Tutorial 7
Introduction to Caffe

Kai KANG (kkang@ee.cuhk.edu.hk)

Outline

- General Introduction
  - What is Caffe?
  - Why using Caffe?
  - How to get started?
- Caffe in details
  - Model and protocol buffers
  - Forward and backward computations
  - Loss and supervision
  - Solver and optimizations
  - Data and database
  - Interfaces and tools

What is Caffe?

- Open source deep learning framework maintained by Berkeley Vision and Learning Center (BVLC)
- Created by Yangqing Jia as an improved version of DeCaf
- Mainly written in C++ and CUDA C with Python and Matlab interfaces
- Tools, reference models and sample recipes included

Why Using Caffe? (Official)

- Expression: models and optimizations are defined as plaintext schemas instead of code.

- Speed: for research and industry alike speed is crucial for state-of-the-art models and massive data.

- Modularity: new tasks and settings require flexibility and extension.

- Openness: scientific and applied progress call for common code, reference models, and reproducibility.

- Community: academic research, startup prototypes, and industrial applications all share strength by joint discussion and development in a BSD-2 project.

Why We Use Caffe?

- Good starting point
- Reliability, especially for large scale problem
- Speed
- Popularity

How to Get Started?

- Official website (http://caffe.berkeleyvision.org)
- Download from the GitHub page (https://github.com/BVLC/caffe)
- Follow the installation instructions to compile and test (http://caffe.berkeleyvision.org/installation.html)
- Try the tutorials and reference models (http://caffe.berkeleyvision.org/tutorial/)
- Look through the detailed API documentations (http://caffe.berkeleyvision.org/doxygen/annotated.html)

Model and Protocol Buffers

What is a Network?

- Text representations: human-readable network configurations

- Parameter representations: trained network parameters

- Class representations: network data and behavior implementations

What is a Caffe net?

```
name: "CaffeNet"
layers {
  name: "data"
  type: IMAGE_DATA
  ...
}
layers {
  name: "conv1"
  type: CONVOLUTION
  ...
}

...

layers {
  name: "loss"
  type: SOFTMAX_LOSS
  ...
}
```

Text Representations

⇩

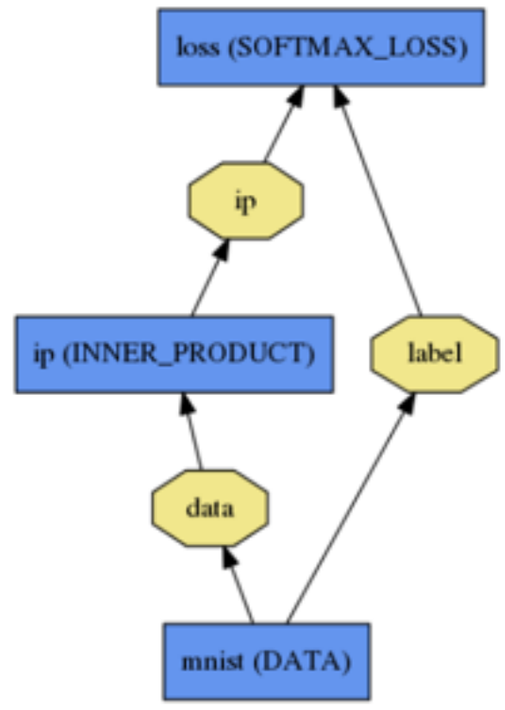
Net Definitions

```
message NetParameter {
  optional string name = 1;
  repeated LayerParameter layers =
2;
  repeated string input = 3;
  repeated int32 input_dim = 4;
  optional bool force_backward = 5
[default = false];
  optional NetState state = 6;
}
```

Parameter Representations

⇩

Data Structure



Class Representations

⇩

Implementation

Protocol Buffers

- Google open-source project
- Represent structured data
- Support serialization and de-serialization
- Include C++, Python and Java interfaces
- Compilers compile protocol buffers into classes to access the data

Example - Blobs

- A Blob is a wrapper over the actual data in Caffe
- It's a 4D tensor with dimensions of (num, channels, height and width)
- It can represent data, results and network parameters
    - Data: 20 * 3 * 256 * 256
    - Results: 20 * 96 * 128 * 128
    - Convolution kernels: 96 * 3 * 7 * 7
    - Fully-connected vector: 4096 * 128 * 1 * 1
- Blobs store results of both forward and backward propagations

