Caffe Tutorial

Maximally accurate	Maximally specific
espresso	2.23192
coffee	2.19914
beverage	1.93214
liquid	1.89367
fluid	1.85519

Brewing Deep Networks With Caffe

Rohit Girdhar

Many slides from Xinlei Chen (16-824 tutorial), Caffe CVPR'15 tutorial

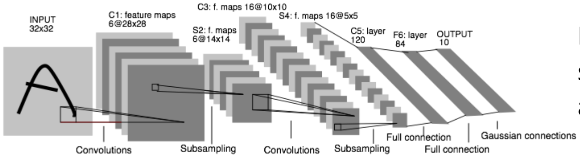
Overview

- Motivation and comparisons
- Training/Finetuning a simple model
- Deep dive into Caffe!

! this->tutorial

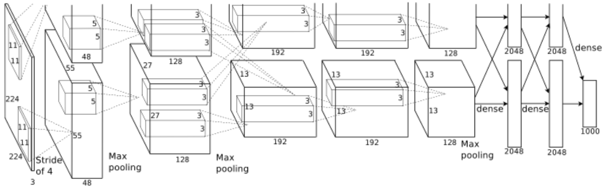
- What is Deep Learning?
- Why Deep Learning?
  - The Unreasonable Effectiveness of Deep Features
- History of Deep Learning.

CNNs 1989



LeNet: a layered model composed of convolution and subsampling operations followed by a holistic representation and ultimately a classifier for handwritten digits. [LeNet]

CNNs 2012



AlexNet: a layered model composed of convolution, subsampling, and further operations followed by a holistic representation and all-in-all a landmark classifier on ILSVRC12. [AlexNet]
+ data, + gpu, + non-saturating nonlinearity, + regularization

Other Frameworks

- Torch7
  - NYU
  - scientific computing framework in Lua
  - supported by Facebook
- TensorFlow
  - Google
  - Good for deploying
- Theano/Pylearn2
  - U. Montreal
  - scientific computing framework in Python
  - symbolic computation and automatic differentiation
- Cuda-Convnet2
  - Alex Krizhevsky
  - Very fast on state-of-the-art GPUs with Multi-GPU parallelism
  - C++ / CUDA library
- MatConvNet
  - Oxford U.
  - Deep Learning in MATLAB
- CXXNet
- Marvin

Framework Comparison

- More alike than different
  - All express deep models
  - All are open-source (contributions differ)
  - Most include scripting for hacking and prototyping

- No strict winners – experiment and choose the framework that best fits your work

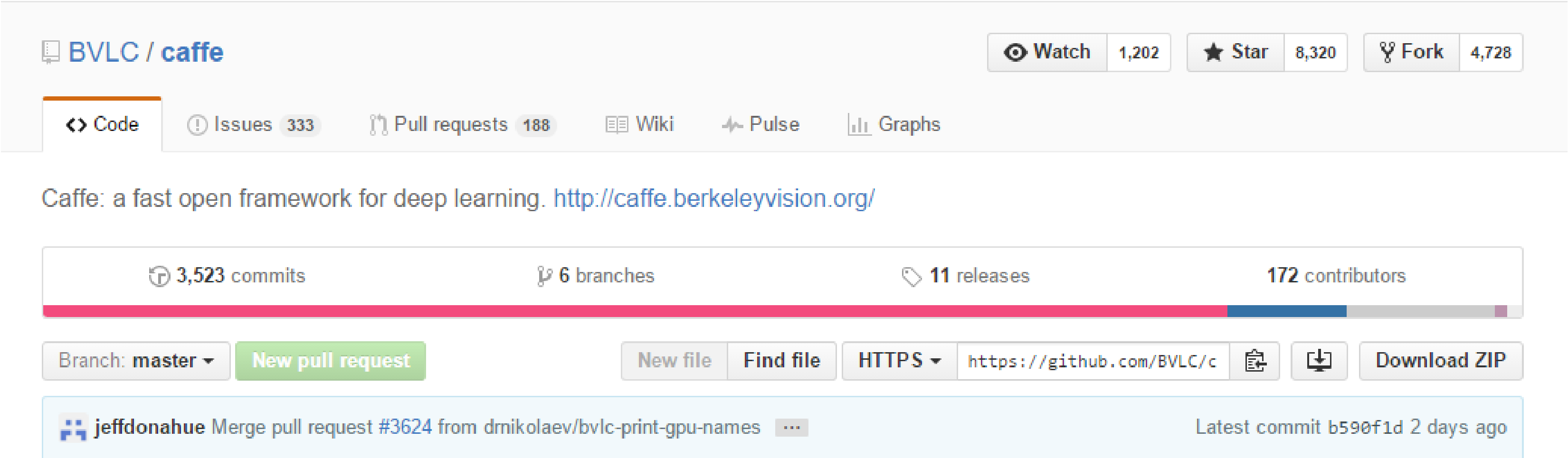
Torch vs Caffe vs TensorFlow?

- Torch has more functionality built-in (more variety of layers etc.) and is in general more flexible
- However, more flexibility => writing more code! If you have a million images and want to train a mostly standard architecture, go with caffe!
- TensorFlow is best at deployment! Even works on mobile devices.

What is Caffe?

Open framework, models, and worked examples for deep learning

- 600+ citations, 150+ contributors, 7,000+ stars, 4,700+ forks, >1 pull request / day average
- focus has been vision, but branching out:
  sequences, reinforcement learning, speech + text



So what is Caffe?

- Pure C++ / CUDA architecture for deep learning
    - command line, Python, MATLAB interfaces
- Fast, well-tested code
- Tools, reference models, demos, and recipes
- Seamless switch between CPU and GPU
    - `Caffe::set_mode(Caffe::GPU);`

Prototype

Training

Deployment

All with essentially the same code!

Brewing by the Numbers...

- Speed with Krizhevsky's 2012 model:
  – 2 ms / image on K40 GPU
  – <1 ms inference with Caffe + cuDNN v2 on Titan X
  – 72 million images / day with batched IO
  – 8-core CPU: ~20 ms/image
- 9k lines of C++ code (20k with tests)
- https://github.com/soumith/convnet-benchmarks: A pretty reliable benchmark

C++ 84.2% Python 10.5% Cuda 3.9% Other 1.4%

Why Caffe? In one sip…

Expression: models + optimizations are plaintext schemas, not code.

Speed: for state-of-the-art models and massive data.

Modularity: to extend to new tasks and settings.

Openness: common code and reference models for reproducibility.

Community: joint discussion and development through BSD-2 licensing.

Caffe Tutorial

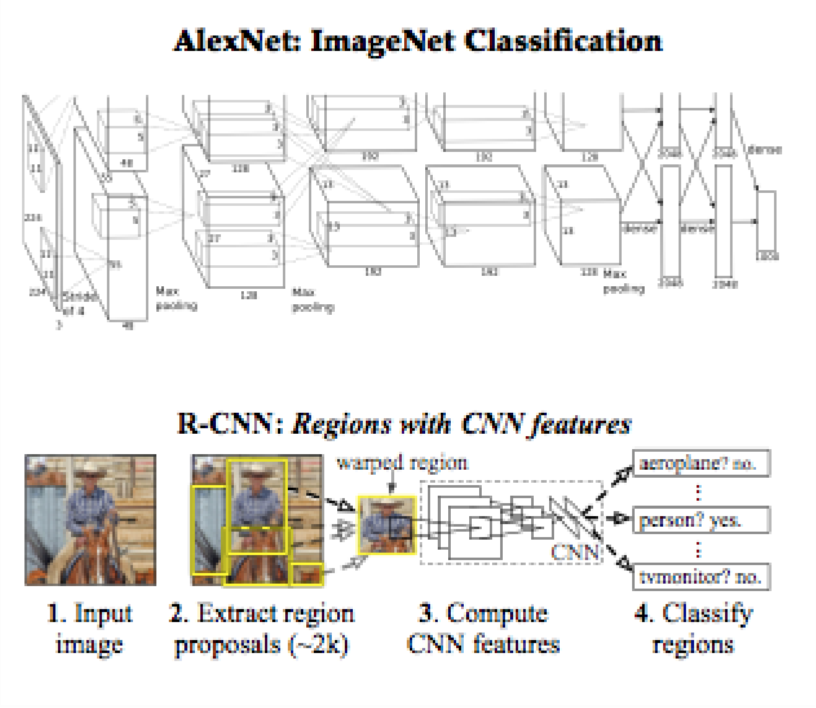
http:/caffe.berkeleyvision.org/tutorial/

- Nets, Layers, and Blobs: the anatomy of a Caffe model.
- Forward / Backward: the essential computations of layered compositional models.
- Loss: the task to be learned is defined by the loss.
- Solver: the solver coordinates model optimization.
- Layer Catalogue: the layer is the fundamental unit of modeling and computation – Caffe's catalogue includes layers for state-of-the-art models.
- Interfaces: command line, Python, and MATLAB Caffe.
- Data: how to caffeinate data for model input.

For a closer look at a few details:

- Caffeinated Convolution: how Caffe computes convolutions.

Reference 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.

