Part 1.
Caffe minimum background

Introduction

What is Caffe?

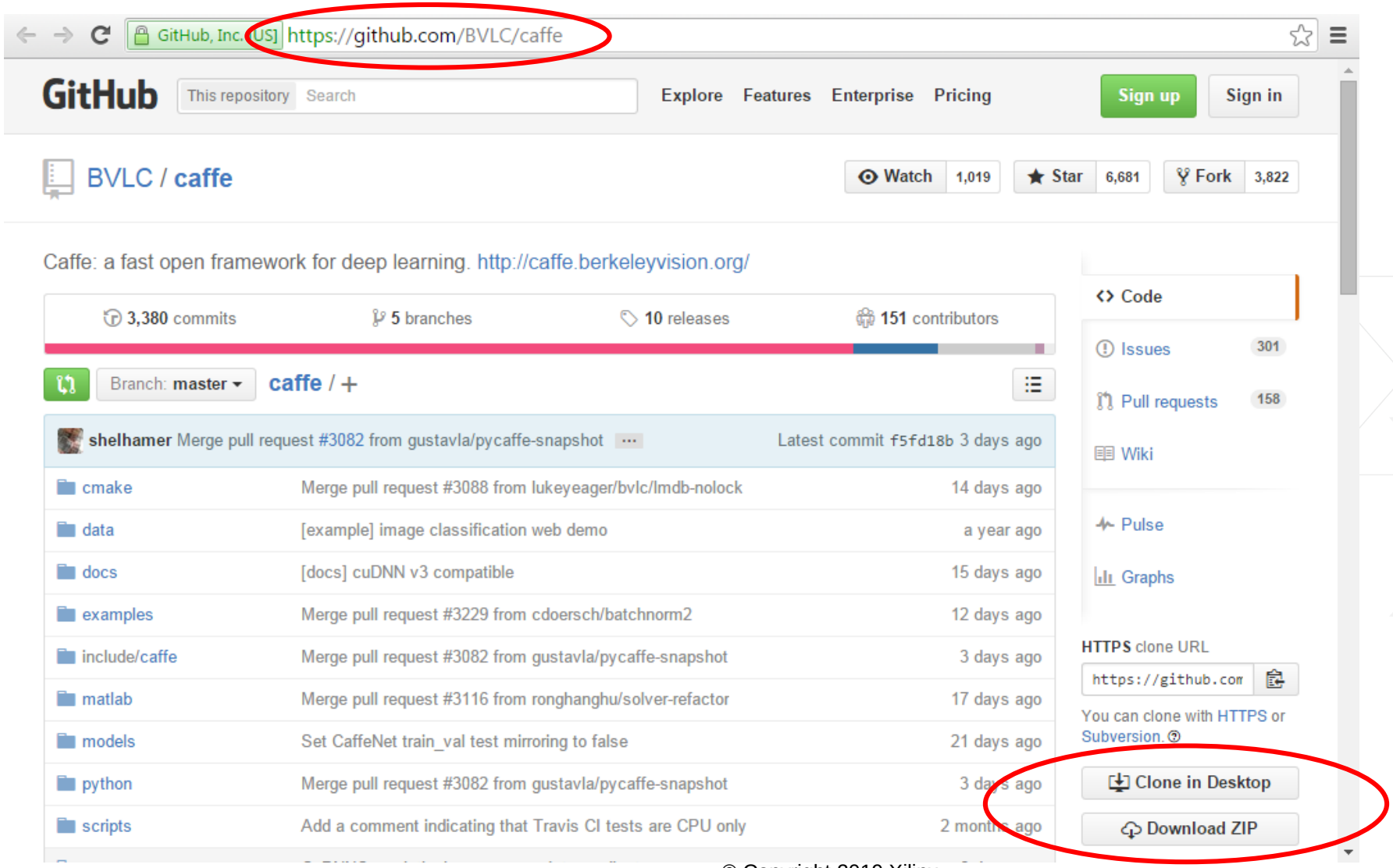
Convolution Architecture For Feature Extraction (CAFFE)

An open framework for deep learning developed by the Berkeley Vision and Learning Center (BVLC)

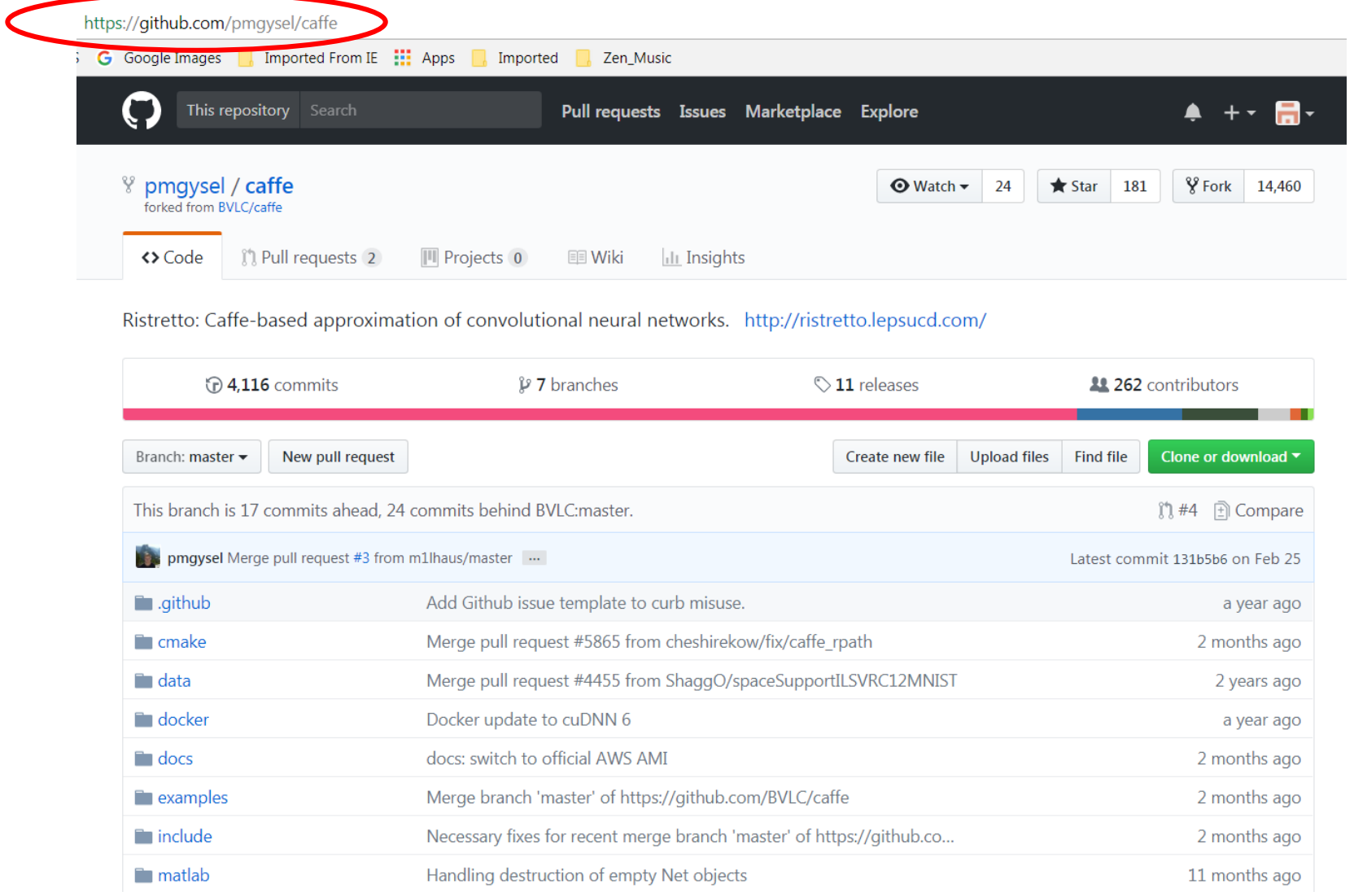
caffe.berkeleyvision.org
http://github.com/BVLC/caffe

- Pure C++/CUDA architecture
- Command line, Python, MATLAB interfaces
- Fast, well-tested code
- Pre-processing and deployment tools, reference models and examples
- Image data management
- Seamless GPU acceleration
- Large community of contributors to the open-source project

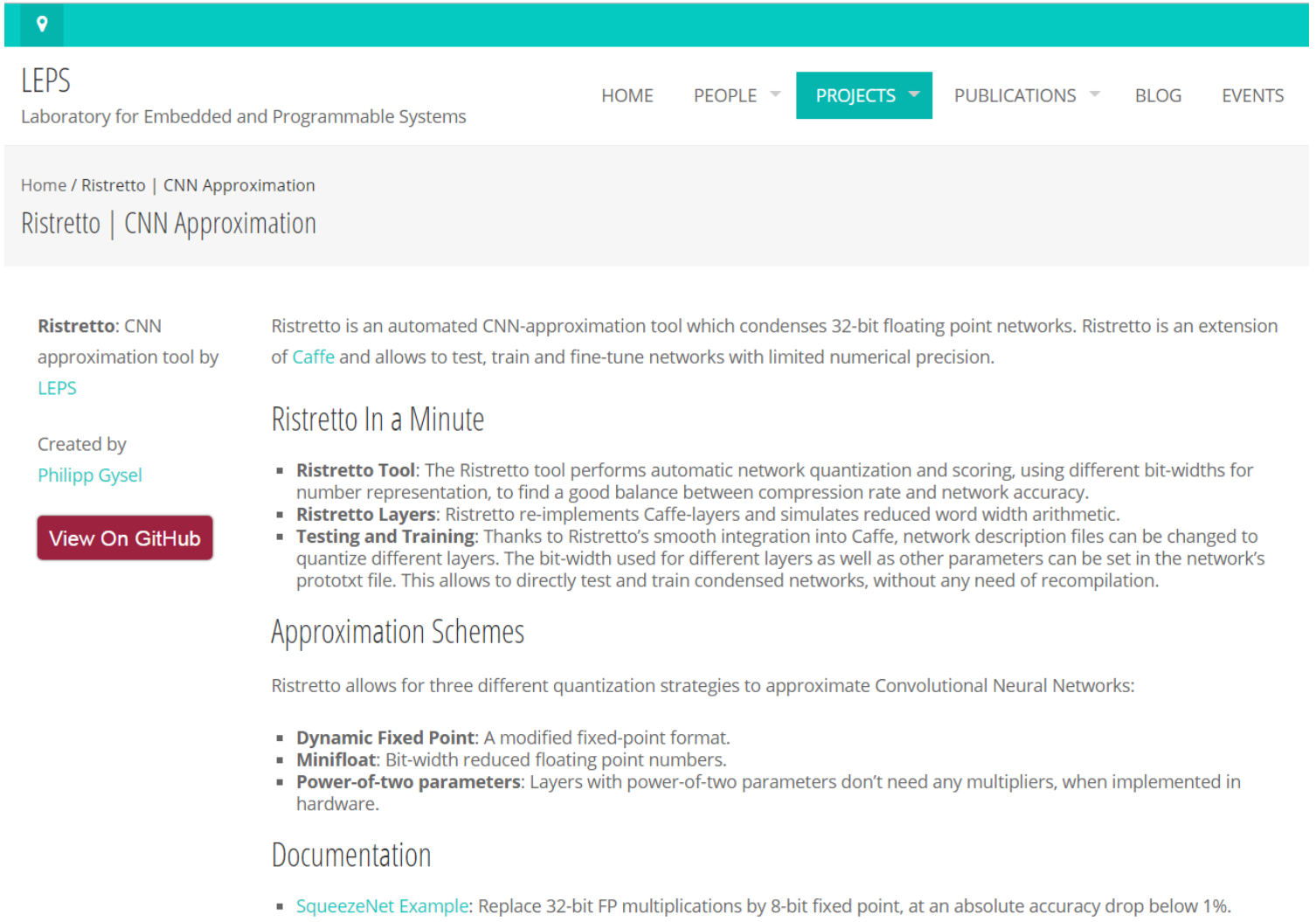
BVLC Caffe from github



Download Caffe Ristretto from github



http://lepsucd.com/?page_id=621



LEPS
Laboratory for Embedded and Programmable Systems

HOME PEOPLE PROJECTS PUBLICATIONS BLOG EVENTS

Home / Ristretto | CNN Approximation

Ristretto | CNN Approximation

Ristretto: CNN approximation tool by LEPS

Created by Philipp Gysel

View On GitHub

Ristretto is an automated CNN-approximation tool which condenses 32-bit floating point networks. Ristretto is an extension of Caffe and allows to test, train and fine-tune networks with limited numerical precision.