Example - Blobs

Protocol Buffers definitions

```
message BlobProto {
  optional int32 num = 1 [default = 0];
  optional int32 channels = 2 [default = 0];
  optional int32 height = 3 [default = 0];
  optional int32 width = 4 [default = 0];
  repeated float data = 5 [packed = true];
  repeated float diff = 6 [packed = true];
}
```

Represent multiple blobs

```
message BlobProtoVector {
  repeated BlobProto blobs = 1;
}
```

Example - Blobs

C++ class definitions

```
class Blob {
 public:
  Blob()
       : data_(), diff_(), num_(0), channels_(0), height_(0), width_(0),
       count_(0), capacity_(0) {}
  ...
  inline int num() const { return num_; }
  inline int channels() const { return channels_; }
  inline int height() const { return height_; }
  inline int width() const { return width_; }
  inline int count() const { return count_; }
  ...
  const Dtype* cpu_data() const;
  void set_cpu_data(Dtype* data);
  const Dtype* gpu_data() const;
  const Dtype* cpu_diff() const;
  const Dtype* gpu_diff() const;
  Dtype* mutable_cpu_data();
  Dtype* mutable_gpu_data();
  Dtype* mutable_cpu_diff();
  Dtype* mutable_gpu_diff();
  ...
};  // class Blob
```

Example - Layers

Protocol Buffers

```
message LayerParameter {
  repeated string bottom = 2; // the name of the bottom blobs
  repeated string top = 3; // the name of the top blobs
  optional string name = 4; // the layer name
  repeated NetStateRule include = 32;
  repeated NetStateRule exclude = 33;
  enum LayerType {
    ...
    CONVOLUTION = 4;
    ...
  }
  optional LayerType type = 5; // the layer type from the enum above

  repeated BlobProto blobs = 6;
  repeated float blobs_lr = 7;
  repeated float weight_decay = 8;
  repeated float loss_weight = 35;
  ...
  optional ContrastiveLossParameter contrastive_loss_param = 40;
  ...
  optional TransformationParameter transform_param = 36;

}
```

Example - Layers

- C++ class implementations
  - Setup: initialize the layer and connections
  - Forward: given input from bottoms compute outputs and pass to tops
  - Backward: given gradients wrt to top outputs, compute gradients wrt to input and sent to bottom. A layer with parameters computes gradients wrt to its parameters and store then internally
- A layer can have both CPU and GPU implementations for forward and backward propagations

Caffe Net

```
name: "CaffeNet"
layers {
  name: "data"
  type: IMAGE_DATA
  ...
}
layers {
  name: "conv1"
  type: CONVOLUTION
  ...
}

...

layers {
  name: "loss"
  type: SOFTMAX_LOSS
  ...
}
```

Text Representations

⇩

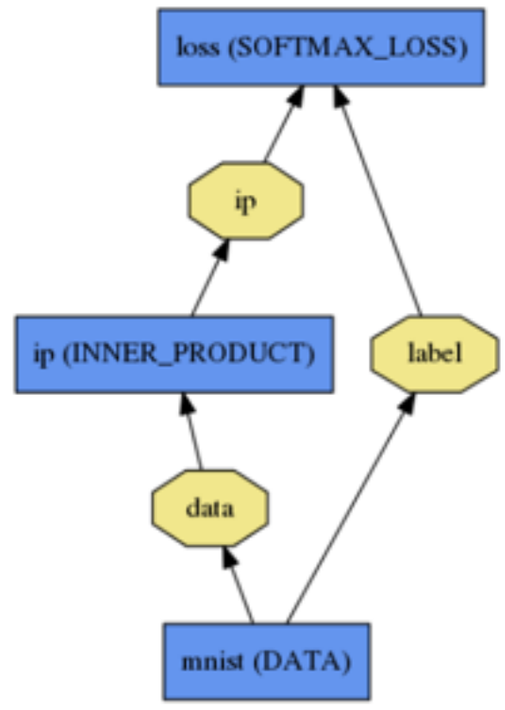
Net Definitions

```
message NetParameter {
  optional string name = 1;
  repeated LayerParameter layers =
2;
  repeated string input = 3;
  repeated int32 input_dim = 4;
  optional bool force_backward = 5
[default = false];
  optional NetState state = 6;
}
```