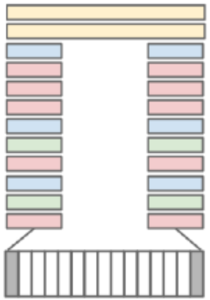
Open Model Collection

The Caffe Model Zoo
- open collection of deep models to share innovation
    - VGG ILSVRC14 + Devil models in the zoo
    - Network-in-Network / CCCP model in the zoo
        - MIT Places scene recognition model in the zoo
- help disseminate and reproduce research
- bundled tools for loading and publishing models

Share Your Models! with your citation + license of course

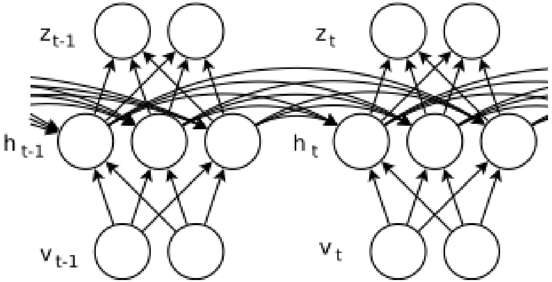
Architectures

DAGs
multi-input
multi-task



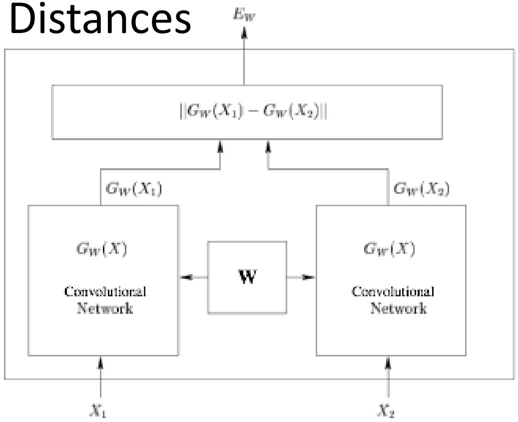
[Karpathy14]

Weight Sharing
Recurrent (RNNs)
Sequences



[Sutskever13]

Siamese Nets
Distances



[Chopra05]

Define your own model from our catalogue of layers types and start learning.

Installation Hints

- We have already compiled the latest version of caffe (as on 5 Feb’16) on LateDays!
- However, you might want to customize and compile your own caffe (esp. if you want to create new layers)

Installation

- http://caffe.berkeleyvision.org/installation.html
- **CUDA, OPENCV**
- **BLAS** (Basic Linear Algebra Subprograms): operations like matrix multiplication, matrix addition, both implementation for CPU(cBLAS) and GPU(cuBLAS). provided by MKL(INTEL), ATLAS, openBLAS, etc.
- **Boost**: a c++ library. > Use some of its math functions and shared_pointer.
- **glog,gflags** provide logging & command line utilities. > Essential for debugging.
- **leveldb, lmdb**: database io for your program. > Need to know this for preparing your own data.
- **protobuf**: an efficient and flexible way to define data structure. > Need to know this for defining new layers.

TRAINING AND FINE-TUNING

Training: Step 1

Create a `lenet_train.prototxt`

Data

```
layer {
  name: "data"
  type: "Data"
  top: "data"
  top: "label"
  transform_param {
    scale: 0.00392156862745
  }
  data_param {
    source: "examples/mnist/mnist_train_lmdb"
    batch_size: 64
    backend: LMDB
  }
}
```

Layers

```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  convolution_param {
    num_output: 20
    kernel_size: 5
    weight_filler {
      type: "xavier"
    }
  }
}
```

Loss

```
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "ip2"
  bottom: "label"
  top: "loss"
}
```

Training: Step 2

Create a `lenet_solver.prototxt`

```
train_net:          "lenet_train.prototxt"
base_lr:            0.01
momentum:           0.9
weight_decay:       0.0005
max_iter:           10000
snapshot_prefix:    "lenet_snapshot"
# … and some other options …
```

Training: Step 2

Some details on SGD parameters

$$V_{t+1} = \underbrace{\mu}_{\text{Momentum}} V_t - \underbrace{\alpha}_{\text{LR}} (\nabla L(W_t) + \underbrace{\lambda}_{\text{Decay}} W_t)$$

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

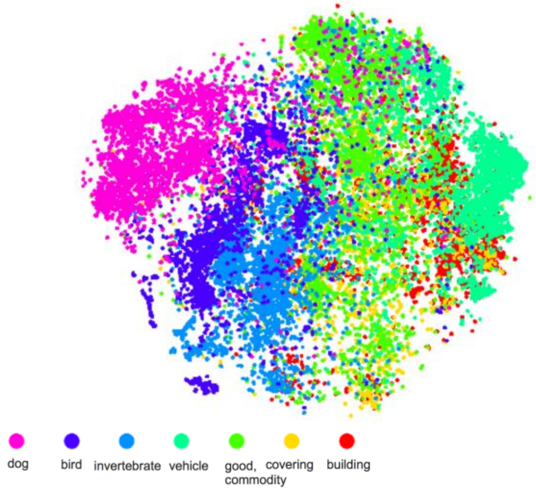
Training: Step 3

Train!

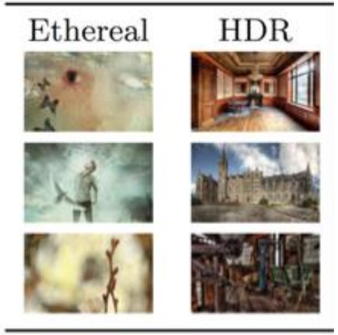
```
$ build/tools/caffe train \
    -solver lenet_solver.prototxt \
    -gpu 0
```

Fine-tuning Transferring learned weights to kick-start models

- Take a pre-trained model and fine-tune to new tasks [DeCAF] [Zeiler-Fergus] [OverFeat]



Your Task



Style Recognition

© kaggle.com

Dogs vs. Cats
top 10 in 10 minutes

From ImageNet to Style

- Simply change a few lines in the layer definition

```
layers {
  name: "data"
  type: DATA
  data_param {
    source: "ilsvrc12_train_leveldb"
    mean_file: "../../data/ilsvrc12"
    ...
  }
  ...
}
...
layers {
  name: "fc8"
  type: INNER_PRODUCT
  blobs_lr: 1
  blobs_lr: 2
  weight_decay: 1
  weight_decay: 0
  inner_product_param {
    num_output: 1000
    ...
  }
}
```

```
layers {
  name: "data"
  type: DATA
  data_param {
    source: "style_leveldb"
    mean_file: "../../data/ilsvrc12"
    ...
  }
  ...
}
...
layers {
  name: "fc8-style"
  type: INNER_PRODUCT
  blobs_lr: 1
  blobs_lr: 2
  weight_decay: 1
  weight_decay: 0
  inner_product_param {
    num_output: 20
    ...
  }
}
```

new name = new params

Input: A different source

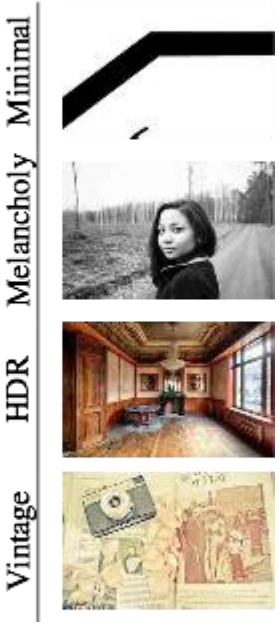
Last Layer: A different classifier

From ImageNet to Style

```
$ caffe train -solver models/finetune_flickr_style/solver.prototxt \
              -gpu 0 \
              -weights bvlc_reference_caffenet.caffemodel
```

Under the hood (loosely speaking):

```
net = new Caffe::Net(
    "style_solver.prototxt");
net.CopyTrainedNetFrom(
    pretrained_model);
solver.Solve(net);
```



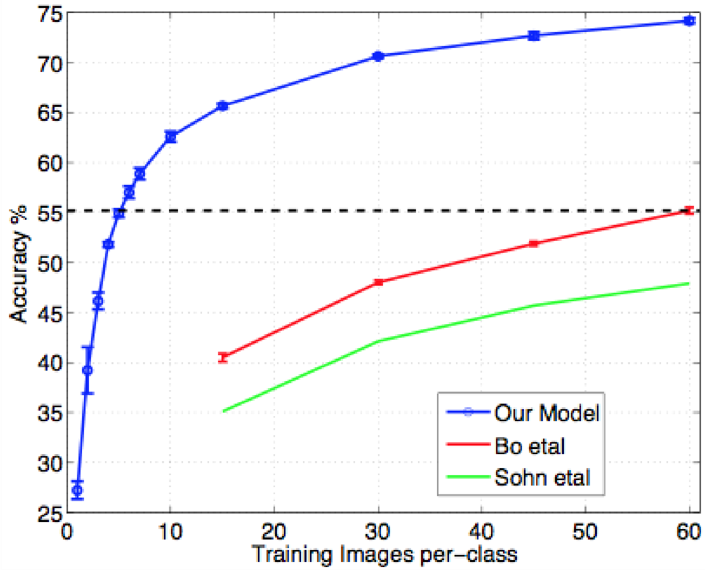
When to Fine-tune?

