I have **not** tested the native extension on Windows personally.

Compile Native Extension Manually

The native code is 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.

2.9.3 CUDA Backend

GPUs have 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 implementes a CUDA backend by combining cuDNN, cuBLAS and plain CUDA kernels.

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

Note: Mocha is tested on CUDA 8.0 and cuDNN 5.1 on Linux. Since cuDNN typically do not keep backward compatibility in the APIs, it is not guaranteed to run on different versions.

Before using the CUDA backend, the 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 command for compiling 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

backend = GPUBackend()
init(backend)

# ...

shutdown(backend)
```

Note that instead of instantiating a `CPUBackend`, you now construct a `GPUBackend`. The environment variable needs to be set **before** loading Mocha. It is designed to use conditional loading so that the pure CPU backend can still run on machines which don't have a GPU device or don't have the CUDA library installed. If you have multiple GPU devices on one node, the environment variable `MOCHA_CUDA_DEVICE` can be used to specify the device ID to use. The default device ID is `0`.

Recompiling Kernels

When you upgrade Mocha to a higher version, the source code for some CUDA kernel implementations might have changed. Mocha will compile the timestamps for the compiled kernel and the source files. An error will be raised if the compiled kernel file is found to be older than the kernel source files. Simply following the procedures above to compile the kernel again will solve this problem.

2.10 Tools

2.10.1 Importing Trained Model from Caffe

Overview

Mocha provides a tool to help with importing Caffe's trained models. Importing Caffe's models 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 correspondences.
2. **Importing the learned network parameters**: this can 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 the 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 contain some extra information, but what we need are 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 the layer parameters to a HDF5 file. In the `tools` directory of Mocha's source root, you can find this in `dump_network_hdf5.cpp`.

Put that file 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 follows:

```
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 the 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 can not be found in the HDF5 archive, it will fail with an error. But you can also change this behavior by passing `false` as the third argument, indicating not to 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 other deep learning tools into Mocha, as long as you can export them to HDF5 files. The HDF5 file that Mocha tries to 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`.

  The 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 decided 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.

Exporting Caffe's Mean File

Sometimes Caffe's model includes a *mean file*, which is the mean data point computed over all the training data. This information might be needed in *data preprocessing*. Of course we could compute the mean from the training data manually. But if the training data is too large or is not easily obtainable, it might be easier to load Caffe's pre-computed mean file instead.

In the `tools` directory of Mocha's source root, you can find `dump_mean_file.cpp`. Similar to exporting Caffe's model file, you can copy this file to Caffe's `tools` directory and compile Caffe. After that, you can export Caffe's mean file:

```
build/tools/dump_mean_file \
    data/ilsvrc12/imagenet_mean.binaryproto \
    ilsvr12_mean.hdf5
```

The exported HDF5 file can then be loaded in Mocha using `DataTransformers.SubMean`.

2.10.2 Image Classifier

A simple image classifier interface is provided in `tools/image-classifier.jl`. It wraps a network and provides an easy-to-use interface that takes an arbitrary number of images and returns the classification results as both class probabilities and symbolic class names. Please see *Image Classification with Pre-trained Model* for an example on how to use this interface.

CHAPTER 3

Developer's Guide

3.1 Blob

A blob is the fundamental data representation in Mocha. It is used for both data (e.g. mini-batch of data samples) and parameters (e.g. filters of a convolution layer). Conceptually, a blob is an N-dimensional tensor.

For example, in vision, a data blob is usually a 4D-tensor. Following the vision (and Caffe) convention, the four dimensions are called *width*, *height*, *channels* and *num*. The fastest changing dimension is *width* and slowest changing dimension is *num*.

Note: The memory layout of a blob in Mocha is compatible with Caffe's blob. So a blob (e.g. layer parameters) in Mocha can be saved to HDF5 and loaded from Caffe without doing any dimension permutation, and vise versa. However, since Julia uses the column-major convention for tensor and matrix data, and Caffe uses the row-major convention, in Mocha API, the order of the four dimensions is width, height, channels, and num, while in Caffe API, it is num, channels, height, width.