Ristretto In a Minute

- **Ristretto Tool**: The Ristretto tool performs automatic network quantization and scoring, using different bit-widths for number representation, to find a good balance between compression rate and network accuracy.
- **Ristretto Layers**: Ristretto re-implements Caffe-layers and simulates reduced word width arithmetic.
- **Testing and Training**: Thanks to Ristretto's smooth integration into Caffe, network description files can be changed to quantize different layers. The bit-width used for different layers as well as other parameters can be set in the network's prototxt file. This allows to directly test and train condensed networks, without any need of recompilation.

Approximation Schemes

Ristretto allows for three different quantization strategies to approximate Convolutional Neural Networks:

- **Dynamic Fixed Point**: A modified fixed-point format.
- **Minifloat**: Bit-width reduced floating point numbers.
- **Power-of-two parameters**: Layers with power-of-two parameters don't need any multipliers, when implemented in hardware.

Documentation

- SqueezeNet Example: Replace 32-bit FP multiplications by 8-bit fixed point, at an absolute accuracy drop below 1%.

Why Caffe Ristretto

- Caffe generates 32-bit floating point NN models. There are 2 output files:
  - float_deploy.prototxt is a text file with the CNN layers description
  - float.caffemodel is the binary file of 32-bit floating point weights
- Xilinx CHaiDNN-v2 can run only fixed point (8- or 16-bit) NN models
- Caffe Ristretto quantizes the original Caffe model and add parameters in the prototxt file to describe their quantization. The 2 new output files are:
  - fixed_deploy.prototxt
  - quantized.caffemodel
- You can only install Caffe Ristretto, as it contains the original Caffe too.
  - It is up to you: FYI, I have installed both plus a 3rd fork of Caffe suitable for SSD Object Detection
  - Support to Ristretto can be obtained by our partner GL-Research, who implemented it

Caffe / Ristretto after having been installed

```
(caffe_python_2) danieleb@CentOS63-x86-64:~/ML/ristretto_caffe_python_2$ ls
build                          docker           Makefile.config.example
caffe.cloc                     docs             matlab
caffe_python_2_requirements.txt  examples       models
cmake                          include          python
CMakeLists.txt                 INSTALL.md       README.md
CMakeLists.txt~                LICENSE          scripts
CONTRIBUTING.md                Makefile         src
CONTRIBUTORS.md                Makefile~        tools
data                           Makefile.config
distribute                     Makefile.config~
(caffe_python_2) danieleb@CentOS63-x86-64:~/ML/ristretto_caffe_python_2$ ls -l distribute/
total 20
drwxrwxr-x 2 danieleb danieleb 4096 apr  6 23:25 bin
drwxrwxr-x 4 danieleb danieleb 4096 apr  6 23:25 include
drwxrwxr-x 2 danieleb danieleb 4096 apr  6 23:25 lib
drwxrwxr-x 2 danieleb danieleb 4096 apr  6 23:25 proto
drwxrwxr-x 3 danieleb danieleb 4096 apr  6 23:25 python
(caffe_python_2) danieleb@CentOS63-x86-64:~/ML/ristretto_caffe_python_2$ ls -l models/
total 28
drwxrwxr-x 2 danieleb danieleb 4096 nov  2 21:28 bvlc_alexnet
drwxrwxr-x 2 danieleb danieleb 4096 nov  2 21:28 bvlc_googlenet
drwxrwxr-x 2 danieleb danieleb 4096 nov  2 21:28 bvlc_reference_caffenet
drwxrwxr-x 2 danieleb danieleb 4096 nov  2 21:28 bvlc_reference_rcnn_ilsvrc13
drwxrwxr-x 2 danieleb danieleb 4096 nov  2 21:28 finetune_flickr_style
drwxrwxr-x 3 danieleb danieleb 4096 nov  2 21:28 SqueezeNet
drwxrwxr-x 2 danieleb danieleb 4096 nov  2 21:28 xilinx_googlenet
(caffe_python_2) danieleb@CentOS63-x86-64:~/ML/ristretto_caffe_python_2$ ls -l examples/
total 4904
-rw-rw-r-- 1 danieleb danieleb  813348 nov  2 21:28 00-classification.ipynb
-rw-rw-r-- 1 danieleb danieleb  376291 nov  2 21:28 01-learning-lenet.ipynb
-rw-rw-r-- 1 danieleb danieleb  480501 nov  2 21:28 02-fine-tuning.ipynb
-rw-rw-r-- 1 danieleb danieleb  452886 nov  2 21:28 brewing-logreg.ipynb
drwxrwxr-x 4 danieleb danieleb    4096 apr  8 00:50 cifar10
-rw-rw-r-- 1 danieleb danieleb    1063 nov  2 21:28 CMakeLists.txt
drwxrwxr-x 2 danieleb danieleb    4096 nov  2 21:28 cpp_classification
-rw-rw-r-- 1 danieleb danieleb  702461 nov  2 21:28 detection.ipynb
drwxrwxr-x 2 danieleb danieleb    4096 nov  2 21:28 feature_extraction
drwxrwxr-x 2 danieleb danieleb    4096 nov  2 21:28 finetune_flickr_style
drwxrwxr-x 2 danieleb danieleb    4096 nov  2 21:28 finetune_pascal_detection
drwxrwxr-x 2 danieleb danieleb    4096 nov  2 21:28 hdf5_classification
drwxrwxr-x 2 danieleb danieleb    4096 nov  2 21:28 imagenet
drwxrwxr-x 2 danieleb danieleb    4096 nov  2 21:28 images
drwxrwxr-x 2 danieleb danieleb    4096 nov  2 21:28 mnist
drwxrwxr-x 2 danieleb danieleb    4096 nov  2 21:28 net_surgery
-rw-rw-r-- 1 danieleb danieleb  583249 nov  2 21:28 net_surgery.ipynb
-rw-rw-r-- 1 danieleb danieleb 1539559 nov  2 21:28 pascal-multilabel-with-datalayer.ipynb
drwxrwxr-x 3 danieleb danieleb    4096 nov  2 21:28 pycaffe
drwxrwxr-x 3 danieleb danieleb    4096 nov  2 21:28 ristretto
drwxrwxr-x 2 danieleb danieleb    4096 nov  2 21:28 siamese
drwxrwxr-x 3 danieleb danieleb    4096 nov  2 21:28 web_demo
```

Learning using gradient descent

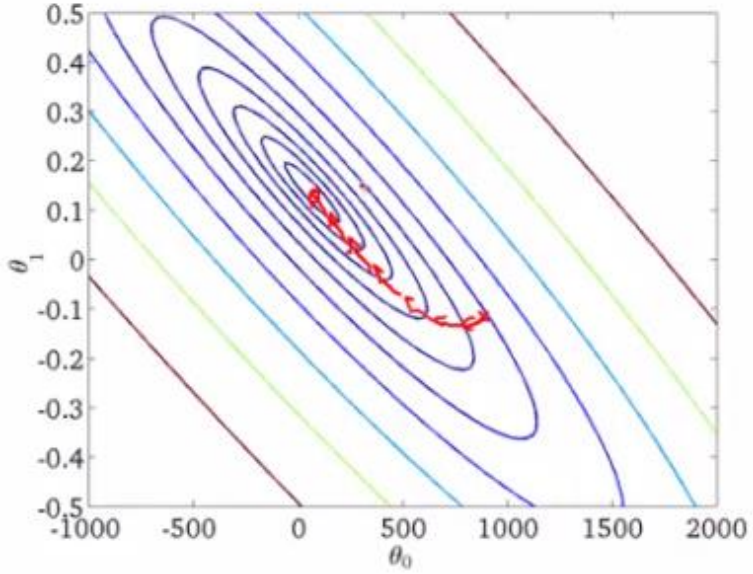
- Loss function we are minimizing

$$Loss(w) = \frac{1}{N}\sum_{i=1}^{N} \ell(w, x_i, y_i)$$

- Gradient descent

$$w_{t+1} = w_t - \frac{1}{N}\eta_t \sum_{i=1}^{N} \nabla_w \ell(w_t, x_i, y_i)$$

- Problem – for large N, a single step is very expensive.



Gradient descent

- Gradient descent is a first-order optimization algorithm.
- To find a local minimum of a function using gradient descent, one takes steps proportional to the negative of the gradient (or of the approximate gradient) of the function at the current point.
- One starts with a guess w_0 for a local minimum of F, and considers the sequence w_0, w_1, w_2,… such that

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \nabla F(\mathbf{w}_t),\ t \geq 0.$$

  If function F is defined and differentiable in a neighborhood of a point w_t, then for small enough step size eta we get that:

$$F(\mathbf{w}_0) \geq F(\mathbf{w}_1) \geq F(\mathbf{w}_2) \geq \cdots,$$

  so hopefully the sequence w_t converges to the desired local minimum.

- Note that the value of the step size eta is allowed to change at every iteration.

Stochastic gradient descent

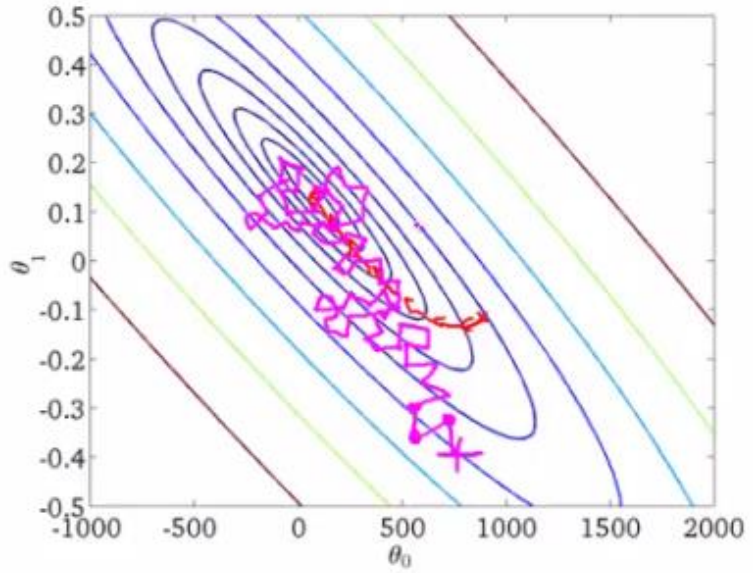
- Loss function we are minimizing

$$Loss(w) = \frac{1}{N}\sum_{i=1}^{N} \ell(w, x_i, y_i)$$

- Gradient descent

$$w_{t+1} = w_t - \frac{1}{N}\eta_t \sum_{i=1}^{N} \nabla_w \ell(w_t, x_i, y_i)$$

- Problem – for large N, a single step is very expensive.