A good first step!

- More robust optimization – good initialization helps
- Needs less data
- Faster learning

State-of-the-art results in

- recognition
- detection
- segmentation



[Zeiler-Fergus]

Fine-tuning Tricks

Learn the last layer first

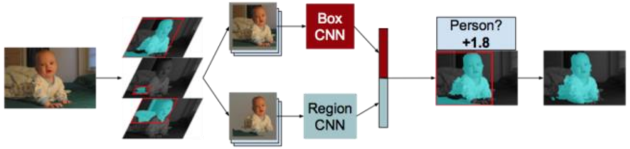
- Caffe layers have local learning rates: `blobs_lr`
- Freeze all but the last layer for fast optimization and avoiding early divergence.
- Stop if good enough, or keep fine-tuning

Reduce the learning rate

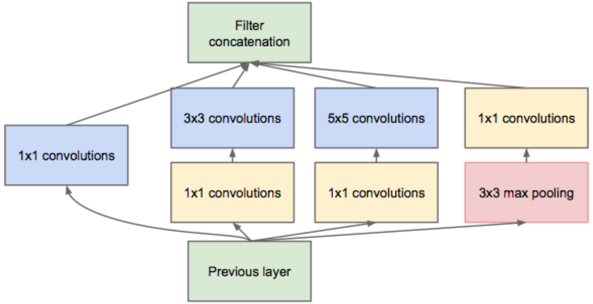
- Drop the solver learning rate by 10x, 100x
- Preserve the initialization from pre-training and avoid thrashing

DEEPER INTO CAFFE

DAG

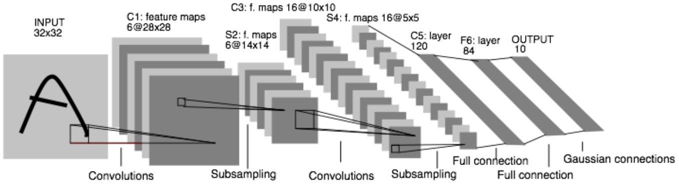


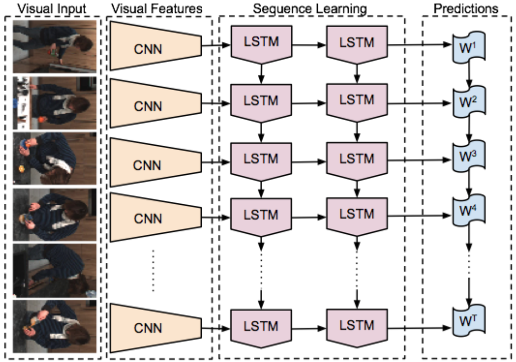
SDS two-stream net



GoogLeNet Inception Module

Many current deep models have linear structure





but Caffe nets can have any directed acyclic graph (DAG) structure.

Define `bottoms` and `tops` and Caffe will connect the net.

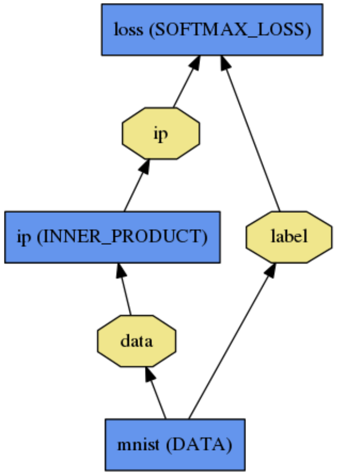
LRCN joint vision-sequence model

Net

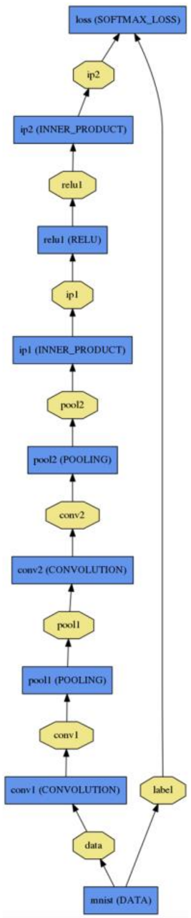
- A network is a set of layers connected as a DAG:

```
name: "dummy-net"
layers { name: "data" …}
layers { name: "conv" …}
layers { name: "pool" …}
     … more layers …
layers { name: "loss" …}
```

- Caffe creates and checks the net from the definition.
- Data and derivatives flow through the net as *blobs* – a an array interface

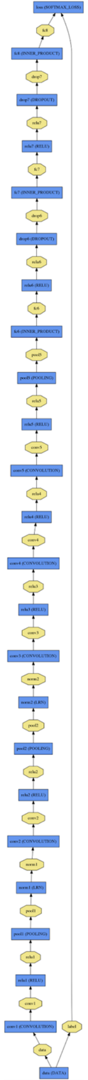


LogReg ↑



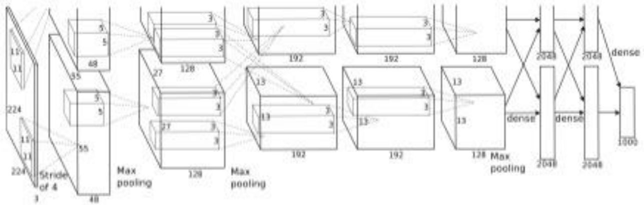
LeNet →

ImageNet, Krizhevsky 2012 →



Forward / Backward the essential Net computations

Forward: inference $f_W(x)$



"espresso" + loss

$\nabla f_W(x)$ Backward: learning

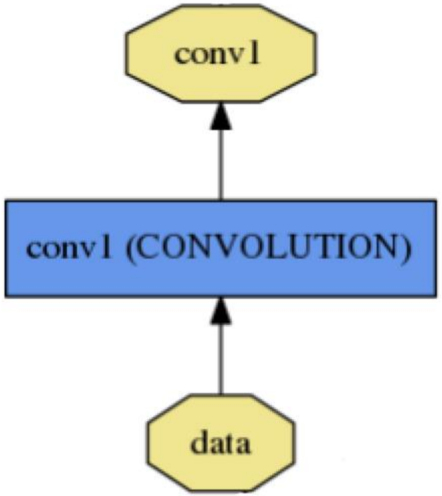
Caffe models are complete machine learning systems for inference and learning.
The computation follows from the model definition. Define the model and run.

Layer

```
name: "conv1"
type: "Convolution"
bottom: "data"
top: "conv1"
convolution_param {
    num_output: 20
    kernel_size: 5
    stride: 1
    weight_filler {
        type: "xavier"
    }
}
```

name, type, and the connection structure (input blobs and output blobs)

layer-specific parameters



- Every layer type defines
  - **Setup**
  - **Forward**
  - **Backward**

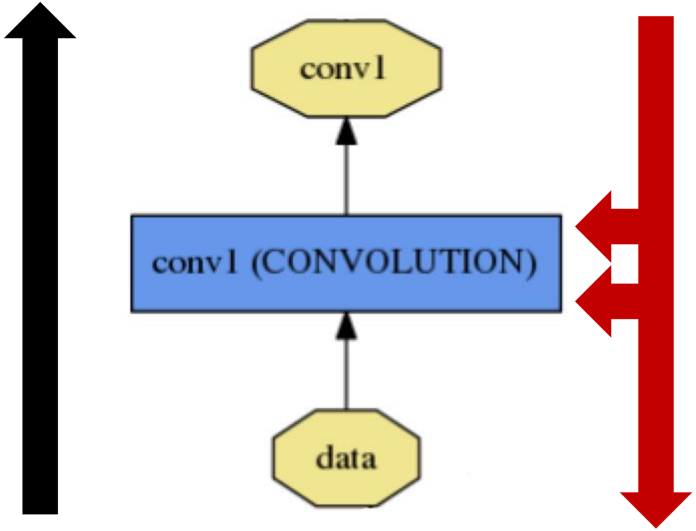
* Nets + Layers are defined by protobuf schema

Layer Protocol

Setup: run once for initialization.

Forward: make output given input.

Backward: make gradient of output
- w.r.t. bottom
- w.r.t. parameters (if needed)



Model Composition
The Net forward and backward passes are the composition the layers'.

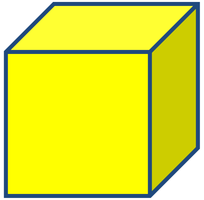
[Layer Development Checklist]

Blob

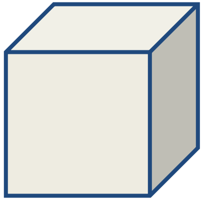
```
name: "conv1"
type: "Convolution"
bottom: "data"
top: "conv1"
… definition …
```

Blobs are ~~4~~ N-D arrays for storing and communicating information.