Parameter Representations

⇩

Data Structure



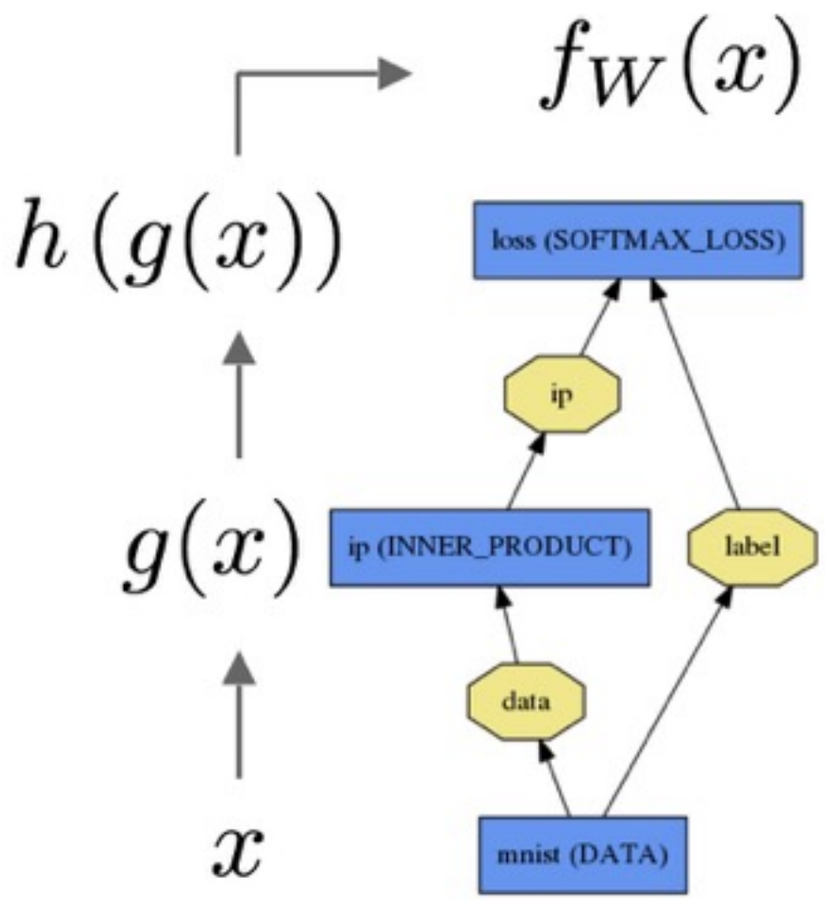
Class Representations

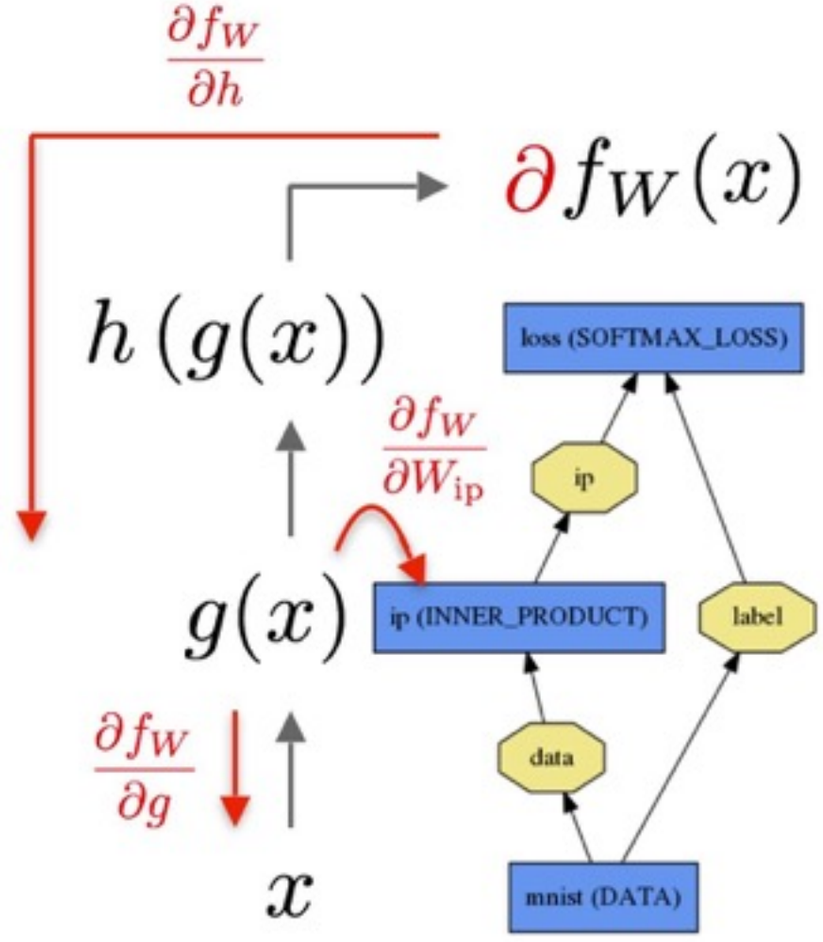
⇩

Implementation

Forward and Backward Propagation

Forward and Backward






```
Net::Forward();
Layer::Forward();
```

```
Net::Backward();
Layer::Backward();
```

