Caffe tutorial

V1.0

Based on official web tutorial
(http://caffe.berkeleyvision.org/tutorial/)
and rewrite by Cari depend on new version of Caffe's definition on Github.(https://github.com/BVLC/caffe)

2015.4@SDU

Catalogue

1.Blobs, Layers, and Nets: anatomy of a Caffe model

Deep networks are compositional models that are naturally represented as a collection of inter-connected layers that work on chunks of data. Caffe defines a net layer-by-layer in its own model schema. The network defines the entire model bottom-to-top from input data to loss. As data and derivatives flow through the network in the forward and backward passes Caffe stores, communicates, and manipulates the information as *blobs*: the blob is the standard array and unified memory interface for the framework. The layer comes next as the foundation of both model and computation. The net follows as the collection and connection of layers. The details of blob describe how information is stored and communicated in and across layers and nets.
Solving is configured separately to decouple modeling and optimization.
We will go over the details of these components in more detail.

1.1Blob storage and communication

A Blob is a wrapper over the actual data being processed and passed along by Caffe, and also under the hood provides synchronization capability between the CPU and the GPU. Mathematically, a blob is a 4-dimensional array that stores things in the order of (Num, Channels, Height and Width), from major to minor, and stored in a C-contiguous fashion. The main reason for putting Num (the name is due to legacy reasons, and is equivalent to the notation of “batch” as in minibatch SGD).

Caffe stores and communicates data in 4-dimensional arrays called blobs. Blobs provide a unified memory interface, holding data e.g. batches of images, model parameters, and derivatives for optimization.

Blobs conceal the computational and mental overhead of mixed CPU/GPU operation by synchronizing from the CPU host to the GPU device as needed. Memory on the host and device is allocated on demand (lazily) for efficient memory usage.

The conventional blob dimensions for data are number N x channel K x height H x width W. Blob memory is row-major in layout so the last / rightmost dimension changes fastest. For example, the value at index (n, k, h, w) is physically located at index ((n * K + k) * H + h) * W + w.

- Number / N is the batch size of the data. Batch processing achieves better throughput for communication and device processing. For an ImageNet training batch of 256 images B = 256.
- Channel / K is the feature dimension e.g. for RGB images K = 3.

Note that although we have designed blobs with its dimensions corresponding to image applications, they are named purely for notational purpose and it is totally valid for you to do non-image applications. For example, if you simply need fully-connected networks like the conventional multi-layer perceptron, use blobs of dimensions (Num, Channels, 1, 1) and call the InnerProductLayer (which we will cover soon).

Caffe operations are general with respect to the channel dimension / K. Grayscale and hyperspectral imagery are fine. Caffe can likewise model and process arbitrary vectors in blobs with singleton. That is, the shape of blob holding 1000 vectors of 16 feature dimensions is 1000 x 16 x 1 x 1.

Parameter blob dimensions vary according to the type and configuration of the layer. For a convolution layer with 96 filters of 11 x 11 spatial dimension and 3 inputs the blob is 96 x 3 x 11 x 11. For an inner product / fully-connected layer with 1000 output channels and 1024 input channels the parameter blob is 1 x 1 x 1000 x 1024.

For custom data it may be necessary to hack your own input preparation tool or data layer. However once your data is in your job is done. The modularity of layers accomplishes the rest of the work for you.

1.1.1Implementation Details

As we are often interested in the values as well as the gradients of the blob, a Blob stores two chunks of memories, *data* and *diff*. The former is the normal data that we pass along, and the latter is the gradient computed by the network.
Further, as the actual values could be stored either on the CPU and on the GPU, there are two different ways to access them: the const way, which does not change the values, and the mutable way, which changes the values:

```
const Dtype* cpu_data() const;
Dtype* mutable_cpu_data();
```

(similarly for gpu and diff).

The reason for such design is that, a Blob uses a SyncedMem class to synchronize values between the CPU and GPU in order to hide the synchronization details and to minimize data transfer. A rule of thumb is, always use the const call if you do not want to change the values, and never store the pointers in your own object. Every time you work on a blob, call the functions to get the pointers, as the SyncedMem will need this to figure out when to copy data.

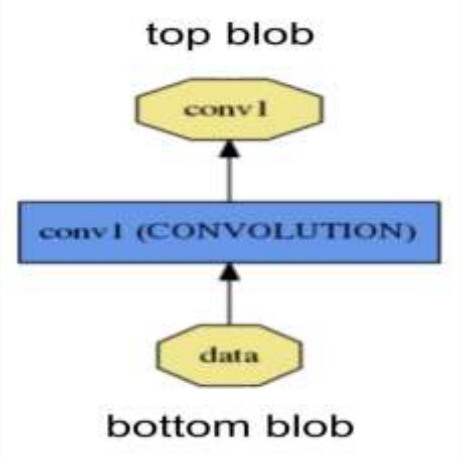
In practice when GPUs are present, one loads data from the disk to a blob in CPU code, calls a device kernel to do GPU computation, and ferries the blob off to the next layer, ignoring low-level details while maintaining a high level of performance. As long as all layers have GPU implementations, all the intermediate data and gradients will remain in the GPU.

If you want to check out when a Blob will copy data, here is an illustrative example:

```
// Assuming that data are on the CPU initially, and we have a blob.
const Dtype* foo;
Dtype* bar;
foo = blob.gpu_data(); // data copied cpu->gpu.
foo = blob.cpu_data(); // no data copied since both have up-to-date contents.
bar = blob.mutable_gpu_data(); // no data copied.
// ... some operations ...
bar = blob.mutable_gpu_data(); // no data copied when we are still on GPU.
foo = blob.cpu_data(); // data copied gpu->cpu, since the gpu side has modified
the data
foo = blob.gpu_data(); // no data copied since both have up-to-date contents
bar = blob.mutable_cpu_data(); // still no data copied.
bar = blob.mutable_gpu_data(); // data copied cpu->gpu.
bar = blob.mutable_cpu_data(); // data copied gpu->cpu.
```

1.2Layer computation and connections

The layer is the essence of a model and the fundamental unit of computation. Layers convolve filters, pool, take inner products, apply nonlinearities like rectified-linear and sigmoid and other elementwise transformations, normalize, load data, and compute losses like softmax and hinge.See the layer catalogue for all operations. Most of the types needed for state-of-the-art deep learning tasks are there.



A layer takes input through *bottom* connections and makes output through *top* connections.

Each layer type defines three critical computations: *setup*, *forward*, and *backward*.

- Setup: initialize the layer and its connections once at model initialization.
- Forward: given input from bottom compute the output and send to the top.
- Backward: given the gradient w.r.t. the top output compute the gradient w.r.t. to the input and send to the bottom. A layer with parameters computes the gradient w.r.t. to its parameters and stores it internally.

More specifically, there will be two Forward and Backward functions implemented, one for CPU and one for GPU. If you do not implement a GPU version, the layer will fall back to the CPU functions as a backup option. This may come handy if you would like to do quick experiments, although it may come with additional data transfer cost (its inputs will be copied from GPU to CPU, and its outputs will be copied back from CPU to GPU).