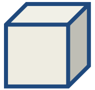
- hold data, derivatives, and parameters
- lazily allocate memory
- shuttle between CPU and GPU



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

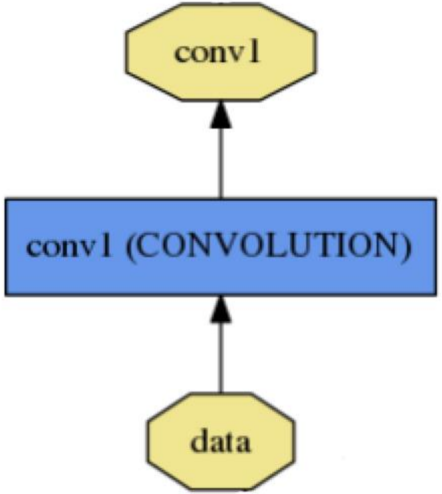


Parameter: Convolution Weight
N Output x *K* Input x *H*eight x *W*idth
96 x 3 x 11 x 11 for CaffeNet conv1



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

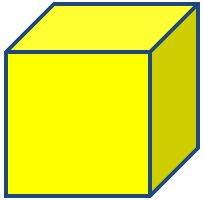
top
blob



bottom
blob

Blob

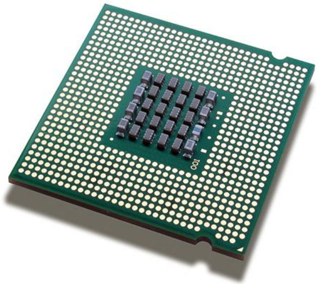
Blobs provide a unified memory interface.



Reshape(num, channel, height, width)
- declare dimensions
- make *SyncedMem* -- but only lazily allocate

cpu_data(), mutable_cpu_data()
- host memory for CPU mode
gpu_data(), mutable_gpu_data()
- device memory for GPU mode

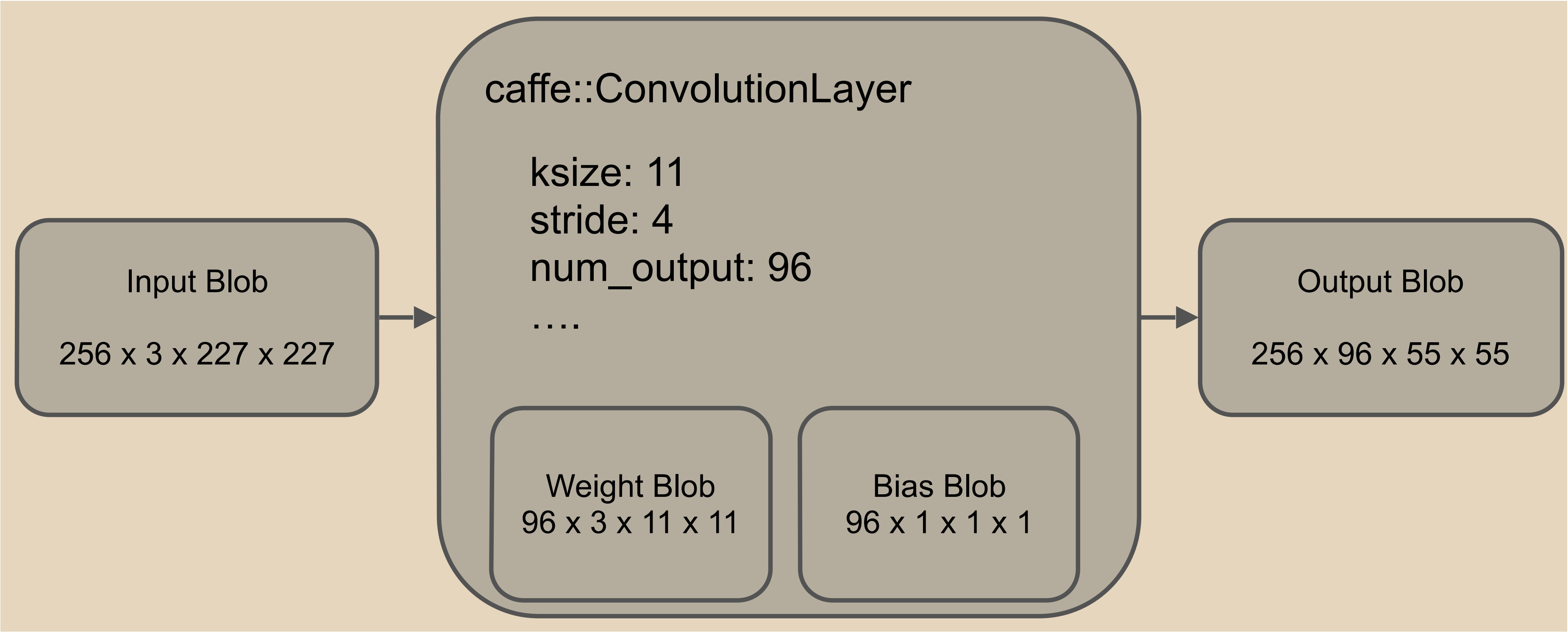
{cpu,gpu}_diff(), mutable_{cpu,gpu}_diff()
- derivative counterparts to data methods
- easy access to data + diff in forward / backward



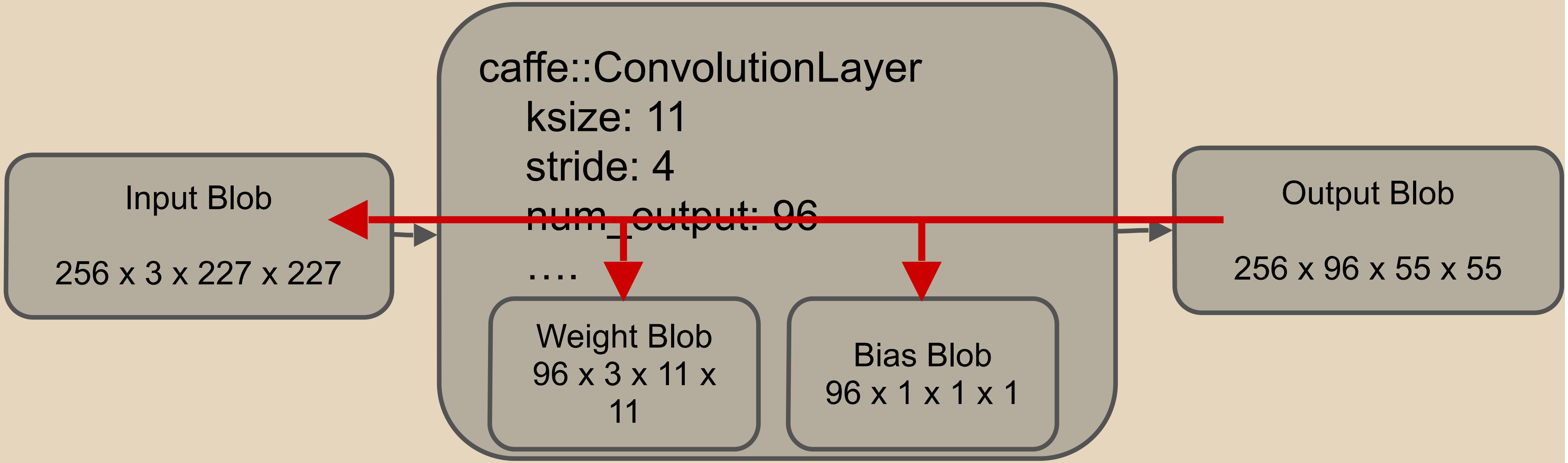
SyncedMem
allocation + communication

- Earlier, CAFFE only supported 4-D blobs and 2-D convolutions (NxCxHxW)
- Since October’15, it supports
  - n-D blobs and
  - (n-2)-D convolutions

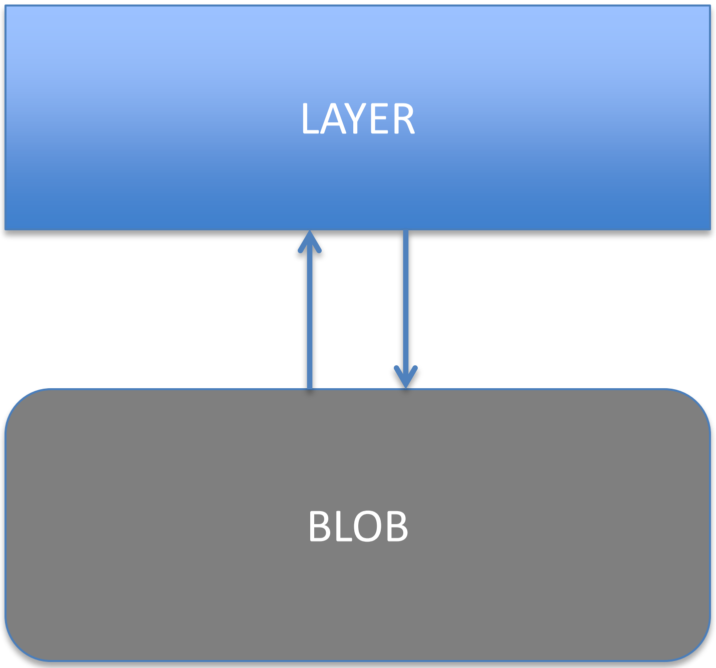

Input Blob
caffe::Net
Output Blob


caffe::ConvolutionLayer
ksize: 11
stride: 4
num_output: 96
….
Input Blob
256 x 3 x 227 x 227
Output Blob
256 x 96 x 55 x 55
Weight Blob
96 x 3 x 11 x 11
Bias Blob
96 x 1 x 1 x 1

- Forward: given input, computes the output.
- Backward: given the gradient w.r.t. the output, compute the gradient w.r.t. the input and its internal parameters.
- Setup: how to initialize the layer.

