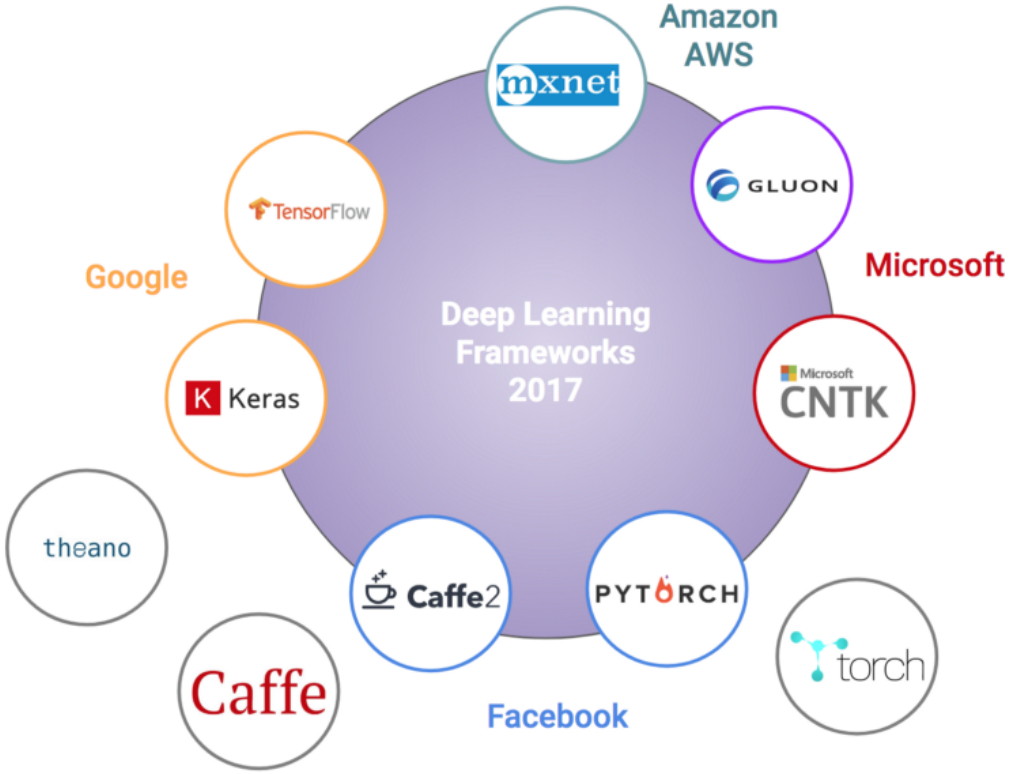
Deep Learning Framework

Yunkai Li
Megvii(Face++) Researcher
liyunkai@megvii.com
Apr, 2018

Outline

- Framework Survey
  - Framework components
- Caffe
  - Structure
  - How to define a model
  - How to train a model
- PyTorch
  - Structure
  - Train a model
  - Some key Parts

Framework Survey



README.md

ONNX

Linux	Windows
build passing	build passing

Open Neural Network Exchange (ONNX) is the first step toward an open ecosystem that empowers AI developers to choose the right tools as their project evolves. ONNX provides an open source format for AI models. It defines an extensible computation graph model, as well as definitions of built-in operators and standard data types. Initially we focus on the capabilities needed for inferencing (evaluation).

Caffe2, PyTorch, Microsoft Cognitive Toolkit, Apache MXNet and other tools are developing ONNX support. Enabling interoperability between different frameworks and streamlining the path from research to production will increase the speed of innovation in the AI community. We are an early stage and we invite the community to submit feedback and help us further evolve ONNX.

Software	Creator	Software license[a]	Open source	Platform	Written in	Interface	OpenMP support	OpenCL support	CUDA support	Automatic differentiation [1]	Has pretrained models	Recurrent nets	Convolutional nets	RBM/ DBNs	Parallel execution (multi node)
Caffe	Berkeley Vision and Learning Center	BSD license	Yes	Linux, macOS, Windows[2]	C++	Python, MATLAB	Yes	Under development[3]	Yes	Yes	Yes[4]	Yes	Yes	No	?
Keras	François Chollet	MIT license	Yes	Linux, macOS, Windows	Python	Python, R	Only if using Theano as backend	Under development for the Theano backend (and on roadmap for the TensorFlow backend)	Yes	Yes	Yes[15]	Yes	Yes	Yes	Yes[16]
Apache MXNet	Apache Software Foundation	Apache 2.0	Yes	Linux, macOS, Windows, [3][4] AWS, Android,[5] iOS, JavaScript [6]	Small C++ core library	C++, Python, Julia, Matlab, JavaScript, Go, R, Scala, Perl	Yes	On roadmap[35]	Yes	Yes[36]	Yes[37]	Yes	Yes	Yes	Yes[38]
PyTorch	Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan	BSD license	Yes	Linux, macOS	Python, C , CUDA	Python	Yes		Yes	Yes	Yes	Yes	Yes		Yes
TensorFlow	Google Brain team	Apache 2.0	Yes	Linux, macOS, Windows, [7] Android	C++, Python, CUDA	Python (Keras), C/ C++, Java, Go, R[8]	No	On roadmap[41] but already with SYCL[42] support	Yes	Yes[43]	Yes[44]	Yes	Yes	Yes	Yes
Theano	Université de Montréal	BSD license	Yes	Cross-platform	Python	Python (Keras)	Yes	Under development[45]	Yes	Yes[46][47]	Through Lasagne's model zoo[48]	Yes	Yes	Yes	Yes[49]
Torch	Ronan Collobert, Koray Kavukcuoglu, Clement Farabet	BSD license	Yes	Linux, macOS, Windows, [9] Android, [10] iOS	C, Lua	Lua, LuaJIT,[11] C, utility library for C++/ OpenCL[12]	Yes	Third party implementations[54][55]	Yes[56][57]	Through Twitter's Autograd[58]	Yes[59]	Yes	Yes	Yes	Yes[60]

Framework Components

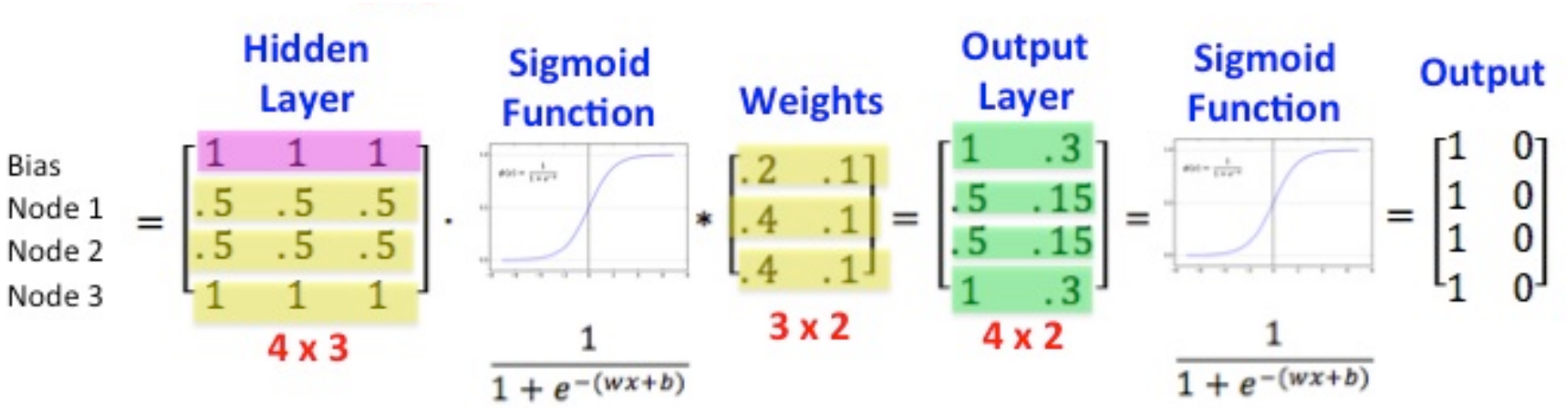
- Tensor
- Operator
- Computation Graph
  - Static Declaration and Dynamic Declaration
- Auto-differentiation tools
- BLAS / cuBLAS and cuDNN extensions

Tensor

- At the heart of the framework is the tensor object.
- A tensor is a generalization of a matrix to n-dimensions (think numpy’ s ndarrays).
- An easy way to understand tensors is to consider them as N-D Arrays.

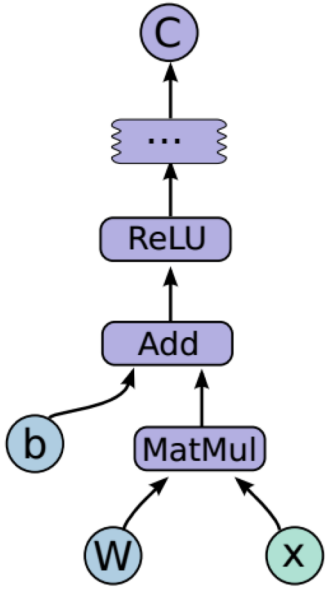
Operator

- A neural network can be considered as a series of Operations performed on an input tensor to give an output.



Computation Graph and optimization

- A Computation Graph is basically an object that contains links to the instances of various Ops and the relations between which operation takes the output of which operation as well as additional information.