Layers have two key responsibilities for the operation of the network as a whole: a *forward pass*that takes the inputs and produces the outputs, and a *backward pass* that takes the gradient with respect to the output, and computes the gradients with respect to the parameters and to the inputs, which are in turn back-propagated to earlier layers. These passes are simply the composition of each layer's forward and backward.

Developing custom layers requires minimal effort by the compositionality of the network and modularity of the code. Define the setup, forward, and backward for the layer and it is ready for inclusion in a net.

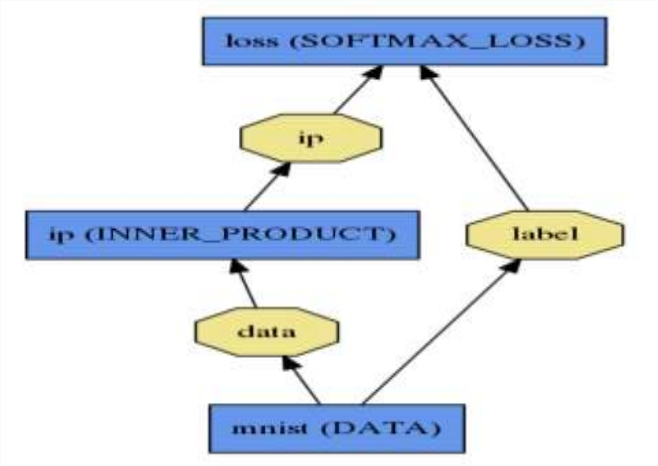
1.3Net definition and operation

The net jointly defines a function and its gradient by composition and auto-differentiation. The composition of every layer's output computes the function to do a given task, and the

composition of every layer's backward computes the gradient from the loss to learn the task. Caffe models are end-to-end machine learning engines.

The net is a set of layers connected in a computation graph – a directed acyclic graph (DAG) to be exact. Caffe does all the bookkeeping for any DAG of layers to ensure correctness of the forward and backward passes. A typical net begins with a data layer that loads from disk and ends with a loss layer that computes the objective for a task such as classification or reconstruction.

The net is defined as a set of layers and their connections in a plaintext modeling language. A simple logistic regression classifier



is defined by

```
name: "LogReg"
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  data_param {
    source: "input_leveldb"
    batch_size: 64
    backend: LMDB
  }
}
layer {
  name: "ip"
  type: "InnerProduct"
```

```
  bottom: "data"
  top: "ip"
  inner_product_param {
    num_output: 2
  }
}
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "ip"
  bottom: "label"
  top: "loss"
}
```

Model initialization is handled by `Net::Init()`. The initialization mainly does two things: scaffolding the overall DAG by creating the blobs and layers (for C++ geeks: the network will retain ownership of the blobs and layers during its lifetime), and calls the layers' `SetUp()` function. It also does a set of other bookkeeping things, such as validating the correctness of the overall network architecture. Also, during initialization the Net explains its initialization by logging to INFO as it goes:

```
I0902 22:52:17.931977 2079114000 net.cpp:39] Initializing net from parameters:
name: "LogReg"
[...model prototxt printout...]
# construct the network layer-by-layer
I0902 22:52:17.932152 2079114000 net.cpp:67] Creating Layer mnist
I0902 22:52:17.932165 2079114000 net.cpp:356] mnist -> data
I0902 22:52:17.932188 2079114000 net.cpp:356] mnist -> label
I0902 22:52:17.932200 2079114000 net.cpp:96] Setting up mnist
I0902 22:52:17.935807 2079114000 data_layer.cpp:135] Opening leveldb
input_leveldb
I0902 22:52:17.937155 2079114000 data_layer.cpp:195] output data size:
64,1,28,28
```

Note that the construction of the network is device agnostic - recall our earlier explanation that blobs and layers hide implementation details from the model definition. After construction, the network is run on either CPU or GPU by setting a single switch defined in `Caffe::mode()` and set by `Caffe::set_mode()`. Layers come with corresponding CPU and GPU routines that produce identical results (up to numerical errors, and with tests to guard it). The CPU / GPU switch is seamless and independent of the model definition. For research and deployment alike it is best to divide model and implementation.

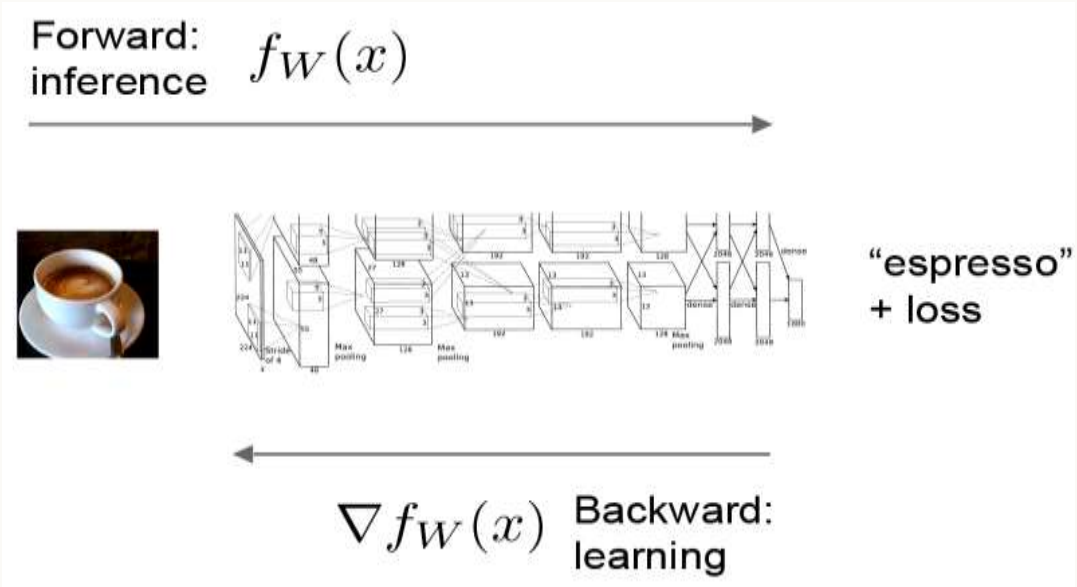
1.3.1Model format

The models are defined in plaintext protocol buffer schema (prototxt) while the learned models are serialized as binary protocol buffer (binaryproto) .caffemodel files.

The model format is defined by the protobuf schema in caffe.proto. The source file is mostly self-explanatory so one is encouraged to check it out.
Caffe speaks Google Protocol Buffer for the following strengths: minimal-size binary strings when serialized, efficient serialization, a human-readable text format compatible with the binary version, and efficient interface implementations in multiple languages, most notably C++ and Python. This all contributes to the flexibility and extensibility of modeling in Caffe.