- Solution – Stochastic gradient descent. At each iteration select random data points $\{i_1, ..., i_B\}$ with batch size B, then

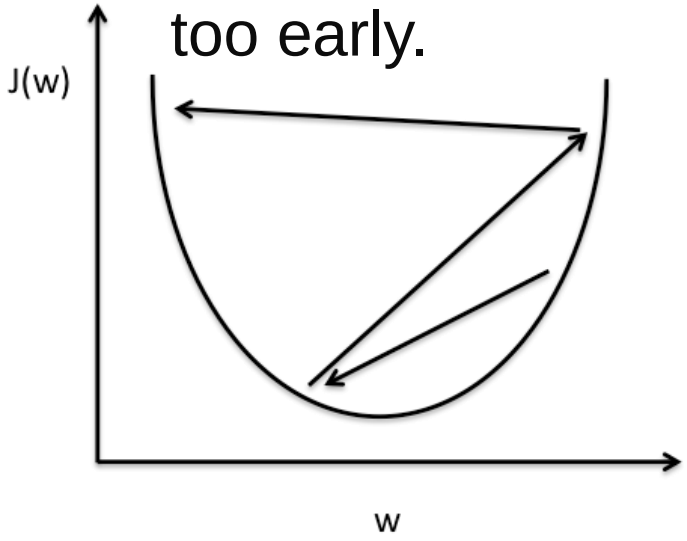
$$w_{t+1} = w_t - \frac{1}{B}\eta_t \sum_{k=1}^{B} \nabla_w \ell(w_t, x_{i_k}, y_{i_k})$$

- As B grows, we get a better approximation of the real gradient but at a higher computational cost.

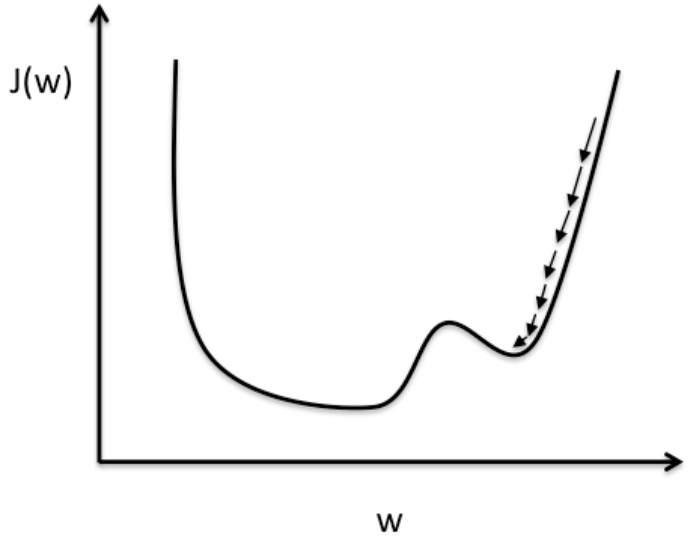
Learning rate η_t

$$w_{t+1} = w_t - \frac{1}{N}\eta_t \sum_{i=1}^{N} \nabla_w \ell(w_t, x_i, y_i)$$

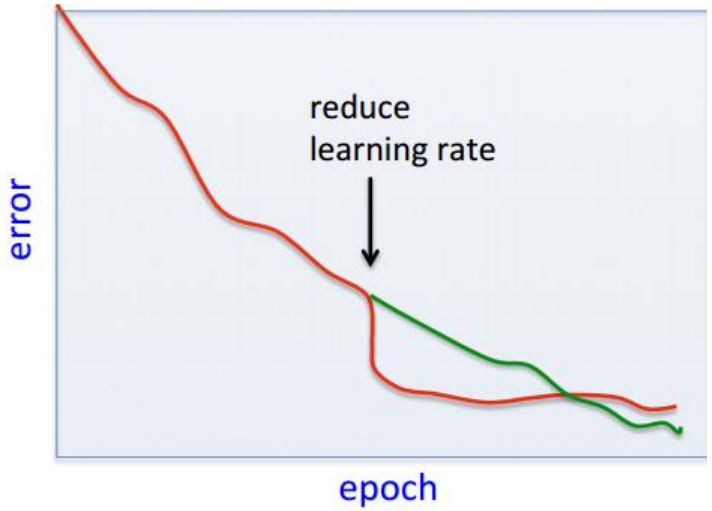
- Don’t start too big, and not too small.
- Start as big as you can without diverging, then when getting to a plateau start reducing the learning rate. Be careful not to reduce the learning rate too early.



Large learning rate: Overshooting.



Small learning rate: Many iterations until convergence and trapping in local minima.



Momentum

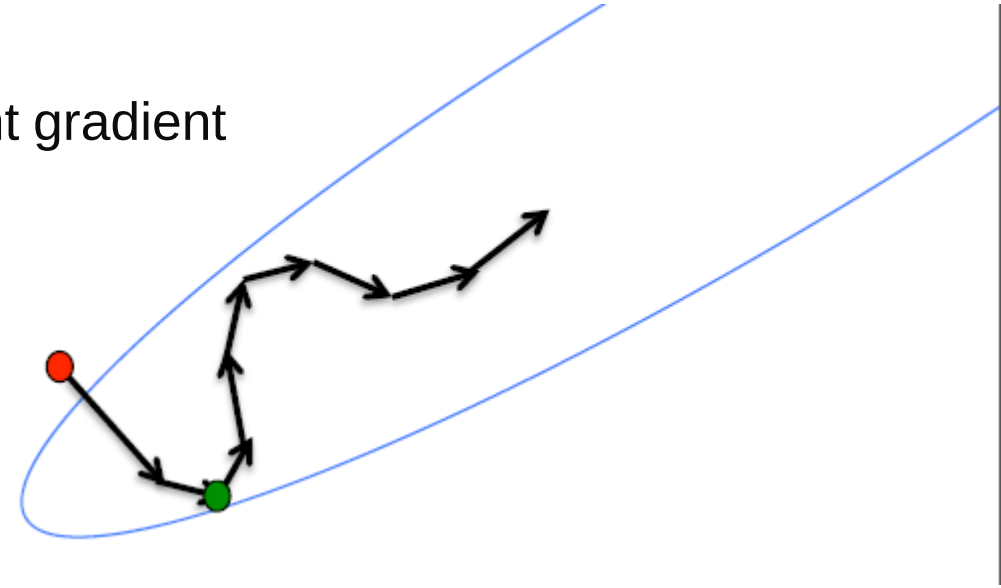
- The momentum method is a technique for accelerating gradient descent that accumulates a velocity vector in directions of persistent reduction in the objective across iterations.
- The momentum $\mu \in [0, 1]$ is the weight of the previous update.
- The update value Vt+1 and the updated weights Wt+1 at iteration t+1:

$$\mathbf{v}_{t+1} = \mu \mathbf{v}_t - \eta_t \nabla L(\mathbf{w}_t)$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \mathbf{v}_{t+1}$$

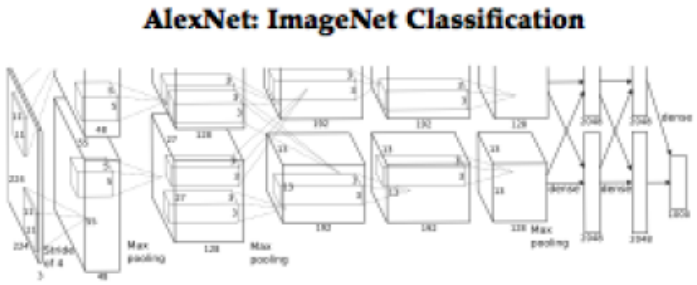
The intuition behind the momentum method

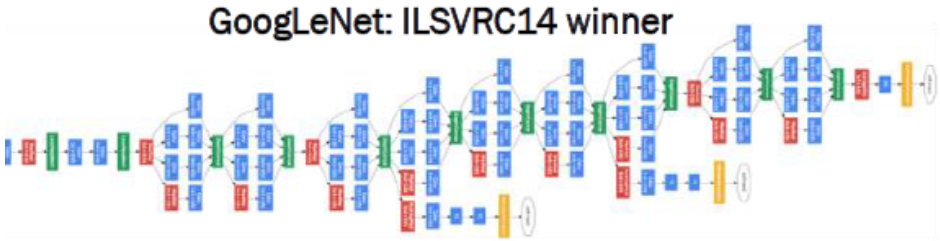
- Imagine a ball on the error surface.
- The ball starts off by following the gradient, but once it has velocity, it no longer does steepest descent.
- Its momentum makes it keep going in the previous direction.
- It damps oscillations in directions of high curvature by combining gradients with opposite signs.
- It builds up speed in directions with a gentle but consistent gradient



Anatomy of Caffe

Reference Caffe Models





Caffe offers the

- model definitions
- optimization settings
- pre-trained weights

so you can start right away.

The BVLC models are licensed for unrestricted use.

The community shares models in the Model Zoo.

Slide credit: Evan Shelhamer, Jeff Donahue, Jon Long, Yangqing Jia, and Ross Girshick

Caffe features: data pre-processing and management

Data ingest formats

LevelDB or LMDB database

In-memory (C++ and Python only)

HDF5

Image files

Pre-processing tools

LevelDB/LMDB creation from raw images

Training and validation set creation with shuffling

Mean-image generation

```
$CAFFE_ROOT/build/tools
```

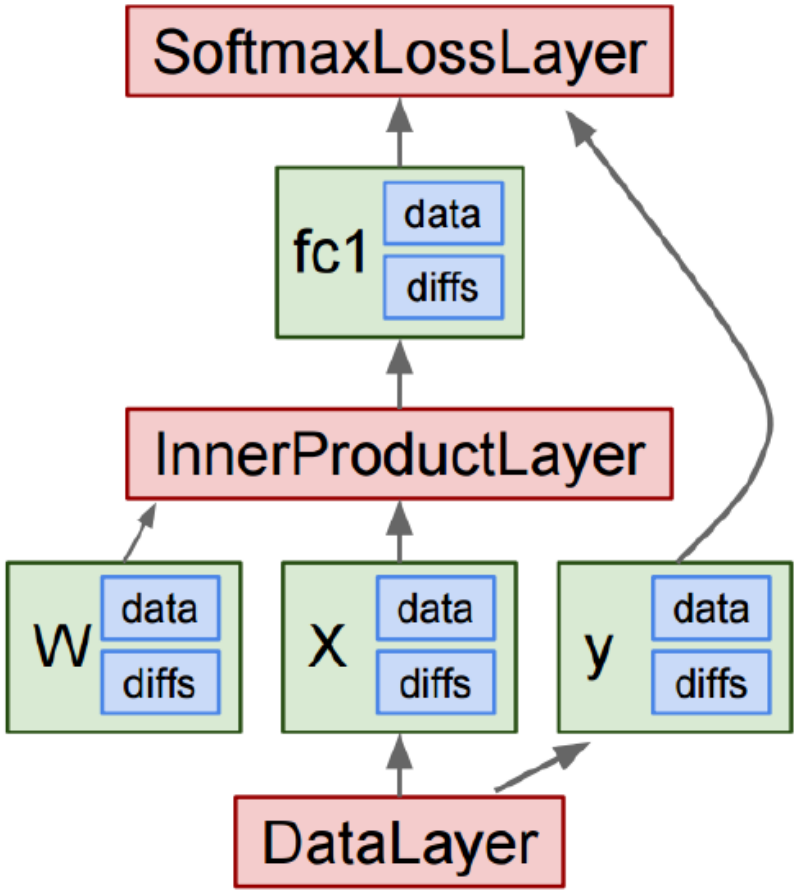
Data transformations

Image cropping, resizing, scaling and mirroring

Mean subtraction

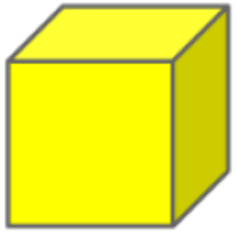
Caffe Features: Main Classes

- **Blob:** Stores data and derivatives
- **Layer**: Transforms bottom blobs to top blobs
- **Net:** Many layers; computes gradients via Forward / Backward
- **Solver:** Uses gradients to update weights