Static Vs Dynamic

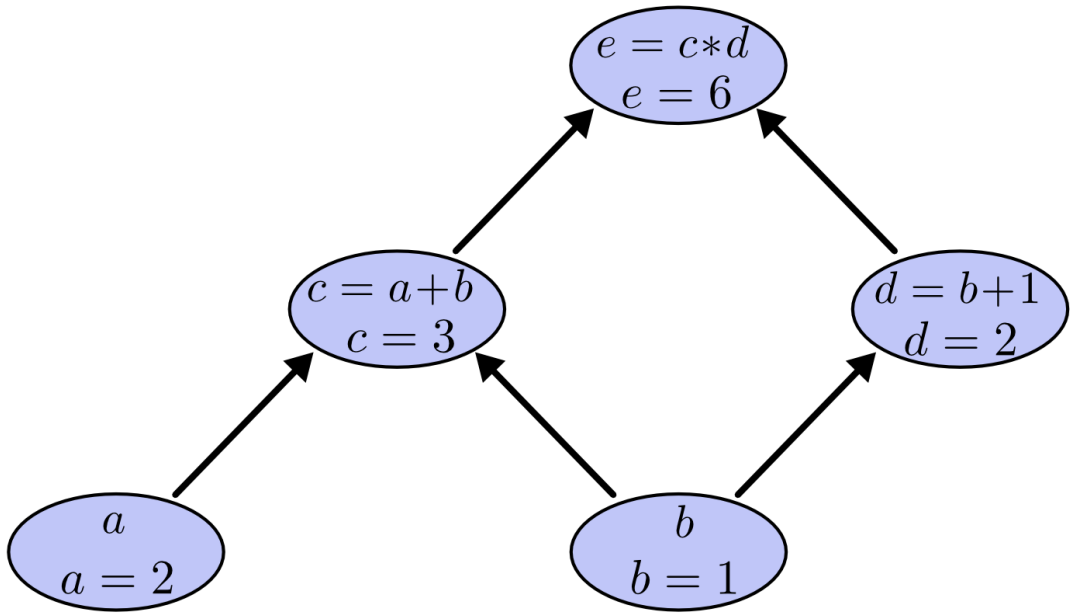
- Static declaration
  - Phase 1: define an architecture (maybe with some primitive flow control like loops and conditionals)
  - Phase 2: run a bunch of data through it to train the model and/or make predictions

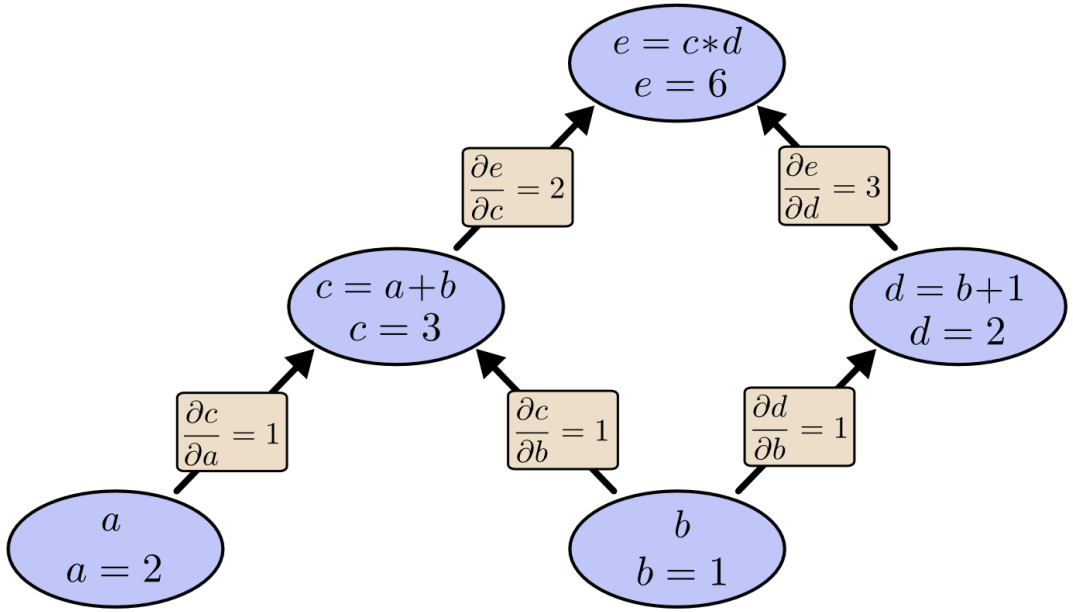
- Dynamic declaration
  - Graph is defined implicitly (e.g., using operator overloading) as the forward computation is executed

Auto-differentiation tools

- Another benefit of having the computational graph is that calculating gradients used in the learning phase becomes modular and straightforward to compute.
- The chain rule that lets you calculate derivatives of composition of functions in a systematic way.

Auto-differentiation tools





BLAS / cuBLAS and cuDNN extensions

- The most time comsuming operation in deep learning is matrix multiplication
- Deep learning framework heavily depends on accelerating library
- By using these packages, you could gain significant speed-ups in your framework.

- CPU
  - Eigen, OpenBLAS, MKL
- GPU
  - cuBLAS, cnDNN

Caffe

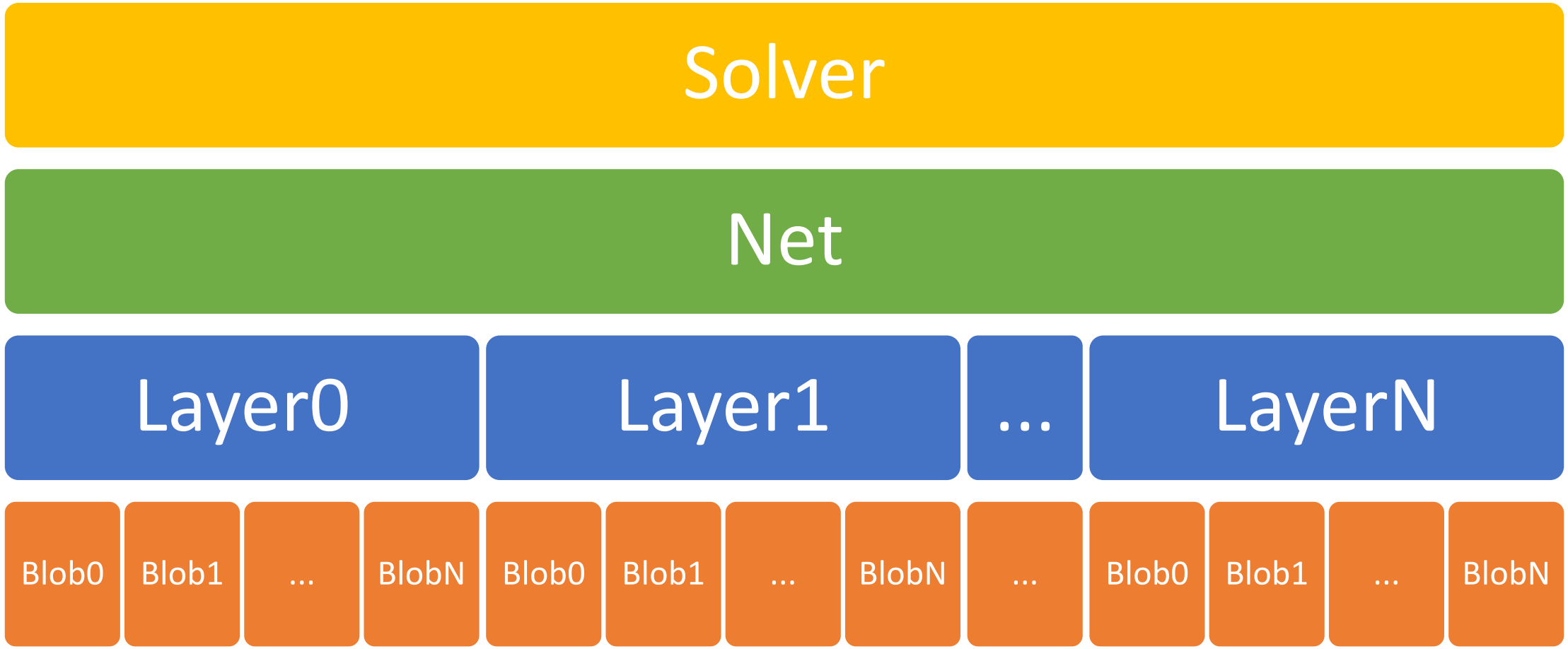
- What is caffe?
- Structure
  - Blob
  - Layer
  - Net
  - Solver
  - Protobuffer
- How to train a model

What is Caffe?

Convolution **A**rchitecture **F**or **F**eature **E**xtraction (**CAFFE**)

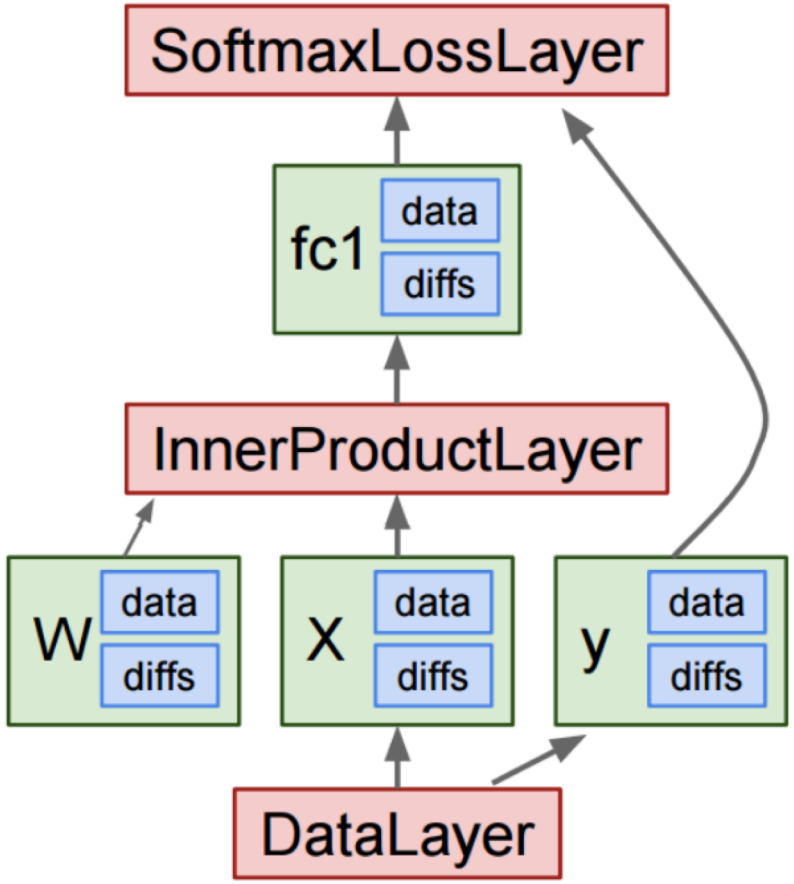
- Open framework, models, and examples for deep learning
- Pure C++ / CUDA architecture for deep learning
- Command line, Python, MATLAB interfaces
- Fast, well-tested code
- Tools, reference models, demos, and recipes
- Seamless switch between CPU and GPU