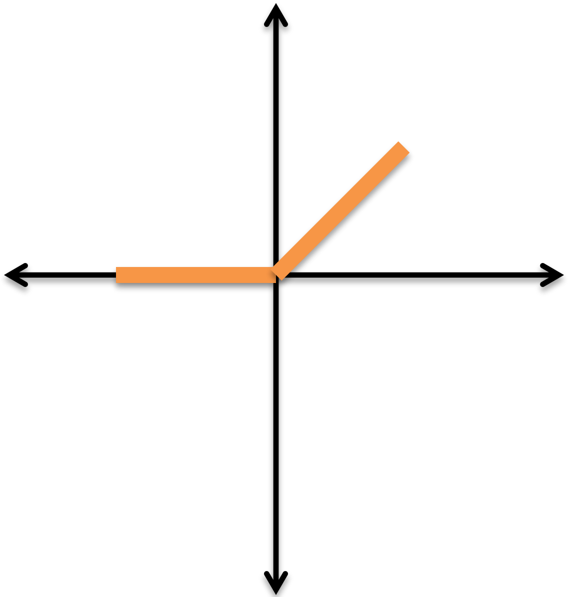


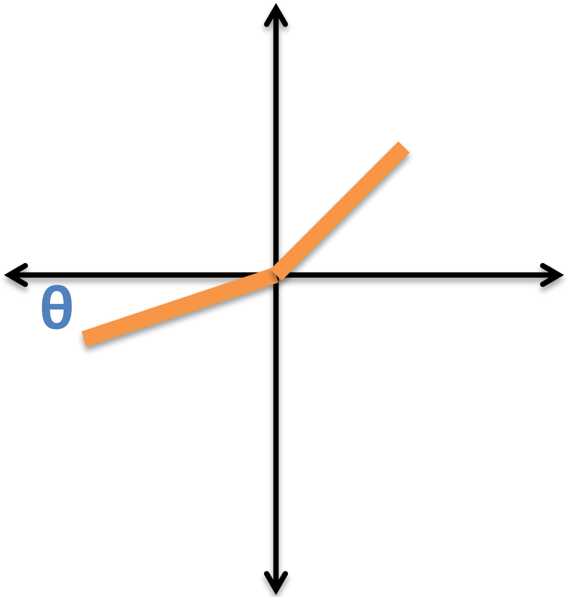
Can also have..



In-place
updates

Example: ReLU or PReLU





(PReLU) He *et al.* ICCV'15

How much memory would a PReLU require?

- It does an in-place update, so say requires B for blob
- Say it requires P for parameters (could be per-channel, or just a single scalar)
- Does it need any more?
  - Yes! Need to keep the original input around for computing the derivative for parameters => $+B$
- Q: Can parameterized layers do in-place updates?

GPU/CPU Switch with Blob

- Use synchronized memory
- Mutable/non-mutable determines whether to copy. Use of `mutable_*` may lead to data copy
- Rule of thumb:

  Use ~~mutable_~~{cpu|gpu}_data whenever possible

Layers

```
src/caffe/layers/accuracy_layer.cpp            src/caffe/layers/eltwise_layer.cpp                      src/caffe/layers/pooling_layer.cpp
src/caffe/layers/argmax_layer.cpp              src/caffe/layers/euclidean_loss_layer.cpp               src/caffe/layers/power_layer.cpp
src/caffe/layers/base_data_layer.cpp           src/caffe/layers/flatten_layer.cpp                      src/caffe/layers/relu_layer.cpp
src/caffe/layers/bnll_layer.cpp                src/caffe/layers/hdf5_data_layer.cpp                    src/caffe/layers/sigmoid_cross_entropy_loss_layer.cpp
src/caffe/layers/concat_layer.cpp              src/caffe/layers/hdf5_output_layer.cpp                  src/caffe/layers/sigmoid_layer.cpp
src/caffe/layers/contrastive_loss_layer.cpp    src/caffe/layers/hinge_loss_layer.cpp                   src/caffe/layers/silence_layer.cpp
src/caffe/layers/conv_layer.cpp                src/caffe/layers/im2col_layer.cpp                       src/caffe/layers/slice_layer.cpp
src/caffe/layers/cudnn_conv_layer.cpp          src/caffe/layers/image_data_layer.cpp                   src/caffe/layers/softmax_layer.cpp
src/caffe/layers/cudnn_pooling_layer.cpp       src/caffe/layers/infogain_loss_layer.cpp                src/caffe/layers/softmax_loss_layer.cpp
src/caffe/layers/cudnn_relu_layer.cpp          src/caffe/layers/inner_product_layer.cpp                src/caffe/layers/split_layer.cpp
src/caffe/layers/cudnn_sigmoid_layer.cpp       src/caffe/layers/loss_layer.cpp                         src/caffe/layers/tanh_layer.cpp
src/caffe/layers/cudnn_softmax_layer.cpp       src/caffe/layers/lrn_layer.cpp                          src/caffe/layers/threshold_layer.cpp
src/caffe/layers/cudnn_tanh_layer.cpp          src/caffe/layers/memory_data_layer.cpp                  src/caffe/layers/window_data_layer.cpp
src/caffe/layers/data_layer.cpp                src/caffe/layers/multinomial_logistic_loss_layer.cpp
src/caffe/layers/dropout_layer.cpp             src/caffe/layers/mvn_layer.cpp
[abhi@aurora]~/research/codes/caffe-latest :>  ls src/caffe/layers/*.cu
src/caffe/layers/absval_layer.cu              src/caffe/layers/dropout_layer.cu            src/caffe/layers/relu_layer.cu
src/caffe/layers/base_data_layer.cu           src/caffe/layers/eltwise_layer.cu            src/caffe/layers/sigmoid_cross_entropy_loss_layer.cu
src/caffe/layers/bnll_layer.cu                src/caffe/layers/euclidean_loss_layer.cu     src/caffe/layers/sigmoid_layer.cu
src/caffe/layers/concat_layer.cu              src/caffe/layers/flatten_layer.cu            src/caffe/layers/silence_layer.cu
src/caffe/layers/contrastive_loss_layer.cu    src/caffe/layers/hdf5_data_layer.cu          src/caffe/layers/slice_layer.cu
src/caffe/layers/conv_layer.cu                src/caffe/layers/hdf5_output_layer.cu        src/caffe/layers/softmax_layer.cu
src/caffe/layers/cudnn_conv_layer.cu          src/caffe/layers/im2col_layer.cu             src/caffe/layers/softmax_loss_layer.cu
src/caffe/layers/cudnn_pooling_layer.cu       src/caffe/layers/inner_product_layer.cu      src/caffe/layers/split_layer.cu
src/caffe/layers/cudnn_relu_layer.cu          src/caffe/layers/lrn_layer.cu                src/caffe/layers/tanh_layer.cu
src/caffe/layers/cudnn_sigmoid_layer.cu       src/caffe/layers/mvn_layer.cu                src/caffe/layers/threshold_layer.cu
```

More about Layers

- Data layers
- Vision layers
- Common layers
- Activation/Neuron layers
- Loss layers

Data Layers

- Data enters 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/.mat or common image formats.
- Common input preprocessing (mean subtraction, scaling, random cropping, and mirroring) is available by specifying `TransformationParameters`.

Data Layers

- `Data (Backend: LevelDB,LMDB)`
- `MemoryData`
- `HDF5Data`
- `ImageData`
- `WindowData`
- `DummyData`
- `Write your own! In Python!`

Data Layers

Database

- Layer type: `Data`
- Parameters
  - Required
    - `source`: the name of the directory containing the database
    - `batch_size`: the number of inputs to process at one time
  - Optional
    - `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`

Data Layers

```
name: "LeNet"
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    scale: 0.00390625
  }
  data_param {
    source: "examples/mnist/mnist_train_lmdb"
    batch_size: 64
    backend: LMDB
  }
}
```

Data Layers

In-Memory

- Layer type: `MemoryData`
- Parameters
  - Required
    - `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

- Layer type: `HDF5Data`
- Parameters
  - Required
    - `source`: the name of the file to read from
    - `batch_size`

HDF5 Output

- Layer type: `HDF5Output`
- Parameters
  - Required
    - `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

- Layer type: `ImageData`
- Parameters
  - Required
    - `source`: name of a text file, with each line giving an image filename and label
    - `batch_size`: number of images to batch together
  - Optional
    - `rand_skip`
    - `shuffle` [default false]
    - `new_height`, `new_width`: if provided, resize all images to this size