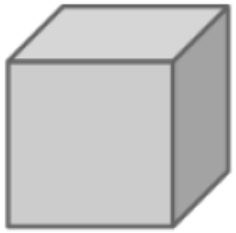


Slide credit: Stanford Vision CS231

Caffe Features: Blobs



Data
*N*umber x *K* Channel x *H*eight x *W*idth
256 x 3 x 227 x 227 for ImageNet train input



Parameter: Convolution Weight
N Output x *K* Input x *H*eight x *W*idth
96 x 3 x 11 x 11 for CaffeNet conv1

Parameter: Convolution Bias
96 x 1 x 1 x 1 for CaffeNet conv1

N-D arrays for storing and communicating data

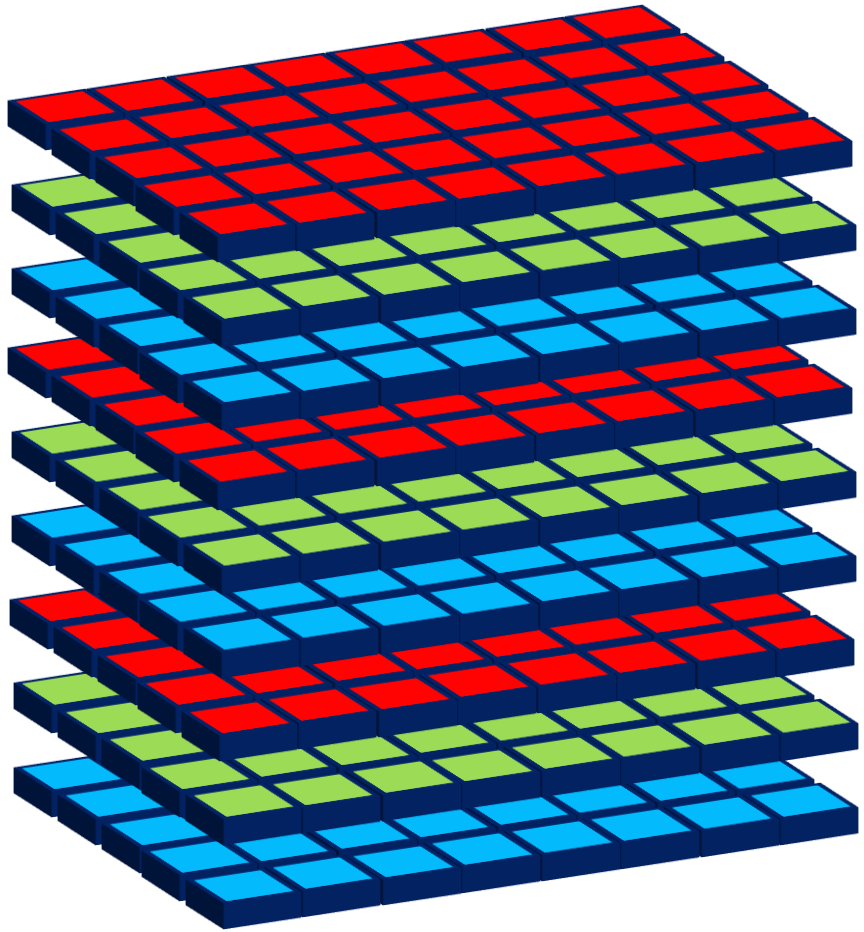
- Hold data, derivatives and parameters
- Lazily allocate memory
- Shuttle between CPU and GPU

Slide credit: Evan Shelhamer, Jeff Donahue, Jon Long, Yangqing Jia, and Ross Girshick

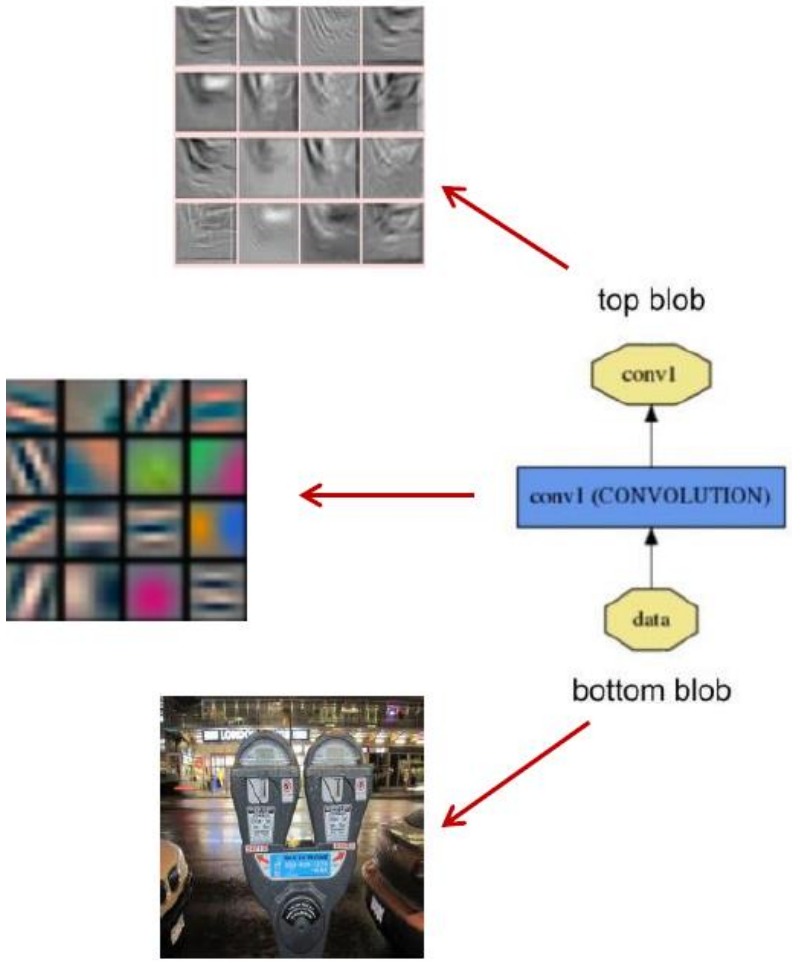
Caffe - Storing Data In Memory

Blob size: N x C x H x W

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



Caffe Features: Layers



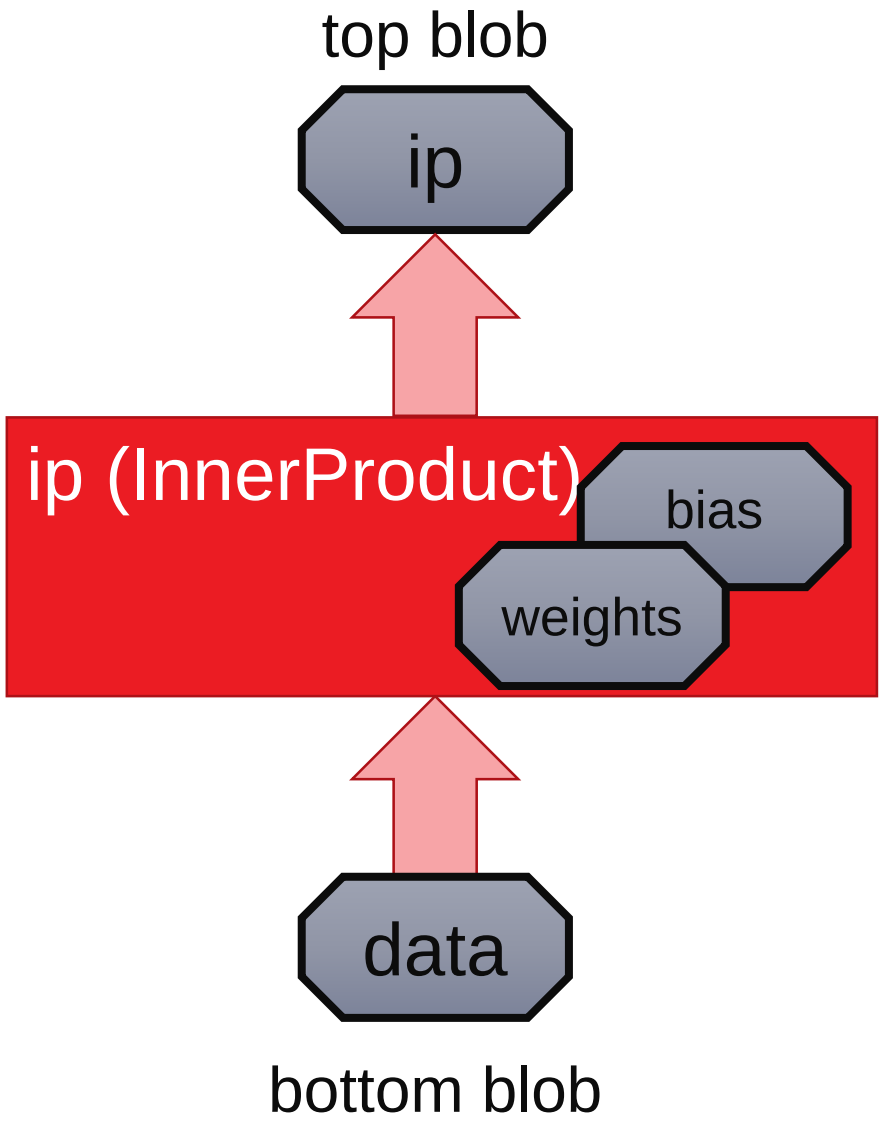
Caffe's fundamental unit of computation

Implemented as layers:

- Data access
- Convolution
- Pooling
- Activation Functions
- Loss Functions
- Dropout
- etc.

