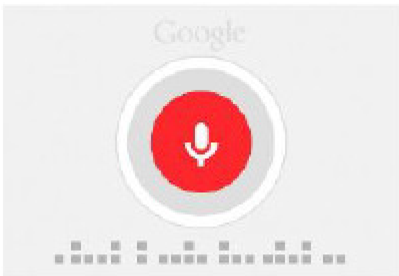
Introduction to Caffe Framework

Der-Hau Lee

Caffe: Convolutional Architecture for Deep Learning

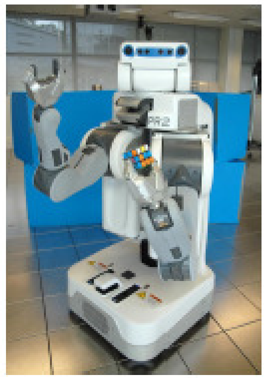
- Paper: Yangqing Jia *et al*., Proceedings of the 22nd ACM international conference on Multimedia, 675 (2014)
- Citation: 5896
- Deep learning: end-to-end learning for many tasks

vision

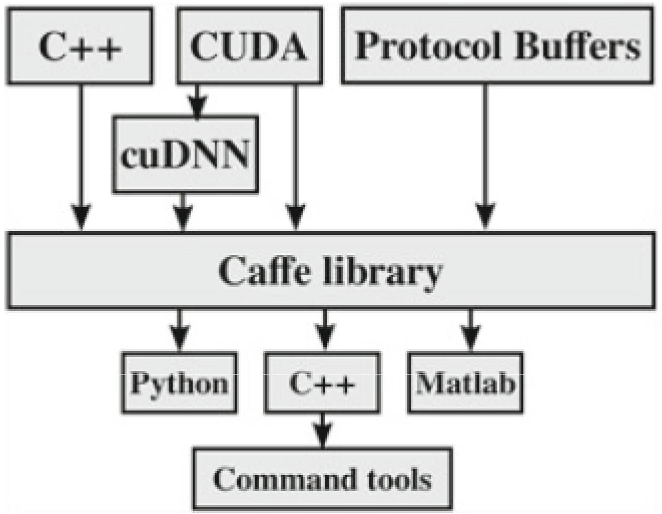


speech

text


control

Third-party libraries in Caffe



- C++
- CUDA & cuDNN: for GPU computation
- Protocol Buffers: for model format

Protocol Buffers (Protobuf)

- Google發展Protobuf以處理big data。
- 傳輸優點: Protobuf編譯器可自動生成C++、Python等程式碼。
- 執行優點: Protobuf語法簡潔，如省去了不必要的 { 或 :。
- Caffe利用Protobuf作為model & parameter的輸入方式。
- Protobuf語法以”message”為基本組成單位。

Helloworld programming

lm.helloworld.proto

compiler

Protobuf (C++)