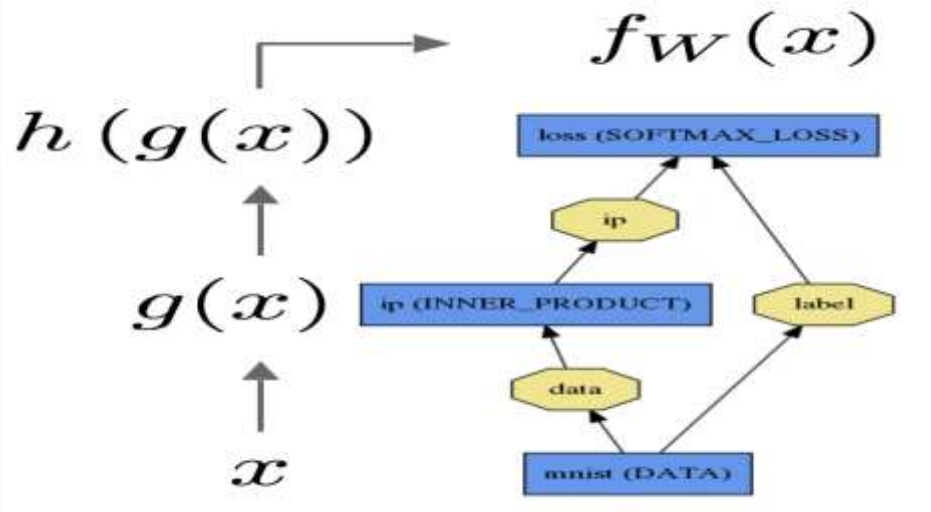
2.Forward and Backward

The forward and backward passes are the essential computations of a Net.



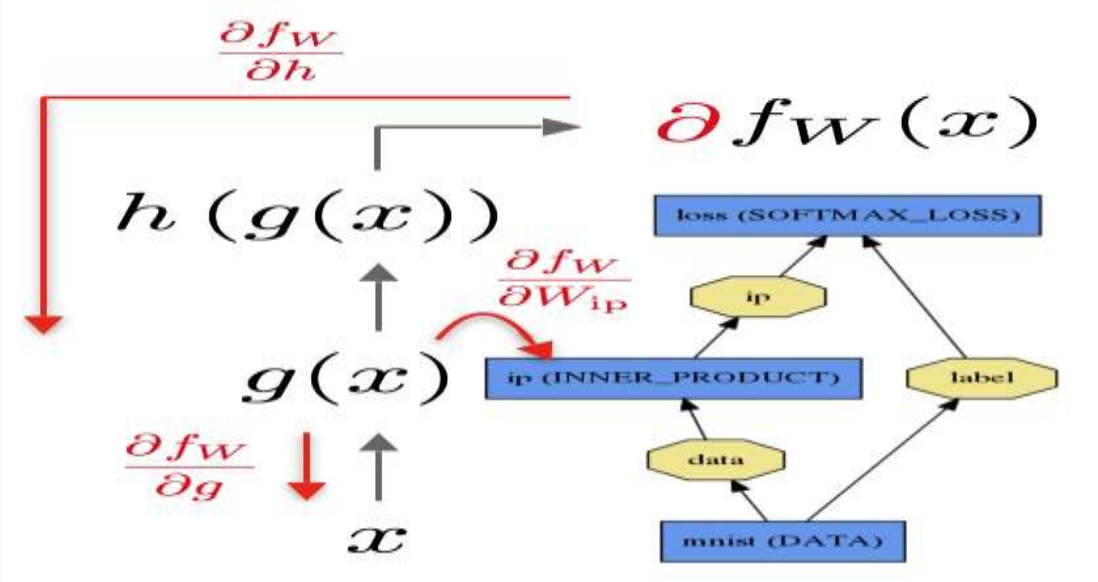
Let's consider a simple logistic regression classifier.

The **forward** pass computes the output given the input for inference. In forward Caffe composes the computation of each layer to compute the "function" represented by the model. This pass goes from bottom to top.



The data x is passed through an inner product layer for $g(x)$ then through a softmax for $h(g(x))$ and softmax loss to give $fw(x)$.

The **backward** pass computes the gradient given the loss for learning. In backward Caffe reverse-composes the gradient of each layer to compute the gradient of the whole model by automatic differentiation. This is back-propagation. This pass goes from top to bottom.



The backward pass begins with the loss and computes the gradient with respect to the output $\partial fw \partial h$. The gradient with respect to the rest of the model is computed layer-by-layer through the chain rule. Layers with parameters, like the `INNER_PRODUCT` layer, compute the gradient with respect to their parameters $\partial fw \partial W_{ip}$ during the backward step.
These computations follow immediately from defining the model: Caffe plans and carries out the forward and backward passes for you.

- The `Net::Forward()` and `Net::Backward()` methods carry out the respective passes while`Layer::Forward()` and `Layer::Backward()` compute each step.
- Every layer type has `forward_{cpu,gpu}()` and `backward_{cpu,gpu}` methods to compute its steps according to the mode of computation. A layer may only implement CPU or GPU mode due to constraints or convenience.

The Solver optimizes a model by first calling forward to yield the output and loss, then calling backward to generate the gradient of the model, and then incorporating the gradient into a weight update that attempts to minimize the loss. Division of labor between the Solver, Net, and Layer keep Caffe modular and open to development.

For the details of the forward and backward steps of Caffe's layer types, refer to the layer catalogue.

3.Loss

In Caffe, as in most of machine learning, learning is driven by a **loss** function (also known as an**error**, **cost**, or **objective** function). A loss function specifies the goal of learning by mapping parameter settings (i.e., the current network weights) to a scalar value specifying the "badness" of these parameter settings. Hence, the goal of learning is to find a setting of the weights that*minimizes* the loss function.

The loss in Caffe is computed by the Forward pass of the network. Each layer takes a set of input (`bottom`) blobs and produces a set of output (`top`) blobs. Some of these layers' outputs may be used in the loss function. A typical choice of loss function for one-versus-all classification tasks is the `SOFTMAX_LOSS` function, used in a network definition as follows, for example:

```
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "pred"
  bottom: "label"
  top: "loss"
}
```

In a `SOFTMAX_LOSS` function, the `top` blob is a scalar (dimensions $1 \times 1 \times 1 \times 1$) which averages the loss (computed from predicted labels `pred` and actuals labels `label`) over the entire mini-batch.

3.1.Loss weights

For nets with multiple layers producing a loss (e.g., a network that both classifies the input using a `"SoftmaxWithLoss"` layer and reconstructs it using a `euclideanLoss` layer), *loss weights* can be used to specify their relative importance.
By convention, Caffe layer types with the suffix –'`Loss`' contribute to the loss function, but other layers are assumed to be purely used for intermediate computations. However, **any layer can be used as a loss by adding a field `loss_weight: <float>`** to a layer definition for each `top` blob produced by the layer. Layers with the suffix –'`Loss`' have an implicit `loss_weight: 1` for the first`top` blob (and `loss_weight: 0` for any additional `top`s); other layers have an implicit `loss_weight: 0` for all `top`s. So, the above `SOFTMAX_LOSS` layer could be equivalently written as:

```
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "pred"
  bottom: "label"
  top: "loss"
  loss_weight: 1
}
```

However, *any* layer able to backpropagate may be given a non-zero `loss_weight`, allowing one to, for example, regularize the activations produced by some intermediate layer(s) of the network if desired. For non-singleton outputs with an associated non-zero loss, the loss is computed simply by summing over all entries of the blob.
The final loss in Caffe, then, is computed by summing the total weighted loss over the network, as in the following pseudo-code:

```
loss := 0
for layer in layers:
  for top, loss_weight in layer.tops, layer.loss_weights:
    loss += loss_weight * sum(top)
```

4.Solver

The solver orchestrates model optimization by coordinating the network's forward inference and backward gradients to form parameter updates that attempt to improve the loss. The responsibilities of learning are divided between the Solver for overseeing the optimization and generating parameter updates and the Net for yielding loss and gradients.