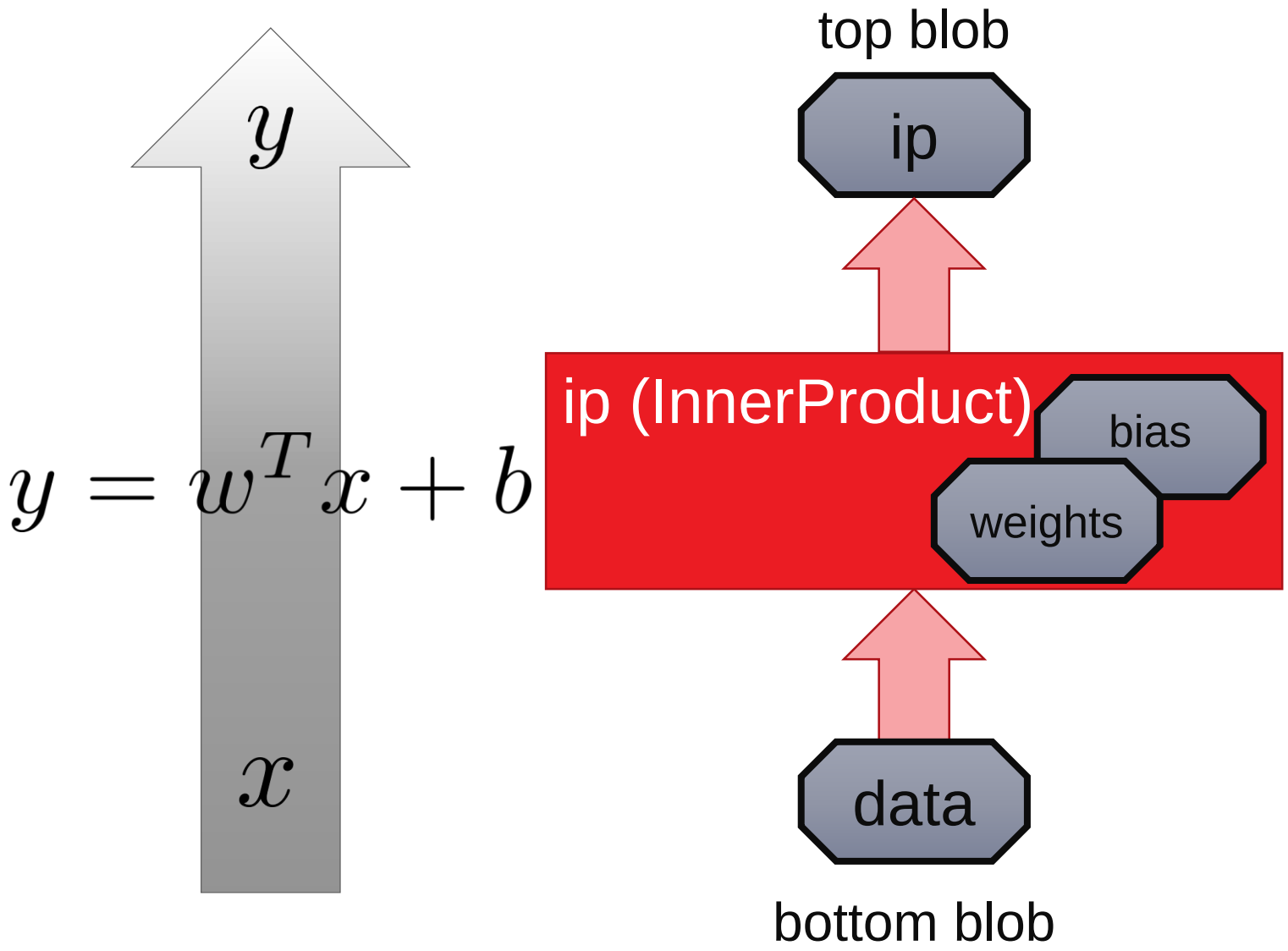
Caffe Features: Layers

- A layer takes input through *bottom* connections and makes output through *top* connections
- Each layer type defines three computations: *setup*, *forward*, and *backward*.



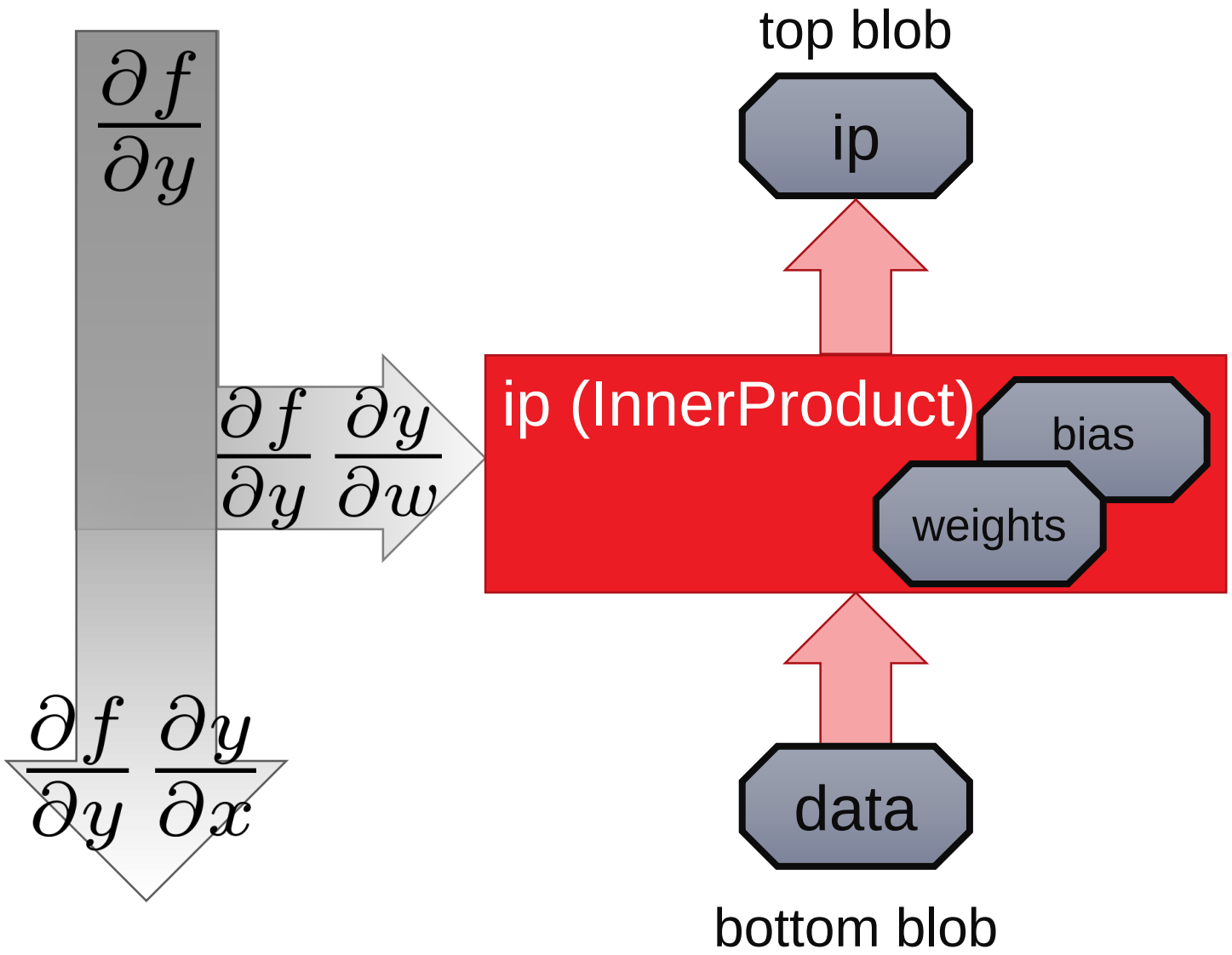
Layer Forward

- The forward pass goes from bottom to top
- During forward pass Caffe composes the computation of each layer to compute the “function” represented by the model.



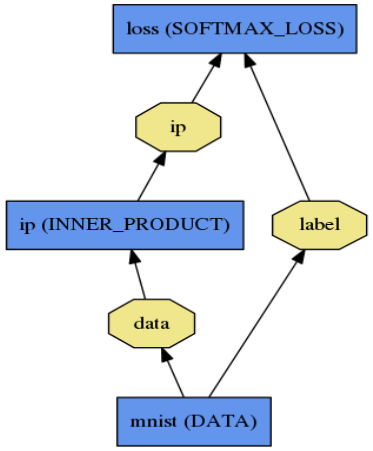
Layer Backward

- The backward pass performs back-propagation
- Given the gradient w.r.t. the top output Caffe computes the gradient w.r.t. to the input and sends to the bottom.
- A layer with parameters computes the gradient w.r.t. to its parameters and stores it internally.

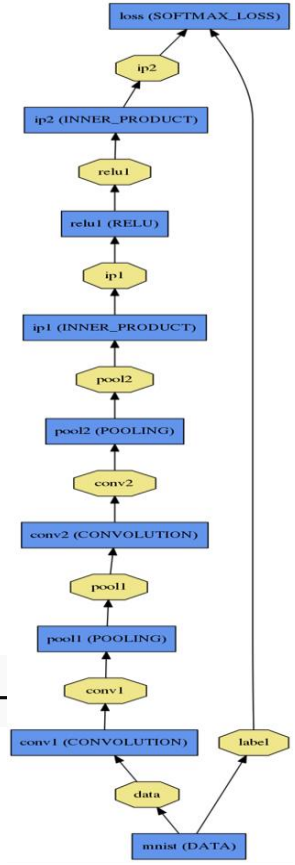


Caffe features: Net

- A network is a set of layers and their connections
- Most of the time linear graph
- Could be any directed acyclic graph (DAG).
- end-to-end machine learning: needs to start from data and ends in loss.



LogReg ↑

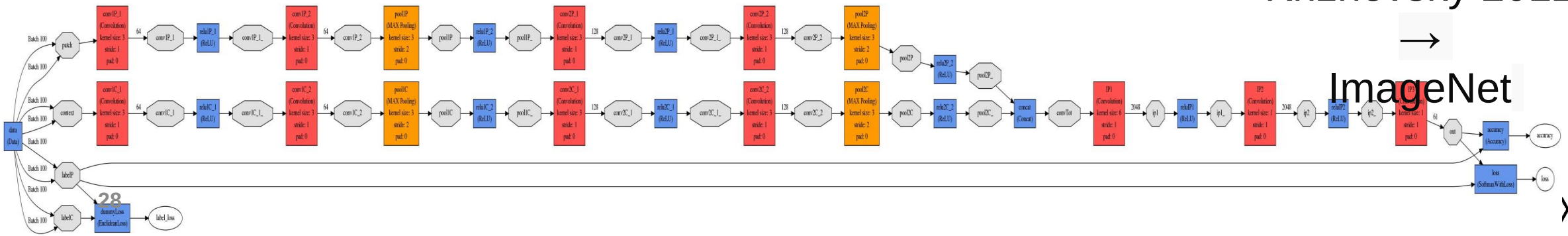


LeNet –

Krizhevsky 2012

→

ImageNet



Layers Overview

- **Data Layers**

  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. This layer has common input preprocessing (mean subtraction, scaling, random cropping, and mirroring)

- **Common Layers**

  Various commonly used layers, such as: Inner Product, Reshape, Concatenation, Softmax, …

- **Vision Layers**

  Vision layers usually take images as input and produce other images as output.

- **Neuron Layers**

  Neuron layers are element-wise operators, taking one bottom blob and producing one top blob of the same size.

- **Loss Layers**

  Loss drives learning by comparing an output to a target and assigning cost to minimize. The loss is computed by the forward pass.

