从零开始的深度学习课程

II 工具解析

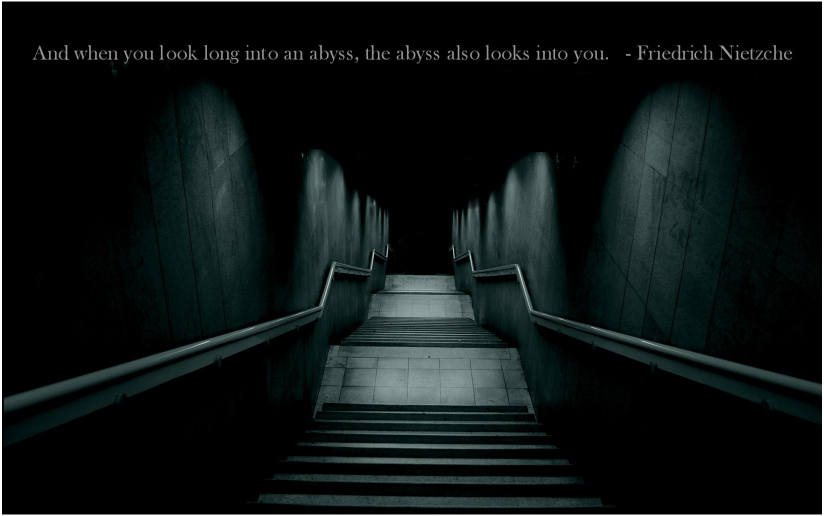
谢晋璟

Commerce Data Science Team

CDL数据科学社区（原BDA社区）
数据科学工作室
技术茶会

2016-07-22

窥视深渊者,必为深渊所窥视



目录

一个比喻：GPU vs 集群

GPU

集群

	Throughput	Latency
GPU	High	High
CPU	Low	Low

Table: GPU和CPU对比

GPU计算能力– 例子

桌面– GeForce GTX 1080

- Double Flops: NA
- Single Flops: 8.2T
- CUDA core: 2560
- Compute Capability: 6.1
- GPU Architecture: Pascal
- GPU Memory: 8GB
- Price: about 5000￥

服务器/工作站– Tesla K40

- Double Flops: 1.43T
- Single Flops: 4.29T
- CUDA core: 2880
- Compute Capability: 3.5
- GPU Architecture: Kepler
- GPU Memory: 12GB
- Price: around 20000￥

对比CPU的flops，比如i7-6700K(超频到4.6GHz)最多只达到211GFlops。一些简单Layer（比如Affine）不需要复杂的指令集，分支预测，乱序执行等现代CPU的高级功能，但它是强计算密集型的，正是发挥GPU高能耗比的地方。

Show me the code – “Hello, World”

这是CUDA下的“Hello, World”–向量加法，计算

$$\text{out} \leftarrow \text{in1} + \text{in2}$$

下面是设备（CUDA只支持GPU，OPENCL理论上还支持其他设备FPGA等）端代码

Device Side – Kernel Code

```
__global__ void vecAdd(float *in1, float *in2, float *out, int len) {
   int i = threadIdx.x + blockDim.x * blockIdx.x ;
   if (i < len)    out[i] = in1[i] + in2[i] ;
}
```

下一页是主机端代码

主机端代码

```
//apply for device memory
int size = sizeof(float) * inputLength;
cudaMalloc((void **) &deviceInput1, size);
cudaMalloc((void **) &deviceInput2, size);
cudaMalloc((void **) &deviceOutput, size);
//copy data to device
cudaMemcpy(deviceInput1, hostInput1, size, cudaMemcpyHostToDevice);
cudaMemcpy(deviceInput2, hostInput2, size, cudaMemcpyHostToDevice);
//define grid for computation
dim3 DimGrid((inputLength-1)/256 + 1, 1, 1);
dim3 DimBlock(256 , 1, 1);
//invoke the Kernel for computation
vecAdd<<<DimGrid, DimBlock>>>(deviceInput1, deviceInput2, deviceOutput,
      inputLength);
//need to synchronize between host and device
cudaDeviceSynchronize();
//After device finished its computation, copy the results back
cudaMemcpy(hostOutput, deviceOutput, size, cudaMemcpyDeviceToHost);
//release the device resources
cudaFree(deviceInput1); cudaFree(deviceInput2);
cudaFree(deviceOutput);
```

Bandwidth的诅咒

上面的例子里计算和读写内存的比例是1:1, 这个比值称为

CGMA(compute to global memory access ratio)

比如NVIDIA G80只有86.4GB/s带宽，最多调入21.6G的单精度浮点数，但CGMA=1时，最多只能执行21.6GFlop的浮点计算，远远小于设备的峰值吞吐量367GFlops。

如何逼近Throughput峰值?

解决办法

使用片上shared memory

其他影响Bandwidth的内容

合并访存

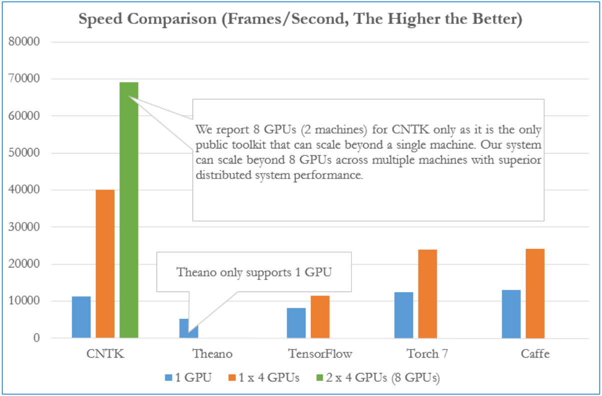
NVIDIA的库

编写GPU计算程序要考虑的细节非常繁多，需要对硬件和编译器有较深入的理解才能写出高效的代码，不然速度可能还不如CPU。好在已经有了很多现成的library可以使用，大部分情况我们只要调用一下，不必从头造轮子。比如NVIDIA自己就提供了这些

- cuFFT Fast Fourier Transforms Library
- cuBLAS Complete BLAS library
- cuSolver Collection of dense and sparse direct solvers/cusolver
- cuSPARSE Sparse Matrix library
- cuRAND Random Number Generator
- NPP Thousands of Performance Primitives for Image & Video Processing
- Thrust Templated Parallel Algorithms & Data Structures
- CUDA Math Library high performance math routines
- cuDNN library for deep neural networks

现有DL工具对GPU的需求

回到我们的主题，所有的DL工具都默认使用GPU。
最常用的DL工具就是Caffe, Torch7, Theano和TensorFlow。
下面借了大MS的一张广告图：




Decaf
Coffee
is Caffeine
Free

前言

Caffe是最早流行的DL工具，它基本上只是下面两个东西

- C++ 写的library
- 命令行下的训练、预测工具

后来它也提供了python的接口，可以在python中比较方便的存取它的数据对象，调用向前向后命令等。

Caffe的代码量不大，写的比较清晰，和我们在上一讲中的介绍是一一对应的，所以它是个理解DL框架的完美例子。下面一一解释它的核心对象。

目录

Blob是存储张量的数据结构

Blob的域

```
protected:
    shared_ptr<SyncedMemory> data_;       //  X
    shared_ptr<SyncedMemory> diff_;       //  δX = ∂l/∂X
    shared_ptr<SyncedMemory> shape_data_; // same as shape_
    vector<int> shape_;                   // dimension vector
    int count_;                           // number of elements
    int capacity_;
```

基础数据对象Blob

用来存储所有的（张量）数据

- 输入数据和神经元的激活：$X_{u,v}^{c,j}$
- Score：S
- 反馈信号：$\delta X_{u,v}^{c,j}$, δS
- 待估参数：$W_{u,v}^{\iota}, K(\iota)(u,v)_{c}^{c'}$

Blob是绑定了data和loss对此data偏导的数据结构

它的update方法就是用自己的$\mathrm{d}X$修正自己的X.