The Caffe solvers are Stochastic Gradient Descent (SGD), Adaptive Gradient (ADAGRAD), and Nesterov's Accelerated Gradient (NESTEROV).

The solver:

1. scaffolds the optimization bookkeeping and creates the training network for learning and test network(s) for evaluation.
2. iteratively optimizes by calling forward / backward and updating parameters
3. (periodically) evaluates the test networks
4. snapshots the model and solver state throughout the optimization

where each iteration:

1. calls network forward to compute the output and loss
2. calls network backward to compute the gradients
3. incorporates the gradients into parameter updates according to the solver method
4. updates the solver state according to learning rate, history, and method

to take the weights all the way from initialization to learned model.

Like Caffe models, Caffe solvers run in CPU / GPU modes.

4.1Methods

The solver methods address the general optimization problem of loss minimization. For dataset D, the optimization objective is the average loss over all $|D|$ data instances throughout the dataset

$$L(W)=1|D|\sum_{i|D|}f_W(X_{(i)})+\lambda r(W)$$

where $f_W(X_{(i)})$ is the loss on data instance $X_{(i)}$ and $r(W)$ is a regularization term with weight λ. $|D|$ can be very large, so in practice, in each solver iteration we use a stochastic approximation of this objective, drawing a mini-batch of $N << |D|$ instances:

$$L(W) \approx \frac{1}{N} \sum_{i}^{N} f_W(X_{(i)}) + \lambda r(W)$$

The model computes f_W in the forward pass and the gradient ∇f_W in the backward pass. The parameter update ΔW is formed by the solver from the error gradient ∇f_W, the regularization gradient $\nabla r(W)$, and other particulars to each method.

4.1.1SGD

Stochastic gradient descent (`solver_type: SGD`) updates the weights W by a linear combination of the negative gradient $\nabla L(W)$ and the previous weight update V_t.

The **learning rate** α is the weight of the negative gradient. The **momentum** μ is the weight of the previous update.

Formally, we have the following formulas to compute the update value V_{t+1} and the updated weights W_{t+1} at iteration $t+1$, given the previous weight update V_t and current weights W_t:

$$V_{t+1} = \mu V_t - \alpha \nabla L(W_t)$$

$$W_{t+1} = W_t + V_{t+1}$$

The learning “hyperparameters” (α and μ) might require a bit of tuning for best results. If you’re not sure where to start, take a look at the “Rules of thumb” below, and for further information you might refer to Leon Bottou’s Stochastic Gradient Descent Tricks [1].

[1] L. Bottou. Stochastic Gradient Descent Tricks. *Neural Networks: Tricks of the Trade*: Springer, 2012.

4.1.1.1.Rules of thumb for setting the learning rate α and momentum μ

A good strategy for deep learning with SGD is to initialize the learning rate α to a value around $\alpha \approx 0.01 = 10^{-2}$, and dropping it by a constant factor (e.g., 10) throughout training when the loss begins to reach an apparent “plateau”, repeating this several times. Generally, you probably want to use a momentum $\mu = 0.9$ or similar value. By smoothing the weight updates across iterations, momentum tends to make deep learning with SGD both stabler and faster.
This was the strategy used by Krizhevsky et al. [1] in their famously winning CNN entry to the ILSVRC-2012 competition, and Caffe makes this strategy easy to implement in a `SolverParameter`, as in our reproduction of [1] at `./examples/imagenet/alexnet_solver.prototxt`.
To use a learning rate policy like this, you can put the following lines somewhere in your solver prototxt file:

```
base_lr: 0.01     # begin training at a learning rate of 0.01 = 1e-2

lr_policy: "step" # learning rate policy: drop the learning rate in "steps"
                  # by a factor of gamma every stepsize iterations

gamma: 0.1        # drop the learning rate by a factor of 10
                  # (i.e., multiply it by a factor of gamma = 0.1)

stepsize: 100000  # drop the learning rate every 100K iterations

max_iter: 350000  # train for 350K iterations total

momentum: 0.9
```

Under the above settings, we’ll always use `momentum` $\mu = 0.9$. We’ll begin training at a `base_lr` of $\alpha = 0.01 = 10^{-2}$ for the first 100,000 iterations, then multiply the learning rate by `gamma` (γ) and train at $\alpha' = \alpha\gamma = (0.01)(0.1) = 0.001 = 10^{-3}$ for iterations 100K-200K, then at $\alpha'' = 10^{-4}$ for iterations 200K-300K, and finally train until iteration 350K (since we have`max_iter: 350000`) at $\alpha''' = 10^{-5}$.

Note that the momentum setting μ effectively multiplies the size of your updates by a factor of $\frac{1}{1-\mu}$ after many iterations of training, so if you increase μ, it may be a good idea to **decrease** αaccordingly (and vice versa).

For example, with $\mu=0.9$, we have an effective update size multiplier of $\frac{1}{1-0.9}=10$. If we increased the momentum to $\mu=0.99$, we've increased our update size multiplier to 100, so we should drop α (`base_lr`) by a factor of 10.

Note also that the above settings are merely guidelines, and they're definitely not guaranteed to be optimal (or even work at all!) in every situation. If learning diverges (e.g., you start to see very large or `NaN` or `inf` loss values or outputs), try dropping the `base_lr` (e.g., `base_lr: 0.001`) and re-training, repeating this until you find a `base_lr` value that works.

[1] A. Krizhevsky, I. Sutskever, and G. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems*, 2012.

4.1.2AdaGrad

The **adaptive gradient** (`solver_type: ADAGRAD`) method (Duchi et al. [1]) is a gradient-based optimization method (like SGD) that attempts to "find needles in haystacks in the form of very predictive but rarely seen features," in Duchi et al.'s words. Given the update information from all previous iterations $(\nabla L(W))_{t'}$ for $t' \in \{1,2,...,t\}$, the update formulas proposed by [1] are as follows, specified for each component i of the weights W:

$$(W_{t+1})_i = (W_t)_i - \alpha \frac{(\nabla L(W_t))_i}{\sqrt{\sum_{t'=1}^{t} (\nabla L(W_{t'}))_i^2}}$$

Note that in practice, for weights $W \in R^d$, AdaGrad implementations (including the one in Caffe) use only $O(d)$ extra storage for the historical gradient information (rather than the $O(dt)$ storage that would be necessary to store each historical gradient individually).

[1] J. Duchi, E. Hazan, and Y. Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *The Journal of Machine Learning Research*, 2011.

