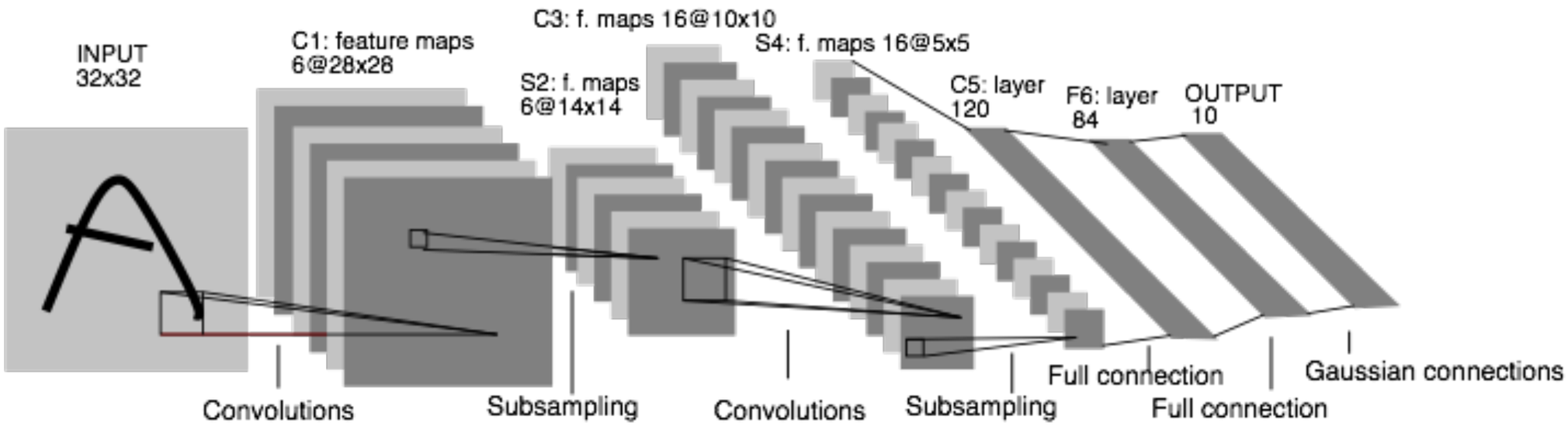
Caffe tutorial

borrowed slides from:
caffe official tutorials

Recap Convnet



Supervised learning trained by stochastic gradient descend

$$J(W,b) = \frac{1}{2}||h(x) - y||^2$$

1. feedforward: get the activations for each layer and the cost
2. backward: get the gradient for all the parameters
3. update: gradient descend

Outline

- For people who use CNN as a blackbox
- For people who want to define new layers & cost functions
- A few training tricks.

* there is a major update for caffe recently,
we might get different versions

Blackbox Users

http://caffe.berkeleyvision.org/tutorial/

highly recommended!

Installation

detailed documentation:

http://caffe.berkeleyvision.org/installation.html

required packages:

- **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.

Preparing data

—> If you want to run CNN on other dataset:

- caffe reads data in a standard database format.
- You have to convert your data to leveldb/lmdb manually.

```
layers {
  name: "mnist"
  type: DATA
  top: "data"
  top: "label"
  # the DATA layer configuration
  data_param {
    # path to the DB
    source: "examples/mnist/mnist_train_lmdb"
    # type of DB: LEVELDB or LMDB (LMDB supports concurrent reads)
    backend: LMDB
    # batch processing improves efficiency.
    batch_size: 64
  }
  # common data transformations
  transform_param {
    # feature scaling coefficient: this maps the [0, 255] MNIST data t
```

database type

Preparing data

this is the only coding needed (chenyi has experience)

declare database

```
// declare databases
// lmdb
MDB_env *mdb_env;
MDB_dbi mdb_dbi;
MDB_val mdb_key, mdb_data;
MDB_txn *mdb_txn;
// leveldb
leveldb::DB* db;
```

open database

```
// Open db
if (db_backend == "leveldb") {  // leveldb
  LOG(INFO) << "Opening leveldb " << db_path;
  leveldb::Status status = leveldb::DB::Open(
      options, db_path, &db);
} else if (db_backend == "lmdb") {  // lmdb
  LOG(INFO) << "Opening lmdb " << db_path;
  CHECK_EQ(mdb_open(mdb_txn, NULL, 0, &mdb_dbi), MDB_SUCCESS)
      << "mdb_open failed. Does the lmdb already exist? ";
```

write database

```
Datum datum;
datum.set_channels(1);
datum.set_height(rows);
datum.set_width(cols);
for (int item_id = 0; item_id < num_items; ++item_id) {
  image_file.read(pixels, rows * cols);
  label_file.read(&label, 1);
  datum.set_data(pixels, rows*cols);
  datum.set_label(label);
  snprintf(key_cstr, kMaxKeyLength, "%08d", item_id);
  datum.SerializeToString(&value);
  string keystr(key_cstr);

  // Put in db
  if (db_backend == "leveldb") {  // leveldb
    batch->Put(keystr, value);
  } else if (db_backend == "lmdb") {  // lmdb
    mdb_data.mv_size = value.size();
    mdb_data.mv_data = reinterpret_cast<void*>(&value[0]);
    mdb_key.mv_size = keystr.size();
    mdb_key.mv_data = reinterpret_cast<void*>(&keystr[0]);
    mdb_put(mdb_txn, mdb_dbi, &mdb_key, &mdb_data, 0), MD
```