Windows

`WindowData`

Dummy

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

Writing your own data layer in python

- Compile CAFFE, uncommenting in Makefile.config

  # WITH_PYTHON_LAYER := 1

- Example: See Fast-RCNN

Prototxt

```
name: "CaffeNet"
layer {
  name: 'data'
  type: 'Python'
  top: 'data'
  top: 'rois'
  top: 'labels'
  top: 'bbox_targets'
  top: 'bbox_loss_weights'
  python_param {
    module: 'roi_data_layer.layer'
    layer: 'RoIDataLayer'
    param_str: "'num_classes': 21"
  }
}
```

Python

```python
import caffe

class RoIDataLayer(caffe.Layer):
    """Fast R-CNN data layer used for training."""

    def setup(self, bottom, top):
        """Setup the RoIDataLayer."""
        # ...
        pass

    def forward(self, bottom, top):
        # ...
        pass

    def backward(self, top, propagate_down, bottom):
        """This layer does not propagate gradients."""
        pass

    def reshape(self, bottom, top):
        """Reshaping happens during the call to fwd."""
        pass
```

Transformations

```
layer {
  name: "data"
  type: "Data"
  [...]
  transform_param {
    scale: 0.1
    mean_file_size: mean.binaryproto
    # for images in particular horizontal mirroring and random cropping
    # can be done as simple data augmentations.
    mirror: 1  # 1 = on, 0 = off
    # crop a `crop_size` x `crop_size` patch:
    # - at random during training
    # - from the center during testing
    crop_size: 227
  }
}
```

Note that all layers do not support transformations, like HDF5

More about Layers

- Data layers
- Vision layers
- Common layers
- Activation/Neuron layers
- Loss layers

Vision Layers

- Images as input and produce other images as output.
- Non-trivial height h>1 and width w>1.
- 2D geometry naturally lends itself to certain decisions about how to process the input.
  - Since Oct'15, supports nD convolutions
- 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 Layer

Convolution

- Layer type: `Convolution`
- CPU implementation: `./src/caffe/layers/convolution_layer.cpp`
- CUDA GPU implementation: `./src/caffe/layers/convolution_layer.cu`
- Parameters (`ConvolutionParameter convolution_param`)
  - Required
    - `num_output` (`c_o`): the number of filters
    - `kernel_size` (or `kernel_h` and `kernel_w`): specifies height and width of each filter
  - Strongly Recommended
    - `weight_filler` [default `type: 'constant' value: 0`]
  - Optional
    - `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.

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.

```
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
    }
  }
}
```

Pooling Layer

- LayerType: `POOLING`
- CPU implementation: `./src/caffe/layers/pooling_layer.cpp`
- CUDA GPU implementation: `./src/caffe/layers/pooling_layer.cu`
- Parameters (`PoolingParameter pooling_param`)
    - Required
        - `kernel_size` (or `kernel_h` and `kernel_w`): specifies height and width of each filter
    - Optional
        - `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

```
layers {
  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
  }
}
```

Vision Layers

- Convolution
- Pooling
- Local Response Normalization (LRN)
- Im2col -- helper

More about Layers

- Data layers
- Vision layers
- Common layers
- Activation/Neuron layers
- Loss layers

Common Layers

- `INNER_PRODUCT` $W^T x+b$ (fully connected)
- `SPLIT`
- `FLATTEN`
- `CONCAT`
- `SLICE`
- `ELTWISE` (element wise operations)
- `ARGMAX`
- `SOFTMAX`
- `MVN` (mean-variance normalization)

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

More about Layers

- Data layers
- Vision layers
- Common layers
- Activation/Neuron layers
- Loss layers

Activation/Neuron layers

- One Input Blob
- One Output Blob
  - Both same size
- Or a single blob – in-place updates

Activation/Neuron layers

- `ReLU / PReLU`
- `Sigmoid`
- `Tanh`
- `Absval`
- `Power`
- `BNLL` (binomial normal log likelihood)

ReLU / Rectified-Linear and Leaky-ReLU

- Layer type: `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 `./models/bvlc_reference_caffenet/train_val.prototxt`)

```
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "conv1"
  top: "conv1"
}
```

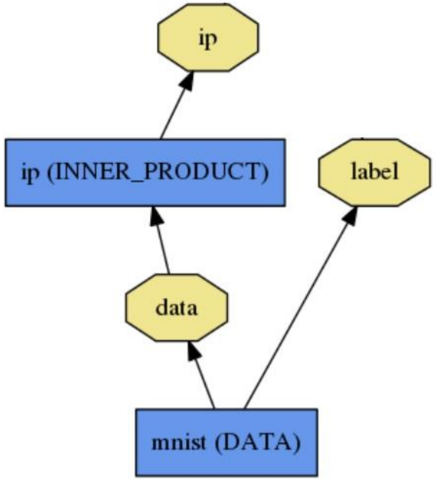
Given an input value x, The `ReLU` layer computes the output as x if x > 0 and negative_slope * x if x <= 0. When the negative slope parameter is not set, it is equivalent to the standard ReLU function of taking max(x, 0). It also supports in-place computation, meaning that the bottom and the top blob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

More about Layers

- Data layers
- Vision layers
- Common layers
- Activation/Neuron layers
- Loss layers

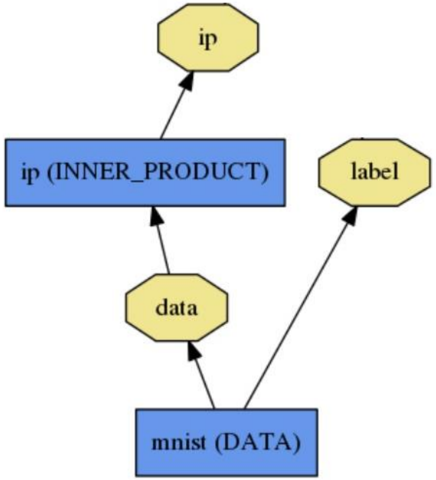
Loss

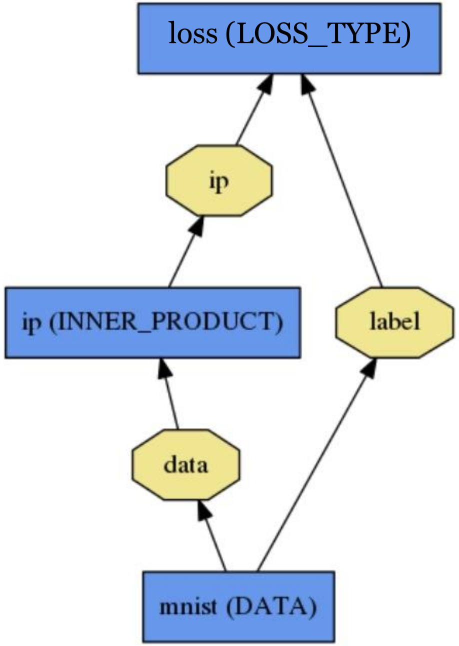
What kind of model is this?



Loss

What kind of model is this?





Who knows! Need a **loss function**.

Classification

`SOFTMAX_LOSS`

`HINGE_LOSS`

Linear Regression

`EUCLIDEAN_LOSS`

Attributes / Multiclassification

`SIGMOID_CROSS_ENTROPY_LOSS`

Others…

New Task

`NEW_LOSS`

Loss

- **Loss function** determines the learning task.
- Given data D, a Net typically minimizes:

$$L(W) = \frac{1}{|D|} \sum_{i}^{|D|} f_W\left(X^{(i)}\right) + \lambda r(W)$$

Data term: error averaged over instances

Regularization term: penalize large weights to improve generalization

Loss

- The data error term $f_W\left(X^{(i)}\right)$ is computed by Net::Forward
- Loss is computed as the output of Layers
- Pick the loss to suit the task – many different losses for different needs

Loss Layers

- SOFTMAX_LOSS
- HINGE_LOSS
- EUCLIDEAN_LOSS
- SIGMOID_CROSS_ENTROYPY_LOSS
- INFOGAIN_LOSS

- ACCURACY
- TOPK

Loss Layers

- SOFTMAX_LOSS
- HINGE_LOSS
- EUCLIDEAN_LOSS
- SIGMOID_..._LOSS
- INFOGAIN_LOSS
- ACCURACY
- TOPK
- **NEW_LOSS**

Classification

Linear Regression

Attributes / Multiclassification

Other losses

Not a loss

Softmax Loss Layer

- Multinomial logistic regression: used for predicting a single class of K mutually exclusive classes

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

$$\hat{p}_{nk} = \exp(x_{nk}) / \left[\sum_{k'} \exp(x_{nk'})\right]$$

$$E = \frac{-1}{N} \sum_{n=1}^{N} \log(\hat{p}_{n,l_n}),$$

Sigmoid Cross-Entropy Loss

- Binary logistic regression: used for predicting K independent probability values in [0, 1]

```
layers {
  name: "loss"
  type: "SigmoidCrossEntropyLoss"
  bottom: "pred"
  bottom: "label"
  top: "loss"
}
```

$$y = (1 + \exp(-x))^{-1}$$

$$E = \frac{-1}{n} \sum_{n=1}^{N} \left[p_n \log \hat{p}_n + (1 - p_n) \log(1 - \hat{p}_n) \right]$$