4.1.3NAG

Nesterov's accelerated gradient (`solver_type: NESTEROV`) was proposed by Nesterov [1] as an "optimal" method of convex optimization, achieving a convergence rate of $O(1/t^2)$ rather than the $O(1/t)$. Though the required assumptions to achieve the $O(1/t^2)$ convergence typically will not hold for deep networks trained with Caffe (e.g., due to non-smoothness and non-convexity), in practice NAG can be a very effective method for optimizing certain types of deep learning architectures, as demonstrated for deep MNIST autoencoders by Sutskever et al. [2].
The weight update formulas look very similar to the SGD updates given above:

$$V_{t+1} = \mu V_t - \alpha \nabla L(W_t + \mu V_t)$$

$$W_{t+1} = W_t + V_{t+1}$$

What distinguishes the method from SGD is the weight setting W on which we compute the error gradient $\nabla L(W)$ – in NAG we take the gradient on weights with added momentum $\nabla L(W_t + \mu V_t)$; in SGD we simply take the gradient $\nabla L(W_t)$ on the current weights themselves.

[1] Y. Nesterov. A Method of Solving a Convex Programming Problem with Convergence Rate $O(1/k\sqrt{})$. *Soviet Mathematics Doklady*, 1983.

[2] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the Importance of Initialization and Momentum in Deep Learning. *Proceedings of the 30th International Conference on Machine Learning*, 2013.

4.2Scaffolding

The solver scaffolding prepares the optimization method and initializes the model to be learned in `Solver::Presolve()`.

```
> caffe train -solver examples/mnist/lenet_solver.prototxt
```

```
I0902 13:35:56.474978 16020 caffe.cpp:90] Starting Optimization
I0902 13:35:56.475190 16020 solver.cpp:32] Initializing solver from parameters:
test_iter: 100
test_interval: 500
base_lr: 0.01
display: 100
max_iter: 10000
lr_policy: "inv"
gamma: 0.0001
power: 0.75
momentum: 0.9
weight_decay: 0.0005
snapshot: 5000
snapshot_prefix: "examples/mnist/lenet"
solver_mode: GPU
net: "examples/mnist/lenet_train_test.prototxt"
```

Net initialization

```
I0902 13:35:56.655681 16020 solver.cpp:72] Creating training net from net file:
examples/mnist/lenet_train_test.prototxt
[...]
I0902 13:35:56.656740 16020 net.cpp:56] Memory required for data: 0
I0902 13:35:56.656791 16020 net.cpp:67] Creating Layer mnist
I0902 13:35:56.656811 16020 net.cpp:356] mnist -> data
I0902 13:35:56.656846 16020 net.cpp:356] mnist -> label
I0902 13:35:56.656874 16020 net.cpp:96] Setting up mnist
I0902 13:35:56.694052 16020 data_layer.cpp:135] Opening lmdb
I0902 13:35:56.729085 16020 net.cpp:220] Memory required for data: 5169924
I0902 13:35:56.729277 16020 solver.cpp:156] Creating test net (#0) specified by
net file: examples/mnist/lenet_train_test.prototxt
```

Completion

```
I0902 13:35:56.806970 16020 solver.cpp:46] Solver scaffolding done.
I0902 13:35:56.806984 16020 solver.cpp:165] Solving LeNet
```

4.3Updating Parameters

The actual weight update is made by the solver then applied to the net parameters in`Solver::ComputeUpdateValue()`. The `ComputeUpdateValue` method incorporates any

weight decay $r(W)$ into the weight gradients (which currently just contain the error gradients) to get the final gradient with respect to each network weight. Then these gradients are scaled by the learning rate α and the update to subtract is stored in each parameter Blob's `diff` field. Finally, the `Blob::Update` method is called on each parameter blob, which performs the final update (subtracting the Blob's `diff` from its `data`).

4.4Snapshotting and Resuming

The solver snapshots the weights and its own state during training in `Solver::Snapshot()` and `Solver::SnapshotSolverState()`. The weight snapshots export the learned model while the solver snapshots allow training to be resumed from a given point. Training is resumed by `Solver::Restore()` and `Solver::RestoreSolverState()`.
Weights are saved without extension while solver states are saved with `.solverstate` extension. Both files will have an `_iter_N` suffix for the snapshot iteration number.
Snapshotting is configured by:

```
# The snapshot interval in iterations.
snapshot: 5000
# File path prefix for snapshotting model weights and solver state.
# Note: this is relative to the invocation of the `caffe` utility, not the
# solver definition file.
snapshot_prefix: "/path/to/model"
# Snapshot the diff along with the weights. This can help debugging training
# but takes more storage.
snapshot_diff: false
# A final snapshot is saved at the end of training unless
# this flag is set to false. The default is true.
snapshot_after_train: true
```

in the solver definition prototxt.

5.Layers

To create a Caffe model you need to define the model architecture in a protocol buffer definition file (prototxt).

Caffe layers and their parameters are defined in the protocol buffer definitions for the project incaffe.proto. The latest definitions are in the dev caffe.proto.
TODO complete list of layers linking to headings

Vision Layers

- Header: `./include/caffe/vision_layers.hpp`

Vision layers usually take *images* as input and produce other *images* as output. A typical "image" in the real-world may have one color channel ($c=1$), as in a grayscale image, or three color channels ($c=3$) as in an RGB (red, green, blue) image. But in this context, the distinguishing characteristic of an image is its spatial structure: usually an image has some non-trivial height $h>1$ and width $w>1$. This 2D geometry naturally lends itself to certain decisions about how to process the input. In particular, most of the vision layers work by applying a particular operation to some region of the input to produce a corresponding region of the output. In contrast, other layers (with few exceptions) ignore the spatial structure of the input, effectively treating it as "one big vector" with dimension chw.

Convolution

- LayerType: `Convolution`
- CPU implementation: `./src/caffe/layers/convolution_layer.cpp`
- CUDA GPU implementation: `./src/caffe/layers/convolution_layer.cu`
- Parameters (`ConvolutionParameter convolution_param`)
  - Optional(**there are only Optional items in new version Caffe-protofile**)
    - `num_output` (`c_o`): the number of **filters**
    - `kernel_size` (or `kernel_h` and `kernel_w`): specifies height and width of each filter
    - `bias_term` [default `true`]: specifies whether to learn and apply a set of additive biases to the filter outputs

- `pad` (or `pad_h` and `pad_w`) [default 0]: specifies the number of pixels to (implicitly) add to each side of the input
- `stride` (or `stride_h` and `stride_w`) [default 1]: specifies the intervals at which to apply the filters to the input
- `group` (g) [default 1]: If g > 1, we restrict the connectivity of each filter to a subset of the input. Specifically, the input and output channels are separated into g groups, and the ith output group channels will be only connected to the ith input group channels.