---

how caffe loads data in `data_layer.cpp`

(you don't have to know)

```
// Initialize the leveldb
leveldb::DB* db_temp;
db_.reset(db_temp);
iter_.reset(db_->NewIterator(leveldb::ReadOptions()));
iter_->SeekToFirst();
// Read a data point, and use it to initialize the top blob.
Datum datum;
datum.ParseFromString(iter_->value().ToString());
```

example from mnist: `examples/mnist/convert_mnist_data.cpp`

define your network

—> If you want to define your own architecture

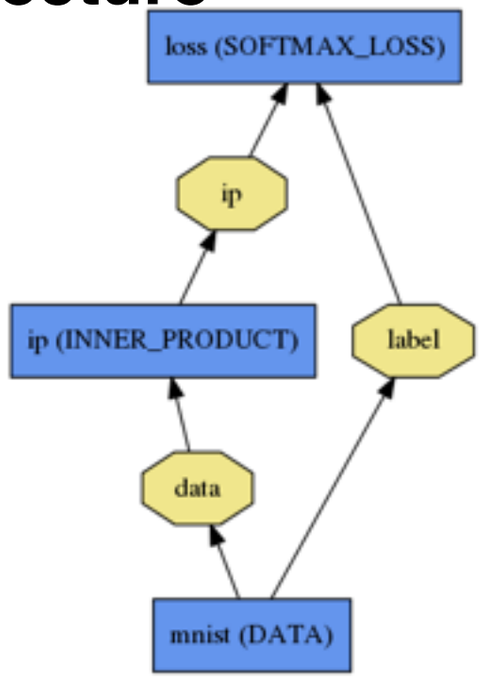
net:

blue: layers you need to define

yellow: data blobs

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

examples/mnist/lenet_train.prototxt



LogReg ↑

LeNet →

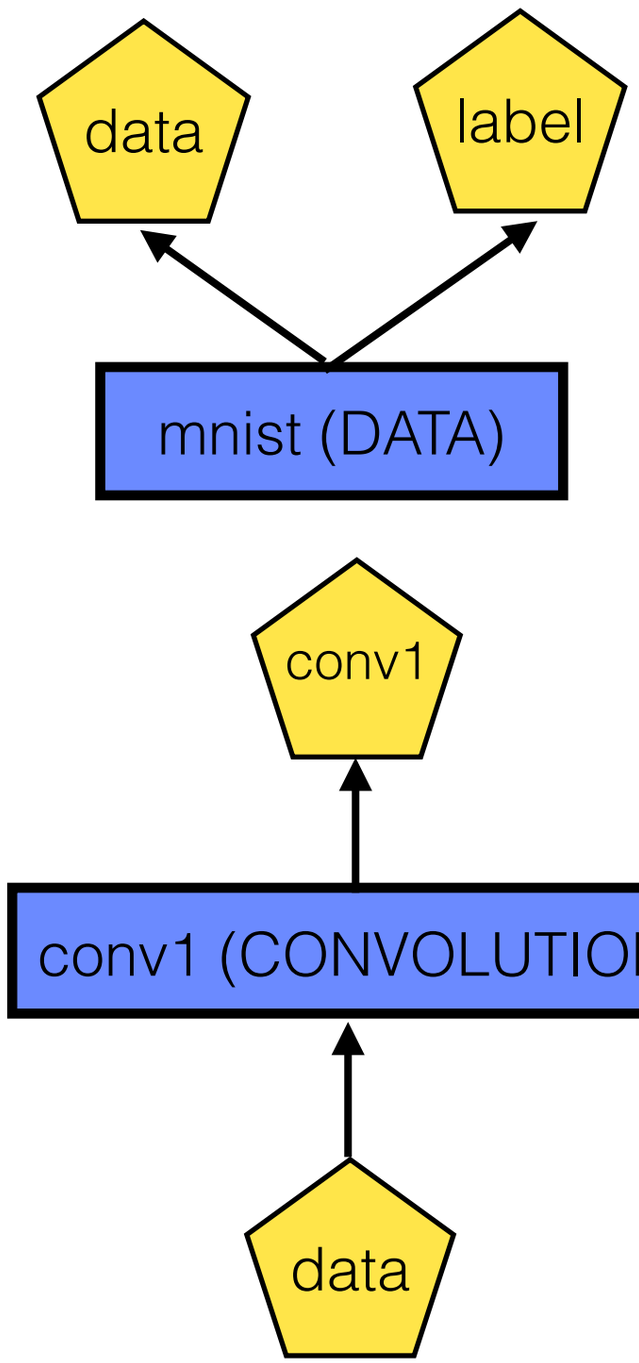
ImageNet, Krizhevsky 2012 →

define your network

```
name: "mnist"
type: DATA
top: "data"
top: "label"
data_param {
    source:
"mnist-train-
leveldb"
    scale:
0.00390625
    batch_size: 64
}
```