Blob的update方法

$$W \leftarrow W + (-1) \cdot \mathrm{d}W$$

```
template <typename Dtype>
void Blob<Dtype>::Update() {
  // We will perform update based on where the data is located.
  switch (data_->head()) {
  case SyncedMemory::HEAD_AT_CPU:
    // perform computation on CPU
    caffe_axpy<Dtype>(count_, Dtype(-1),
        static_cast<const Dtype*>(diff_->cpu_data()),
        static_cast<Dtype*>(data_->mutable_cpu_data()));
    break;
  case SyncedMemory::HEAD_AT_GPU:
  case SyncedMemory::SYNCED:
#ifndef CPU_ONLY
    // perform computation on GPU
    caffe_gpu_axpy<Dtype>(count_, Dtype(-1),
        static_cast<const Dtype*>(diff_->gpu_data()),
        static_cast<Dtype*>(data_->mutable_gpu_data()));
    // . . .
}
```

目录

Layer就是我们的多元向量映射$f(X;W)$

Layer类的域

```
protected:
 /** The protobuf that stores the layer parameters */
 LayerParameter layer_param_;
 /** The phase: TRAIN or TEST */
 Phase phase_;
 /** The vector that stores the learnable parameters as a set of blobs. */
 vector<shared_ptr<Blob<Dtype> > > blobs_;
 /** Vector indicating whether to compute the diff of each param blob. */
 vector<bool> param_propagate_down_;

 /** The vector that indicates whether each top blob has a non-zero weight in
  * the objective function. */
 vector<Dtype> loss_;
```

其中，blobs_ 用于存储W，而LayerParameter 才是记录输入输出，类型的地方。
它是由protobuf定义的，定义文件的一部分如下：

LayerParameter

```
message LayerParameter {
  optional string name = 1; // the layer name
  optional string type = 2; // the layer type
  repeated string bottom = 3; // the name of each bottom blob
  repeated string top = 4; // the name of each top blob

  // The train / test phase for computation.
  optional Phase phase = 10;

  // The amount of weight to assign each top blob in the objective.
  // Each layer assigns a default value, usually of either 0 or 1,
  // to each top blob.
  repeated float loss_weight = 5;

  // Specifies training parameters (multipliers on global learning constants,
  // and the name and other settings used for weight sharing).
  repeated ParamSpec param = 6;

  // The blobs containing the numeric parameters of the layer.
  repeated BlobProto blobs = 7;

  // Specifies whether to backpropagate to each bottom. If unspecified,
  // Caffe will automatically infer whether each input needs backpropagation
  // to compute parameter gradients. If set to true for some inputs,
  // backpropagation to those inputs is forced; if set false for some inputs,
  // backpropagation to those inputs is skipped.
  //
  // The size must be either 0 or equal to the number of bottoms.
  repeated bool propagate_down = 11;
```

LayerParameter（续）

```
// Rules controlling whether and when a layer is included in the network,
// based on the current NetState.  You may specify a non-zero number of rules
// to include OR exclude, but not both.  If no include or exclude rules are
// specified, the layer is always included.  If the current NetState meets
// ANY (i.e., one or more) of the specified rules, the layer is
// included/excluded.
repeated NetStateRule include = 8;
repeated NetStateRule exclude = 9;

// Parameters for data pre-processing.
optional TransformationParameter transform_param = 100;

// Parameters shared by loss layers.
optional LossParameter loss_param = 101;

// . . .
optional ConvolutionParameter convolution_param = 106;
optional InnerProductParameter inner_product_param = 117;
optional InputParameter input_param = 143;
optional ReLUParameter relu_param = 123;
optional SoftmaxParameter softmax_param = 125;
optional DataParameter data_param = 107;
// . . .
```

它包含了具体的Layer的parameter。根据type不同，里面出现的可能是convolution_param, relu_param等，下面举个ConvolutionParameter的例子：

ConvolutionParameter(例子)

```
message ConvolutionParameter {
  optional uint32 num_output = 1; // The number of outputs for the layer
  optional bool bias_term = 2 [default = true]; // whether to have bias terms

  // Pad, kernel size, and stride are all given as a single value for equal
  // dimensions in all spatial dimensions, or once per spatial dimension.
  repeated uint32 pad = 3; // The padding size; defaults to 0
  repeated uint32 kernel_size = 4; // The kernel size
  repeated uint32 stride = 6; // The stride; defaults to 1
  // Factor used to dilate the kernel, (implicitly) zero-filling the resulting
  // holes. (Kernel dilation is sometimes referred to by its use in the
  // algorithme trous from Holschneider et al. 1987.)
  repeated uint32 dilation = 18; // The dilation; defaults to 1

  // For 2D convolution only, the *_h and *_w versions may also be used to
  // specify both spatial dimensions.
  optional uint32 pad_h = 9 [default = 0]; // The padding height (2D only)
  optional uint32 pad_w = 10 [default = 0]; // The padding width (2D only)
  optional uint32 kernel_h = 11; // The kernel height (2D only)
  optional uint32 kernel_w = 12; // The kernel width (2D only)
  optional uint32 stride_h = 13; // The stride height (2D only)
  optional uint32 stride_w = 14; // The stride width (2D only)

  optional uint32 group = 5 [default = 1]; // The group size for group conv

  optional FillerParameter weight_filler = 7; // The filler for the weight
  optional FillerParameter bias_filler = 8; // The filler for the bias
```

ConvolutionParameter(续)

```
  enum Engine {
    DEFAULT = 0;
    CAFFE = 1;
    CUDNN = 2;
  }
  optional Engine engine = 15 [default = DEFAULT];

  // The axis to interpret as "channels" when performing convolution.
  // Preceding dimensions are treated as independent inputs;
  // succeeding dimensions are treated as "spatial".
  // With (N, C, H, W) inputs, and axis == 1 (the default), we perform
  // N independent 2D convolutions, sliding C-channel (or (C/g)-channels, for
  // groups g>1) filters across the spatial axes (H, W) of the input.
  // With (N, C, D, H, W) inputs, and axis == 1, we perform
  // N independent 3D convolutions, sliding (C/g)-channels
  // filters across the spatial axes (D, H, W) of the input.
  optional int32 axis = 16 [default = 1];

  // Whether to force use of the general ND convolution, even if a specific
  // implementation for blobs of the appropriate number of spatial dimensions
  // is available. (Currently, there is only a 2D-specific convolution
  // implementation; for input blobs with num_axes != 2, this option is
  // ignored and the ND implementation will be used.)
  optional bool force_nd_im2col = 17 [default = false];
}
```