Data layer formats

- **Caffe supports leveldb, lmdb, HDF5 and images inputs.**

- **HDF5**
  - very flexible and easy to use.
  - Problem – loads all the data into memory at once (problematic on large datasets).
- **Leveldb & LMDB**
  - works sequentially
  - less flexible (Caffe-wise)
  - much faster
- **Images**
  - takes a text file with image paths and labels (imagenet).

Getting data in: Data Augmentation

- **Happens on-the-fly!**
  - Random crops
  - Random horizontal flips
  - Subtract mean image
- **See TransformationParameter proto at:**
  - https://github.com/BVLC/caffe/blob/85bb397acfd383a676c125c75d877642d6b39ff6/src/caffe/proto/caffe.proto#L338
- **DataLayer, ImageDataLayer, WindowDataLayer**
- **NOT HDF5Layer**

Data Layer – leveldb & LMDB

```
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    crop_Size: 227
    mirror: true
  }
  data_param {
    source:
"examples/mnist/mnist_train_lmdb"
    batch_size: 64
    backend: LMDB
  }
}
```

- Name – for reusing net params (finetunning)
- Bottom – on every layer except data

- Top – for leveldb&LMBD always 2 top, data blob and label blob. Label is integer and of size Nx1x1x1.

- Phases – select when to use layer, default == both.
- For TEST phase define another layer with the same name

- Transform_param – do simple preprocessing: i.e. crops an image (at random during training and at the center image during testing)

- Data_param – tell caffe where (and what type) the data is.
- Batch_size – how many examples per batch. Small batch_size is faster, but more oscillatory.
- Backend – leveldb / LMDB

Prototxt: Layer Detail

Learning rates (weight + bias)
Set these to 0 to freeze a layer

Convolution-specific parameters

Parameter Initialization

```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  # learning rate and decay multipliers for the filters
  param { lr_mult: 1 decay_mult: 1 }
  # learning rate and decay multipliers for the biases
  param { lr_mult: 2 decay_mult: 0 }
  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
    }
  }
}
```

Example from ./models/bvlc_reference_caffenet/train_val.prototxt

Common Layer - Inner product layer

```
layer {
  name: "fc8"
  type: "InnerProduct"
  # learning rate and decay multipliers
  # for the weights and biases
  param { lr_mult: 1 decay_mult: 1 }
  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
    }
  }
  bottom: "fc7"
  top: "fc8"
}
```

- Linear function $f : \mathbb{R}^{in} \rightarrow \mathbb{R}^{out}$, $f(x) = w^T x + b$
- In = $C \cdot H \cdot W$, Bottom blob is of size (N,C,H,W)
- Out is a layer parameter (num_output). Top blob is (N,out,1,1).
- Number of parameters: $(C \cdot H \cdot W + 1) \cdot out$
- Param allows you to change specific layer learning rate, and separates weights and biases.
- During Net Finetunning: fixed layer – lr_mult: 0
- Can run on a single axis, see documentation.

Convolutional layer

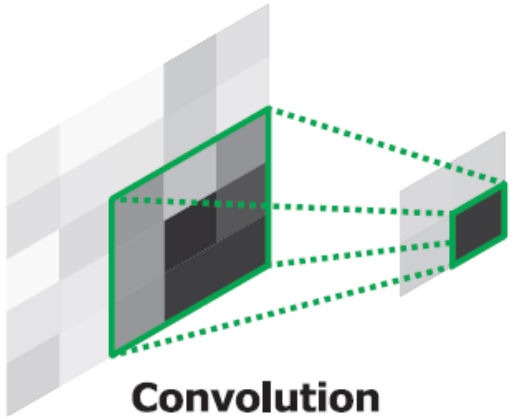
```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param { lr_mult: 1 }
  param { lr_mult: 2 }
  convolution_param {
    num_output: 20
    kernel_size: 5
    pad: 2
    stride: 1
    weight_filler {
      type: "xavier" }
    bias_filler {
      type: "constant" }
  }
}
```

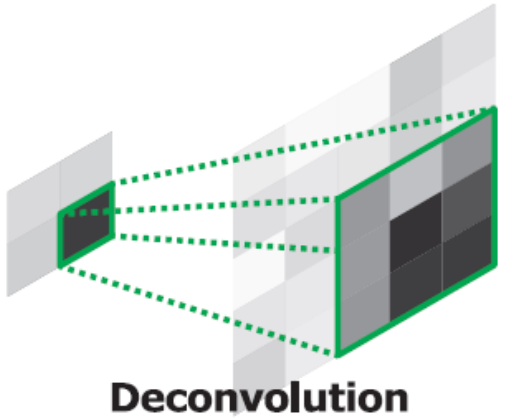
- Kernel_size – doesn't have to be symmetric.
- pad – specifies the number of pixels to (implicitly) add to each side of the input
- stride – step size in pixels between each filter application, reduce output size by a factor.
- weight_filler – random weight initialization. Break symmetry. "Xavier" picks std according to blob size.
- Performing a convolution with kernel size (C,H,W) is equivalent to performing inner product.

Deconvolution layer (Convolution Transpose)

```
layer {
  name: "upscore2"
  type:
"Deconvolution"
  bottom: "score59"
  top: "upscore2"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  convolution_param {
    num_output: 60
    bias_term: false
    kernel_size: 4
    stride: 2
  }
}
```

- Multiplies each input value by a filter elementwise, and sums over the resulting output windows. Resulting in convolution-like operations with multiple learned filters
- Reuses ConvolutionParameter for its parameters, but in the opposite sense as in ConvolutionLayer (so padding is removed from the output rather than added to the input, and stride results in upsampling rather than downsampling).

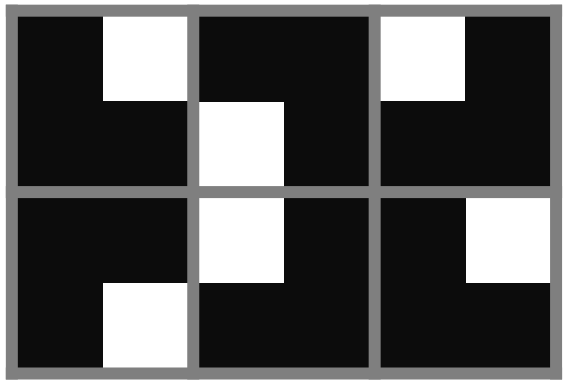




Pooling layer

```
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}
```

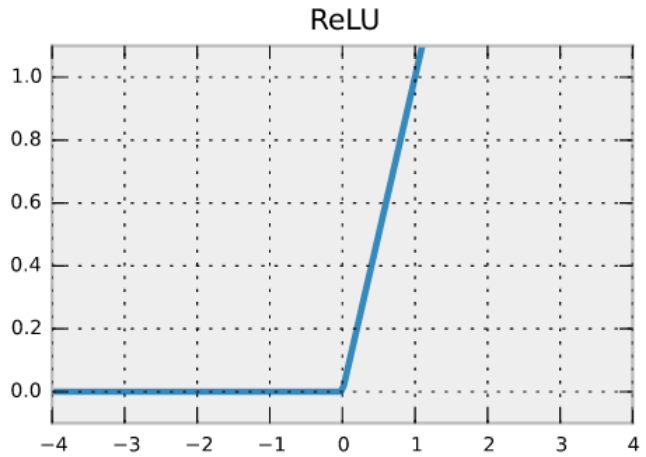
- Like convolution, just uses a fixed function MAX/AVE
- Use stride to reduce dimensionality.
- Allows for small translation invariance (MAX).

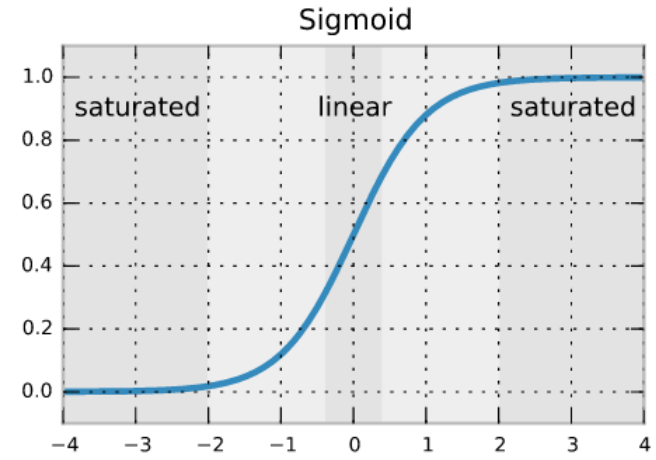


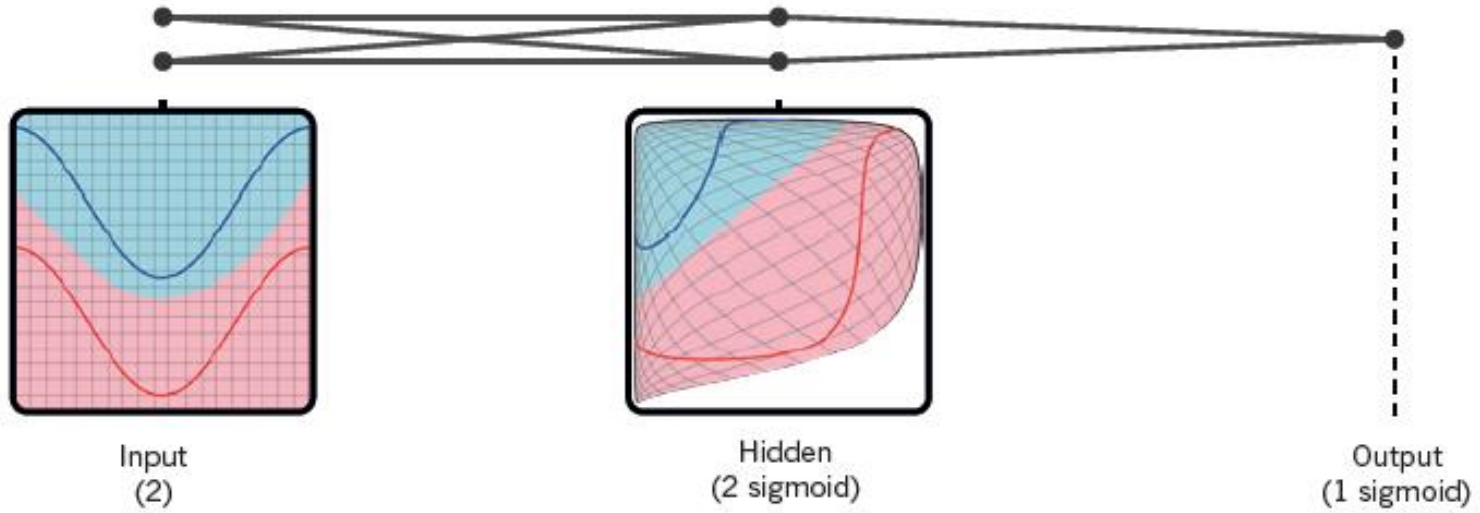
Neuron layer

```
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "pool1"
  top: "pool1"
}
```

- For each value x in the blob, return f(x).
- Size of input == Size of output
- Computation done in place.
- ReLU, sigmoid, tanh…