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

layer-specific parameters




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

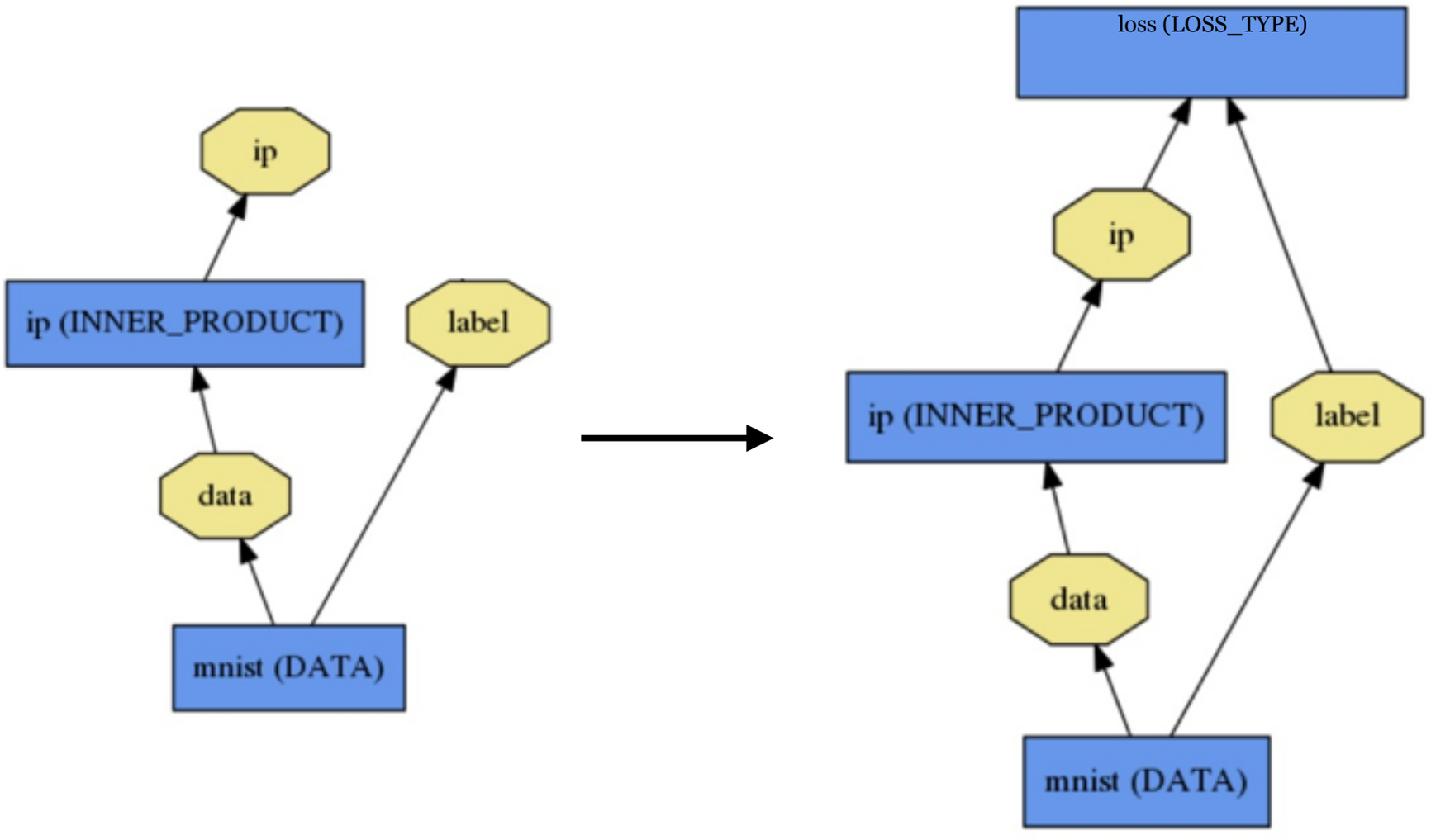
name, type, and the connection structure (input blobs and output blobs)

layer-specific parameters

examples/mnist/lenet_train.prototxt

define your network

loss:



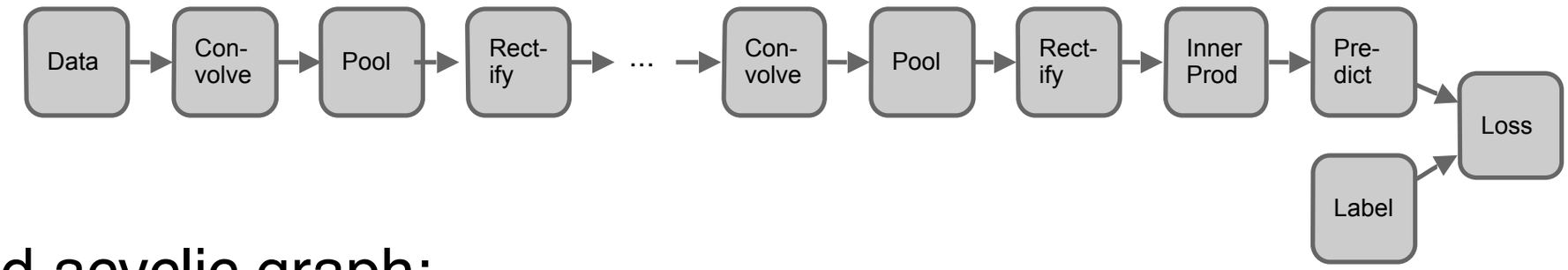

```
layers {
  name: "loss"
  type: SOFTMAX_LOSS
  bottom: "ip"
  bottom: "label"
  top: "loss"
}
```

define your network

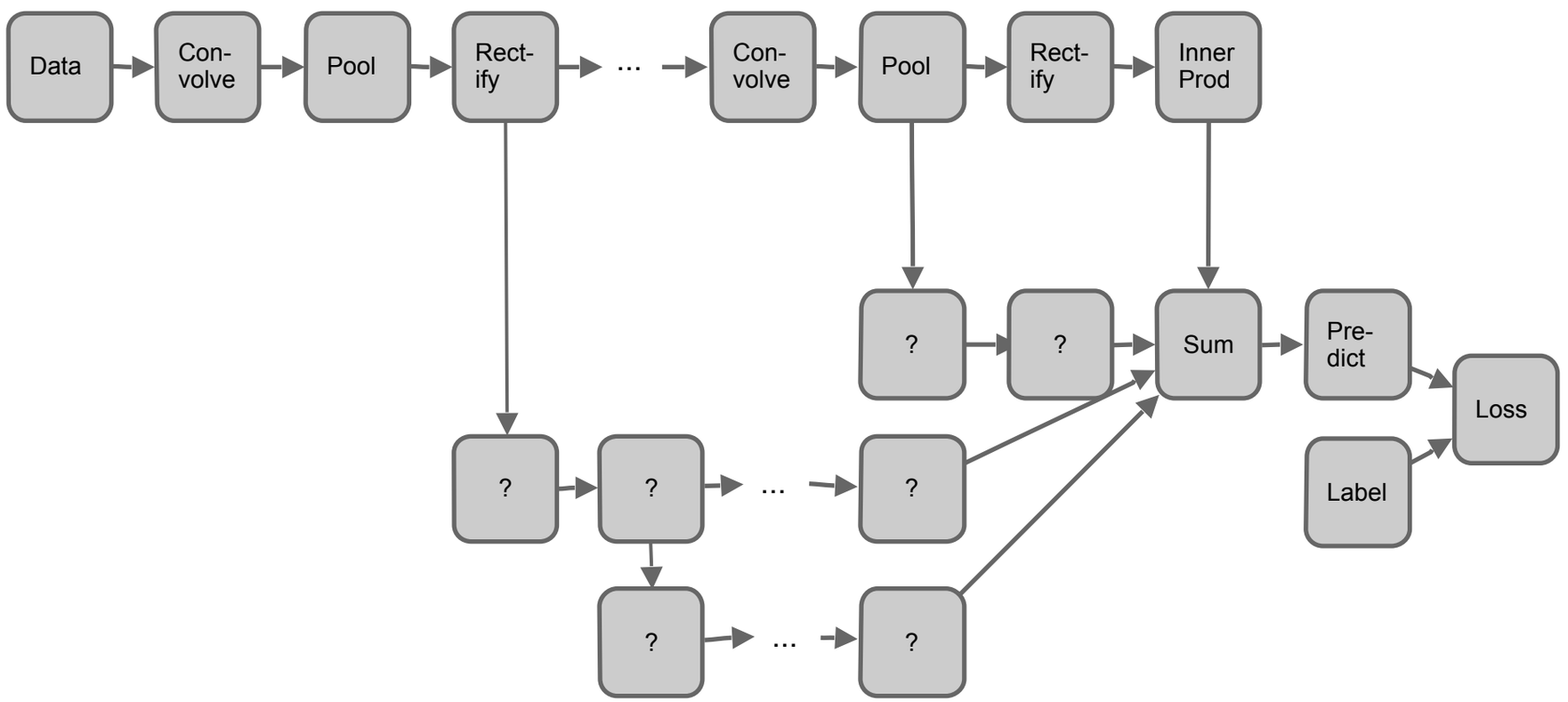
—> a little more about the network

- **network does not need to be linear**

linear network:



directed acyclic graph:



define your solver

- solver is for setting training parameters.