可以看到上一讲提到的概念padding, stride等都出现在里面。

DataParameter–负责数据的输入的也被当成Layer

```
message DataParameter {
  // Specify the data source.
  optional string source = 1;
  // Specify the batch size.
  optional uint32 batch_size = 4;

  optional DB backend = 8 [default = LEVELDB];

  optional string mean_file = 3;

  // Force the encoded image to have 3 color channels
  optional bool force_encoded_color = 9 [default = false];

  // Prefetch queue (Number of batches to prefetch to host memory, increase if
  // data access bandwidth varies).
  optional uint32 prefetch = 10 [default = 4];
}
```

DataLayer是非常重要的Layer，它虽然不负责计算，也没有向后方法，但是它的Forward方法负责读取一个Batch的数据。

Layer的最重要方法：向前和向后

下面是它的Interface：

```
/**
  * The Forward wrapper calls the relevant device wrapper function
  * (Forward_cpu or Forward_gpu) to compute the top blob values given the
  * bottom blobs.  If the layer has any non-zero loss_weights, the wrapper
  * then computes and returns the loss.
  *
  * Your layer should implement Forward_cpu and (optionally) Forward_gpu.
  */
 inline Dtype Forward(const vector<Blob<Dtype>*>& bottom,
     const vector<Blob<Dtype>*>& top);

 /**
  * @param propagate_down
  *     a vector with equal length to bottom, with each index indicating
  *     whether to propagate the error gradients down to the bottom blob at
  *     the corresponding index
  *
  * The Backward wrapper calls the relevant device wrapper function
  * (Backward_cpu or Backward_gpu) to compute the bottom blob diffs given the
  * top blob diffs.
  *
  * Your layer should implement Backward_cpu and (optionally) Backward_gpu.
  */
 inline void Backward(const vector<Blob<Dtype>*>& top,
     const vector<bool>& propagate_down,
     const vector<Blob<Dtype>*>& bottom);
```

目录

Net就是hypotheses

Net把各个计算用的Layer的输入输出和参数按照一定的方式组织在一起。它负责计算loss值和loss的对参数的梯度。下面详解它的类定义。

Net的域(1)：层和块

```
/// @brief The network name
string name_;
/// @brief The phase: TRAIN or TEST
Phase phase_;

/// @brief Individual layers in the net
vector<shared_ptr<Layer<Dtype> > > layers_;
vector<string> layer_names_;
map<string, int> layer_names_index_;
vector<bool> layer_need_backward_;

/// @brief the blobs storing intermediate results between the layer.
vector<shared_ptr<Blob<Dtype> > > blobs_;
vector<string> blob_names_;
map<string, int> blob_names_index_;
vector<bool> blob_need_backward_;
```

可以发现它是由Layer和描述激活度的Blob连接成的。

Net的域(2)：每层的输入输出，整体的输入输出块

```
/// bottom_vecs stores the vectors containing the input for each layer.
/// They don't actually host the blobs (blobs_ does), so we simply store
/// pointers.
vector<vector<Blob<Dtype>*> > bottom_vecs_;
vector<vector<int> > bottom_id_vecs_;
vector<vector<bool> > bottom_need_backward_;

/// top_vecs stores the vectors containing the output for each layer
vector<vector<Blob<Dtype>*> > top_vecs_;
vector<vector<int> > top_id_vecs_;

/// Vector of weight in the loss (or objective) function of each net blob,
/// indexed by blob_id.
vector<Dtype> blob_loss_weights_;

/// blob indices for the input and the output of the net
vector<int> net_input_blob_indices_;
vector<int> net_output_blob_indices_;
vector<Blob<Dtype>*> net_input_blobs_;
vector<Blob<Dtype>*> net_output_blobs_;
```

它们都是一些book keeping的指针，记录哪些Blob是哪些Layer的输入输出，以及整体的输入是哪个Blob，输出是哪个Blob。

Net的域(3)：参数全体–固定的和可学习的

```
vector<vector<int> > param_id_vecs_;
vector<int> param_owners_;
vector<string> param_display_names_;
vector<pair<int, int> > param_layer_indices_;
map<string, int> param_names_index_;

/// The parameters in the network.
vector<shared_ptr<Blob<Dtype> > > params_;
vector<Blob<Dtype>*> learnable_params_;
/**
 * The mapping from params_ -> learnable_params_: we have
 * learnable_param_ids_.size() == params_.size(),
 * and learnable_params_[learnable_param_ids_[i]] == params_[i].get()
 * if and only if params_[i] is an "owner"; otherwise, params_[i] is a sharer
 * and learnable_params_[learnable_param_ids_[i]] gives its owner.
 */
vector<int> learnable_param_ids_;
```

Net的域(4)：调节网络在训练中的行为的参数，learning rate等

```
/// the learning rate multipliers for learnable_params_
vector<float> params_lr_;
vector<bool> has_params_lr_;
/// the weight decay multipliers for learnable_params_
vector<float> params_weight_decay_;
vector<bool> has_params_decay_;
/// The bytes of memory used by this net
size_t memory_used_;
/// Whether to compute and display debug info for the net.
bool debug_info_;
/// The root net that actually holds the shared layers in data parallelism
const Net* const root_net_;
```

Solver会读取这些数值来控制对整个网络的更新，比如，某层的learning rate设0，它在训练中就被冻结了，参数不再改变。

Net的核心方法–当然也是向前向后方法

Net的前后方法

```
/**
 * The From and To variants of Forward and Backward operate on the
 * (topological) ordering by which the net is specified. For general DAG
 * networks, note that (1) computing from one layer to another might entail
 * extra computation on unrelated branches, and (2) computation starting in
 * the middle may be incorrect if all of the layers of a fan-in are not
 * included.
 */
const vector<Blob<Dtype>*>& Forward(Dtype* loss = NULL)
Dtype ForwardFromTo(int start, int end);
Dtype ForwardFrom(int start);
Dtype ForwardTo(int end);

/**
 * The network backward should take no input and output, since it solely
 * computes the gradient w.r.t the parameters, and the data has already been
 * provided during the forward pass.
 */
void Backward();
void BackwardFromTo(int start, int end);
void BackwardFrom(int start);
void BackwardTo(int end);
```