- Strongly Recommended
- `weight_filler` [default `type: 'constant' value: 0`]

- Input
  - `n * c_i * h_i * w_i`
- Output
  - `n * c_o * h_o * w_o`, where `h_o = (h_i + 2 * pad_h - kernel_h) / stride_h + 1` and `w_o` likewise.
- Sample (as seen in `./examples/imagenet/imagenet_train_val.prototxt`)

```
layer{
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param{
   lr_mult :1  # learning rate multiplier for the filters
   decay_mult:1 # learning rate multiplier for the biases
 }
  param{
  lr_mult :1   # weight decay multiplier for the filters
  decay_mult:0   # weight decay multiplier for the biases

  convolution_param {
    num_output: 96     # learn 96 filters
    kernel_size: 11    # each filter is 11x11
    stride: 4         # step 4 pixels between each filter application
    weight_filler {
      type: "gaussian" # initialize the filters from a Gaussian
      std: 0.01       # distribution with stdev 0.01 (default mean: 0)
    }
    bias_filler {
      type: "constant" # initialize the biases to zero (0)
      value: 0
    }
```

The `CONVOLUTION` layer convolves the input image with a set of learnable filters, each producing one feature map in the output image.

Pooling

- LayerType: `Pooling`
- CPU implementation: `./src/caffe/layers/pooling_layer.cpp`
- CUDA GPU implementation: `./src/caffe/layers/pooling_layer.cu`
- Parameters (`PoolingParameter pooling_param`)
  - Optional
    - `kernel_size` (or `kernel_h` and `kernel_w`): specifies height and width of each filter
    - `pool` [default MAX]: the pooling method. Currently MAX, AVE, or STOCHASTIC
    - `pad` (or `pad_h` and `pad_w`) [default 0]: specifies the number of pixels to (implicitly) add to each side of the input
    - `stride` (or `stride_h` and `stride_w`) [default 1]: specifies the intervals at which to apply the filters to the input
- Input
  - `n * c * h_i * w_i`
- Output
  - `n * c * h_o * w_o`, where h_o and w_o are computed in the same way as convolution.
- Sample (as seen in `./examples/imagenet/imagenet_train_val.prototxt`)

```
layer{
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 3 # pool over a 3x3 region
    stride: 2     # step two pixels (in the bottom blob) between pooling regions
  }
}
```

Local Response Normalization (LRN)

- LayerType: `LRN`
- CPU Implementation: `./src/caffe/layers/lrn_layer.cpp`
- CUDA GPU Implementation: `./src/caffe/layers/lrn_layer.cu`
- Parameters (`LRNParameter lrn_param`)

- Optional
  - `local_size` [default 5]: the number of channels to sum over (for cross channel LRN) or the side length of the square region to sum over (for within channel LRN)
  - `alpha` [default 1]: the scaling parameter (see below)
  - `beta` [default 5]: the exponent (see below)
  - `norm_region` [default `ACROSS_CHANNELS`]: whether to sum over adjacent channels (`ACROSS_CHANNELS`) or nearby spatial locaitons (`WITHIN_CHANNEL`)

The local response normalization layer performs a kind of "lateral inhibition" by normalizing over local input regions. In `ACROSS_CHANNELS` mode, the local regions extend across nearby channels, but have no spatial extent (i.e., they have shape `local_size` x 1 x 1). In `WITHIN_CHANNEL`mode, the local regions extend spatially, but are in separate channels (i.e., they have shape 1 x `local_size` x `local_size`). Each input value is divided by $(1+(\alpha/n)\sum_i x_{2i})_\beta$, where n is the size of each local region, and the sum is taken over the region centered at that value (zero padding is added where necessary).

Im2col

`IM2COL` is a helper for doing the image-to-column transformation that you most likely do not need to know about. This is used in Caffe's original convolution to do matrix multiplication by laying out all patches into a matrix.

Loss Layers

Loss drives learning by comparing an output to a target and assigning cost to minimize. The loss itself is computed by the forward pass and the gradient w.r.t. to the loss is computed by the backward pass.

Softmax

- LayerType: "`SoftmaxWithLoss`"

The softmax loss layer computes the multinomial logistic loss of the softmax of its inputs. It's conceptually identical to a softmax layer followed by a multinomial logistic loss layer, but provides a more numerically stable gradient.

Sum-of-Squares / Euclidean

- LayerType: "EuclideanLoss"

The Euclidean loss layer computes the sum of squares of differences of its two inputs, $\frac{1}{2N}\sum_{i=1}^{N} \lVert x_{1i} - x_{2i} \rVert_2^2$.

Hinge / Margin

- LayerType: “HingeLoss”
- CPU implementation: `./src/caffe/layers/hinge_loss_layer.cpp`
- CUDA GPU implementation: none yet
- Parameters (`HingeLossParameter hinge_loss_param`)
  - Optional
    - `norm` [default L1]: the norm used. Currently L1, L2
- Inputs
  - `n * c * h * w` Predictions
  - `n * 1 * 1 * 1` Labels
- Output
  - `1 * 1 * 1 * 1` Computed Loss
- Samples

```
# L1 Norm
layer{
  name: "loss"
  type: "HingeLoss"
  bottom: "pred"
  bottom: "label"
}

# L2 Norm
layer {
  name: "loss"
  type: "HingeLoss"
  bottom: "pred"
  bottom: "label"
  top: "loss"
  hinge_loss_param {
    norm: L2
  }
}
The hinge loss layer computes a one-vs-all hinge or squared hinge loss.
```

Sigmoid Cross-Entropy

`SigmoidCrossEntropyLoss`

Infogain

`InfogainLoss`

Accuracy and Top-k(we may use this layer to validate the quality of our training net)

`Accuracy` scores the output as the accuracy of output with respect to target – it is not actually a loss and has no backward step.

Activation / Neuron Layers

In general, activation / Neuron layers are element-wise operators, **taking one bottom blob and producing one top blob of the same size**. In the layers below, we will ignore the input and out sizes as they are identical:

- Input
  - n * c * h * w
- Output
  - n * c * h * w

ReLU / Rectified-Linear and Leaky-ReLU

- LayerType: `ReLu`
- CPU implementation: `./src/caffe/layers/relu_layer.cpp`
- CUDA GPU implementation: `./src/caffe/layers/relu_layer.cu`
- Parameters (`ReLUParameter relu_param`)
  - Optional
    - `negative_slope` [default 0]: specifies whether to leak the negative part by multiplying it with the slope value rather than setting it to 0.
- Sample (as seen in `./examples/imagenet/imagenet_train_val.prototxt`)

```
layer {
  name: "relu1"
  type: "ReLu"
  bottom: "conv1"
  top: "conv1"
```

```
}
```

Given an input value x, The `RELU` layer computes the output as x if x > 0 and negative_slope * x if x <= 0. When the negative slope parameter is not set, it is equivalent to the standard ReLU function of taking max(x, 0). It also supports in-place computation, meaning that the bottom and the top blob could be the same to preserve memory consumption.

Sigmoid

- LayerType: `Sigmoid`
- CPU implementation: `./src/caffe/layers/sigmoid_layer.cpp`
- CUDA GPU implementation: `./src/caffe/layers/sigmoid_layer.cu`
- Sample (as seen in `./examples/imagenet/mnist_autoencoder.prototxt`)

```
layer {
  name: "encode1neuron"
  bottom: "encode1"
  top: "encode1neuron"
  type: "Sigmoid"
}
```

The `SIGMOID` layer computes the output as sigmoid(x) for each input element x.