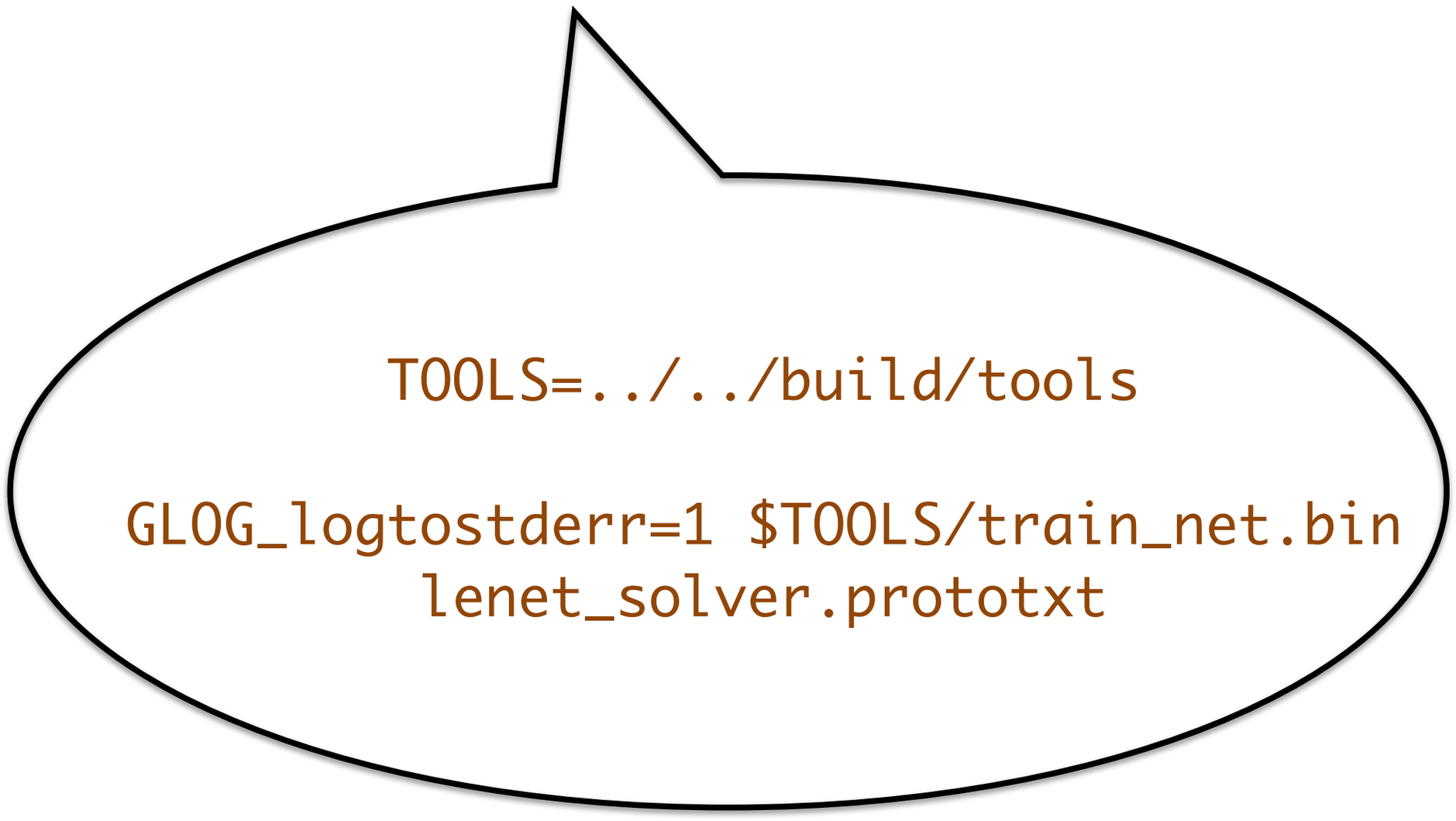
```
train_net: "lenet_train.prototxt"
base_lr: 0.01
lr_policy: “constant”
momentum: 0.9
weight_decay: 0.0005
max_iter: 10000
snapshot_prefix: "lenet_snapshot"
solver_mode: GPU
```

examples/mnist/lenet_solver.prototxt

train your model

—> you can now train your model by

```
./train_lenet.sh
```



finetuning models

—> what if you want to transfer the weight of a existing model to finetune another dataset / task