Structure

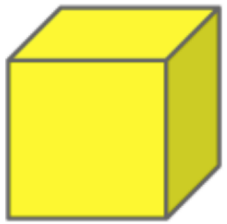

Solver
Net
Layer0
Layer1
...
LayerN
Blob0
Blob1
...
BlobN
Blob0
Blob1
...
BlobN
...
Blob0
Blob1
...
BlobN

Structure

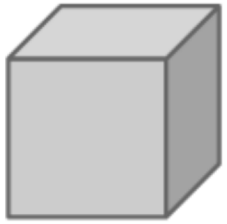
- **Blob**: Stores data and derivatives
- **Layer**: Transforms bottom blobs to top blobs
- **Net**: Many layers; computes gradients via Forward / Backward
- **Solver**: Uses gradients to update weights



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

Blobs

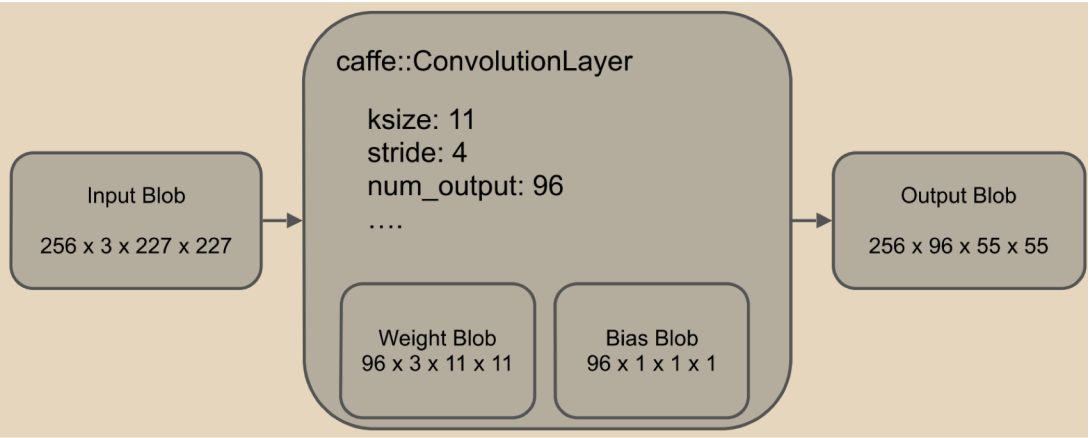
- **cpu_data(), mutable_cpu_data()**
  - host memory for CPU mode
- **gpu_data(), mutable_gpu_data()**
  - device memory for gpu mode
- **{cpu, gpu}_diff(), mutable_{cpu, gpu}_diff()**
  - derivative counterparts to data methods
  - easy access to data and diff in forward / backward

```
class SyncedMemory {
 public:
  SyncedMemory();
  explicit SyncedMemory(size_t size);
  ~SyncedMemory();
  const void* cpu_data();
  void set_cpu_data(void* data);
  const void* gpu_data();
  void set_gpu_data(void* data);
  void* mutable_cpu_data();
  void* mutable_gpu_data();
  enum SyncedHead { UNINITIALIZED, HEAD_AT_CPU, HEAD_AT_GPU, SYNCED };
  SyncedHead head() { return head_; }
  size_t size() { return size_; }

#ifndef CPU_ONLY
  void async_gpu_push(const cudaStream_t& stream);
#endif

 private:
  void check_device();

  void to_cpu();
  void to_gpu();
  void* cpu_ptr_;
  void* gpu_ptr_;
  size_t size_;
  SyncedHead head_;
  bool own_cpu_data_;
  bool cpu_malloc_use_cuda_;
  bool own_gpu_data_;
  int device_;

  DISABLE_COPY_AND_ASSIGN(SyncedMemory);
};  // class SyncedMemory

}  // namespace caffe
```

set/get data from xpu

switch device cpu/gpu

Layers



- DataLoader
  - data_layer
  - image_data_layer
- Basic Operations
  - conv_layer
  - batch_norm_layer
  - pooling_layer
- Activation
  - relu_layer
  - eltwise_layer
- Loss
  - softmax_loss_layer
  - contrastive_loss_layer

Layers

```
template <typename Dtype>
class InnerProductLayer : public Layer<Dtype> {
 public:
  explicit InnerProductLayer(const LayerParameter& param)
      : Layer<Dtype>(param) {}
  virtual void LayerSetUp(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top);
  virtual void Reshape(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top);

  virtual inline const char* type() const { return "InnerProduct"; }
  virtual inline int ExactNumBottomBlobs() const { return 1; }
  virtual inline int ExactNumTopBlobs() const { return 1; }

 protected:
  virtual void Forward_cpu(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top);
  virtual void Forward_gpu(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top);
  virtual void Backward_cpu(const vector<Blob<Dtype>*>& top,
      const vector<bool>& propagate_down, const vector<Blob<Dtype>*>& bottom);
  virtual void Backward_gpu(const vector<Blob<Dtype>*>& top,
      const vector<bool>& propagate_down, const vector<Blob<Dtype>*>& bottom);

  int M_;
  int K_;
  int N_;
  bool bias_term_;
  Blob<Dtype> bias_multiplier_;
  bool transpose_;  ///< if true, assume transposed weights
};

}  // namespace caffe
```

layer setup

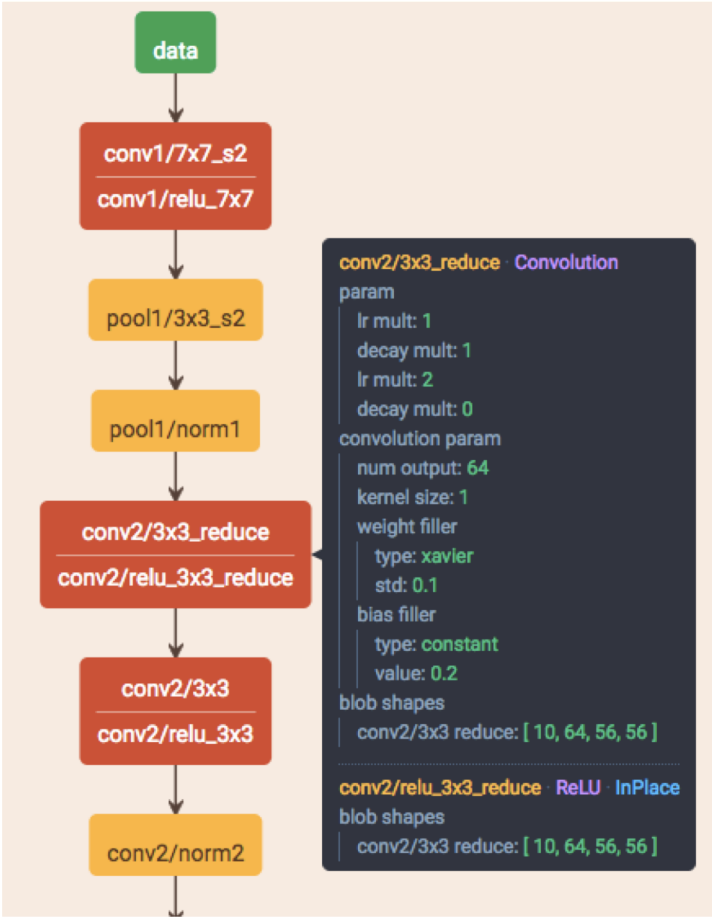
layer input check

- Layer.hpp
- inline Dtype **Forward**();
- inline void **Backward**();
- virtual void **Forward_{cpu, gpu}**() = 0;
- virtual void **Backward_{cpu, gpu}**() = 0;

```
void SetUp() {
    CheckBlobCounts(bottom, top);
    LayerSetUp(bottom, top);
    Reshape(bottom, top);
    SetLossWeights(top);
  }
```

Net

- A DAG of layers and the blobs that connect them
- Caffe creates and checks the net from a definition file (more later)
- Exposes Forward / Backward methods



Solver

- Scaffolds the optimization bookkeeping and creates the training network for learning and test network(s) for evaluation
- Calls Forward / Backward and updates net parameters
- Periodically evaluates model on the test network(s)
- Snapshots model and solver state

- Solvers available:
  - SGD
  - AdaDelta
  - AdaGrad
  - Adam
  - Nesterov
  - RMSprop

Protobuffer

- Net / Layer / Solver / parameters are messages defined in .prototxt files
- Available message types defined in ./src/caffe/proto/caffe.proto

```
message NetParameter {
  optional string name = 1; // consider giving the network a name
  // DEPRECATED. See InputParameter. The input blobs to the network.
  repeated string input = 3;
  // DEPRECATED. See InputParameter. The shape of the input blobs.
  repeated BlobShape input_shape = 8;

  // 4D input dimensions -- deprecated.  Use "input_shape" instead.
  // If specified, for each input blob there should be four
  // values specifying the num, channels, height and width of the input blob.
  // Thus, there should be a total of (4 * #input) numbers.
  repeated int32 input_dim = 4;

  // Whether the network will force every layer to carry out backward operation.
  // If set False, then whether to carry out backward is determined
  // automatically according to the net structure and learning rates.
  optional bool force_backward = 5 [default = false];
  // The current "state" of the network, including the phase, level, and stage.
  // Some layers may be included/excluded depending on this state and the states
  // specified in the layers' include and exclude fields.
  optional NetState state = 6;

  // Print debugging information about results while running Net::Forward,
  // Net::Backward, and Net::Update.
  optional bool debug_info = 7 [default = false];

  // The layers that make up the net.  Each of their configurations, including
  // connectivity and behavior, is specified as a LayerParameter.
  repeated LayerParameter layer = 100;  // ID 100 so layers are printed last.

  // DEPRECATED: use 'layer' instead.
  repeated V1LayerParameter layers = 2;
}
```

How to train a model?