Net的“前后”方法和向前方法的实现

```
Dtype ForwardBackward() {
    Dtype loss;
    Forward(&loss);
    Backward();
    return loss;
  }

template <typename Dtype>
const vector<Blob<Dtype>*>& Net<Dtype>::Forward(Dtype* loss) {
  if (loss != NULL) {
    *loss = ForwardFromTo(0, layers_.size() - 1);
  } else {
    ForwardFromTo(0, layers_.size() - 1);
  }
  return net_output_blobs_;
}

template <typename Dtype>
Dtype Net<Dtype>::ForwardFromTo(int start, int end) {
  CHECK_GE(start, 0);
  CHECK_LT(end, layers_.size());
  Dtype loss = 0;
  for (int i = start; i <= end; ++i) {
    // LOG(ERROR) << "Forwarding " << layer_names_[i];
    Dtype layer_loss = layers_[i]->Forward(bottom_vecs_[i], top_vecs_[i]);
    loss += layer_loss;
    if (debug_info_) { ForwardDebugInfo(i); }
  }
  return loss;
}
```

Net的向后方法

Backward 实际是由BackwardFromTo 实现的

```
template <typename Dtype>
void Net<Dtype>::BackwardFromTo(int start, int end) {
  CHECK_GE(end, 0);
  CHECK_LT(start, layers_.size());
  for (int i = start; i >= end; --i) {
    if (layer_need_backward_[i]) {
      layers_[i]->Backward(
          top_vecs_[i], bottom_need_backward_[i], bottom_vecs_[i]);
      if (debug_info_) { BackwardDebugInfo(i); }
    }
  }
}
```

Net提供的Update

```
template <typename Dtype>
void Net<Dtype>::Update() {
  for (int i = 0; i < learnable_params_.size(); ++i) {
    learnable_params_[i]->Update();
  }
}
```

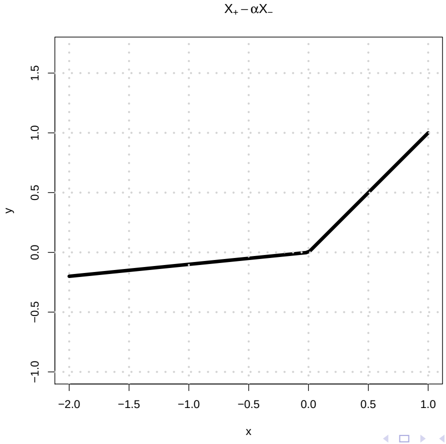
最naive的方式就是：走完一个向前向后过程后，Blob里就有$\frac{\partial l}{\partial \text{this blob}}$的数据，调用上面的Net的update方法更新参数。后面我们会看到有些Solver并不调用Net的Update，而是根据某些修正，去直接修改parameter的值。

另一点值得注意的是，最终的Forward的计算还是要依赖底层Layer的实现的，下面是具体的例子：Leaky ReLU。

Layer Forward/Backward的例子–Leaky ReLU定义

Leaky ReLU：和普通ReLU的差别是小于0部分的斜率不再是0，而是一个可学习的参数α，反馈信号可以α比例“漏”过去。公式：

$$Y = X_+ - \alpha X_-$$



Layer Forward/Backward的例子–Leaky ReLU

向前实现: CPU版本

```
template <typename Dtype>
void ReLULayer<Dtype>::Forward_cpu(const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top) {
  const Dtype* bottom_data = bottom[0]->cpu_data();
  Dtype* top_data = top[0]->mutable_cpu_data();
  const int count = bottom[0]->count();
  Dtype negative_slope = this->layer_param_.relu_param().negative_slope();
  for (int i = 0; i < count; ++i) {
    // This is the line do the computation
    top_data[i] = std::max(bottom_data[i], Dtype(0))
        + negative_slope * std::min(bottom_data[i], Dtype(0));
  }
}
```

相对应的向前GPU Kernel

```
template <typename Dtype>
__global__ void ReLUForward(const int n, const Dtype* in, Dtype* out,
    Dtype negative_slope) {
  CUDA_KERNEL_LOOP(index, n) {
    out[index] = in[index] > 0 ? in[index] : in[index] * negative_slope;
  }
}
```

Layer Forward/Backward的例子–Leaky ReLU

向后实现: CPU版本

```
template <typename Dtype>
void ReLULayer<Dtype>::Backward_cpu(const vector<Blob<Dtype>*>& top,
    const vector<bool>& propagate_down,
    const vector<Blob<Dtype>*>& bottom) {
  if (propagate_down[0]) {
    const Dtype* bottom_data = bottom[0]->cpu_data();
    const Dtype* top_diff = top[0]->cpu_diff();
    Dtype* bottom_diff = bottom[0]->mutable_cpu_diff();
    const int count = bottom[0]->count();
    Dtype negative_slope = this->layer_param_.relu_param().negative_slope();
    for (int i = 0; i < count; ++i) {
    // the local gradient is 1 or negative_slope depend on X>0
      bottom_diff[i] = top_diff[i] * ((bottom_data[i] > 0) + negative_slope * (
           bottom_data[i] <= 0));
    } } }
```

相对应的向后GPU Kernel

```
template <typename Dtype>
__global__ void ReLUBackward(const int n, const Dtype* in_diff,
    const Dtype* in_data, Dtype* out_diff, Dtype negative_slope) {
  CUDA_KERNEL_LOOP(index, n) {
    out_diff[index] = in_diff[index] * ((in_data[index] > 0)
        + (in_data[index] <= 0) * negative_slope);
  } }
```

目录

求解器

求解器的用处

Net 定义了Hypotheses的结构，统计上相当于某种先验的知识。它的向前方法负责计算loss，向后方法计算反馈。
而求解器负责如何从数据中取出一个个batch，通过计算loss和反馈来调整参数值，使得Net 收敛到一个我们满意的结果。

Solver类的核心域

```
SolverParameter param_;
int iter_;
int current_step_;
shared_ptr<Net<Dtype> > net_;
vector<shared_ptr<Net<Dtype> > > test_nets_;
vector<Callback*> callbacks_;
vector<Dtype> losses_;
Dtype smoothed_loss_;
```

这里面控制求解器行为的最重要参数是param_ ，它的类SolverParameter 是由Protobuf定义的。我们下面详细解释。

SolverParameter的ProtoBuf定义

learning policy

```
// Proto filename for the train net, possibly combined with one or more test nets.
optional string net = 24;
optional float base_lr = 5; // The base learning rate
optional int32 max_iter = 7; // the maximum number of iterations

// The learning rate decay policy. The currently implemented learning rate policies
     are as follows:
//    - fixed: always return base_lr.
//    - step: return base_lr * gamma ^ (floor(iter / step))
//    - exp: return base_lr * gamma ^ iter
//    - inv: return base_lr * (1 + gamma * iter) ^ (- power)
//    - multistep: similar to step but it allows non uniform steps defined by
//      stepvalue
//    - poly: the effective learning rate follows a polynomial decay, to be
//      zero by the max_iter. return base_lr (1 - iter/max_iter) ^ (power)
//    - sigmoid: the effective learning rate follows a sigmod decay
//      return base_lr ( 1/(1 + exp(-gamma * (iter - stepsize))))
//
// where base_lr, max_iter, gamma, step, stepvalue and power are defined
// in the solver parameter protocol buffer, and iter is the current iteration.
optional string lr_policy = 8;
optional float gamma = 9; // The parameter to compute the learning rate.
optional float power = 10; // The parameter to compute the learning rate.
optional int32 stepsize = 13; // the stepsize for learning rate policy "step"
repeated int32 stepvalue = 34; // the stepsize for learning rate policy "multistep"
```