- Simply change a few lines in the layer definition

new name = new params

Input:
A different source

```
layers {
  name: "data"
  type: DATA
  data_param {
    source:
"ilsvrc12_train_leveldb"
    mean_file: "../../data/
ilsvrc12"
    ...
  }
```

```
layers {
  name: "data"
  type: DATA
  data_param {
    source: "style_leveldb"
    mean_file: "../../data/
ilsvrc12"
    ...
  }
  ...
```

Last Layer:
A different classifier

```
...
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: "fc8-style"
  type: INNER_PRODUCT
  blobs_lr: 1
  blobs_lr: 2
  weight_decay: 1
  weight_decay: 0
  inner_product_param {
    num_output: 20
    ...
  }
```

finetuning models

old caffe:

```
> finetune_net.bin solver.prototxt model_file
```

new caffe:

```
> caffe train -solver models/finetune_flickr_style/solver.prototxt
    -weights bvlc_reference_caffenet.caffemodel
```

Under the hood (loosely speaking):

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

extracting features

examples/
feature_extraction/
imagenet_val.prototxt

```
layers {
  name: "data"
  type: IMAGE_DATA
  top: "data"
  top: "label"
  image_data_param {
    source: "file_list.txt"
    mean_file: "imagenet_mean.binaryproto"
    crop_size: 227
    new_height: 256
    new_width: 256
  }
}
```

image list you want to process

Run:

```
build/tools/extract_features.bin imagenet_model
imagenet_val.prototxt fc7 temp/features 10
```

model_file

batch_size

network definition

data blobs you want to extract

output_file

MATLAB wrappers

—> What about importing the model into Matlab memory?

install the wrapper:

```
> make matcaffe
```

- RCNN provides a function for this:

```
> model = rcnn_load_model(model_file, use_gpu);
```