- Define Model
- Define Solver
- Run
- Gradient Check

Face++ 旷视

Define model

```
name: "AlexNet"
layer {
  name: "data"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    mirror: true
    crop_size: 227
    mean_file: "data/ilsvrc12/imagenet_mean.binaryproto"
  }
  data_param {
    source: "examples/imagenet/ilsvrc12_train_lmdb"
    batch_size: 256
    backend: LMDB
  }
```

layer type

blobs

data preprocessing

data type

```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 96
    kernel_size: 11
    stride: 4
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
```

learning rate and weight decay

params of conv layer

param blob init

Define solver

```
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 config file

testing scheme

learning policy

snapshot config

Face++ 旷视

Run

```
16076 I1105 15:17:42.207845 37642 solver.cpp:218] Iteration 0 (0 iter/s, 19.4624s/20 iters), loss = 4.64535
16077 I1105 15:17:42.207979 37642 solver.cpp:237]     Train net output #0: loss = 4.64535 (* 1 = 4.64535 loss)
16078 I1105 15:17:42.208039 37642 sgd_solver.cpp:105] Iteration 0, lr = 0.001
16079 I1105 15:17:57.559537 37642 solver.cpp:218] Iteration 20 (1.3028 iter/s, 15.3516s/20 iters), loss = 4.57324
16080 I1105 15:17:57.559942 37642 solver.cpp:237]     Train net output #0: loss = 4.64743 (* 1 = 4.64743 loss)
16081 I1105 15:17:57.559981 37642 sgd_solver.cpp:105] Iteration 20, lr = 0.001
16082 I1105 15:18:08.101182 37642 solver.cpp:218] Iteration 40 (1.89731 iter/s, 10.5412s/20 iters), loss = 4.42709
16083 I1105 15:18:08.101240 37642 solver.cpp:237]     Train net output #0: loss = 4.52784 (* 1 = 4.52784 loss)
16084 I1105 15:18:08.101276 37642 sgd_solver.cpp:105] Iteration 40, lr = 0.001
16085 I1105 15:18:18.704746 37642 solver.cpp:218] Iteration 60 (1.88618 iter/s, 10.6035s/20 iters), loss = 4.21881
16086 I1105 15:18:18.704798 37642 solver.cpp:237]     Train net output #0: loss = 4.19613 (* 1 = 4.19613 loss)
16087 I1105 15:18:18.704839 37642 sgd_solver.cpp:105] Iteration 60, lr = 0.001
16088 I1105 15:18:29.363610 37642 solver.cpp:218] Iteration 80 (1.87639 iter/s, 10.6588s/20 iters), loss = 4.12232
16089 I1105 15:18:29.363962 37642 solver.cpp:237]     Train net output #0: loss = 4.23099 (* 1 = 4.23099 loss)
16090 I1105 15:18:29.367856 37642 sgd_solver.cpp:105] Iteration 80, lr = 0.001
16091 I1105 15:18:39.861603 37642 solver.cpp:218] Iteration 100 (1.90519 iter/s, 10.4976s/20 iters), loss = 3.7957
16092 I1105 15:18:39.861654 37642 solver.cpp:237]     Train net output #0: loss = 3.34173 (* 1 = 3.34173 loss)
16093 I1105 15:18:39.867529 37642 sgd_solver.cpp:105] Iteration 100, lr = 0.001
16094 I1105 15:18:50.422592 37642 solver.cpp:218] Iteration 120 (1.89378 iter/s, 10.5609s/20 iters), loss = 3.70076
16095 I1105 15:18:50.422655 37642 solver.cpp:237]     Train net output #0: loss = 2.99149 (* 1 = 2.99149 loss)
16096 I1105 15:18:50.422672 37642 sgd_solver.cpp:105] Iteration 120, lr = 0.001
16097 I1105 15:19:00.914120 37642 solver.cpp:218] Iteration 140 (1.90632 iter/s, 10.4914s/20 iters), loss = 3.5228
16098 I1105 15:19:00.914424 37642 solver.cpp:237]     Train net output #0: loss = 3.05207 (* 1 = 3.05207 loss)
16099 I1105 15:19:00.919710 37642 sgd_solver.cpp:105] Iteration 140, lr = 0.001
16100 I1105 15:19:11.310906 37642 solver.cpp:218] Iteration 160 (1.92374 iter/s, 10.3964s/20 iters), loss = 3.25225
16101 I1105 15:19:11.311002 37642 solver.cpp:237]     Train net output #0: loss = 3.15632 (* 1 = 3.15632 loss)
16102 I1105 15:19:11.311029 37642 sgd_solver.cpp:105] Iteration 160, lr = 0.001
16103 I1105 15:19:21.796437 37642 solver.cpp:218] Iteration 180 (1.90742 iter/s, 10.4854s/20 iters), loss = 3.11067
16104 I1105 15:19:21.796509 37642 solver.cpp:237]     Train net output #0: loss = 3.03842 (* 1 = 3.03842 loss)
16105 I1105 15:19:21.803426 37642 sgd_solver.cpp:105] Iteration 180, lr = 0.001
16106 I1105 15:19:32.237669 37642 solver.cpp:218] Iteration 200 (1.91551 iter/s, 10.4411s/20 iters), loss = 3.01716
16107 I1105 15:19:32.238009 37642 solver.cpp:237]     Train net output #0: loss = 3.15964 (* 1 = 3.15964 loss)
16108 I1105 15:19:32.238041 37642 sgd_solver.cpp:105] Iteration 200, lr = 0.001
16109 I1105 15:19:42.776481 37642 solver.cpp:218] Iteration 220 (1.89782 iter/s, 10.5384s/20 iters), loss = 2.71478
16110 I1105 15:19:42.776583 37642 solver.cpp:237]     Train net output #0: loss = 1.67323 (* 1 = 1.67323 loss)
16111 I1105 15:19:42.776610 37642 sgd_solver.cpp:105] Iteration 220, lr = 0.001
16112 I1105 15:19:53.307315 37642 solver.cpp:218] Iteration 240 (1.89922 iter/s, 10.5307s/20 iters), loss = 2.64442
16113 I1105 15:19:53.307368 37642 solver.cpp:237]     Train net output #0: loss = 2.25393 (* 1 = 2.25393 loss)
16114 I1105 15:19:53.307410 37642 sgd_solver.cpp:105] Iteration 240, lr = 0.001
16115 I1105 15:20:03.903152 37642 solver.cpp:218] Iteration 260 (1.88756 iter/s, 10.5957s/20 iters), loss = 2.55155
16116 I1105 15:20:03.903445 37642 solver.cpp:237]     Train net output #0: loss = 2.59057 (* 1 = 2.59057 loss)
16117 I1105 15:20:03.912786 37642 sgd_solver.cpp:105] Iteration 260, lr = 0.001
16118 I1105 15:20:14.349661 37642 solver.cpp:218] Iteration 280 (1.91458 iter/s, 10.4461s/20 iters), loss = 2.37144
16119 I1105 15:20:14.349723 37642 solver.cpp:237]     Train net output #0: loss = 2.51418 (* 1 = 2.51418 loss)
16120 I1105 15:20:14.349755 37642 sgd_solver.cpp:105] Iteration 280, lr = 0.001
16121 I1105 15:20:24.909660 37642 solver.cpp:218] Iteration 300 (1.89397 iter/s, 10.5599s/20 iters), loss = 2.39962
16122 I1105 15:20:24.909718 37642 solver.cpp:237]     Train net output #0: loss = 2.74425 (* 1 = 2.74425 loss)
16123 I1105 15:20:24.913538 37642 sgd_solver.cpp:105] Iteration 300, lr = 0.001
```

./build/tools/caffe train
-gpu 0,1,2,3
-solver alexnet_solver.prototxt
-weights pretrained.caffemodel

```
-rw-rw-r-- 1 liyunkai liyunkai 252M Dec 20 09:46 tsn_resnet152_rgb_iter_108000.solverstate
-rw-rw-r-- 1 liyunkai liyunkai 253M Dec 20 09:46 tsn_resnet152_rgb_iter_108000.caffemodel
-rw-rw-r-- 1 liyunkai liyunkai 252M Dec 20 08:24 tsn_resnet152_rgb_iter_100000.solverstate
-rw-rw-r-- 1 liyunkai liyunkai 253M Dec 20 08:24 tsn_resnet152_rgb_iter_100000.caffemodel
-rw-rw-r-- 1 liyunkai liyunkai 252M Dec 20 06:42 tsn_resnet152_rgb_iter_90000.solverstate
-rw-rw-r-- 1 liyunkai liyunkai 253M Dec 20 06:42 tsn_resnet152_rgb_iter_90000.caffemodel
-rw-rw-r-- 1 liyunkai liyunkai 252M Dec 20 04:56 tsn_resnet152_rgb_iter_80000.solverstate
-rw-rw-r-- 1 liyunkai liyunkai 253M Dec 20 04:56 tsn_resnet152_rgb_iter_80000.caffemodel
-rw-rw-r-- 1 liyunkai liyunkai 252M Dec 20 03:10 tsn_resnet152_rgb_iter_70000.solverstate
-rw-rw-r-- 1 liyunkai liyunkai 253M Dec 20 03:10 tsn_resnet152_rgb_iter_70000.caffemodel
-rw-rw-r-- 1 liyunkai liyunkai 252M Dec 20 01:23 tsn_resnet152_rgb_iter_60000.solverstate
-rw-rw-r-- 1 liyunkai liyunkai 253M Dec 20 01:23 tsn_resnet152_rgb_iter_60000.caffemodel
-rw-rw-r-- 1 liyunkai liyunkai 252M Dec 19 23:36 tsn_resnet152_rgb_iter_50000.solverstate
-rw-rw-r-- 1 liyunkai liyunkai 253M Dec 19 23:36 tsn_resnet152_rgb_iter_50000.caffemodel
-rw-rw-r-- 1 liyunkai liyunkai 252M Dec 19 21:47 tsn_resnet152_rgb_iter_40000.solverstate
-rw-rw-r-- 1 liyunkai liyunkai 253M Dec 19 21:47 tsn_resnet152_rgb_iter_40000.caffemodel
-rw-rw-r-- 1 liyunkai liyunkai 252M Dec 19 20:00 tsn_resnet152_rgb_iter_30000.solverstate
-rw-rw-r-- 1 liyunkai liyunkai 253M Dec 19 20:00 tsn_resnet152_rgb_iter_30000.caffemodel
-rw-rw-r-- 1 liyunkai liyunkai 252M Dec 19 18:13 tsn_resnet152_rgb_iter_20000.solverstate
-rw-rw-r-- 1 liyunkai liyunkai 253M Dec 19 18:13 tsn_resnet152_rgb_iter_20000.caffemodel
-rw-rw-r-- 1 liyunkai liyunkai 252M Dec 19 16:11 tsn_resnet152_rgb_iter_10000.solverstate
-rw-rw-r-- 1 liyunkai liyunkai 253M Dec 19 16:11 tsn_resnet152_rgb_iter_10000.caffemodel
```

Face++ 旷视

Gradient Check