SolverParameter的ProtoBuf定义

Regularization

```
optional float weight_decay = 12; // The weight decay.
// regularization types supported: L1 and L2
// controlled by weight_decay
optional string regularization_type = 29 [default = "L2"];

// Set clip_gradients to >= 0 to clip parameter gradients to that L2 norm,
// whenever their actual L2 norm is larger.
optional float clip_gradients = 35 [default = -1];
```

SolverParameter的ProtoBuf定义

Solver Type and Sovler related parameters

```
// type of the solver: SGD, NESTEROV, ADAGRAD, RMSPROP, ADADELTA, ADAM
optional string type = 40 [default = "SGD"];

optional float momentum = 11; // The momentum value.

// numerical stability for RMSProp, AdaGrad and AdaDelta and Adam
optional float delta = 31 [default = 1e-8];

// parameters for the Adam solver
optional float momentum2 = 39 [default = 0.999];

// RMSProp decay value
// MeanSquare(t) = rms_decay*MeanSquare(t-1) + (1-rms_decay)*SquareGradient(t)
optional float rms_decay = 38;
```

求解器的核心方法是Step

因为Solve 方法的核心只有下面一句：

```
Step(param_.max_iter() - iter_);
```

Step里面的loop，每个loop是一个iteration。由于DataLayer也是Layer，它也有Forward方法（当然没有Backward）。所以每次iteration是由DataLayer负责去读取一个新的batch到data Blob中，再由实际计算的Layer向上传。

Step方法

```
template <typename Dtype> void Solver<Dtype>::Step(int iters) {
  const int start_iter = iter_;
  const int stop_iter = iter_ + iters;
  int average_loss = this->param_.average_loss();
  losses_.clear();
  smoothed_loss_ = 0;

  while (iter_ < stop_iter) {
    // zero-init the params
    net_->ClearParamDiffs();

    Dtype loss = 0;
    for (int i = 0; i < param_.iter_size(); ++i) {
      loss += net_->ForwardBackward();
    }
    loss /= param_.iter_size();
    // average the loss across iterations for smoothed reporting
    UpdateSmoothedLoss(loss, start_iter, average_loss);

    ApplyUpdate();
    ++iter_;
  }
}
```

ApplyUpdate 方法

Step 里面用向前向后来计算每个系数相关的更新量，放入Blob 的diff 中；然而具体如何使用d W去更新W，是依赖于具体方法实现的。

现有代码有下面的衍生类实现了相应的ApplyUpdate 方法:

- SGDSolver 带momentum的随机梯度下降
- NesterovSolver 用未来位置梯度更新的变种SGD
- AdaGradSolver 拟二阶的方法
- RMSPropSolver 相比AdaGrad，用指数平滑代替累加
- AdaDeltaSolver 相比AdaGrad，同时对梯度和参数改变做修正
- AdamSolver 同时使用一、二阶矩来光滑化修正。

我们下面一一解释：

SGDSolver –带动量的随机梯度下降

SGDSolver::ApplyUpdate 的核心部分就是

```
ClipGradients();
for (int param_id = 0; param_id < this->net_->learnable_params().size();
     ++param_id) {
  Normalize(param_id);
  Regularize(param_id);
  ComputeUpdateValue(param_id, rate);
}
this->net_->Update();
```

前面计算好修正量，写回diff 部分，用Net 的update 方法更新之。

SGDSolver::ComputeUpdateValue 实际计算是由下面完成的

```
caffe_cpu_axpby(net_params[param_id]->count(), local_rate,
          net_params[param_id]->cpu_diff(), momentum,
          history_[param_id]->mutable_cpu_data());
caffe_copy(net_params[param_id]->count(),
    history_[param_id]->cpu_data(),
    net_params[param_id]->mutable_cpu_diff());
```

对应的公式是:

$$\text{cache} \leftarrow \text{local\_rate} * \mathrm{d}W + \text{momentum} * \text{cache}$$

$$\mathrm{d}W \leftarrow \text{cache}$$

NesterovSolver–“一秒之后的未来”

用$v^{(t)}$代表history，用$\theta^{(t)}$代表迭代的当前值。用r代表learning rate而momentum的比例记作μ。上一页的公式就是下面

SGDSolver

$$v^{(t)} \leftarrow r \cdot \nabla_\theta f(\theta^{(t-1)}) + \mu \cdot v^{(t-1)}$$
$$\theta^{(t)} \leftarrow \theta^{(t-1)} - v^{(t)}$$

NesterovSolver: 使用一步之后的未来梯度

$$v^{(t)} \leftarrow r \cdot \nabla_\theta f(\theta^{(t-1)} - \mu v^{(t-1)}) + \mu \cdot v^{(t-1)}$$
$$\theta^{(t)} \leftarrow \theta^{(t-1)} - v^{(t)}$$

方便起见，把括号中用$\phi^{(t-1)} = \theta^{(t-1)} - \mu v^{(t-1)}$代替，那么

$$v^{(t)} \leftarrow r \cdot \nabla_\theta f(\phi^{(t-1)}) + \mu \cdot v^{(t-1)}$$
$$\phi^{(t)} \leftarrow \phi^{(t-1)} + \mu v^{(t-1)} - (1+\mu) v^{(t)}$$

带二阶信息的方法–AdaGrad和RMSPropSolver

前面方法用到∇f，如果引入$(\nabla f)^2$（点态），那么就有

AdaGradSolver

$$v^{(t)} \leftarrow v^{(t-1)} + (\nabla_\theta f)^2$$

$$\theta^{(t)} \leftarrow \theta^{(t-1)} - r \cdot \nabla_\theta f / (\sqrt{v^{(t)}} + \epsilon)$$

修正的意义相当于利用方差标准化。

RMSPropSolver

$$v^{(t)} \leftarrow \mu \cdot v^{(t-1)} + (1-\mu) \cdot (\nabla_\theta f)^2$$

$$\theta^{(t)} \leftarrow \theta^{(t-1)} - r \cdot \nabla_\theta f / (\sqrt{v^{(t)}} + \epsilon)$$