https://github.com/rbgirshick/rcnn

More curious Users

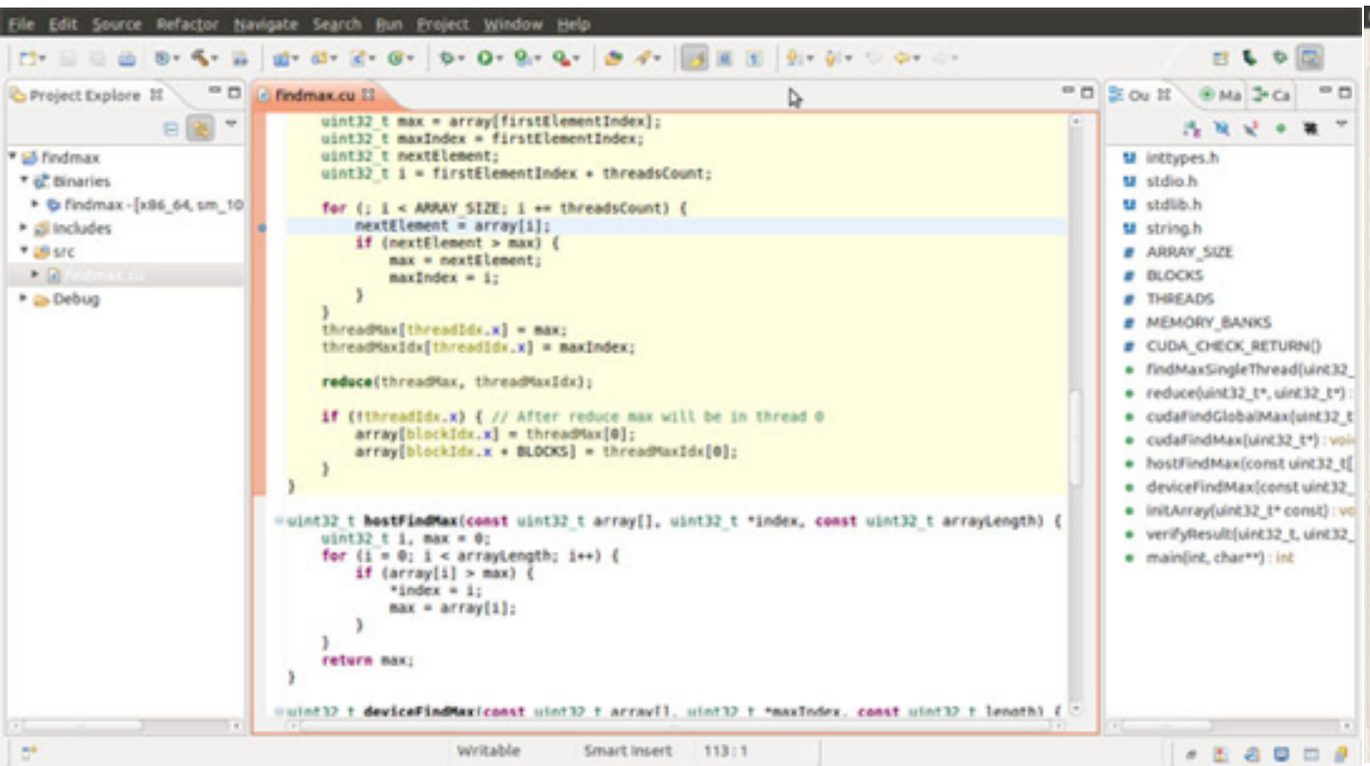
nsight IDE

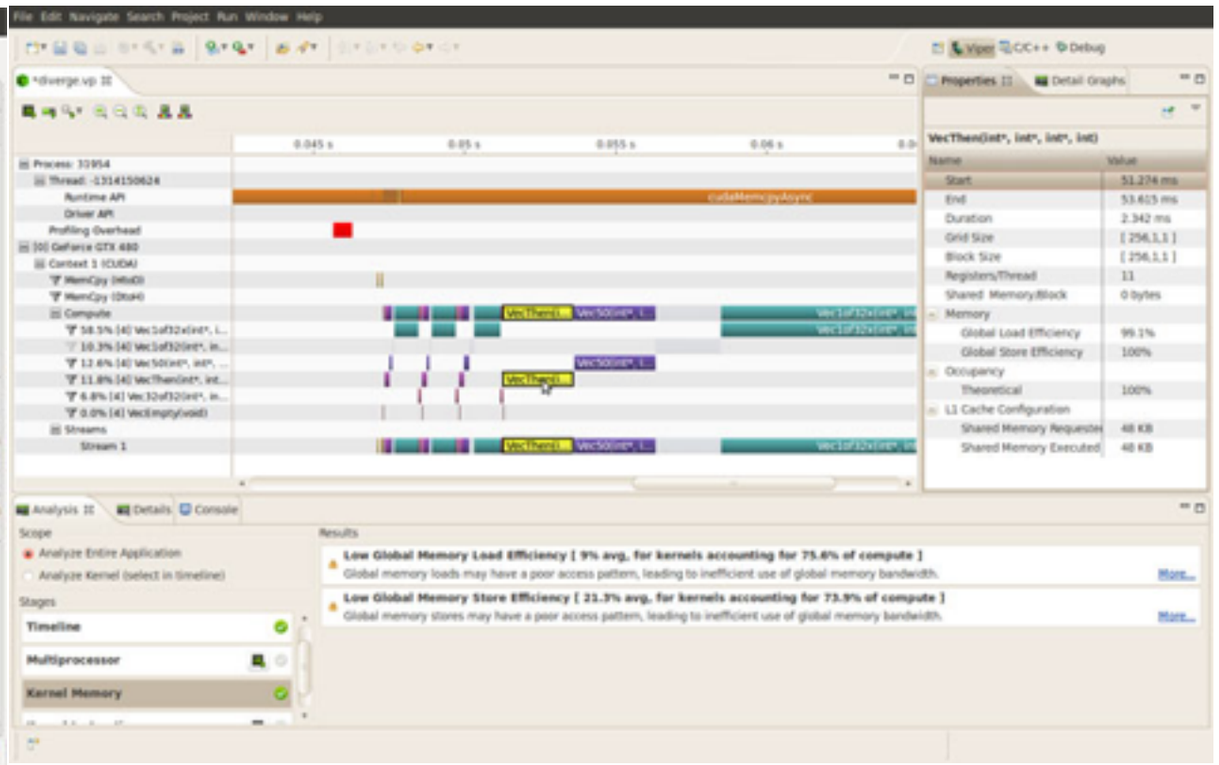
—> needs an environment to program caffe? use nsight

- nsight automatically comes with CUDA, in the terminal hit “nsight”

For this nsight eclipse edition, it supports nearly all we need:

- an editor with highlight and function switches
- debug c++ code and CUDA code
- profile your code





Protobuf

- understanding protobuf is very important to develop your own code on caffe
- protobuf is used to define data structure for multiple programming languages

```
message student {
  string name = 3;
    int ID = 2;}
```

```
    student mary;
mary.set_name("mary");
```

- the protobuf compiler can compile code into c++ .o file and .h headers
- using these structure in C++ is just like other class you defined in C++
- protobuf provide get_ set_ has_ function like has_name()
- protobuf complier can also compile the code for java, python

Protobuf — a example

caffe reads `solver.prototxt` into a SolverParameter object

protobuf definition

```
message SolverParameter {
  optional string train_net = 1; // The pro
  optional string test_net = 2; // The prot
  // The number of iterations for each test
  optional int32 test_iter = 3 [default = 0
  // The number of iterations between two t
  optional int32 test_interval = 4 [default
  optional bool test_compute_loss = 19 [def
  optional float base_lr = 5; // The base l
  optional float base_flip = 21; // The bas
  // the number of iterations between displ
  // will be displayed.
  optional int32 display = 6;
  optional int32 max_iter = 7; // the maxim
  optional string lr_policy = 8; // The lea
  optional float lr_gamma = 9; // The param
  optional float lr_power = 10; // The para
```