```
231 TYPED_TEST(ConvolutionLayerTest, TestSimpleConvolution) {
232   typedef typename TypeParam::Dtype Dtype;
233   this->blob_bottom_vec_.push_back(this->blob_bottom_2_);
234   this->blob_top_vec_.push_back(this->blob_top_2_);
235   LayerParameter layer_param;
236   ConvolutionParameter* convolution_param =
237       layer_param.mutable_convolution_param();
238   convolution_param->add_kernel_size(3);
239   convolution_param->add_stride(2);
240   convolution_param->set_num_output(4);
241   convolution_param->mutable_weight_filler()->set_type("gaussian");
242   convolution_param->mutable_bias_filler()->set_type("constant");
243   convolution_param->mutable_bias_filler()->set_value(0.1);
244   shared_ptr<Layer<Dtype> > layer(
245       new ConvolutionLayer<Dtype>(layer_param));
246   layer->SetUp(this->blob_bottom_vec_, this->blob_top_vec_);
247   layer->Forward(this->blob_bottom_vec_, this->blob_top_vec_);
248   // Check against reference convolution.
249   const Dtype* top_data;
250   const Dtype* ref_top_data;
251   caffe_conv(this->blob_bottom_, convolution_param, layer->blobs(),
252       this->MakeReferenceTop(this->blob_top_));
253   top_data = this->blob_top_->cpu_data();
254   ref_top_data = this->ref_blob_top_->cpu_data();
255   for (int i = 0; i < this->blob_top_->count(); ++i) {
256     EXPECT_NEAR(top_data[i], ref_top_data[i], 1e-4);
257   }
258   caffe_conv(this->blob_bottom_2_, convolution_param, layer->blobs(),
259       this->MakeReferenceTop(this->blob_top_2_));
260   top_data = this->blob_top_2_->cpu_data();
261   ref_top_data = this->ref_blob_top_->cpu_data();
262   for (int i = 0; i < this->blob_top_->count(); ++i) {
263     EXPECT_NEAR(top_data[i], ref_top_data[i], 1e-4);
264   }
265 }
```

Gradient approximate

```
TYPED_TEST(ConvolutionLayerTest, TestGradient) {
  typedef typename TypeParam::Dtype Dtype;
  LayerParameter layer_param;
  ConvolutionParameter* convolution_param =
      layer_param.mutable_convolution_param();
  this->blob_bottom_vec_.push_back(this->blob_bottom_2_);
  this->blob_top_vec_.push_back(this->blob_top_2_);
  convolution_param->add_kernel_size(3);
  convolution_param->add_stride(2);
  convolution_param->set_num_output(2);
  convolution_param->mutable_weight_filler()->set_type("gaussian");
  convolution_param->mutable_bias_filler()->set_type("gaussian");
  ConvolutionLayer<Dtype> layer(layer_param);
  GradientChecker<Dtype> checker(1e-2, 1e-3);
  checker.CheckGradientExhaustive(&layer, this->blob_bottom_vec_,
      this->blob_top_vec_);
}
```

Gradient Check

- Definition of derivative

$$\frac{d}{d\theta}J(\theta) = \lim_{\epsilon \to 0} \frac{J(\theta + \epsilon) - J(\theta - \epsilon)}{2\epsilon}$$

- Numerically approximate
  - EPSILON set to be around 10e-4. Dont set extreme small, say 10e-10, which will lead to numeriacal roundoff errors

$$g(\theta) \approx \frac{J(\theta + \text{EPSILON}) - J(\theta - \text{EPSILON})}{2 \times \text{EPSILON}}$$

Face++ 旷视

PyTorch

- What is PyTorch?
- Structure
  - Tensor
    - NumpyBridge
  - Variable
  - Autograd
- Train a model
  - Linear Regression
  - Training Loop
- Some key Parts
  - nn.Sequential
  - nn.Module
  - hook
  - nn.functional
  - torch.optim
  - Dataloader

special thanks to xingyu liao

Face++ 旷视

PyTorch

- **What is PyTorch**?
- Structure
  - Tensor
    - NumpyBridge
  - Variable
  - Autograd
- Train a model
  - Linear Regression
  - Training Loop
- Some key Parts
  - nn.Sequential
  - nn.Module
  - hook
  - nn.functional
  - torch.optim
  - Dataloader

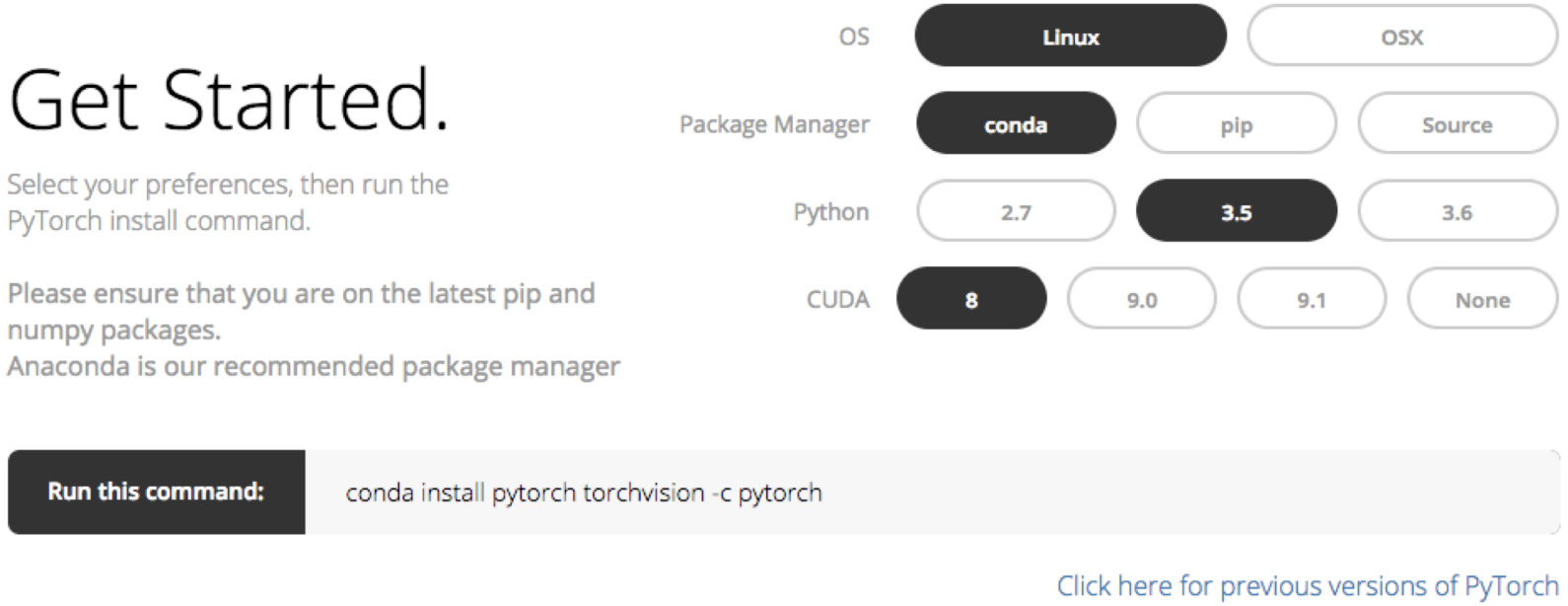
special thanks to xingyu liao

What is PyTorch?

PyTorch 由 Facebook 的 AI 研究团队开发，是一个以 python 优先的深度学习框架，在 GPU 加速基础上实现张量和动态神经网络

官网: pytorch.org

Github主页: https://github.com/pytorch/pytorch



Why PyTorch?

- More Pythonic (imperative)
  - Flexible
  - Intuitive and cleaner code
  - Easy to debug

- More Neural Networkic
  - Write code as the network works
  - forward/backward clear

Companies & Universities developing PyTorch

Face++ 旷视

TensorFlow vs PyTorch: while loop

tensorflow

```
import tensorflow as tf

first_counter = tf.constant(0)
second_counter = tf.constant(10)

def cond(first_counter, second_counter, *args):
    return first_counter < second_counter

def body(first_counter, second_counter):
    first_counter = tf.add(first_counter, 2)
    second_counter = tf.add(second_counter, 1)
    return first_counter, second_counter

c1, c2 = tf.while_loop(cond, body, [first_counter, second_counter])

with tf.Session() as sess:
    counter_1_res, counter_2_res = sess.run([c1, c2])

print(counter_1_res)
print(counter_2_res)
```

```
20
20
```

pytorch

```
import torch

first_counter = torch.Tensor([0])
second_counter = torch.Tensor([10])

while (first_counter < second_counter)[0]:
    first_counter += 2
    second_counter += 1

print(first_counter)
print(second_counter)
```

```
 20
[torch.FloatTensor of size 1]


 20
[torch.FloatTensor of size 1]
```

Face++ 旷视

Pytorch

- What is PyTorch?
- **Structure**
  - Tensor
    - NumpyBridge
  - Variable
  - Autograd
- Train a model
  - Linear Regression
  - Training Loop
- Some key Parts
  - nn.Sequential
  - nn.Module
  - hook
  - nn.functional
  - torch.optim
  - Dataloader

special thanks to xingyu liao

Tensor

Tensor 是 PyTorch 中最基本的操作单元，是一个能够在 GPU 上加速的高维数组

```
import torch
a = torch.Tensor(2, 3) # shape=(2, 3)
print(a)
```

```
 0.0000e+00 -0.0000e+00 -2.2023e-34
 2.0005e+00 -2.0914e-34 -3.6902e+19
[torch.FloatTensor of size 2x3]
```

Data type	CPU tensor	GPU tensor
32-bit floating point	`torch.FloatTensor`	`torch.cuda.FloatTensor`
64-bit floating point	`torch.DoubleTensor`	`torch.cuda.DoubleTensor`
16-bit floating point	`torch.HalfTensor`	`torch.cuda.HalfTensor`
8-bit integer (unsigned)	`torch.ByteTensor`	`torch.cuda.ByteTensor`
8-bit integer (signed)	`torch.CharTensor`	`torch.cuda.CharTensor`
16-bit integer (signed)	`torch.ShortTensor`	`torch.cuda.ShortTensor`
32-bit integer (signed)	`torch.IntTensor`	`torch.cuda.IntTensor`
64-bit integer (signed)	`torch.LongTensor`	`torch.cuda.LongTensor`

Face++ 旷视

Numpy Bridge

In [9]:
```
import numpy as np
x = np.ones((2, 3), dtype=np.float32)
y = torch.Tensor(2, 2)