Forward and Backward

- Each layer has CPU and GPU implementations for forward and backward:
  - Forward_cpu(), Forward_gpu()
  - Backward_cpu(), Backward_gpu()
- Solver calls forward first to get output and loss, then calls backward to calculate gradients of the model and update parameters

Loss and Supervision

Loss and Supervision

- Caffe is mainly used in supervised learning
- A loss function is the target function that we want to minimize
- Built-in loss functions:
    - Euclidean loss
    - Hinge loss
    - Multinomial logistic regression loss
    - Sigmoid cross entropy loss
    - Softmax loss
    - …
- Multiple loss functions for multitasking

Loss and Supervision

Single loss function

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

Multiple loss functions with weights

```
layers {
  name: "loss_softmax"
  type: SOFTMAX_LOSS
  bottom: "pred"
  bottom: "label"
  top: "loss_softmax"
  loss_weight: 1
}
```

```
layers {
  name: "loss_eclidean"
  type: ECLIDEAN_LOSS
  bottom: "pred"
  bottom: "label"
  top: "loss_eclidean"
  loss_weight: 0.5
}
```

Caffe sums up all losses with non-zero loss_weight

Solver and optimizations

Solver and optimizations

- Solver
  - Creates training and validation networks
  - Iteratively calls forward/backward and updates parameters
  - Evaluates validation networks periodically
  - Snapshots model and solver states periodically

Solver and optimizations

Solver example

```
net: "examples/hdf5_classification/train_val.prototxt"
test_iter: 1000
test_interval: 1000
base_lr: 0.01
lr_policy: "step"
gamma: 0.1
stepsize: 5000
display: 1000
max_iter: 10000
momentum: 0.9
weight_decay: 0.0005
snapshot: 10000
snapshot_prefix: "examples/hdf5_classification/data/train"
solver_mode: GPU
```

Data and Database

Data and Database

- Data are represented in Caffe as Blobs
- Data layers provide data to the network
    - Data can also represented in Protocol Buffers
        - Structured data are serialized in binary format and stored in databases
- Databases
    - LevelDB
    - LMDB
    - HDF5
    - …

Data Layers

- Tops and bottoms: multiple tops (data, label, …), no bottoms
- Transformations: scaling, extracting mean, cropping, mirroring
- Prefetching: loading data in another thread
- Built-in data layers: DataLayer, ImageDataLayer, WindowDataLayer, HDF5DataLayer,…
- Multiple data layers or custom data layers for specific applications

Example - DataLayer

```
layers {
  name: "data"
  type: DATA
  top: "data"
  top: "label"
  data_param {
    source: "examples/imagenet/ilsvrc12_train_leveldb"
    batch_size: 256
  }
  transform_param {
    crop_size: 227
    mean_file: "data/ilsvrc12/imagenet_mean.binaryproto"
    mirror: true
  }
  include: { phase: TRAIN }
}
```

Example - ImageDataLayer

```
layers {
  name: "data"
  type: IMAGE_DATA
  top: "data"
  top: "label"
  image_data_param {
    source: "examples/_temp/file_list.txt"
    batch_size: 50
    new_height: 256
    new_width: 256
  }
  transform_param {
    crop_size: 227
    mean_file: "data/ilsvrc12/imagenet_mean.binaryproto"
    mirror: false
  }
}
```

Interfaces and Tools

Interfaces

- Command-line interfaces
- Python interfaces
- Matlab interfaces

Command-line Interfaces

Training

```
# 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
```

Fine-tuning

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

Testing

```
# testing LeNet in lenet_train_test.prototxt
caffe test -model examples/mnist/lenet_train_test.prototxt \
-weights examples/mnist/lenet_iter_10000.caffemodel -gpu 0 \
-iterations 100
```

Tools

- convert_imageset.bin: convert image datasets
- compute_image_mean.bin: compute mean values
- extract_features.bin: extract features using trained models
- parse_log.sh: parse training log files
- …