TanH / Hyperbolic Tangent

- LayerType: TanH
- CPU implementation: `./src/caffe/layers/tanh_layer.cpp`
- CUDA GPU implementation: `./src/caffe/layers/tanh_layer.cu`
- Sample

```
layer{
  name: "layer"
  bottom: "in"
  top: "out"
  type: "TanH"
}
```

The `TANH` layer computes the output as tanh(x) for each input element x.

Absolute Value

- LayerType: `AbsVal`
- CPU implementation: `./src/caffe/layers/absval_layer.cpp`
- CUDA GPU implementation: `./src/caffe/layers/absval_layer.cu`
- Sample

```
layer {
  name: "layer"
  bottom: "in"
  top: "out"
  type: "AbsVal"
}
```

The `ABSVAL` layer computes the output as abs(x) for each input element x.

Power

- LayerType: `Power`
- CPU implementation: `./src/caffe/layers/power_layer.cpp`
- CUDA GPU implementation: `./src/caffe/layers/power_layer.cu`
- Parameters (`PowerParameter power_param`)
  - Optional
    - `power` [default 1]
    - `scale` [default 1]
    - `shift` [default 0]
- Sample

```
layer{
  name: "layer"
  bottom: "in"
  top: "out"
  type: "Power"
  power_param {
    power: 1
    scale: 1
    shift: 0
  }
}
```

The `POWER` layer computes the output as (shift + scale * x) ^ power for each input element x.

BNLL

- LayerType: `BNLL`
- CPU implementation: `./src/caffe/layers/bnll_layer.cpp`
- CUDA GPU implementation: `./src/caffe/layers/bnll_layer.cu`
- Sample

```
layers {
  name: "layer"
  bottom: "in"
  top: "out"
  type: "BNLL"
}
```

The `BNLL` (binomial normal log likelihood) layer computes the output as log(1 + exp(x)) for each input element x.

Data Layers

Data enters Caffe through data layers: they lie at the bottom of nets. Data can come from efficient databases (LevelDB or LMDB), directly from memory, or, when efficiency is not critical, from files on disk in HDF5 or common image formats.

Common input preprocessing (mean subtraction, scaling, random cropping, and mirroring) is available by specifying `TransformationParameter`s.

Database

- LayerType: `Data`
- Parameters
  - Optional
    - `source`: the name of the directory containing the database
    - `batch_size`: the number of inputs to process at one time
    - `rand_skip`: skip up to this number of inputs at the beginning; useful for asynchronous sgd
    - `backend` [default `LEVELDB`]: choose whether to use a `LEVELDB` or `LMDB`

In-Memory

- LayerType: `MemoryData`
- Parameters
  - Optional
    - `batch_size`, `channels`, `height`, `width`: specify the size of input chunks to read from memory

The memory data layer reads data directly from memory, without copying it. In order to use it, one must call `MemoryDataLayer::Reset` (from C++)
or `Net.set_input_arrays` (from Python) in order to specify a source of contiguous data (as 4D row major array), which is read one batch-sized chunk at a time.

HDF5 Input

- LayerType: `HDF5Data`
- Parameters
  - Optional
    - `source`: the name of the file to read from
    - `batch_size`
    - `shuffle [default = false] (// Specify whether to shuffle the data.`
      `// If shuffle == true, the ordering of the HDF5 files is shuffled,`
      `// and the ordering of data within any given HDF5 file is shuffled,`
      `// but data between different files are not interleaved; all of a file's`
      `// data are output (in a random order) before moving onto another file.)`

    - `HDF5 Output`
- LayerType: `HDF5Output`
- Parameters
  - Optional
    - `file_name`: name of file to write to

The HDF5 output layer performs the opposite function of the other layers in this section: it writes its input blobs to disk.

Images

- LayerType: `ImageData`
- Parameters
  - Optional
    - `source`: name of a text file, with each line giving an image filename and label

- `batch_size`: number of images to batch together
- `rand_skip`
- `shuffle` [default false]
- `new_height, new_width: if provided, resize all images to this size`

Windows

`WINDOW_DATA`

Dummy

`DUMMY_DATA` is for development and debugging. See `DummyDataParameter`.

Common Layers

Inner Product

- LayerType: `InnerProduct`
  - CPU implementation: `./src/caffe/layers/inner_product_layer.cpp`
  - CUDA GPU implementation: `./src/caffe/layers/inner_product_layer.cu`
  - Parameters (`InnerProductParameter inner_product_param`)
  - Optional
    - `num_output` (`c_o`): the number of filters
    - `bias_filler` [default `type: 'constant' value: 0`]
    - `bias_term` [default `true`]: specifies whether to learn and apply a set of additive biases to the filter outputs
  - Strongly recommended
    - `weight_filler` [default `type: 'constant' value: 0`]
  - Input
  - `n * c_i * h_i * w_i`
  - Output
  - `n * c_o * 1 * 1`
  - Sample

```
layer {
  name: "fc8"
  type: "InnerProduct"
 bottom: "fc7"
  top: "fc8"
  param{
  lr_mult: 1         # learning rate multiplier for the filters
  decay_mult: 0      # decay rate multiplier for the biases
}
 param{
  lr_mult: 2
  Decay_mult: 0
  inner_product_param {
    num_output: 1000
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
 }
```

The `INNER_PRODUCT` layer (also usually **referred to as the fully connected layer**) treats the input as a simple vector and produces an output in the form of a single vector (with the blob's height and width set to 1).

Splitting

The `SPLIT` layer is a utility layer that splits an input blob to multiple output blobs. This is used when a blob is fed into multiple output layers.

Flattening

The `FLATTEN` layer is a utility layer that flattens an input of shape `n * c * h * w` to a simple vector output of shape `n * (c*h*w) * 1 * 1`.

Concatenation

- LayerType: `Concat`

- CPU implementation: `./src/caffe/layers/concat_layer.cpp`
- CUDA GPU implementation: `./src/caffe/layers/concat_layer.cu`
- Parameters (`ConcatParameter concat_param`)
  - Optional
    - `concat_dim` [default 1]: 0 for concatenation along num and 1 for channels.
- Input
  - `n_i * c_i * h * w` for each input blob i from 1 to K.
- Output
  - if `concat_dim = 0`: `(n_1 + n_2 + ... + n_K) * c_1 * h * w`, and all input `c_i` should be the same.
  - if `concat_dim = 1`: `n_1 * (c_1 + c_2 + ... + c_K) * h * w`, and all input `n_i` should be the same.
- Sample

```
layer {
  name: "concat"
  bottom: "in1"
  bottom: "in2"
  top: "out"
  type: "Concat"
  concat_param {
    concat_dim: 1
  }
}
```

The `CONCAT` layer is a utility layer that concatenates its multiple input blobs to one single output blob. Currently, the layer supports concatenation along num or channels only.

Slicing

The `SLICE` layer is a utility layer that slices an input layer to multiple output layers along a given dimension (currently num or channel only) with given slice indices.

- Sample

```
layer{
  name: "slicer_label"
  type: "Slice"
  bottom: "label"
  ## Example of label with a shape N x 3 x 1 x 1
  top: "label1"
  top: "label2"
```

```
  top: "label3"
  slice_param {
      slice_dim: 1
      slice_point: 1
      slice_point: 2
  }
}
```

`slice_dim` indicates the target dimension and can assume only two values: 0 for num or 1 for channel; `slice_point` indicates indexes in the selected dimension (the number of indexes must be equal to the number of top blobs minus one).