# bridge between numpy and tensor
x_tensor = torch.from_numpy(x) # convert numpy.ndarray to tensor
print(x_tensor)

y_array = y.numpy() # convert tensor to numpy.ndarray
print(y_array)
```

```
 1  1  1
 1  1  1
[torch.FloatTensor of size 2x3]

[[ 0.0000000e+00 -2.5243549e-29]
 [-3.9187265e-25  2.5249709e-29]]
```

Cuda Tensor

In [10]:
```
x = torch.randn(2, 3)

x_gpu = x.cuda() # move cpu Tensor to gpu

x_cpu = x_gpu.cpu() # move gpu Tensor to cpu
```

Face++ 旷视

Variable

class **torch.autograd.Variable** [source]

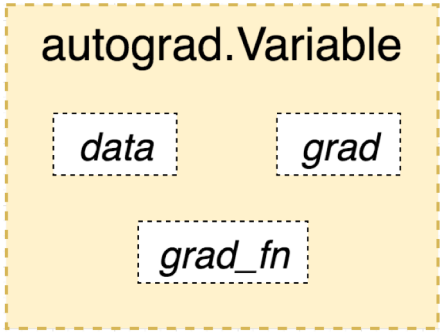
Wraps a tensor and records the operations applied to it.

Variable is a thin wrapper around a Tensor object, that also holds the gradient w.r.t. to it, and a reference to a function that created it. This reference allows retracing the whole chain of operations that created the data. If the Variable has been created by the user, its grad_fn will be `None` and we call such objects *leaf* Variables.

Since autograd only supports scalar valued function differentiation, grad size always matches the data size. Also, grad is normally only allocated for leaf variables, and will be always zero otherwise.

Variables:

- **data** – Wrapped tensor of any type.
- **grad** – Variable holding the gradient of type and location matching the `.data`. This attribute is lazily allocated and can't be reassigned.
- **requires_grad** – Boolean indicating whether the Variable has been created by a subgraph containing any Variable, that requires it. See Excluding subgraphs from backward for more details. Can be changed only on leaf Variables.
- **volatile** – Boolean indicating that the Variable should be used in inference mode, i.e. don't save the history. See Excluding subgraphs from backward for more details. Can be changed only on leaf Variables.
- **is_leaf** – Boolean indicating if the Variable is a graph leaf (i.e if it was created by the user).
- **grad_fn** – Gradient function graph trace.



- data: 保存 variable 中所含的tensor
- grad: 保存 variable 中的梯度
- requires_grad: 是否需要对其求导
- volatile: 为 inference 设计，构建在其上的计算图不会求导
- grad_fn: 记录 tensor 的操作历史，用来构建计算图

Autograd

torch.autograd.backward(*variables, grad_variables=None, retain_graph=None, create_graph=None, retain_variables=None*) [source]

- variables: 需要计算梯度的 variable
- grad_variables: 链式法则中的头梯度, 即 $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$ 中的 $\frac{\partial z}{\partial y}$ 如果 variables 是标量, 则可不声明
- retrain_graph: 反向传播是否需要缓存结果，用于多次反向传播梯度的累加
- create_graph: 对反向传播再次构建计算图，实现高阶导数

Autograd

```
x = Variable(torch.ones(2, 2), requires_grad=True)

# define compute graph
y = x + 2
z = y * y * 3
out = z.mean()

print('z: {}'.format(z))
print('out: {}'.format(out))

# autograd
out.backward()

print('x gradient: {}'.format(x.grad))
```

```
z: Variable containing:
 27  27
 27  27
[torch.FloatTensor of size 2x2]

out: Variable containing:
 27
[torch.FloatTensor of size 1]

x gradient: Variable containing:
 4.5000  4.5000
 4.5000  4.5000
[torch.FloatTensor of size 2x2]
```

```
m = Variable(torch.Tensor([2, 3]), requires_grad=True)
n = Variable(torch.Tensor(2))

# (n1, n2) = (m1 ^2, m2 ^3)
n[0] = m[0] ** 2
n[1] = m[1] ** 3

n.backward(torch.ones(2)) # 传入头梯度
print(m.grad)
```

```
Variable containing:
  4
 27
[torch.FloatTensor of size 2]
```

$$m.grad = (1 \times \frac{\partial n_1}{\partial m_1} + 1 \times \frac{\partial n_2}{\partial m_1}, 1 \times \frac{\partial n_1}{\partial m_2} + 1 \times \frac{\partial n_2}{\partial m_2})$$

More Autograd

```
x = Variable(torch.ones(2, 2), requires_grad=True)
y = x + 2
z = y * y * 3
out = z.mean()
out.backward()

print(z.grad)
```

```
None
```

反向传播的过程从根节点到叶子节点，非叶子节点的梯度在计算之后就会被清空，那么如何取得中间的梯度？

- autograd.grad
- hook

torch.autograd.grad

```
# torch.autograd.grad

x = Variable(torch.ones(2, 2), requires_grad=True)
y = x + 2
z = y * y * 3
out = z.mean()
torch.autograd.grad(out, z)
```

```
(Variable containing:
  0.2500  0.2500
  0.2500  0.2500
 [torch.FloatTensor of size 2x2],)
```

hook

```
# hook 是一个函数，输入梯度，没有返回值
def variable_hook(grad):
    print('z的梯度: {}'.format(grad))

x = Variable(torch.ones(2, 2), requires_grad=True)
y = x + 2
z = y * y * 3
out = z.mean()

# 注册 hook
hook_handle = z.register_hook(variable_hook)
out.backward()

# 除非你每次都要用hook，否则用完之后记得移除hook
hook_handle.remove()
```

```
z的梯度: Variable containing:
 0.2500  0.2500
 0.2500  0.2500
[torch.FloatTensor of size 2x2]
```

Extending torch.autograd

自动求导帮助我们很方便地构建神经网络，是现代深度学习框架的一个非常重要的组成部分。但是如果网络中存在一个无法自动求导的操作，怎么办呢？

我们可以扩展自动求导，写一个 Function，实现他的前向传播和反向传播，我们用这个简单的小例子说明

```python
from torch.autograd import Function
class MulConstant(Function):
    @staticmethod
    def forward(ctx, tensor, constant):
        # ctx 用来缓存前向传播中的信息用于反向传播
        ctx.save_for_backward(constant)
        return tensor * constant

    @staticmethod
    def backward(ctx, grad_output):
        # grad_output 表示头梯度
        # backward 返回和 forward 输入一样多的结果
        # 不需要求梯度的返回 None
        constant, = ctx.saved_variables
        return grad_output * constant, None
```

```python
x = Variable(torch.Tensor([3]), requires_grad=True)
c = Variable(torch.Tensor([10]))
y = MulConstant.apply(x, c)
y.backward()

print(x.grad)
```

```
Variable containing:
 10
[torch.FloatTensor of size 1]
```

Face++ 旷视

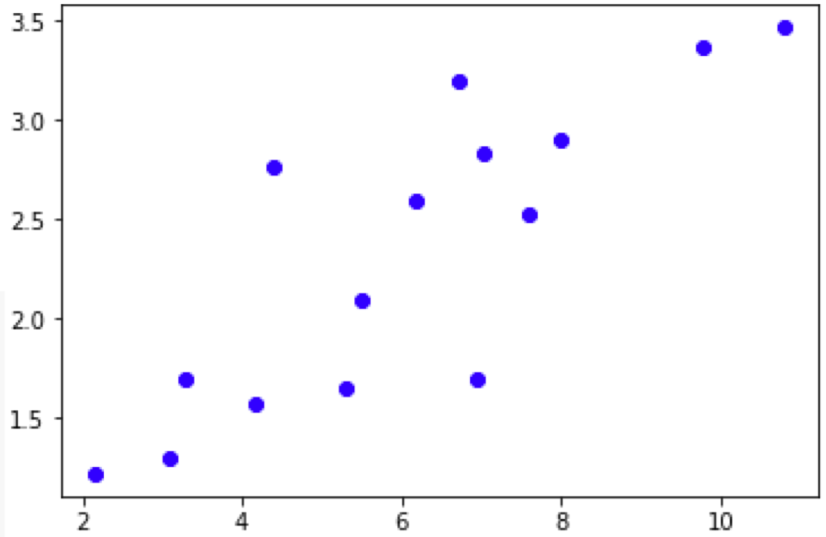
Pytorch

- What is PyTorch?
- Structure
  - Tensor
    - NumpyBridge
  - Variable
  - Autograd
- **Train a model**
  - Linear Regression
  - Training Loop
- Some key Parts
  - nn.Sequential
  - nn.Module
  - hook
  - nn.functional
  - torch.optim
  - Dataloader

special thanks to xingyu liao

Face++ 旷视

Linear Regression




```python
# define parameter w and b
w = Variable(torch.randn(1), requires_grad=True)  # random initialize
b = Variable(torch.zeros(1), requires_grad=True)  # zero initialize


# define linear model
def linear_model(x):
    return x * w + b


# loss function
def get_loss(y_, y):
    return torch.mean((y_ - y) ** 2)


# update parameters
def optimizer_step(w, b, lr=0.01):
    w.data = w.data - lr * w.grad.data
    b.data = b.data - lr * b.grad.data
```

$$\hat{y}_i = wx_i + b$$

$$loss = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

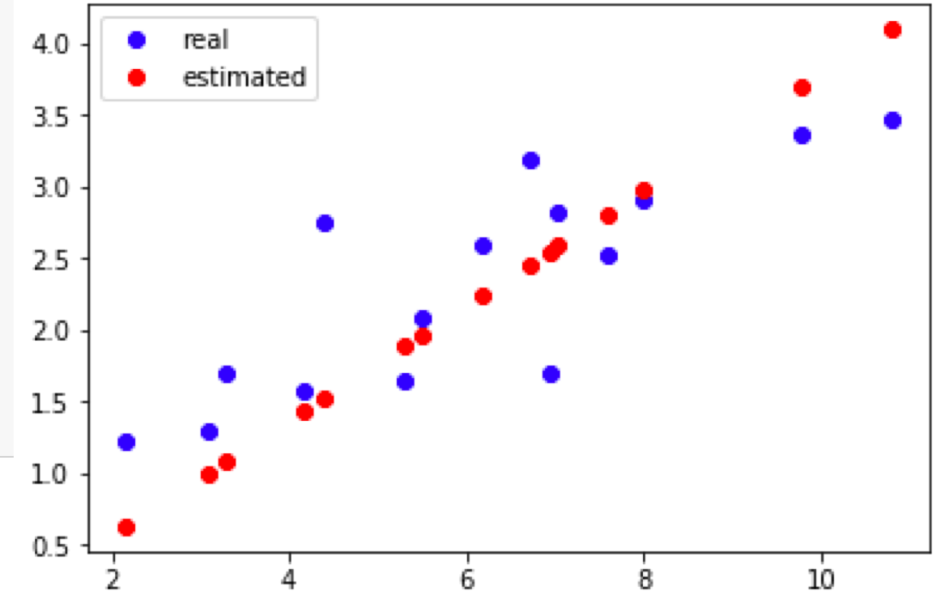
$$w := w - \eta\frac{\partial f(w,\ b)}{\partial w}$$

$$b := b - \eta\frac{\partial f(w,\ b)}{\partial b}$$

Training Loop