Each backend has its own blob implementation, as a subtype of `Blob`. For example, a blob in the CPU backend is a shallow wrapper of a Julia `Array` object, while a blob in the GPU backend references to a piece of GPU memory.

3.1.1 Constructors and Destructors

A backend-dependent blob can be created with the following function:

`make_blob`(*backend*, *data_type*, *dims*)
: `dims` is an `NTuple`, specifying the dimensions of the blob to be created. Currently `data_type` should be either `Float32` or `Float64`.

Several helper functions are also provided:

`make_blob`(*backend*, *data_type*, *dims...*)
: Spell out the dimensions explicitly.

make_blob(*backend*, *array*)
`array` is a Julia `AbstractArray`. This creates a blob with the same data type and shape as `array` and initializes the blob contents with `array`.

make_zero_blob(*backend*, *data_type*, *dims*)
Create a blob and initialize it with zeros.

reshape_blob(*backend*, *blob*, *new_dims*)
Create a reference to an existing blob with a possiblely different shape. The behavior is the same as Julia's `reshape` function on an array: the new blob shares data with the existing one.

destroy(*blob*)
Release the resources of a blob.

Note: The resources need to be released explicitly. A Julia blob object being GC-ed does not release the underlying resource automatically.

3.1.2 Accessing Properties of a Blob

The blob implements a simple API similar to a Julia array:

eltype(*blob*)
Get the element type of the blob.

ndims(*blob*)
Get the tensor dimension of the blob. The same as `length(size(blob))`.

size(*blob*)
Get the shape of the blob. The return value is an `NTuple`.

size(*blob*, *dim*)
Get the size along a particular dimension. `dim` can be negative. For example, `size(blob, -1)` is the same as `size(blob)[end]`. For convenience, if `dim` exceeds `ndims(blob)`, the function returns `1` instead of raising an error.

length(*blob*)
Get the total number of elements in a blob.

get_width(*blob*)
The same as `size(blob, 1)`.

get_height(*blob*)
The same as `size(blob, 2)`.

get_num(*blob*)
The same as `size(blob, -1)`.

get_fea_size(*blob*)
The the *feature size* in a blob, which is the same as `prod(size(blob)[1:end-1])`.

The wrapper `get_chann` was removed when Mocha upgraded from 4D-tensors to general ND-tensors, because the channel dimension is usually ambiguous for general ND-tensors.

3.1.3 Accessing Data of a Blob

Because accessing GPU memory is costly, a blob does not have an interface to do element-wise accessing. The data can be either manipulated in a backend-dependent manner, relying on the underlying implementation details, or in a

backend-independent way by copying the contents from and to a Julia array.

copy! (*dst*, *src*)
Copy the contents of `src` to `dst`. `src` and `dst` can be either a blob or a Julia array.

The following utilities can be used to initialize the contents of a blob

fill! (*blob*, *value*)
Fill every element of `blob` with `value`.

erase! (*blob*)
Fill `blob` with zeros. Depending on the implementation, `erase!(blob)` might be more efficient than `fill!(blob, 0)`.

3.2 Layer

A layer in Mocha is an isolated computation component that (optionally) takes some input blobs and (optionally) produces some output blobs. See *Networks* for an overview of the abstraction of *layer* and *network* in Mocha. Implementing a layer in Mocha means

1. Characterizing the layer (e.g. does this layer define a loss function?) so that the network topology engine knows how to properly glue the layers together to build a network.
2. Implementing the computation of the layer, either in a backend-independent way, or separately for each backend.

3.2.1 Defining a Layer

A layer, like many other computational components in Mocha, consists of two parts:

- A layer configuration, a subtype of `Layer`.
- A layer state, a subtype of `LayerState`.

`Layer` defines how a layer should be constructed and it should behave, while `LayerState` is the realization of a layer which actually holds the data blobs.