用指数平滑代替了AdaGrad中的累加

带二阶信息的方法–AdaDeltaSolver

AdaDeltaSolver

引入$E[g^2]_t$为t时刻的gradient g^2估计，$E[\triangle x^2]_t$为t时刻参数改变量的估计，迭代公式是：

$$E[g^2]_t \leftarrow \mu E[g^2]_{t-1} + (1-\mu)g_t^2$$

$$\triangle x_t \leftarrow -\frac{\mathrm{RMS}[\triangle x]_{t-1}}{\mathrm{RMS}[g]_t} g_t$$

$$E[\triangle x^2]_t \leftarrow \mu E[\triangle x^2]_{t-1} + (1-\mu)\triangle x_t^2$$

$$\text{where} \quad \mathrm{RMS}[y]_t = \sqrt{E[y^2]_t + \epsilon}$$

$$x_{t+1} \leftarrow x_t + r \cdot \triangle x_t$$

含义是用标准化了的$g_t/\sigma(g)$去修正标准化的$\triangle x_t/\sigma(\triangle x)$。

同时进行一二阶信息的平滑–AdamSolver

AdamSolver

$$m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$$

$$v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

$$x_t \leftarrow x_{t-1} - r \cdot \frac{m_t}{1-\beta_1^t} \cdot \frac{1}{\sqrt{\frac{v_t}{1-\beta_2^t}} + \epsilon}$$

实际使用中，基本上就先选AdamSolver就好了。如果它收敛不好，再试弱一点但safter一点的二阶或是一阶方法。

AdamSolver的修正量的解释

这里平滑化是很自然的想法，唯一要小心的是修正。以m_t为例，展开之

$$m_t = \beta_1^{t-1}(1-\beta_1)g_1 + \beta_1^{t-2}(1-\beta_1)g_2 + \cdots + (1-\beta_1)g_t$$

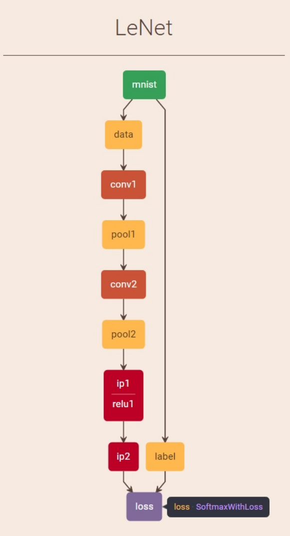
两边取Expectation，有

$$\begin{aligned} E[m_t] &= (\beta_1^{t-1}(1-\beta_1) + \beta_1^{t-2}(1-\beta_1) + \cdots + (1-\beta_1))E[g_t] \\ &= (1-\beta_1)\frac{1-\beta_1^t}{1-\beta_1}E[g_t] \\ &= (1-\beta_1^t)E[g_t] \end{aligned}$$

即

$$E[g_t] = \frac{E[m_t]}{1-\beta_1^t}$$

完整的例子



而传给Solver的参数如下：

```
net: "examples/mnist/lenet_train_test.prototxt"
test_iter: 100
test_interval: 500
base_lr: 0.01
momentum: 0.9
weight_decay: 0.0005
lr_policy: "inv"
gamma: 0.0001
power: 0.75
# Display every 100 iterations
display: 100
max_iter: 10000
snapshot: 5000
snapshot_prefix: "examples/mnist/lenet"
solver_mode: GPU
```

训练命令行:

```
./build/tools/caffe train --solver=examples/mnist/lenet_solver.
    prototxt
```

Caffe的问题

- 开发难度高 新的Layer、Loss一般需要写C++和CUDA代码才能在Caffe中使用；虽然有Python-Layer，可以使用Python来写Layer代码，但它只实现了CPU部分，会拖慢整个的计算流水线。
- Backward开发难 需要手动推导local gradient的公式，才能实现代码
- 搭建Net不灵活 复杂的Net结构需要写非常复杂的protobuf描述文件。
- 修改Layer不灵活 微调某些层的公式只有修改Layer的实现类的代码，重新编译。

如何写出优雅的代码？

答案：

用更加合适的工具

下面的图可能带来一些启发！

流行的深度框架热度对比

Aggregate popularity (30•contrib + 10•issues + 5•forks)•1e-3		
#1:	87.73	tensorflow/tensorflow
#2:	68.06	BVLC/caffe
#3:	39.91	fchollet/keras
#4:	31.17	Theano/Theano
#5:	28.77	dmlc/mxnet
#6:	17.98	deeplearning4j/deeplearning4j
#7:	14.41	Microsoft/CNTK
#8:	13.37	torch/torch7
#9:	8.50	Lasagne/Lasagne
#10:	8.39	pfnet/chainer
#11:	7.77	mila-udem/blocks
#12:	7.33	NVIDIA/DIGITS
#13:	5.79	karpathy/convnetjs
#14:	5.48	NervanaSystems/neon
#15:	3.62	tflearn/tflearn
#16:	1.76	IDSIA/brainstorm
#17:	1.04	torchnet/torchnet

Top libraries by Github issues opened		
#1:	2808	BVLC/caffe
#2:	2348	fchollet/keras
#3:	2126	tensorflow/tensorflow
#4:	1652	Theano/Theano
#5:	1574	dmlc/mxnet
#6:	972	deeplearning4j/deeplearning4j
#7:	610	Microsoft/CNTK
#8:	503	mila-udem/blocks
#9:	468	pfnet/chainer
#10:	455	NVIDIA/DIGITS
#11:	386	Lasagne/Lasagne
#12:	328	torch/torch7
#13:	219	NervanaSystems/neon
#14:	147	tflearn/tflearn
#15:	82	IDSIA/brainstorm
#16:	41	karpathy/convnetjs
#17:	12	torchnet/torchnet

Top libraries by Github stars		
#1:	28092	tensorflow/tensorflow
#2:	11285	BVLC/caffe
#3:	7020	fchollet/keras
#4:	5806	Microsoft/CNTK
#5:	5098	karpathy/convnetjs
#6:	4963	torch/torch7
#7:	4420	dmlc/mxnet
#8:	4123	Theano/Theano
#9:	3375	deeplearning4j/deeplearning4j
#10:	3088	tflearn/tflearn
#11:	2286	Lasagne/Lasagne
#12:	2008	NervanaSystems/neon
#13:	1504	pfnet/chainer
#14:	1298	NVIDIA/DIGITS
#15:	1131	IDSIA/brainstorm
#16:	852	mila-udem/blocks
#17:	690	torchnet/torchnet

Top libraries by Github contributors		
#1:	311	tensorflow/tensorflow
#2:	241	Theano/Theano
#3:	209	fchollet/keras
#4:	199	BVLC/caffe
#5:	161	dmlc/mxnet
#6:	100	torch/torch7
#7:	78	deeplearning4j/deeplearning4j
#8:	73	Microsoft/CNTK
#9:	65	pfnet/chainer
#10:	49	Lasagne/Lasagne
#11:	48	mila-udem/blocks
#12:	41	NervanaSystems/neon

Top libraries by Github forks		
#1:	11428	tensorflow/tensorflow
#2:	6801	BVLC/caffe
#3:	2031	fchollet/keras
#4:	1640	dmlc/mxnet
#5:	1484	Theano/Theano
#6:	1417	torch/torch7
#7:	1223	Microsoft/CNTK
#8:	1183	deeplearning4j/deeplearning4j
#9:	993	karpathy/convnetjs
#10:	634	Lasagne/Lasagne
#11:	411	NervanaSystems/neon
#12:	389	NVIDIA/DIGITS