```
# training loop
for e in range(10):
    y_ = linear_model(x_train)
    loss = get_loss(y_, y_train)

    # make gradient zero
    if w.grad is not None:
        w.grad.zero_()
        b.grad.zero_()

    # backward pass
    loss.backward()

    # optimizer step
    optimizer_step(w, b)
    print('epoch: {}, loss: {}'.format(e, loss.data[0]))
```

```
epoch: 0, loss: 3.1357719898223877
epoch: 1, loss: 0.3550889194011688
epoch: 2, loss: 0.30295443534851074
epoch: 3, loss: 0.30131956934928894
epoch: 4, loss: 0.30062291026115422
epoch: 5, loss: 0.29994693398475647
epoch: 6, loss: 0.299274742603302
epoch: 7, loss: 0.2986060082912445
epoch: 8, loss: 0.2979407012462616
epoch: 9, loss: 0.29727882146835327
```



Face++ 旷视

Pytorch

- What is PyTorch?
- Structure
  - Tensor
    - NumpyBridge
  - Variable
  - Autograd
- Train a model
  - Linear Regression
  - Training Loop
- **Some key Parts**
  - nn.Sequential
  - nn.Module
  - hook
  - nn.functional
  - torch.optim
  - Dataloader

special thanks to xingyu liao

nn.Sequential

nn.Seqeuntial 模仿 keras，使得我们可以非常方便地构建序贯模型

```
net = nn.Sequential(
    nn.Linear(20, 10),
    nn.ReLU(True),
    nn.Linear(10, 5),
    nn.ReLU(True),
    nn.Linear(5, 1)
)
```

直接根据 index 访问每一层

```
net[2]
```

```
Linear(in_features=10, out_features=5, bias=True)
```

nn.Module

对于更复杂的模型，需要用到 nn.Module，这是 torch.nn 中的核心数据结构，既可以表示神经网络的某层，也可以表示很多层，前面讲过的 Sequential 是它的子类

我们先看看如何通过 nn.Module 构建一个网络

```
class perceptron(nn.Module):
    def __init__(self):
        super().__init__()
        self.l1 = nn.Sequential(
            nn.Linear(8, 6),
            nn.ReLU(True)
        )
        self.l2 = nn.Sequential(
            nn.Linear(6, 4),
            nn.ReLU(True)
        )
        self.l3 = nn.Sequential(
            nn.Linear(4, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        out = self.l1(x)
        out = self.l2(out)
        out = self.l3(out)
        return out

model = perceptron()
```

- 在 __init__ 中定义了所有需要学习的参数， Module 能够自动检测到这些 Parameter, 同时还包含着子 Module，主 Module 能够递归查找子 Module 中的子模块，forward 中定义了计算图
- 在使用的时候可以直接将 model 看作函数，调用 model(input) 就可以得到对应的结果，其等价于调用 model__call__(input)，在 __call__ 中主要调用了 model.forward(input)
- 可以通过 named_parameters() 或者 parameters() 返回要学习的参数，这是一个迭代器
- Module 中所有的参数都是 nn.Parameter 的数据类型，这是一种特殊的 Variable，默认 requires_grad=True

Face++ 旷视

nn.Module 的初始化

所有的子类都继承了父类 nn.Module 的 __init__ 初始化，主要是下面这些属性:

- _parameters: 这个字典保存了所有的 parameters
- _modules: 子 Module 会保存到这个字典
- _buffers: 缓存，比如 batch norm 中的 momentum
- _backward_hooks, _forward_hooks: 类似于前面讲的 Variable 的 hook
- training: 是否是训练模式

```
def __init__(self):
    self._parameters = OrderedDict()
    self._modules = OrderedDict()
    self._buffers = OrderedDict()
    self._backward_hooks = OrderedDict()
    self._forward_hooks = OrderedDict()
    self.training = True
```

Face++ 旷视

nn.Module 的构造方法

nn.Module 实现了 __getattr__(self, name) 和 __setattr__(self, name, value)

当模型初始化 module.name = value 的时候，会在 __setattr__ 中判断 value 是否为 Parameter 或者 Module 对象，是的话就会将对象加到 _parameters 和 _modules 中，如果是其他的对象，则会将这些值保存在 __dict__ 中

```
def __setattr__(self, name, value):
    def remove_from(*dicts):
        for d in dicts:
            if name in d:
                del d[name]

    params = self.__dict__.get('_parameters')
    if isinstance(value, Parameter):
        if params is None:
            raise AttributeError(
                "cannot assign parameters before Module.__init__() call")
        remove_from(self.__dict__, self._buffers, self._modules)
        self.register_parameter(name, value)
    elif params is not None and name in params:
        if value is not None:
            raise TypeError("cannot assign '{}' as parameter '{}' "
                            "(torch.nn.Parameter or None expected)"
                            .format(torch.typename(value), name))
        self.register_parameter(name, value)
    else:
        modules = self.__dict__.get('_modules')
        if isinstance(value, Module):
            if modules is None:
                raise AttributeError(
                    "cannot assign module before Module.__init__() call")
            remove_from(self.__dict__, self._parameters, self._buffers)
            modules[name] = value
        elif modules is not None and name in modules:
```

hook

register_forward_hook 和 register_backward_hook，这两个函数的功能类似于 variable 的 register_hook，表示在模型前向传播或者反向传播的时候注册 hook，每次前向传播结束或者反向传播结束就执行 hook 函数

register_forward_hook(*hook*) [source]

Registers a forward hook on the module.

The hook will be called every time after `forward()` has computed an output. It should have the following signature:

```
hook(module, input, output) -> None
```

The hook should not modify the input or output.

Returns: a handle that can be used to remove the added hook by calling `handle.remove()`

Return type: `torch.utils.hooks.RemovableHandle`

register_backward_hook(*hook*) [source]

Registers a backward hook on the module.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that will be used in place of `grad_input` in subsequent computations.

Returns: a handle that can be used to remove the added hook by calling `handle.remove()`

Return type: `torch.utils.hooks.RemovableHandle`

hook 的作用和之前在 variable 中的 hook 作用是相似的，是为了获取中间的结果，比如提取网络中某层的特征，或者得到反向传播中间某层非叶子节点的梯度
用完之后需要将 hook 移除

nn.functional

nn 中还有一个非常常用的模块叫做 nn.functional，nn 中几乎所有的层都有与之对应的函数，而 nn 中的抽象层其实也是 Module 的子模块，在 forward 中调用了 nn.functional 中的函数

nn.Conv2d

```
def __init__(self, in_channels, out_channels, kernel_size, stride=1,
             padding=0, dilation=1, groups=1, bias=True):
    kernel_size = _pair(kernel_size)
    stride = _pair(stride)
    padding = _pair(padding)
    dilation = _pair(dilation)
    super(Conv2d, self).__init__(
        in_channels, out_channels, kernel_size, stride, padding, dilation,
        False, _pair(0), groups, bias)

def forward(self, input):
    return F.conv2d(input, self.weight, self.bias, self.stride,
                    self.padding, self.dilation, self.groups)
```

大多数不需要参数的操作可以在 forward 中使用 nn.functional，比如 functional.max_pool2d，functional.relu

torch.optim

这是一个基类 Optimizer, 参数是:

- params: 需要优化的参数，是一个迭代器，被加入到 param_groups 中等待优化
- defaults: 参数设置，比如学习率，momentum

不同的层设置不同的学习率

```
optimizer = optim.SGD([
                {'params': net.features.parameters()}, # 学习率为1e-5
                {'params': net.classifier.parameters(), 'lr': 1e-2}
            ], lr=1e-5)
```

更多高级用法可以查看文档
http://pytorch.org/docs/0.3.1/optim.html

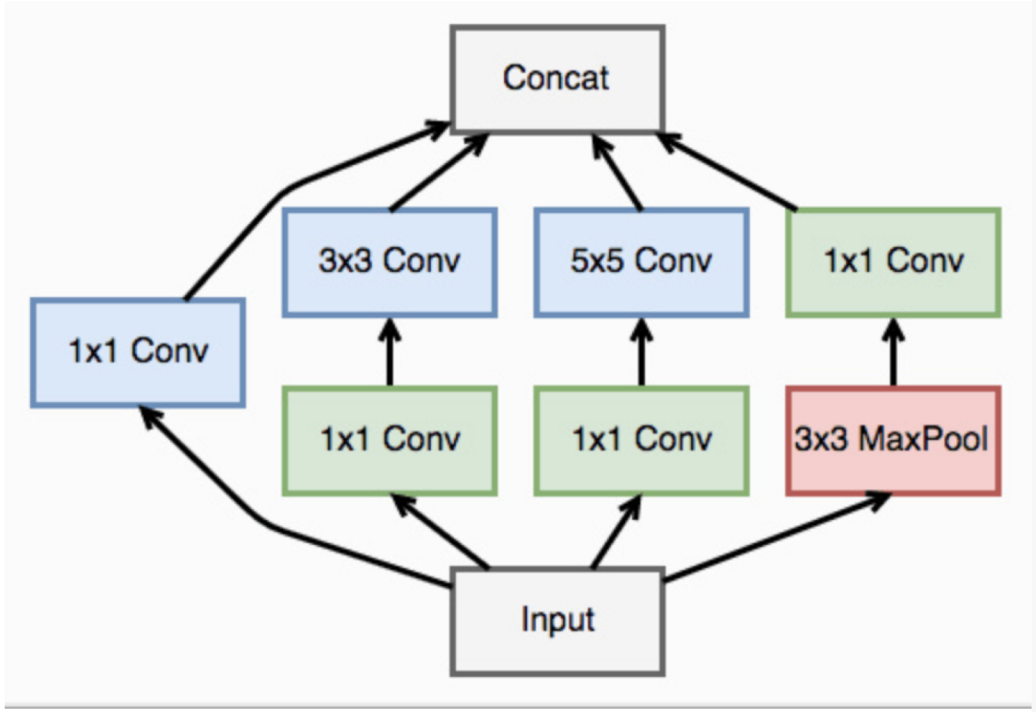
```
class Optimizer(object):
    """Base class for all optimizers.