Elementwise Operations

`ELTWISE`

Argmax

`ARGMAX`

Softmax

`SOFTMAX`

Mean-Variance Normalization

`MVN`

6.Interfaces

Caffe has command line, Python, and MATLAB interfaces for day-to-day usage, interfacing with research code, and rapid prototyping. While Caffe is a C++ library at heart and it exposes a modular interface for development, not every occasion calls for custom compilation. The cmdcaffe, pycaffe, and matcaffe interfaces are here for you.

6.1Command Line

The command line interface – cmdcaffe – is the `caffe` tool for model training, scoring, and diagnostics. Run `caffe` without any arguments for help. This tool and others are found in caffe/build/tools. (The following example calls require completing the LeNet / MNIST example first.)

Training: `caffe train` learns models from scratch, resumes learning from saved snapshots, and fine-tunes models to new data and tasks:

- All training requires a solver configuration through the `-solver solver.prototxt` argument.
- Resuming requires the `-snapshot model_iter_1000.solverstate` argument to load the solver snapshot.
- Fine-tuning requires the `-weights model.caffemodel` argument for the model initialization.

For example, you can run:

```
# train LeNet
caffe train -solver examples/mnist/lenet_solver.prototxt
# train on GPU 2
caffe train -solver examples/mnist/lenet_solver.prototxt -gpu 2
# resume training from the half-way point snapshot
caffe train -solver examples/mnist/lenet_solver.prototxt -snapshot
examples/mnist/lenet_iter_5000.solverstate
```

For a full example of fine-tuning, see examples/finetuning_on_flickr_style, but the training call alone is

```
# fine-tune CaffeNet model weights for style recognition
caffe train -solver examples/finetuning_on_flickr_style/solver.prototxt
-weights models/bvlc_reference_caffenet/bvlc_reference_caffenet.caffemodel
```

Testing: `caffe test` scores models by running them in the test phase and reports the net output as its score. The net architecture must be properly defined to output an accuracy measure or loss as its output. The per-batch score is reported and then the grand average is reported last.

```
#
# score the learned LeNet model on the validation set as defined in the
# model architeture lenet_train_test.prototxt
caffe test -model examples/mnist/lenet_train_test.prototxt -weights
examples/mnist/lenet_iter_10000.caffemodel -gpu 0 -iterations 100
```

Benchmarking: `caffe time` benchmarks model execution layer-by-layer through timing and synchronization. This is useful to check system performance and measure relative execution times for models.

```
# (These example calls require you complete the LeNet / MNIST example first.)
# time LeNet training on CPU for 10 iterations
```

```
caffe time -model examples/mnist/lenet_train_test.prototxt -iterations 10
# time LeNet training on GPU for the default 50 iterations
caffe time -model examples/mnist/lenet_train_test.prototxt -gpu 0
# time a model architecture with the given weights on the first GPU for 10 iterations
caffe time -model examples/mnist/lenet_train_test.prototxt -weights
examples/mnist/lenet_iter_10000.caffemodel -gpu 0 -iterations 10
```

Diagnostics: `caffe device_query` reports GPU details for reference and checking device ordinals for running on a given device in multi-GPU machines.

```
# query the first device
caffe device_query -gpu 0
```

6.2Python

The Python interface – pycaffe – is the `caffe` module and its scripts in caffe/python. `import caffe`to load models, do forward and backward, handle IO, visualize networks, and even instrument model solving. All model data, derivatives, and parameters are exposed for reading and writing.

- `caffe.Net` is the central interface for loading, configuring, and running models.`caffe.Classsifier` and `caffe.Detector` provide convenience interfaces for common tasks.
- `caffe.SGDSolver` exposes the solving interface.
- `caffe.io` handles input / output with preprocessing and protocol buffers.
- `caffe.draw` visualizes network architectures.
- Caffe blobs are exposed as numpy ndarrays for ease-of-use and efficiency.

Tutorial IPython notebooks are found in caffe/examples: do `ipython notebook caffe/examples` to try them. For developer reference docstrings can be found throughout the code.

Compile pycaffe by `make pycaffe`. The module dir caffe/python/caffe should be installed in your PYTHONPATH for `import caffe`.

7.Data: Ins and Outs

Data flows through Caffe as Blobs. Data layers load input and save output by converting to and from Blob to other formats. Common transformations like mean-subtraction and feature-scaling are done by data layer configuration. New input types are supported by

developing a new data layer – the rest of the Net follows by the modularity of the Caffe layer catalogue.
This data layer definition

```
layer {
  name: "mnist"
  # DATA layer loads leveldb or lmdb storage DBs for high-throughput.
  type: "Data"
  # the 1st top is the data itself: the name is only convention
  top: "data"
  # the 2nd top is the ground truth: the name is only convention
  top: "label"
  # the DATA layer configuration
  data_param {
    # path to the DB
    source: "examples/mnist/mnist_train_lmdb"
    # type of DB: LEVELDB or LMDB (LMDB supports concurrent reads)
    backend: LMDB
    # batch processing improves efficiency.
    batch_size: 64
  }
  # common data transformations
  transform_param {
    # feature scaling coefficient: this maps the [0, 255] MNIST data to [0, 1]
    scale: 0.00390625
  }
}
```

loads the MNIST digits.

Tops and Bottoms: A data layer makes **top** blobs to output data to the model. It does not have**bottom** blobs since it takes no input.
Data and Label: a data layer has at least one top canonically named **data**. For ground truth a second top can be defined that is canonically named **label**. Both tops simply produce blobs and there is nothing inherently special about these names. The (data, label) pairing is a convenience for classification models.
Transformations: data preprocessing is parametrized by transformation messages within the data layer definition.

```
layer {
  name: "data"
  type: "Data"
  [...]
```

```
  transform_param {
    scale: 0.1 #make sure the image grayscale stay in the 0-1
    mean_file: mean.binaryproto
    # for images in particular horizontal mirroring and random cropping
    # can be done as simple data augmentations.
    mirror: true  # true = on, false = off
    # crop a `crop_size` x `crop_size` patch:
    # - at random during training
    # - from the center during testing
    crop_size: 227
  }
}
```

Prefetching: for throughput data layers fetch the next batch of data and prepare it in the background while the Net computes the current batch.

Multiple Inputs: a Net can have multiple inputs of any number and type. Define as many data layers as needed giving each a unique name and top. Multiple inputs are useful for non-trivial ground truth: one data layer loads the actual data and the other data layer loads the ground truth in lock-step. In this arrangement both data and label can be any 4D array. Further applications of multiple inputs are found in multi-modal and sequence models. In these cases you may need to implement your own data preparation routines or a special data layer.

Improvements to data processing to add formats, generality, or helper utilities are welcome!

7.1Formats

Refer to the layer catalogue of data layers for close-ups on each type of data Caffe understands.

7.2Deployment Input

For on-the-fly computation deployment Nets define their inputs by `input` fields: these Nets then accept direct assignment of data for online or interactive computation.