目录

TensorFlow的本质

从上一讲，DL里的构造Net的问题本质上只有两个：

- 简单表达式计算的向量（张量）化：比如$\exp(a)+b$，这里a, b都是向量
- 向量化的复合映射的求导

解决了这两个问题，就解决了BP算法中最核心的张量向前向后流动的问题，也就解决了编写Layer困难的问题。而大G站的TensorFlow正是解决这类编程问题的很好的工具。

顾名思义

Tensor(张量)Flow(流动)–正是解决张量计算和自动求导的工具

有了上面两个特性，我们甚至可以用TensorFlow解很多和DeepLearning无关的问题，比如传统的ML甚至一些优化问题。

要优雅

TensorFlow是如何优雅地解决这两个问题的？

解决方法

Computational Graph

TensorFlow可以看作是描述Computational Graph的DSL

下面看具体的例子，TensorFlow的每一次实际执行都被组织成一个个session，我们先准备一个session。

准备

```
from tensorflow import *
import numpy as np
sess = Session()
```

目录

Affine变换：矩阵乘法

一个简单的矩阵乘法

$$X = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \quad W = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \quad Y = X \cdot W$$

定义张量

```
X = constant([[1,2,3],[4,5,6],[7,8,9]],name='X')
W = constant([1,2,3], shape=[3,1], name='W')
```

矩阵乘法

```
Y = matmul(X, W, name = 'Y')
```

计算结果

```
Y0 = sess.run(Y)
>[[14]
> [32]
> [50]]
```

Affine变换：矩阵乘法的梯度

梯度张量$\frac{\partial Y}{\partial X}$

$$\left[\frac{\partial Y_i}{\partial X_{k,j}}\right]_{i,k,j} = \delta_{ik} \cdot W_j$$

梯度张量$\frac{\partial Y}{\partial W}$

$$\left[\frac{\partial Y_i}{\partial W_j}\right]_{i,j} = X_{ij} = X$$

TensorFlow里gradient计算给出的是$\sum_i \frac{\partial Y_i}{\partial \theta_j}$。
对上面的例子，结果是$\mathbb{1} \cdot W^T$和$X^T \cdot \mathbb{1}$.

Affine变换：矩阵乘法的梯度

gradients 能通过链式法则进行符号计算，求出相应的梯度。

求梯度

```
dX = gradients(Y, X, name='dX')
dW = gradients(Y, W, name='dW')
sess.run(dX[0])
>array([[1, 2, 3],
>       [1, 2, 3],
>       [1, 2, 3]], dtype=int32)

sess.run(dW[0])
>array([[12],
>       [15],
>       [18]], dtype=int32)
```

检验结果

```
II = ones((3,1),int32)
sess.run(reduce_all(equal(dX[0] , matmul(II, transpose(W)))))
>True
sess.run(reduce_all(equal(dW[0] , matmul(transpose(X), II))))
>True
```

ReLU – maximum

maximum 做elementwise的比较，返回大的；而且它正确处理了反向求gradient的计算。

local gradients

```
Z = constant([-1,2,0], shape=[3,1], name='Z')
sess.run(gradients(maximum(Z,0), Z))
>[array([[0],
>        [1],
>        [1]], dtype=int32)]
```

注意

0点不可导，TensorFlow的处理，就是令0点的导数为一。

上方有回传的gradient时，ReLU相当于一个开关

```
sess.run(gradients(maximum(Z,0), Z, grad_ys=constant([100,99,98], shape=(3,1))))
>[array([[ 0],
>        [99],
>        [98]], dtype=int32)]
```

上层传来的gradient是$(100, 99, 98)$但只有2, 3位置通过了。

知足而常乐

毕竟这里的符号计算只是规则引擎，能处理链式法则，但不能真的求解数学问题，比如：

隐函数/参数表示的求导

$$x^2 + y^2 = 1 \quad \text{or} \quad \begin{cases} x &= \cos(t) \\ y &= \sin(t) \end{cases}$$

那么，

$$\frac{\mathrm{d}\,y}{\mathrm{d}\,x} = \frac{-\sin(t)\,\mathrm{d}\,t}{\cos(t)\,\mathrm{d}\,t} = -\frac{x}{y}$$

实验：参数方程求gradient

```
t = constant(1.0)
x = cos(t)
y = sin(t)
gradients(y, x)
>[None]
```

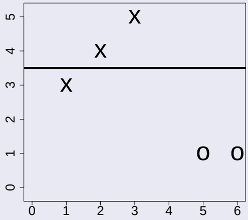
显化：对比正确值

```
sess.run(gradients(sin(acos(x)), x))
>[-0.64209253]

sess.run(-x/y)
>-0.64209259
```

目录

Perceptron例子：问题

问题:平面上找直线分离正例和负例



定义数据和参数

```
# training data
X = constant([[5,1],[6,1],[1,3],[2,4],[3,5]], name='data', dtype=float32)
Y = constant([1, 1, -1, -1, -1], shape=[5,1], name='label', dtype=float32)
# parameters
W = Variable(zeros((2,1),dtype=float32))
b = Variable(1, dtype=float32)
```

Perceptron例子：Loss函数

使用maxmial margin的Loss

Score是

$$S = X \cdot W + b$$

我们希望在$Y = 1$的score越大越好，$Y = -1$的score越小越好，所以Loss是

$$l = (1 - S_{Y>0})_+ + (S_{Y<0} + 1)_+$$

最终分类器的输出是$\mathrm{sign}(S)$，所以训练集上Accuracy是

$$\{i|\,\mathrm{sign}(S_i) = \mathrm{sign}(Y_i)\}\text{的个数}/\text{样本数量}$$

Score，Loss和正确率的计算公式

```
# compute score, loss and accuracy
S    = matmul(X, W) + b
loss = reduce_sum(maximum(1-S[0:1,0],0)) + reduce_sum(maximum(S[2:4,0]+1,0))
accuracy = reduce_mean(to_float(equal(sign(S) , sign(Y))))
```

Perceptron例子：求解

回忆：最简单的参数训练方式

$$W \leftarrow W - r \cdot \mathrm{d}W$$

这里r是learning rate。

定义BP的update rule

```
#learning rate
lr = 0.1
#updating rule
dW, db = gradients(loss, [W,b])
update_param = group( W.assign_sub( lr * dW), b.assign_sub( lr * db ) )
```

梯度下降求解

```
it = 0
while True:
    it = it + 1
    l, acc = sess.run([loss, accuracy])
    print "Iter : %d Loss = %.2f Accuracy=%.2f%%\n" % (it, l, acc*100)
    if (l<0.05):
        break
    sess.run(update_param)
```