    Arguments:
        params (iterable): an iterable of :class:`Variable` s or
            :class:`dict` s. Specifies what Variables should be optimized.
        defaults: (dict): a dict containing default values of optimization
            options (used when a parameter group doesn't specify them).
    """

    def __init__(self, params, defaults):
        self.defaults = defaults

        if isinstance(params, Variable) or torch.is_tensor(params):
            raise TypeError("params argument given to the optimizer should be "
                            "an iterable of Variables or dicts, but got " +
                            torch.typename(params))

        self.state = defaultdict(dict)
        self.param_groups = []

        param_groups = list(params)
        if len(param_groups) == 0:
            raise ValueError("optimizer got an empty parameter list")
        if not isinstance(param_groups[0], dict):
            param_groups = [{'params': param_groups}]

        for param_group in param_groups:
            self.add_param_group(param_group)
```

Serialization semantics

	First way (recommended)	**Second way**
Save model	torch.save(model.state_dict(), PATH)	torch.save(model, PATH)
Load model	model = Model(*args, **kwargs) model.load_state_dict(torch.load(PATH))	model = torch.load(PATH)

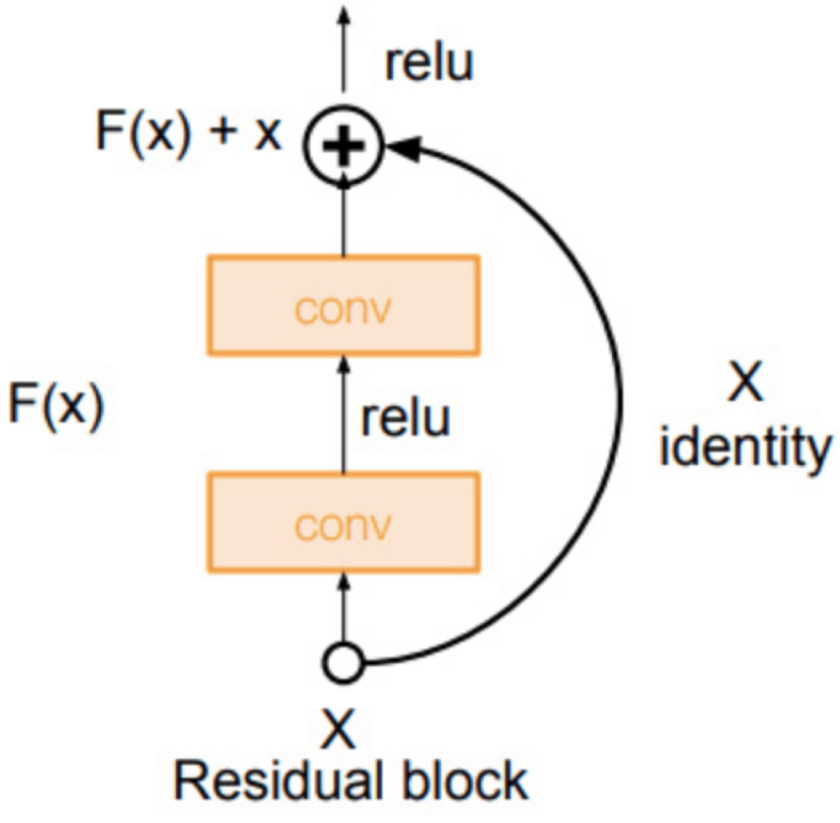
More examples

InceptionNet



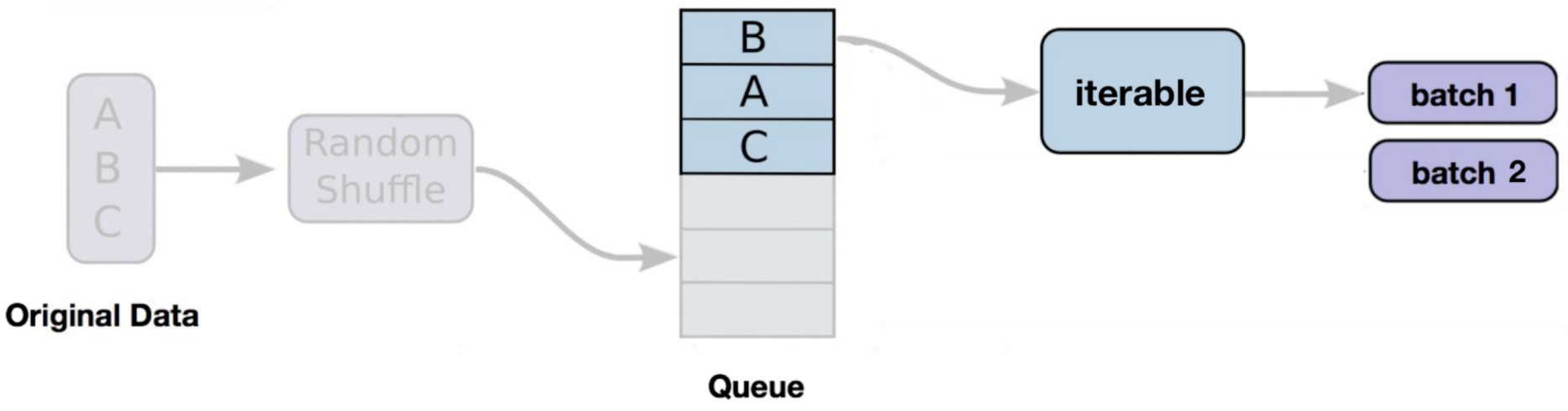

```python
class inception(nn.Module):
    def __init__(self, in_channel, out1_1, out2_1, out2_3, out3_1, out3_5, out4_1):
        super(inception, self).__init__()
        # first branch
        self.branch1x1 = conv_relu(in_channel, out1_1, 1)

        # second branch
        self.branch3x3 = nn.Sequential(
            conv_relu(in_channel, out2_1, 1),
            conv_relu(out2_1, out2_3, 3, padding=1)
        )

        # third branch
        self.branch5x5 = nn.Sequential(
            conv_relu(in_channel, out3_1, 1),
            conv_relu(out3_1, out3_5, 5, padding=2)
        )

        # fourth branch
        self.branch_pool = nn.Sequential(
            nn.MaxPool2d(3, stride=1, padding=1),
            conv_relu(in_channel, out4_1, 1)
        )

    def forward(self, x):
        f1 = self.branch1x1(x)
        f2 = self.branch3x3(x)
        f3 = self.branch5x5(x)
        f4 = self.branch_pool(x)
        output = torch.cat((f1, f2, f3, f4), dim=1)
        return output
```

ResNet




```python
class residual_block(nn.Module):
    def __init__(self, in_channel, out_channel, same_shape=True):
        super(residual_block, self).__init__()
        self.same_shape = same_shape
        stride=1 if self.same_shape else 2

        self.conv1 = nn.Conv2d(in_channel, out_channel, 3,
                               stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channel)

        self.conv2 = nn.Conv2d(out_channel, out_channel, 3,
                               padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channel)
        if not self.same_shape:
            self.conv3 = nn.Conv2d(in_channel, out_channel, 1,
                                   stride=stride)

    def forward(self, x):
        out = self.conv1(x)
        out = F.relu(self.bn1(out), True)
        out = self.conv2(out)
        out = F.relu(self.bn2(out), True)

        if not self.same_shape:
            x = self.conv3(x)
        return F.relu(x+out, True)
```

DataLoader




```
for i, data in enumerate(train_loader, 0):
      # get the inputs
      inputs, labels = data

      # wrap them in Variable
      inputs, labels = Variable(inputs), Variable(labels)

      # Run your training process
      print(epoch, i, "inputs", inputs.data, "labels", labels.data)
```

Custom DataLoader

```
from torch.utils.data import DataLoader

class MyOwnDataset(object):

    # initialize your data, download, etc
    def __init__(self):


    # return one item on the index
    def __getitem__(self, index):
        return

    # return the total data length
    def __len__(self):
        return

dataset = MyOwnDataset()
train_loader = DataLoader(dataset,
                          batch_size=32,
                          shuffle=True,
                          num_workers=4)
```

multiprocessing data loader

1 initialize data and read data

2 return one item on the index

3 return the data length

Custom DataLoader

```
from torch.utils.data import DataLoader

class MyOwnDataset(object):
    def __init__(self, file_name):
        xy = np.loadtxt(file_name)
        self.len = xy.shape[0]
        self.x_data = torch.from_numpy(xy[:, 0:-1])
        self.y_data = torch.from_numpy(xy[:, -1])

    def __getitem__(self, index):
        return self.x_data[index], self.y_data[index]

    def __len__(self):
        return self.len

dataset = MyOwnDataset('data_set.csv')
train_loader = DataLoader(dataset,
                          batch_size=32,
                          shuffle=True,
                          num_workers=4)
```

Using DataLoader

```
dataset = MyOwnDataset('data_set.csv')
train_loader = DataLoader(dataset, batch_size=32, shuffle=True, num_workers=4)

# training loop
for epoch in range(10):
    for data in train_loader:
        inputs, labels = data
        inputs, labels = Variable(inputs), Variable(labels)

        y_pred = model(inputs)
        loss = criterion(y_pred, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

The following dataset are available in torchvision

- MNIST and FashionMNIST
- COCO (Captioning and Detection)
- LSUN Classification
- ImageFolder
- Imagenet-l2
- CIFAR 10 and CIFAR 100
- SVHN
- PhotoTour

More Information

- PyTorch Document

http://pytorch.org/docs/0.3.1/

- PyTorch Forum

https://discuss.pytorch.org/

- PyTorch Tutorial

http://pytorch.org/tutorials/

Face++ 旷视

Q&A