Mocha has a helper macro `@defstruct` to define a `Layer` subtype. For example

```
@defstruct PoolingLayer Layer (
  name :: AbstractString = "pooling",
  (bottoms :: Vector{Symbol} = Symbol[], length(bottoms) > 0),
  (tops :: Vector{Symbol} = Symbol[], length(tops) == length(bottoms)),
  (kernel :: NTuple{2, Int} = (1,1), all([kernel...] .> 0)),
  (stride :: NTuple{2, Int} = (1,1), all([stride...] .> 0)),
  (pad :: NTuple{2, Int} = (0,0), all([pad...] .>= 0)),
  pooling :: PoolingFunction = Pooling.Max(),
  neuron :: ActivationFunction = Neurons.Identity(),
)
```

`@defstruct` can be used to define a general immutable struct. The first parameter is the struct name, the second parameter is the super-type and then a list of struct fields follows. Each field requires a name, a type and a default value. Optionally, an expression can be added to verify the user-supplied value meets the requirements.

This macro will automatically define a constructor with keyword arguments for each field. This makes the interface easier to use for the end-user.

Each layer needs to have a field `name`. When the layer produce output blobs, it has to have a property `tops`, allowing the user to specify a list of names for the output blobs the layer is producing. If the layer takes any number of blobs

as input, it should also have a property `bottoms` for the user to specify the names for the input blobs. Mocha will use the information specified in `tops` and `bottoms` to wire the blobs in a proper data path for network forward and backward iterations.

A subtype of `LayerState` should be defined for each layer, correspondingly. For example

```
type PoolingLayerState <: LayerState
  layer      :: PoolingLayer
  blobs      :: Vector{Blob}
  blobs_diff :: Vector{Blob}

  etc        :: Any
end
```

A layer state should have a field `layer` referencing to the corresponding `Layer` object. If the layer produce output blobs, the state should have a field called `blobs`, and the layer will write output into `blobs` during each *forward* iteration. If the layer needs back-propagation from the upper layers, the state should also have a field called `blobs_diff`. Mocha will pass the blobs in `blobs_diff` to the function computing *backward* iteration in the corresponding upper layer. The back-propagated gradients will be written into `blobs_diff` by the upper layer, and the layer can make use of this when computing the *backward* iteration.

Other fields and/or behaviors are required depending on the layer type (see below).

3.2.2 Characterizing a Layer

A layer is characterized by applying the macro `@characterize_layer` to the defined subtype of `Layer`. The default characterizations are given by

```
@characterize_layer(Layer,
  is_source  => false, # data layer, takes no bottom blobs
  is_sink    => false, # top layer, produces no top blobs (loss, accuracy, etc.)
  has_param  => false, # contains trainable parameters
  has_neuron => false, # has a neuron
  can_do_bp  => false, # can do back-propagation
  is_inplace => false, # does inplace computation, does not have own top blobs
  has_loss   => false, # produces a loss
  has_stats  => false, # produces statistics
)
```

Characterizing a layer can be omitted if all the behaviors are consists with the default specifications. The characterizations should be self-explanatory by the name and comments above. Some characterizations come with extra requirements:

is_source The layer will be used as a source layer of a network. Thus it should take no input blob and the `Layer` object should have no `bottoms` property.

is_sink The layer will be used as a sink layer of a network. Thus it should produce no output blob, and the `Layer` object should have no `tops` property.

has_param The layer has trainable parameters. The `LayerState` object should have a `parameters` field, containing a list of `Parameter` objects.

has_neuron The `Layer` object should have a property called `neuron` of type `ActivationFunction`.

can_do_bp Should be true if the layer has the ability to do back propagation.

is_inplace An inplace `Layer` object should have no `tops` property because the output blobs are the same as the input blobs.

has_loss The `LayerState` object should have a `loss` field.

has_stats The layer computes statistics (e.g. accuracy). The statistics should be accumulated across multiple mini-batches, until the user explicit reset the statistics. The following functions should be implemented for the layer

dump_statistics(*storage*, *layer_state*, *show*)
`storage` is a data storage (typically a `CoffeeLounge` object) that is used to dump statistics into, via the function `update_statistics(storage, key, value)`.