Perceptron结果

结果

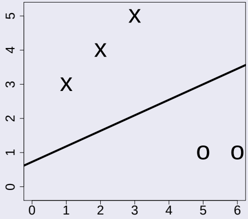
```
>Iter : 1 Loss = 4.00 Accuracy=40.00%
>
>Iter : 2 Loss = 0.30 Accuracy=100.00%
>
>Iter : 3 Loss = 0.60 Accuracy=100.00%
>
>Iter : 4 Loss = 0.10 Accuracy=100.00%
>
>Iter : 5 Loss = 0.00 Accuracy=100.00%

sess.run(S)

>array([[ 2.19999981],
>       [ 2.69999981],
>       [-2.00000024],
>       [-2.60000014],
>       [-3.20000005]], dtype=float32)

sess.run([W,b])

>[array([[ 0.5        ],
>        [-1.10000002]], dtype=float32), 0.79999995]
```

Perceptron结果：可视化

分离直线的方程

从上面的$W = (0.5, -1.1)^T$, $b = 0.8$，我们知道方程是

$$0.5x_1 - 1.1x_2 + 0.8 = 0$$



可以看到我们选取的Loss函数确实使得直线落在比较好的位置。

Perceptron：直接使用优化器

优化器(Optimizer)类可以产生参数的updater，可以取代手写的update_param 。

使用GradientDescentOptimizer取代update_param

```
opt = train.GradientDescentOptimizer(lr)
update_using_opt = opt.minimize(loss)
it = 0
while True:
    it = it + 1
    l, acc = sess.run([loss, accuracy])
    print "Iter : %d Loss = %.2f Accuracy=%.2f%%\n" % (it, l, acc*100)
    if (l<0.05):
        break
    sess.run(update_using_opt)

>Iter : 1 Loss = 4.00 Accuracy=40.00%
>
>Iter : 2 Loss = 0.30 Accuracy=100.00%
>
>Iter : 3 Loss = 0.60 Accuracy=100.00%
>
>Iter : 4 Loss = 0.10 Accuracy=100.00%
>
>Iter : 5 Loss = 0.00 Accuracy=100.00%
```

结果也是一样的。

Perceptron：使用cross entropy loss

使用softmax_cross_entropy_with_logits

```
X = constant([[5,1],[6,1],[1,3],[2,4],[3,5]], name='data', dtype=float32)
# label changed to one_hot format
label_one_hot = one_hot([0,0,1,1,1], 2)
W = Variable(zeros((2,1),dtype=float32))
b = Variable(1, dtype=float32)

S    = matmul(X, W) + b
# score need to be [S,1] format
loss = reduce_mean(nn.softmax_cross_entropy_with_logits(pad(S,[[0,0],[0,1]]),
      label_one_hot, name='cross_entropy'))
# use sign(label_one_hot - 0.5) generate +/-1
accuracy = reduce_mean(to_float(equal(sign(S) , sign(matmul(label_one_hot, constant
      ([1.0,-1.0],shape=(2,1)))))))

#following is same as previouse
#learning rate
lr = 0.1
#updating rule
opt = train.GradientDescentOptimizer(lr)
update_using_opt = opt.minimize(loss)
it = 0
while True:
    it = it + 1
    l, acc = sess.run([loss, accuracy])
    print "Iter : %d Loss = %.2f Accuracy=%.2f%%\n" % (it, l, acc*100)
    if (l<0.05):
        break
    sess.run(update_using_opt)
```

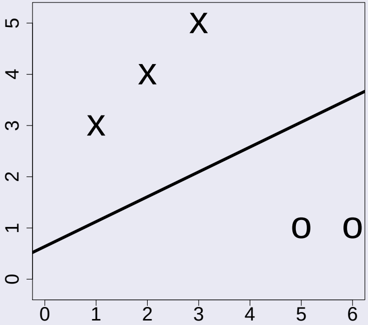
Perceptron：使用cross entropy loss的结果

结果

```
>Iter : 1 Loss = 0.91 Accuracy=40.00%
>
>Iter : 2 Loss = 0.67 Accuracy=40.00%
>
>Iter : 3 Loss = 0.54 Accuracy=80.00%
>
>Iter : 4 Loss = 0.45 Accuracy=100.00%
>
>. . .
>
>Iter : 47 Loss = 0.05 Accuracy=100.00%
>
>Iter : 48 Loss = 0.05 Accuracy=100.00%

sess.run(S)

>array([[ 2.71125937],
>        [ 3.34846902],
>        [-2.4624207 ],
>        [-3.13763237],
>        [-3.81284356]], dtype=float32)

sess.run([W,b])

>[array([[ 0.63720953],
>         [-1.31242108]], dtype=float32), 0.83763283]
>
```

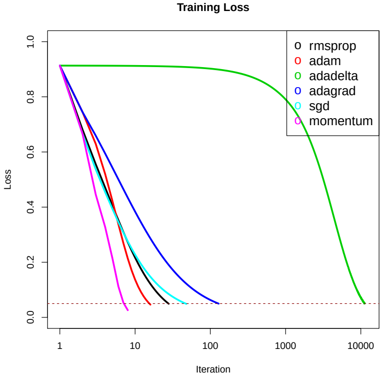
Perceptron：使用cross entropy loss的结果

$$0.6372x_1 - 1.3124x_2 + 0.8376 = 0$$



虽然迭代次数多了（因为loss的尺度不同，不能直接比较），最终效果和SVM loss差不多

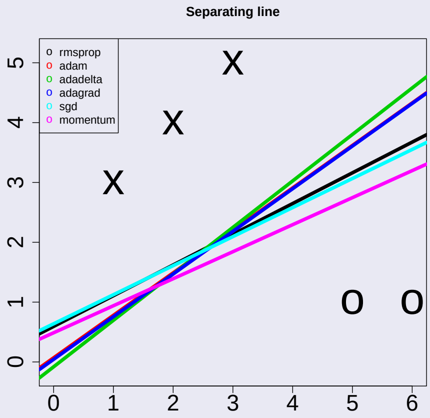
Perceptron：比较不同优化器的结果



	k	b	迭代次数
rmsprop	0.51	0.59	28.00
adam	0.71	0.07	16.00
adadelta	0.78	-0.08	11241.00
adagrad	0.71	0.04	129.00
sgd	0.49	0.64	48.00
momentum	0.45	0.49	8.00

Perceptron比较不同优化器的结果

分离超平面



课后的欢乐小剧场

“他强由他强，清风拂山岗；他横由他横，明月照大江。他自狠来他自恶，我自一口真气足。。。”

“姐姐姐姐，人家是感慨一下理解了基础原理多么重要啊”

“什么鬼，雷姆雷姆，你听课听地画风都变啦”

下周预告

“回顾一下，今天讨论了
● GPU计算的概念和困难
● Caffe框架的结构
● TensorFlow的开发理念
好像有不少收获呢！”

“DeepLearning看起来也不是那么神秘嘛！比起魔法差多了。。。”

“还是姐姐的魔法最棒了！下次我们将开始这场旅程最后的探险，哦。。。”