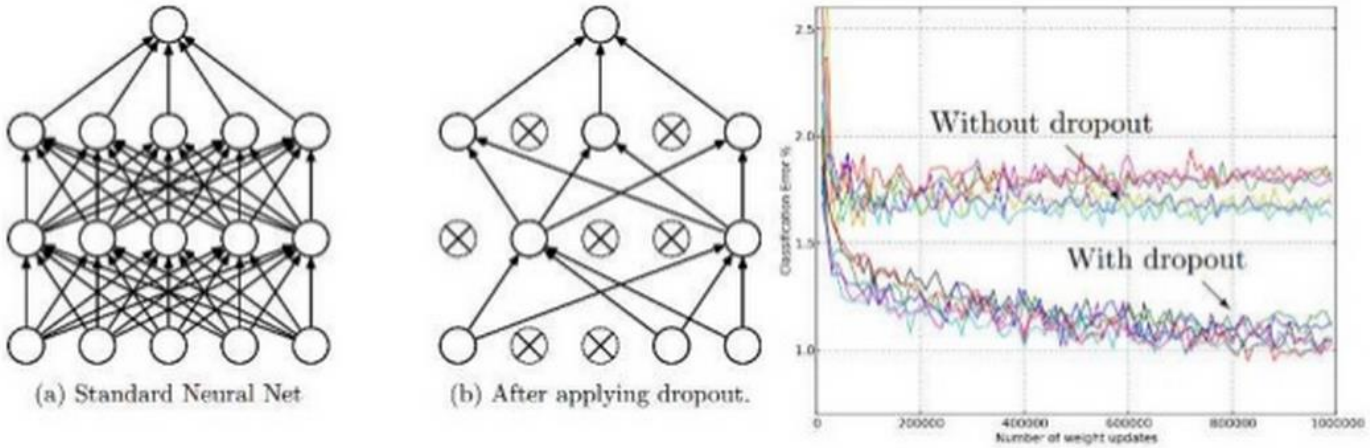
Neuron layer - Dropout

```
layer {
  name: "drop6"
  type: "Dropout"
  bottom: "fc6"
  top: "fc6"
  dropout_param {
    dropout_ratio: 0.5
  }
}
```

- During training only, sets a random portion of x to 0, adjusting the rest of the vector accordingly.

$$y_{\text{train}} = \begin{cases} \frac{x}{1-p} & \text{if } u > p \\ 0 & \text{otherwise} \end{cases} \qquad \text{Where, } u \sim U(0,1)$$

$$E(y_{\text{train}}) = p \cdot 0 + (1-p)\frac{E(x)}{1-p} = E(x)$$



Loss Layer

- 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 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.
- 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.

Loss layers – SoftmaxWithLoss

$$\hat{p}_{nk} = \frac{\exp(x_{nk})}{\sum_{k'=1}^{K} \exp(x_{nk'})}$$

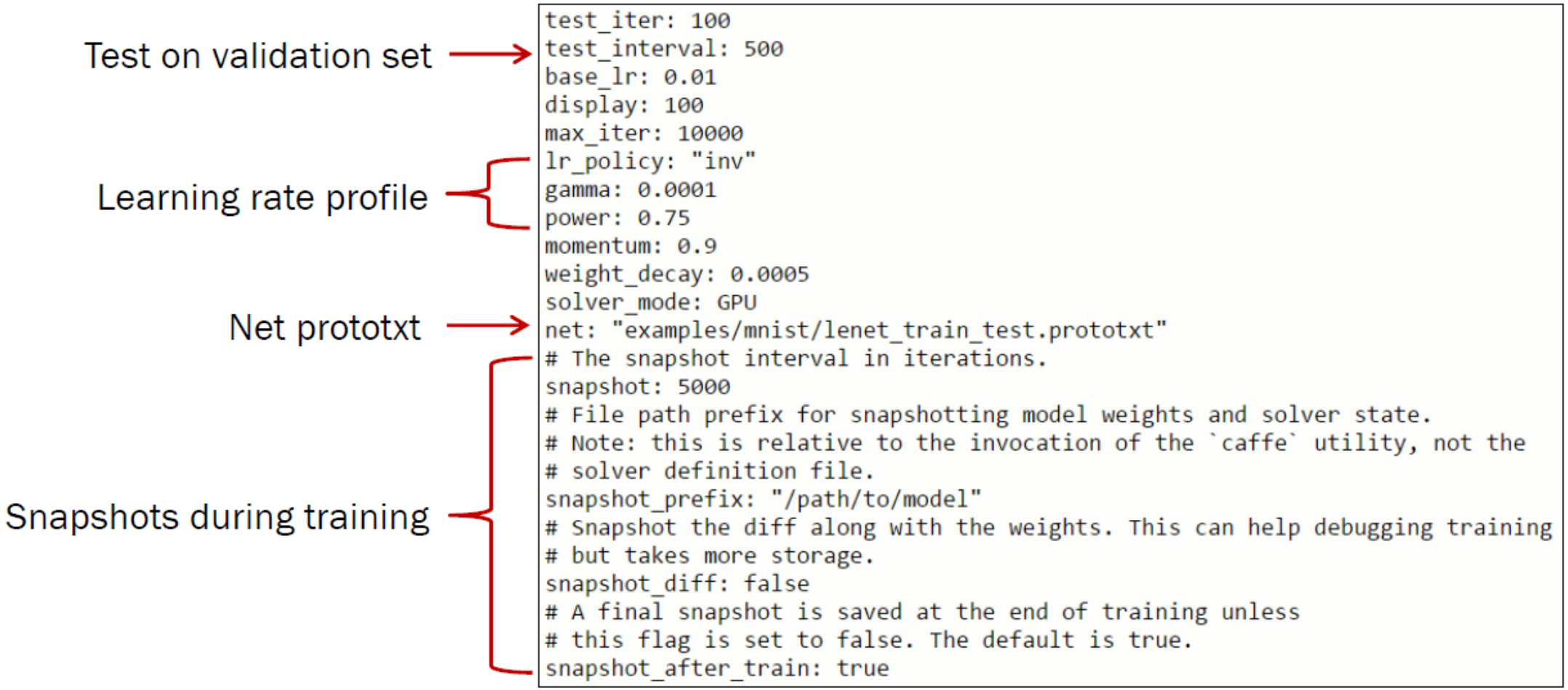
$$E = \frac{-1}{N} \sum_{n=1}^{N} \log(\hat{p}_{n,l_n}) \qquad l_n \in [0, 1, 2, ..., K-1]$$

$$E = \frac{-1}{N} \sum_{n=1}^{N} \log \frac{\exp(x_{nl_n})}{\sum_{k'=1}^{K} \exp(x_{nk'})} = \frac{-1}{N} \sum_{n=1}^{N} (x_{nl_n} - \log \sum_{k'=1}^{K} \exp(x_{nk'}))$$

$$\frac{\partial E}{\partial x_{\tilde{n}k}} = \frac{-1}{N}(-\frac{\exp(x_{\tilde{n}k})}{\sum_{k'=1}^{K} \exp(x_{nk'})}) = \frac{1}{N}\hat{p}_{\tilde{n}k} \qquad \text{when } k \neq l_n$$

$$\frac{\partial E}{\partial x_{\tilde{n}k}} = \frac{-1}{N}(1 - \frac{\exp(x_{\tilde{n}k})}{\sum_{k'=1}^{K} \exp(x_{nk'})}) = \frac{1}{N}(\hat{p}_{\tilde{n}k} - 1) \qquad \text{when } k = l_n$$

Prototxt: Define Solver



Solver prototxt – network run parameters

```
net:
"models/bvlc_alexnet/train_val.prototxt"
test_iter: 1000
test_interval: 1000
base_lr: 0.01
lr_policy: "step"
gamma: 0.1
stepsize: 100000
display: 20
max_iter: 450000
momentum: 0.9
weight_decay: 0.0005
snapshot: 10000
snapshot_prefix:
"models/bvlc_alexnet/caffe_alexnet_train"
solver_mode: GPU
```

- net: Proto filename for the train net, possibly combined with test net
- display: the number of iterations between displaying info
- max_iter : The maximum number of iterations
- Solver_mode: the mode solver will use: CPU or GPU

Solver prototxt – test set parameters

```
net:
"models/bvlc_alexnet/train_val.prototxt"
test_iter: 1000
test_interval: 1000
base_lr: 0.01
lr_policy: "step"
gamma: 0.1
stepsize: 100000
display: 20
max_iter: 450000
momentum: 0.9
weight_decay: 0.0005
snapshot: 10000
snapshot_prefix:
"models/bvlc_alexnet/caffe_alexnet_train"
solver_mode: GPU
```

- test_iter: The number of iterations for each test net
- test_interval: The number of iterations between two testing phases

Learning rate policies

```
base_lr: 0.01
lr_policy:
“fixed"
```

```
base_lr: 0.01
lr_policy:
"step"
gamma: 0.1
stepsize:
100000
```

```
base_lr: 0.01
lr_policy:
"inv"
gamma: 0.0001
power: 0.75
```

- Fixed: always base_lr
- Step: start at base_lr and after each stepsize iterations reduce learning rate by gamma.

$$LR(t) = base\_lr \cdot \gamma^{\lfloor t/stepsize \rfloor}$$

- Inv: start at base_lr and after each iteration reduce learning rate

$$LR(t) = base\_lr \cdot \frac{1}{1+\gamma t}^{power}$$

- If you get NaN/Inf loss values try to reduce base_lr

Solver prototxt – momentum parameter

```
net: "models/bvlc_alexnet/train_val.prototxt"
test_iter: 1000
test_interval: 1000
base_lr: 0.01
lr_policy: "step"
gamma: 0.1
stepsize: 100000
display: 20
max_iter: 450000
momentum: 0.9
weight_decay: 0.0005
snapshot: 10000
snapshot_prefix:
"models/bvlc_alexnet/caffe_alexnet_train"
solver_mode: GPU
```

Weight Decay

- To avoid over-fitting, it is possible to regularize the cost function.
- Here we use L2 regularization, by changing the cost function to:

$$\widetilde{L}(\mathbf{w}) = L(\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^2$$

- In practice this penalizes large weights and effectively limits the freedom in the model.
- The regularization parameter λ determines how you trade off the original loss L with the large weights penalization.

- Applying gradient descent to this new cost function we obtain:

$$w_i \leftarrow w_i - \eta\frac{\partial L}{\partial w_i} - \eta\lambda w_i$$

- The new term $-\eta\lambda w_i$ coming from the regularization causes the weight to decay in proportion to its size.