lm.helloworld.pb.cc

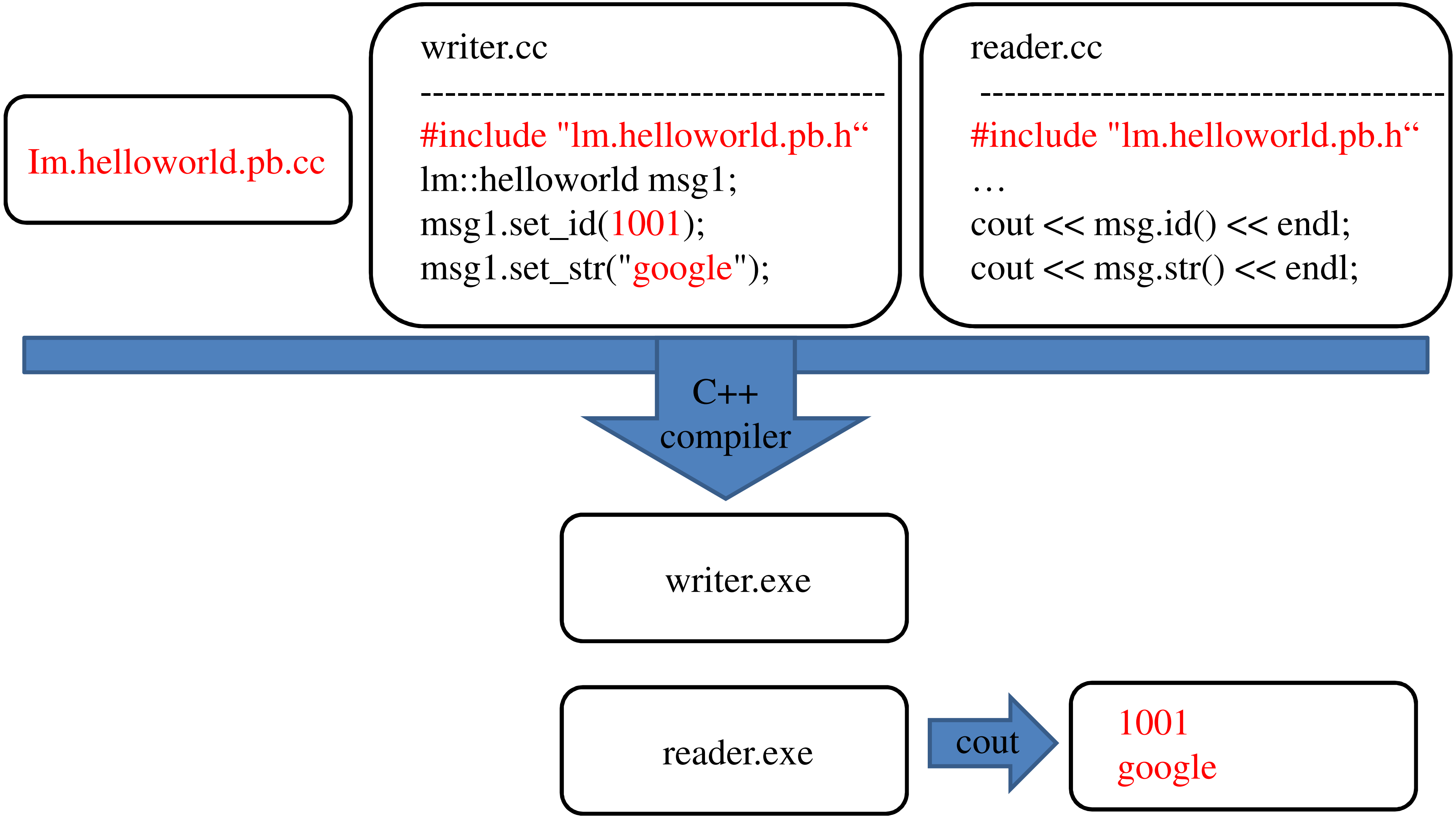
lm.helloworld.pb.h

```
syntax = "proto2";
package lm;

message helloworld {
    required int32      id = 1;
    required string     str = 2;
    optional int32      opt = 3;
}
```

Structure (Class) in C/C++

```
struct person
{
    char name[50];
    int citNo;
    float salary;
};

int main()
{
    struct person person1, person2, person3[20];
    return 0;
}
```