Euclidean Loss

- A loss for regressing to real-valued labels [-inf, inf]

```
layers {
  name: "loss"
  type: "EuclideanLoss"
  bottom: "pred"
  bottom: "label"
  top: "loss"
}
```

$$E = \frac{1}{2N} \sum_{n=1}^{N} ||\hat{y}_n - y_n||_2^2$$

Multiple loss layers

- Your network can contain as many loss functions as you want, as long as it is a DAG!
- Reconstruction and Classification:

$$E = \frac{1}{2N} \sum_{n=1}^{N} ||\hat{y}_n - y_n||_2^2 + \frac{-1}{N} \sum_{n=1}^{N} \log(\hat{p}_{n,l_n})$$

```
layers {
  name: "recon-loss"
  type: "EuclideanLoss"
  bottom: "reconstructions"
  bottom: "data"
  top: "recon-loss"
}

layers {
  name: "class-loss"
  type: "SoftmaxWithLoss"
  bottom: "class-preds"
  bottom: "class-labels"
  top: "class-loss"
}
```

Multiple loss layers

"*Loss" layers have a default loss weight of 1

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

==

```
layers {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "pred"
  bottom: "label"
  top: "loss"
  loss_weight: 1.0
}
```

Multiple loss layers

- Give each loss its own weight
- E.g. give higher priority to classification error
- Or, to balance the values of different loss functions

```
layers {
  name: "recon-loss"
  type: "EuclideanLoss"
  bottom: "reconstructions"
  bottom: "data"
  top: "recon-loss"
}

layers {
  name: "class-loss"
  type: "SoftmaxWithLoss"
  bottom: "class-preds"
  bottom: "class-labels"
  top: "class-loss"
  loss_weight: 100.0
}
```

$$E = \frac{1}{2N} \sum_{n=1}^{N} \|\hat{y}_n - y_n\|_2^2 + 100 * \frac{-1}{N} \sum_{n=1}^{N} \log(\hat{p}_{n,l_n})$$

Any layer can produce a loss!

- Just add `loss_weight: 1.0` to have a layer's output be incorporated into the loss

E = || pred - label ||^2 / (2N)

```
layers {
  name: "loss"
  type: "EuclideanLoss"
  bottom: "pred"
  bottom: "label"
  top: "euclidean_loss"
  loss_weight: 1.0
}
```

==

diff = pred - label

```
layers {
  name: "diff"
  type: "Eltwise"
  bottom: "pred"
  bottom: "label"
  top: "diff"
  eltwise_param {
    op: SUM
    coeff: 1
    coeff: -1
  }
}
```

+

E = || diff ||^2 / (2N)

```
layers {
  name: "loss"
  type: "Power"
  bottom: "diff"
  top: "euclidean_loss"
  power_param {
    power: 2
  }
  # = 1/(2N)
  loss_weight: 0.0078125
}
```

Layers

- Data layers
- Vision layers
- Common layers
- Activation/Neuron layers
- Loss layers

Initialization

- Gaussian [most commonly used]
- Xavier
- Constant [default]

- Goal: keep the variance roughly fixed

Solving: Training a Net

Optimization like model definition is configuration.

```
train_net: "lenet_train.prototxt"
base_lr: 0.01
momentum: 0.9
weight_decay: 0.0005
max_iter: 10000
snapshot_prefix: "lenet_snapshot"
```

All you need to run things on the GPU.

```
> caffe train -solver lenet_solver.prototxt -gpu 0
```

Stochastic Gradient Descent (SGD) + momentum · Adaptive Gradient (ADAGRAD) · Nesterov's Accelerated Gradient (NAG)

```
# The train/test net protocol buffer definition
net: "logreg_train.prototxt"
# test_iter specifies how many forward passes the test should carry out.
# In the case of MNIST, we have test batch size 100 and 100 test iterations,
# covering the full 10,000 testing images.
test_iter: 100
# Carry out testing every 500 training iterations.
test_interval: 500
# The base learning rate, momentum and the weight decay of the network.
base_lr: 0.01
momentum: 0.9
weight_decay: 0.0005
# The learning rate policy
lr_policy: "inv"
gamma: 0.0001
power: 0.75
# Display every 100 iterations
display: 100
# The maximum number of iterations
max_iter: 10000
# snapshot intermediate results
snapshot: 5000
snapshot_prefix: "examples/mnist/lenet"
# solver mode: CPU or GPU
solver_mode: GPU
```

Solver

- **Solver** optimizes the network weights W to minimize the loss L(W) over the data D

$$L(W) = \frac{1}{|D|} \sum_{i}^{|D|} f_W \left(X^{(i)} \right) + \lambda r(W)$$

- Coordinates forward / backward, weight updates, and scoring.

Solver

- Computes parameter update ΔW formed from
  - The stochastic error gradient ∇f_W
  - The regularization gradient $\nabla r(W)$
  - Particulars to each solving method

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

SGD Solver

- Stochastic gradient descent, with momentum
- `solver_type: SGD`

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

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

SGD Solver

- "AlexNet" [1] training strategy:
  - Use momentum 0.9
  - Initialize learning rate at 0.01
  - Periodically drop learning rate by a factor of 10
- Just a few lines of Caffe solver specification:

```
base_lr: 0.01
lr_policy: "step"
gamma: 0.1
stepsize: 100000
max_iter: 350000
momentum: 0.9
```

NAG Solver

- Nesterov's accelerated gradient [1]
- `solver_type: NESTEROV`
- Proven to have optimal convergence rate $\mathcal{O}(1/t^2)$ for convex problems

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

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

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

AdaGrad Solver

- Adaptive gradient (Duchi et al. [1])
- `solver_type: ADAGRAD`
- Attempts to automatically scale gradients based on historical gradients

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

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

Solver Showdown: MNIST Autoencoder

AdaGrad

```
I0901 13:36:30.007884 24952 solver.cpp:232] Iteration 65000, loss = 64.1627
I0901 13:36:30.007922 24952 solver.cpp:251] Iteration 65000, Testing net (#0) # train set
I0901 13:36:33.019305 24952 solver.cpp:289] Test loss: 63.217
I0901 13:36:33.019356 24952 solver.cpp:302]     Test net output #0: cross_entropy_loss = 63.217 (* 1 = 63.217 loss)
I0901 13:36:33.019773 24952 solver.cpp:302]     Test net output #1: l2_error = 2.40951
```

SGD

```
I0901 13:35:20.426187 20072 solver.cpp:232] Iteration 65000, loss = 61.5498
I0901 13:35:20.426218 20072 solver.cpp:251] Iteration 65000, Testing net (#0) # train set
I0901 13:35:22.780092 20072 solver.cpp:289] Test loss: 60.8301
I0901 13:35:22.780138 20072 solver.cpp:302]     Test net output #0: cross_entropy_loss = 60.8301 (* 1 = 60.8301 loss)
I0901 13:35:22.780146 20072 solver.cpp:302]     Test net output #1: l2_error = 2.02321
```

Nesterov

```
I0901 13:36:52.466069 22488 solver.cpp:232] Iteration 65000, loss = 59.9389
I0901 13:36:52.466099 22488 solver.cpp:251] Iteration 65000, Testing net (#0) # train set
I0901 13:36:55.068370 22488 solver.cpp:289] Test loss: 59.3663
I0901 13:36:55.068410 22488 solver.cpp:302]     Test net output #0: cross_entropy_loss = 59.3663 (* 1 = 59.3663 loss)
I0901 13:36:55.068418 22488 solver.cpp:302]     Test net output #1: l2_error = 1.79998
```

Weight sharing

- Parameters can be shared and reused across Layers throughout the Net

- Applications:
  - Convolution at multiple scales / pyramids
  - Recurrent Neural Networks (RNNs)
  - Siamese nets for distance learning

Weight sharing

- Just give the parameter blobs explicit names using the `param` field
- Layers specifying the same `param` name will share that parameter, accumulating gradients accordingly

```
layers: {
  name: 'innerproduct1'
  type: "InnerProduct"
  inner_product_param {
    num_output: 10
    bias_term: false
    weight_filler {
      type: 'gaussian'
      std: 10
    }
  }
  param: 'sharedweights'
  bottom: 'data'
  top: 'innerproduct1'
}
layers: {
  name: 'innerproduct2'
  type: "InnerProduct"
  inner_product_param {
    num_output: 10
    bias_term: false
  }
  param: 'sharedweights'
  bottom: 'data'
  top: 'innerproduct2'
}
```

Interfaces

- Command Line
- Python
- Matlab

CMD

$> Caffe --params

```
# 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
```

CMD

```
# fine-tune CaffeNet model weights for pascal
caffe train \
-solver examples/finetune_pascal_detection/pascal_finetune_solver.prototxt -
weights models/bvlc_reference_caffenet/bvlc_reference_caffenet.caffemodel
```

```
# 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 -gpu 0 -iterations 100
```

CMD

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

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

Python

```
$> make pycaffe
python> import caffe
```

`caffe.Net`: is the central interface for loading, configuring, and running models.