Solver prototxt – weight_decay parameter

```
net: "models/bvlc_alexnet/train_val.prototxt"
test_iter: 1000
test_interval: 1000
base_lr: 0.01
lr_policy: "step"
gamma: 0.1
stepsize: 100000
display: 20
max_iter: 450000
momentum: 0.9
weight_decay: 0.0005
snapshot: 10000
snapshot_prefix:
"models/bvlc_alexnet/caffe_alexnet_train"
solver_mode: GPU
```

Solver prototxt - snapshot

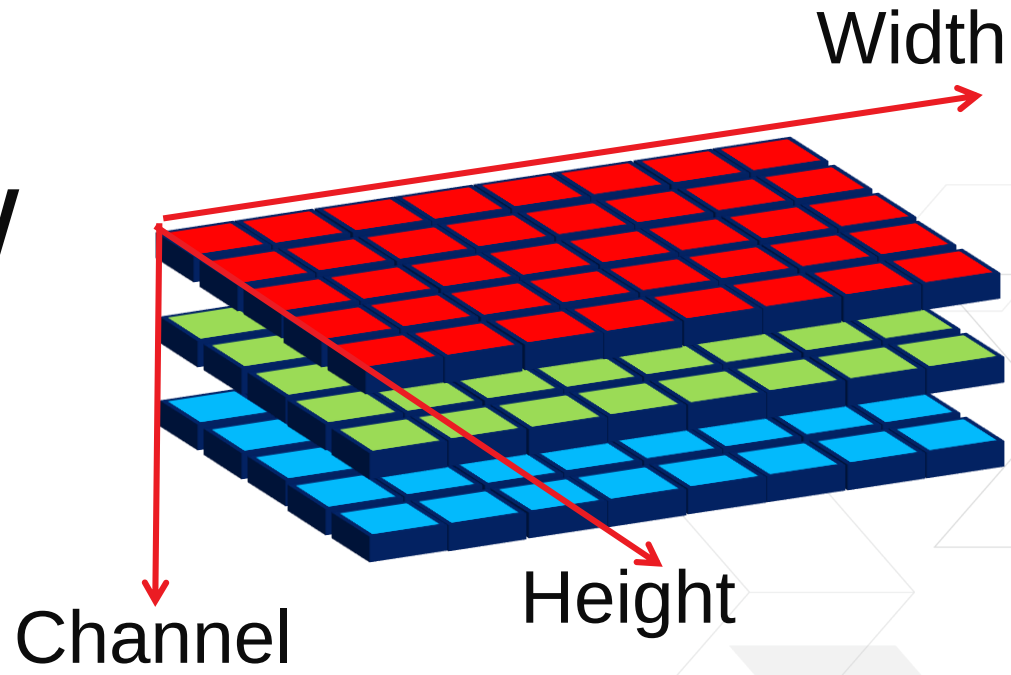
```
net:
"models/bvlc_alexnet/train_val.prototxt"
test_iter: 1000
test_interval: 1000
base_lr: 0.01
lr_policy: "step"
gamma: 0.1
stepsize: 100000
display: 20
max_iter: 450000
momentum: 0.9
weight_decay: 0.0005
snapshot: 10000
snapshot_prefix:
"models/bvlc_alexnet/caffe_alexnet_train"
solver_mode: GPU
```

- The snapshot interval in iterations.
  snapshot: 10000
- 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.
- Can use full path:
  snapshot_prefix: "/path/to/model“

Important things you cannot miss

Caffe - Storing Data In Memory

C x H x W



Note that in Keras/Tensorflow data are stored as HxWxC

Getting data in

- **type: “Data”**
  - Reads images and labels from LMDB file
  - Only good for 1-of-k **Classification**
  - See next page
- **type: “ImageData”**
  - Get images and labels directly from image files
  - No LMDB but probably slower than DataLayer
- **type: “WindowData”**
  - Read windows from image files and class labels
  - Made for **Detection**
- **type: “HDF5_DATA”**
  - Reads arbitrary data from HDF5 files, but it does not allow transform_param
  - Easy to read/write in Python using h5py.
  - **Good for any ML task**
  - Can only store floating point (32, 64) data, no uint8, so image data files are huge

BatchNorm layer 1/3

Normalizes the input to have 0-mean and/or unit (1) variance across the batch.

This layer computes **Batch** Normalization as described in [1]. For each channel in the data (i.e. axis 1), it subtracts the mean and divides by the variance, where both statistics are computed across both spatial dimensions and across the different examples in the batch.

By default, during training time, the network is computing global mean/variance statistics via a running average, which is then used at test time to allow deterministic outputs for each input. You can manually toggle whether the network is accumulating or using the statistics via the use_global_stats option. For reference, these statistics are kept in the layer's three blobs: (0) mean, (1) variance, and (2) moving average factor.

Note that the original paper also included a per-channel learned bias and scaling factor. To implement this in **Caffe**, define a `ScaleLayer` configured with `bias_term: true` after each `BatchNormLayer` to handle both the bias and scaling factor.

[1] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." arXiv preprint arXiv:1502.03167 (2015).

Protected Attributes

Blob< Dtype >	**mean_**
Blob< Dtype >	**variance_**
Blob< Dtype >	**temp_**
Blob< Dtype >	**x_norm_**
bool	**use_global_stats_**
Dtype	**moving_average_fraction_**
int	**channels_**
Dtype	**eps_**
Blob< Dtype >	**batch_sum_multiplier_**
Blob< Dtype >	**num_by_chans_**
Blob< Dtype >	**spatial_sum_multiplier_**

http://caffe.berkeleyvision.org/doxygen/classcaffe_1_1BatchNormLayer.html#details

BatchNorm layer 2/3

- The three param{lr_mult: 0} (mean, variance, and moving average factor) are not necessary since the merge of #4704 to BVLC:master. There was an issue where those blobs would be affected by the weight updates.
- According to the documentation (see page before) there is no longer any need to create separate BatchNormalization layers for test/train phases.
- Note that Caffe has a **InPlace concept** where certain layers (among these relu, batchnorm and scale) can be performed in place on an earlier layer.

```
layer {
  name: "conv1/bn"
  type: "BatchNorm"
  bottom: "conv1"
  top: "conv1"
  batch_norm_param {
    moving_average_fraction: 0.9
  }
}

layer {
  name: "conv1/scale"
  type: "Scale"
  bottom: "conv1"
  top: "conv1"
  scale_param {
    bias_term: true
  }
}
```

BatchNorm layer 3/3

- However both CHaiDNN and DEEPHI support also the other syntax, as shown on the right

- In the DEPLOY of the NN put

```
batch_norm_param {
  use_global_stats: true
}
```

```
layer {
  name: "bn1"
  type: "BatchNorm"
  bottom: "conv1"
  top: "bn1"
  param {    lr_mult: 0  }
  param {    lr_mult: 0  }
  param {    lr_mult: 0  }
  batch_norm_param {
    use_global_stats: false
  }
}
layer {
  name: "scale1"
  type: "Scale"
  bottom: "bn1"
  top: "scale1"
  scale_param { bias_term: true }
}
```

Loss layers – SoftmaxWithLoss (during training)

```
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "fc"
  bottom: "label"
  top: "loss"
  loss_weight: 1
  loss_param {
    ignore_label: 255
  }
}
```

- Used for K-class Classification
- Predictions Input blob of size (N,K,1,1)
- Labels Input blob of size (N,1,1,1)
- Output size (1,1,1,1)
- ignore_label (optional) Specify a label value that should be ignored when computing the loss.
- First preforms softmax then computes the multinomial logistic loss (-log likelihood)

$$\hat{p}_{nk} = \frac{\exp(x_{nk})}{\sum_{k'=1}^{K} \exp(x_{nk'})}$$

$$E = \frac{-1}{N} \sum_{n=1}^{N} \log(\hat{p}_{n,l_n}) \qquad l_n \in [0, 1, 2, ..., K-1]$$

Loss layers – Softmax (during prediction)

```
layer {
  name: “prob"
  type: "Softmax"
  bottom: “fc"

  top: “prob"
  loss_weight: 1
  loss_param {
    ignore_label: 255
  }
}
```

- When you produce a description prototxt file for CNN deploy you have to change this layer:
  - replace “SofmaxWithLoss” with “Softmax”
  - replace “loss” with “prob”
  - remove “label”

Initialization parameters are extremely important

- weight_filter initialization
  - gaussian: samples weights from N(0,std)
  - xavier: samples weights from uniform distribution U(-a,a), where a=sqrt(3/fan_in), where fan_in are the number of incoming neurons
  - MSRAFiller: samples weights from normal distribution N(0,a), where a=sqrt(2/fan_in)
  - for ReLU neurons, MSRAFiller is usually better than xavier which is usually better than gaussian;
  - note for MSRAFiller and xavier, there is no need to manually specify std
- base_lr: initial learning rate (default:.01, change to a smaller number if getting NAN loss in training)
- lr_mult for the bias is usually set to 2x the lr_mult for the non-bias weights
- http://andyljones.tumblr.com/post/110998971763/an-explanation-of-xavier-initialization

Caffe Features: Solver

- Call Forward and Backward and updates the net parameters
- Periodically evaluates mode on the test network(s)
- Snapshots model and solver state
- Solvers available:
  - `type: "SGD"`
  - `type: "AdaDelta"`
  - `type: "AdaGrad"`
  - `type: "Adam"`
  - `type: "Nesterov"`
  - `type: "RMSprop"`
- See: http://caffe.berkeleyvision.org/tutorial/solver.html

Some terminology to set the solver

- **Given the following definitions:**
  - **epoch**: Forward and Backward pass of all training examples (not used in Caffe).
  - **max_epochs**: the amount of times you want to cover completely your training set size
  - **batch**: how many images in one pass
  - **iterations**: how many batches
- **batch_size**
  - defines how many samples are fed through the network at once, followed by a weight-update process.
- **max_iter**
  - is the number of batches (also called "iterations" in Caffe) to feed to the network for training. These are not necessarily different from each other: if your data source is at the end it will just rewind to the beginning to provide more samples.
    - Example: with 60K images batch_size=64 and max_iter=10K there will be 10k*64=640K images of learning. 640K/60k=10.6 epochs
  - **max_iter = max_epochs * (training_set_size / training_batch_size)**
- **test_interval & test_iter**
  - **test_interval** is the number of iterations, after which a test instance is run to process **test_iter** batches. An average accuracy and/or loss value is then computed by the solver. After that the training continues.
  - To be more clear: after every test_interval iterations, test_iter x batch_size images are fetched for testing

Caffe Solver parameters

lr_policy

This parameter indicates how the learning rate should change over time. This value is a quoted string.

Options include:

1. "step" - drop the learning rate in step sizes indicated by the **gamma** parameter.
2. "multistep" - drop the learning rate in step size indicated by the **gamma** at each specified **stepvalue**.
3. "fixed" - the learning rate does not change.
4. "exp" - **gamma**^*iter*
5. "poly" - the effective learning rate follows a polynomial decay, to be zero by the max_iter. **base_lr** * (1 - *iter*/**max_iter**) ^ (**power**)
6. "sigmoid" - the effective learning rate follows a sigmod decay. **base_lr** * (1/(1 + exp(-**gamma** * (*iter* - **stepsize**))))

where **base_lr**, **max_iter**, **gamma**, **step**, **stepvalue** and **power** are defined in the solver parameter protocol buffer, and *iter* is the current iteration.

gamma

This parameter indicates how much the learning rate should change every time we reach the next "step." The value is a real number, and can be thought of as multiplying the current learning rate by said number to gain a new learning rate.

stepsize

This parameter indicates how often (at some iteration count) that we should move onto the next "step" of training. This value is a positive integer.

stepvalue

This parameter indicates one of potentially many iteration counts that we should move onto the next "step" of training. This value is a positive integer. There are often more than one of these

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 $\alpha''' = 10^{-5}$.

Is Caffe behaviour deterministic?

- **According to these sites, Caffe behaviour is not totally deterministic**
  - https://github.com/BVLC/caffe/issues/3168
  - https://github.com/tensorflow/tensorflow/issues/2732

- **Even within the same environment, different runs can generate different results: I have seen up to 2% of average prevision accuracy difference.**
  - If you see the same, this is OK …
  - … but if you see more than 2%, I guess you have some problems

- **Try to have in your solver.prototxt files the following command (which is better than nothing):**
  - `random_seed: 1201`