Helloworld programming



Anatomy of a Caffe model

- Blob: storage and communication
- Layer: the fundamental unit of computation

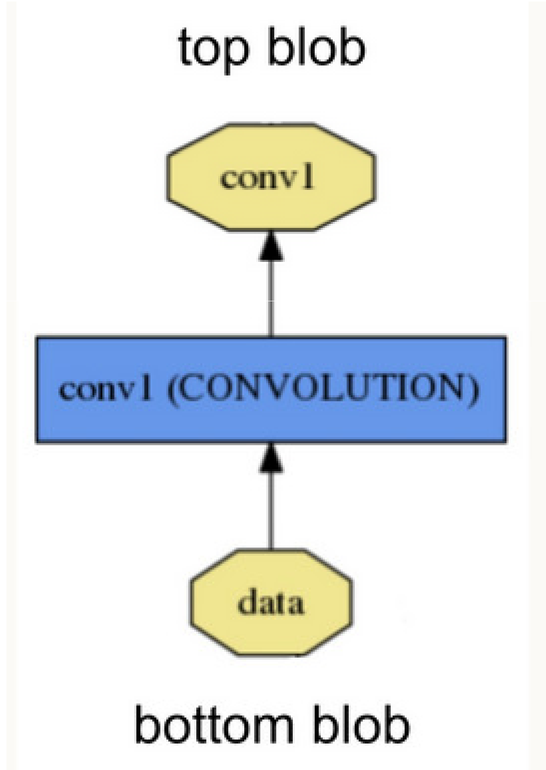
✓Data Layer

✓Convolutional Layer

✓Pooling Layer

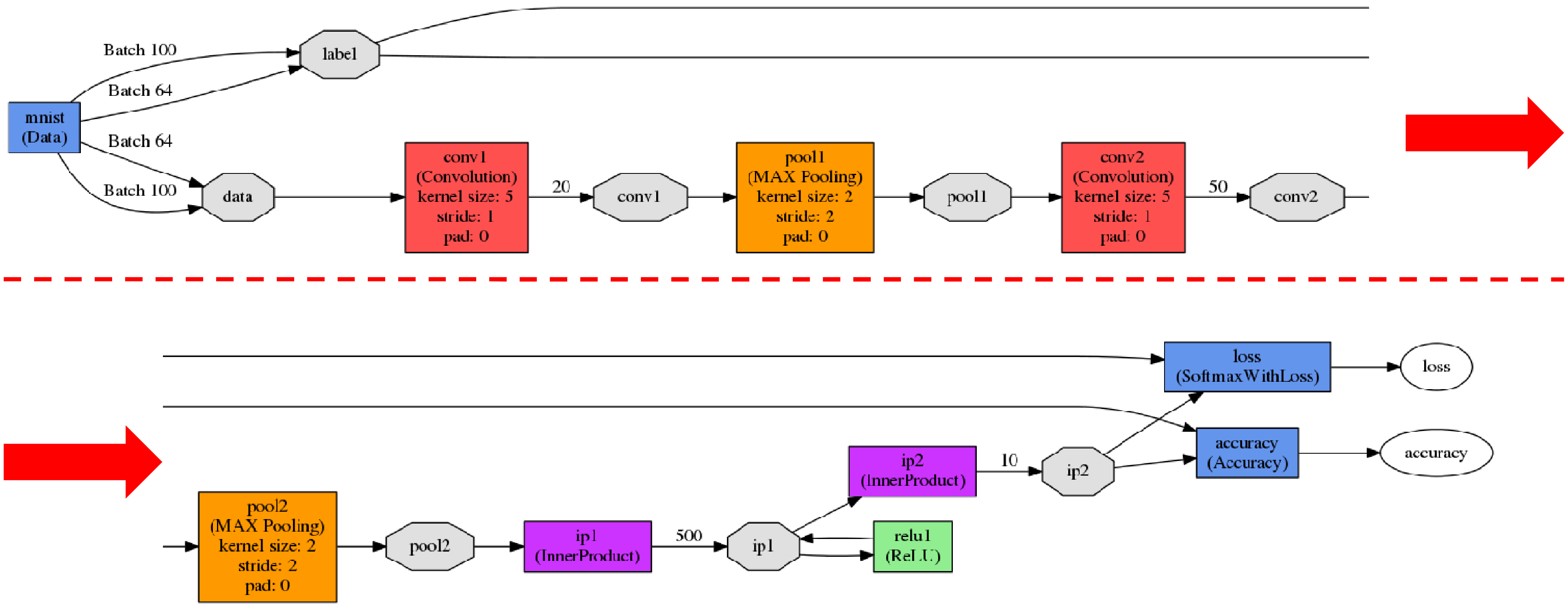
✓ReLU Layer

✓Loss Layer

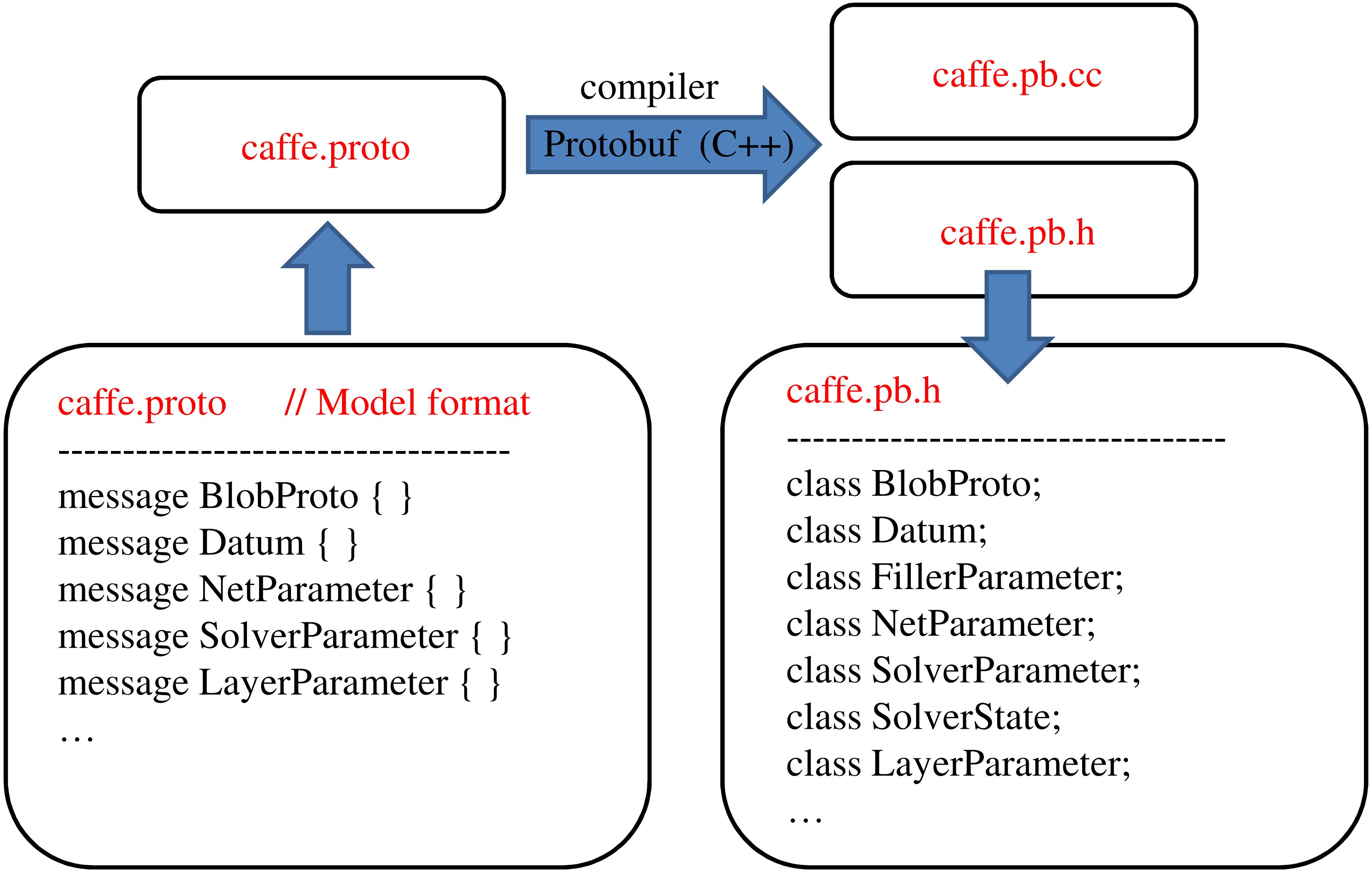


Anatomy of a Caffe model

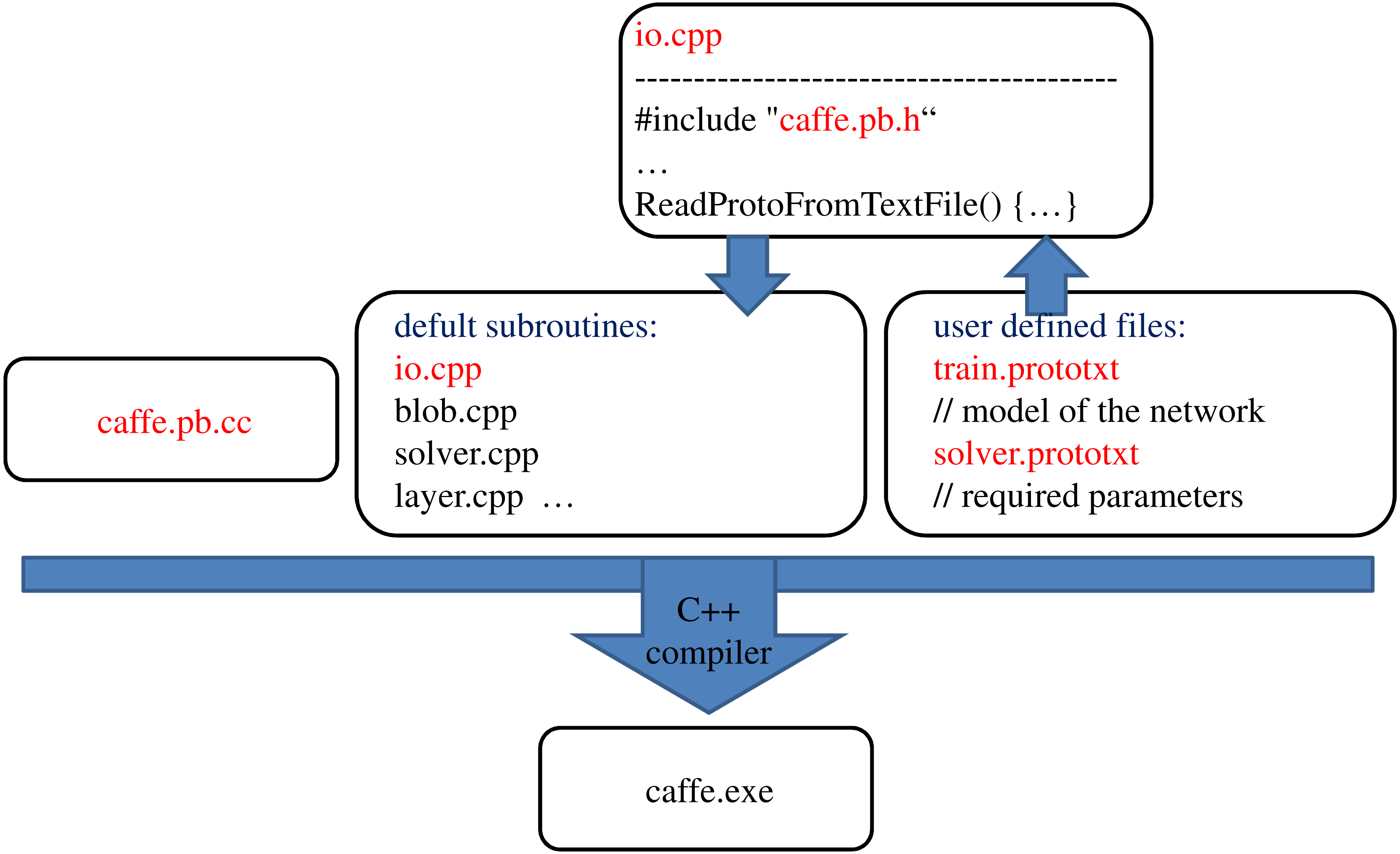
- Net (network): a set of layers and their connections in a plaintext modeling language.
- LeNet (Yann LeCun, 1998) on MNIST data:



Caffe programming



Caffe programming



Input file for training LeNet

solver.prototxt

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

train.prototxt

Data Layer

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

Convolutional Layer

```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
...
```

Pooling Layer

```
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
...
```

Data Layer (type:image, simple)

German Traffic Sign Benchmarks

train.prototxt

```
name:"net1"
layer{
      name:"data"
      type:"ImageData"
      top:"data"
      top:"label"
      image_data_param{
            source:"/home/pc/Desktop/train.txt"
            batch_size:30
            root_folder:"/home/pc/Desktop/"
            is_color:true
            shuffle:true
            new_width:32
            new_height:32
      }
}
```