`caffe.Classsifier & caffe.Detector` for convenience

`caffe.SGDSolver` exposes the solving interface.

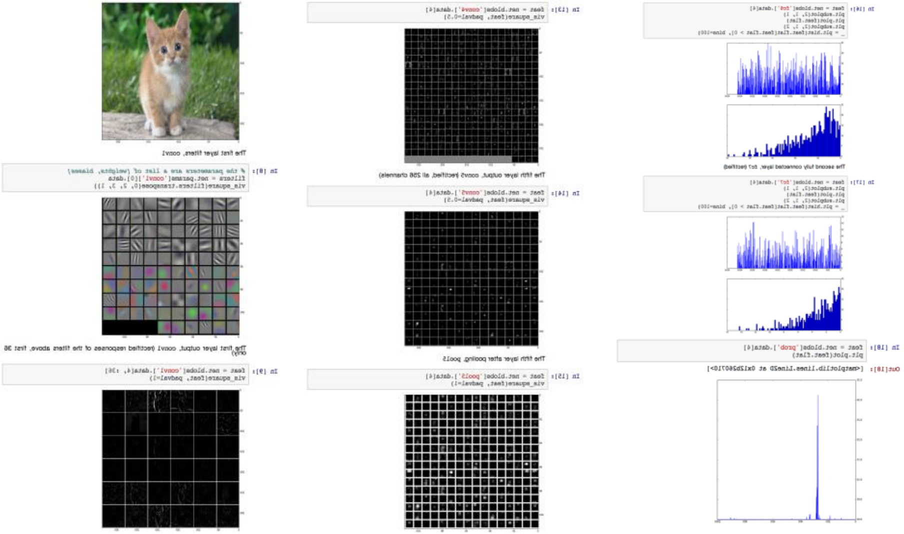
`caffe.io` handles I/O with preprocessing and protocol buffers.

`caffe.draw` visualizes network architectures.

`Caffe blobs` are exposed as numpy ndarrays for ease-of-use and efficiency**

Python

GOTO: IPython Filter Visualization Notebook



MATLAB

```
static handler_registry handlers[] = {
  // Public API functions
  { "get_solver",          get_solver       },
  { "solver_get_attr",     solver_get_attr  },
  { "solver_get_iter",     solver_get_iter  },
  { "solver_restore",      solver_restore   },
  { "solver_solve",        solver_solve     },
  { "solver_step",         solver_step      },
  { "get_net",             get_net          },
  { "net_get_attr",        net_get_attr     },
  { "net_forward",         net_forward      },
  { "net_backward",        net_backward     },
  { "net_copy_from",       net_copy_from    },
  { "net_reshape",         net_reshape      },
  { "net_save",            net_save         },
  { "layer_get_attr",      layer_get_attr   },
  { "layer_get_type",      layer_get_type   },
  { "blob_get_shape",      blob_get_shape   },
  { "blob_reshape",        blob_reshape     },
  { "blob_get_data",       blob_get_data    },
  { "blob_set_data",       blob_set_data    },
  { "blob_get_diff",       blob_get_diff    },
  { "blob_set_diff",       blob_set_diff    },
  { "set_mode_cpu",        set_mode_cpu     },
  { "set_mode_gpu",        set_mode_gpu     },
  { "set_device",          set_device       },
  { "get_init_key",        get_init_key     },
  { "reset",               reset            },
  { "read_mean",           read_mean        },
  { "write_mean",          write_mean       },
  { "version",             version          },
  // The end.
  { "END",                 NULL             },
};
```

RECENT MODELS

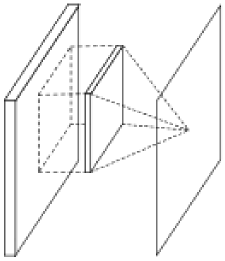
- Network-in-Network (NIN)
- GoogLeNet
- VGG

Questions?

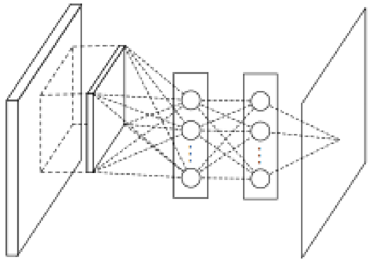
THAT’S ALL! THANKS!

Network-in-Network

- filter with a nonlinear composition instead of a linear filter
- 1x1 convolution + nonlinearity
- reduce dimensionality, deepen the representation

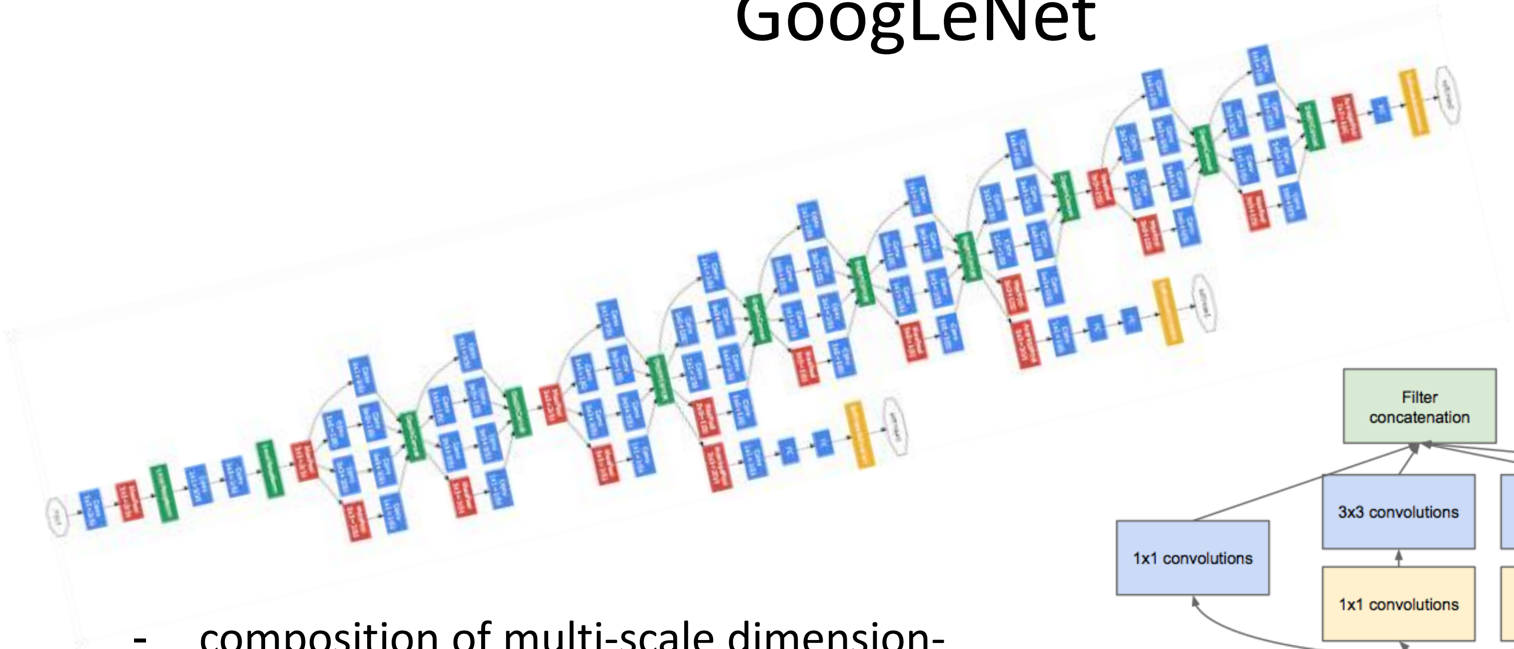


Linear Filter
CONV

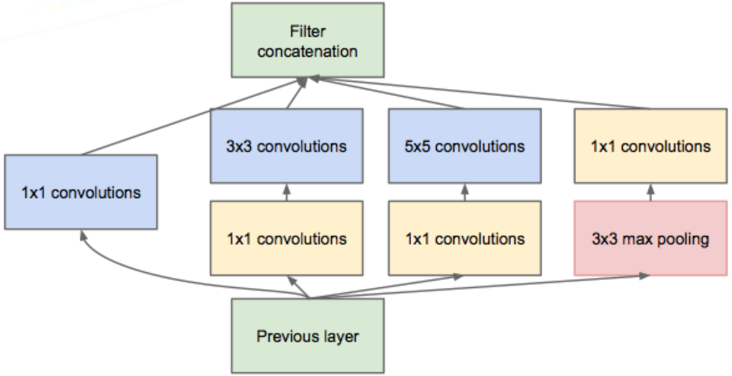


NIN / MLP filter
1x1 CONV

GoogLeNet



- composition of multi-scale dimension-reduced “Inception” modules
- 1x1 conv for dimensionality reduction
- concatenation across filter scales
- multiple losses for training to depth



“Inception” module

VGG

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Table 2: **Number of parameters** (in millions).

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

- 3x3 convolution all the way down...
- fine-tuned progression of deeper models
- 16 and 19 parameter layer variations in the model zoo

Blob Data Management

```
// 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.
```