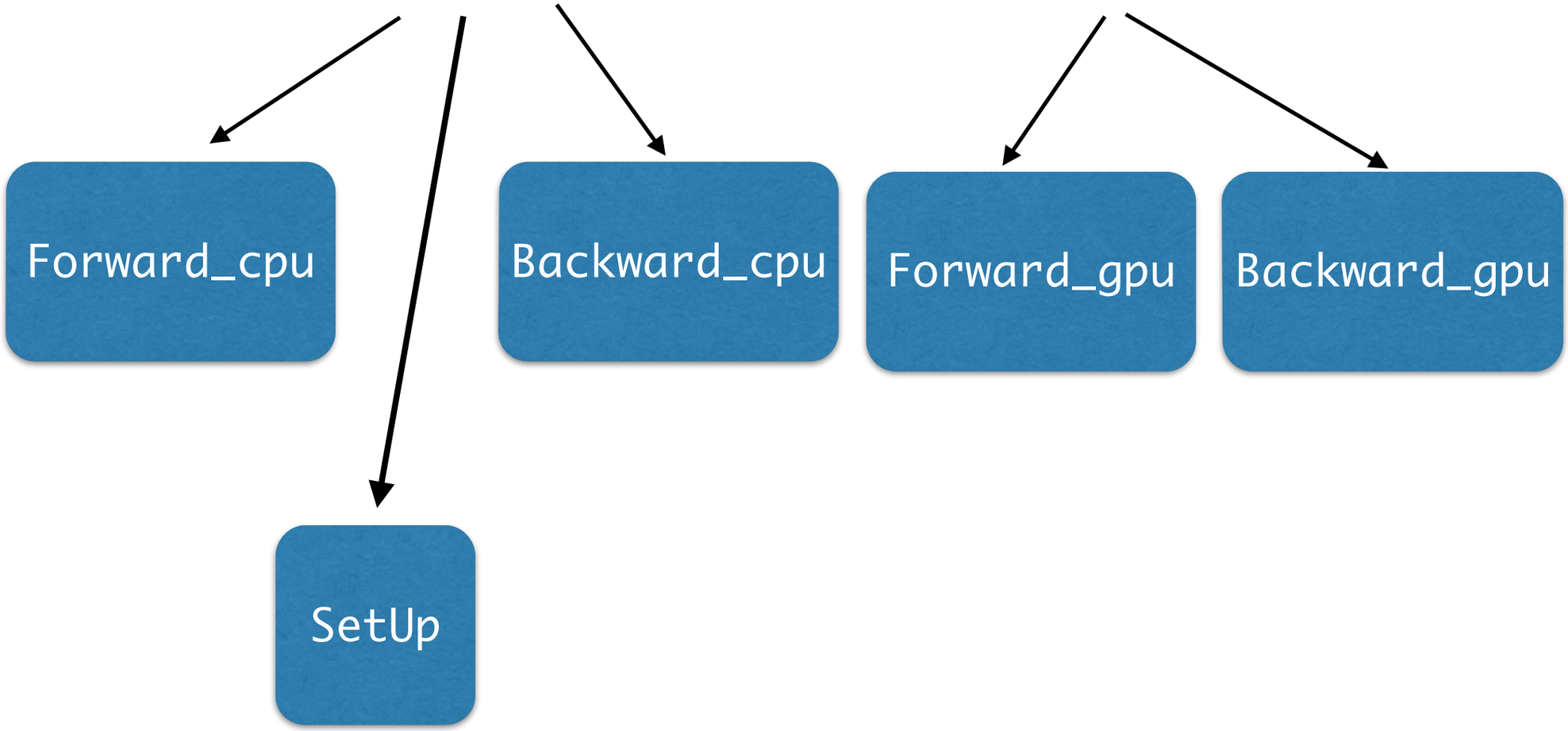
solver.prototxt

```
# The train/test net protocol buffer definition
train_net: “examples/mnist/lenet_train.prototxt"
test_net: "examples/mnist/lenet_test.prototxt"
# test_iter specifies how many forward passes the
# In the case of MNIST, we have test batch size 1
# 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
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
```

Adding layers

`$CAFFE/src/layers`

implement `xx_layer.cpp` and `xx_layer.cu`



Adding layers

show `inner_product.cpp` and `inner_product.cu`

tuning CNN

a few tips

- Our Goal: fitting the data as much as possible —> making the training cost as small as possible.
- Things that we could tune:
  - learning rate: large learning rate would cause the the cost go bigger and finally go to NaN.
  - Parameter Initialization: Bad initialization would give no gradient over parameters —> no learning occurs.
- How to tune those parameters:
  - monitor the testing cost after each several iterations.
  - monitor the gradient and the value of model parameters (abs mean of each layer).