`show` is a boolean value, when true, indicating that a summary of the statistics should also be printed to stdout.

reset_statistics(*layer_state*)
Reset the statistics.

3.2.3 Layer Computation API

The life cycle of a layer is

1. The user defines a `Layer`
2. The user uses `Layers` to construct a `Net`. The `Net` will call `setup_layer` on each `Layer` to construct the corresponding `LayerState`.
3. During training, the solver use a loop to call the `forward` and `backward` functions of the `Net`. The `Net` will then call `forward` and `backward` of each layer in a proper order.
4. The user destroys the `Net`, which will call the `shutdown` function of each layer.

setup_layer(*backend*, *layer*, *inputs*, *diffs*)
Construct a corresponding `LayerState` object given a `Layer` object. `inputs` is a list of blobs, corresponding to the blobs specified by the `bottoms` property of the `Layer` object. If the `Layer` does not have a `bottoms` property, then it will be an empty list.

`diffs` is a list of blobs. Each blob in `diffs` corresponds to a blob in `inputs`. When computing back propagation, the back-propagated gradients for each input blob should be written into the corresponding one in `diffs`. Blobs in `inputs` and `diffs` are taken from `blobs` and `blobs_diff` of the `LayerState` objects of lower layers.

`diffs` is guaranteed to be a list of blobs of the same length as `inputs`. However, when some input blobs do not need back-propagated gradients, the corresponding blob in `diffs` will be a `NullBlob`.

This function should set up its own `blobs` and `blobs_diffs` (if any), matching the shape of its input blobs.

forward(*backend*, *layer_state*, *inputs*)
Do forward computing. It is guaranteed that the blobs in `inputs` are already computed by the lower layers. The output blobs (if any) should be written into the blobs in the `blobs` field of the layer state.

backward(*backend*, *layer_state*, *inputs*, *diffs*)
Do backward computing. It is guaranteed that the back-propagated gradients with respect to all the output blobs for this layer are already computed and written into the blobs in the `blobs_diff` field of the layer state. This function should compute the gradients with respect to its parameters (if any). It is also responsible to compute the back-propagated gradients and write them into the blobs in `diffs`. If a blob in `diffs` is a `NullBlob`, computation for the back-propagated gradients for that blob can be omitted.

The contents in the blobs in `inputs` are the same as in the last call of `forward`, and can be used if necessary.

If a layer does not do backward propagation (e.g. a data layer), an empty `backward` function still has to be defined explicitly.

shutdown(*backend*, *layer_state*)
Release all the resources allocated in `setup_layer`.

3.2.4 Layer Parameters

If a layer has train-able parameters, it should define a `parameters` field in the `LayerState` object, containing a list of `Parameter` objects. It should also define the `has_param` characterization. The only computation the layer needs to do, is to compute the gradients with respect to each parameter and write them into the `gradient` field of each `Parameter` object.

Mocha will handle the updating of parameters during training automatically. Other parameter-related issues like initialization, regularization and norm constraints will also be handled automatically.

3.2.5 Layer Activation Function

When it makes sense for a layer to have an activation function, it can add a `neuron` property to the `Layer` object and define the `has_neuron` characterization. Everything else will be handled automatically.

3.3 Indices and tables

- genindex
- modindex
- search

Bibliography

[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.

[Saxe2013] Andrew M. Saxe, James L. McClelland, Surya Ganguli, *Exact solutions to the nonlinear dynamics of learning in deep linear neural networks*, http://arxiv.org/abs/1312.6120 with a presentation https://www.youtube.com/watch?v=Ap7atx-Ki3Q

Index

E

F

G

H

I

K

L

M

N

X