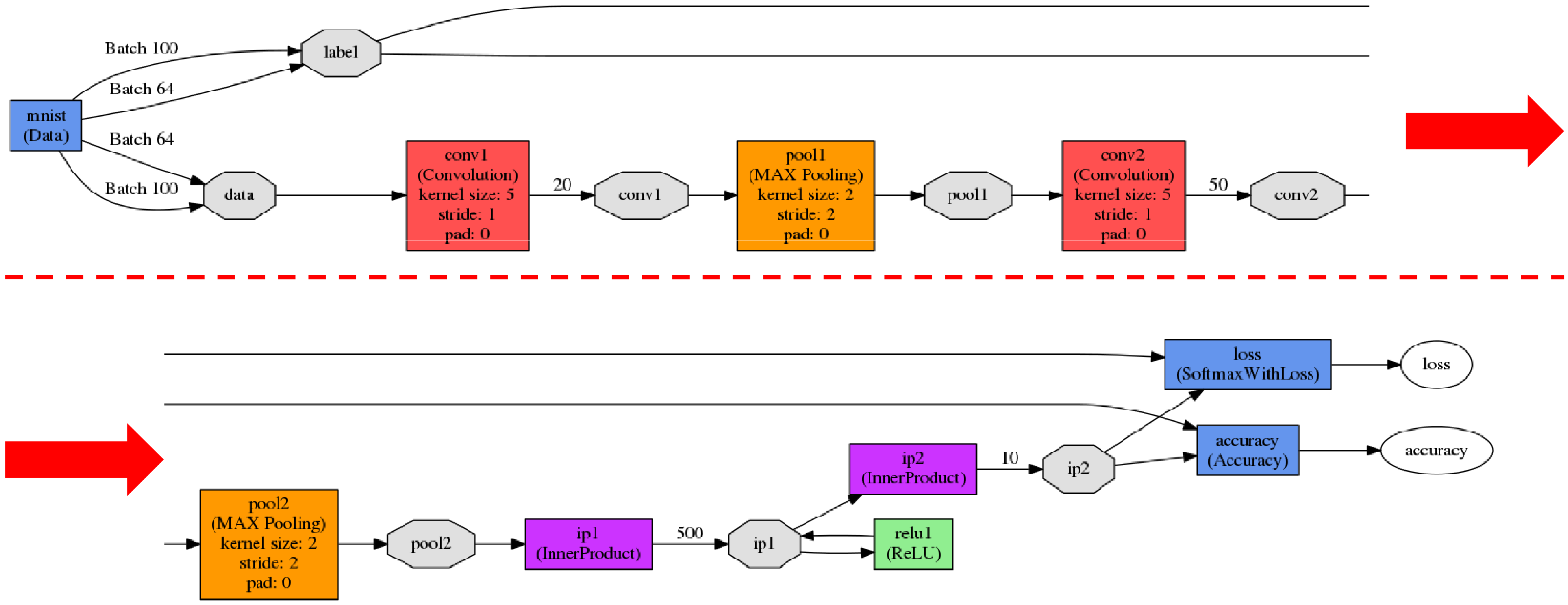
train.txt Label

.ppm image class

```
/00019/00000_00006.ppm 19
/00029/00003_00021.ppm 29
/00010/00054_00008.ppm 10
/00023/00010_00027.ppm 23
/00033/00022_00008.ppm 33
/00021/00000_00005.ppm 21
/00005/00020_00022.ppm 5
/00025/00026_00018.ppm 25
...
```

class=1 class=21 class=42

LeNet on MNIST data



Goal: understanding this plot !

Data Layer (type:data, efficient)

LeNet on MNIST

train.prototxt

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

Input: 1X28X28 image
--1 channel (black/white)

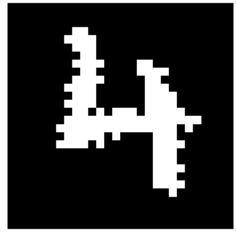
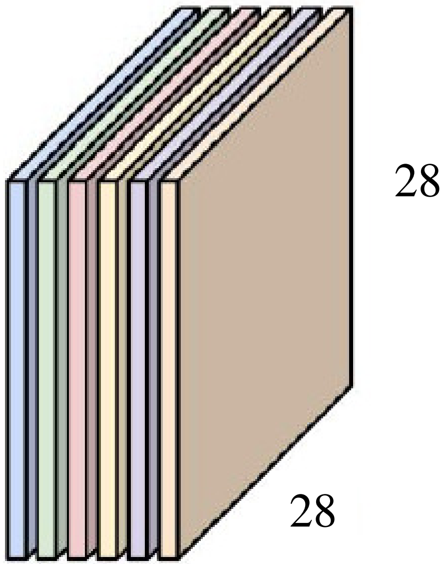


Image format
--Lightning Memory-Mapped Database Manager (LMDB)
-- high performance and memory-efficient



Batch size = 64 (per iteration)

Implementation of the Convolutional Layer

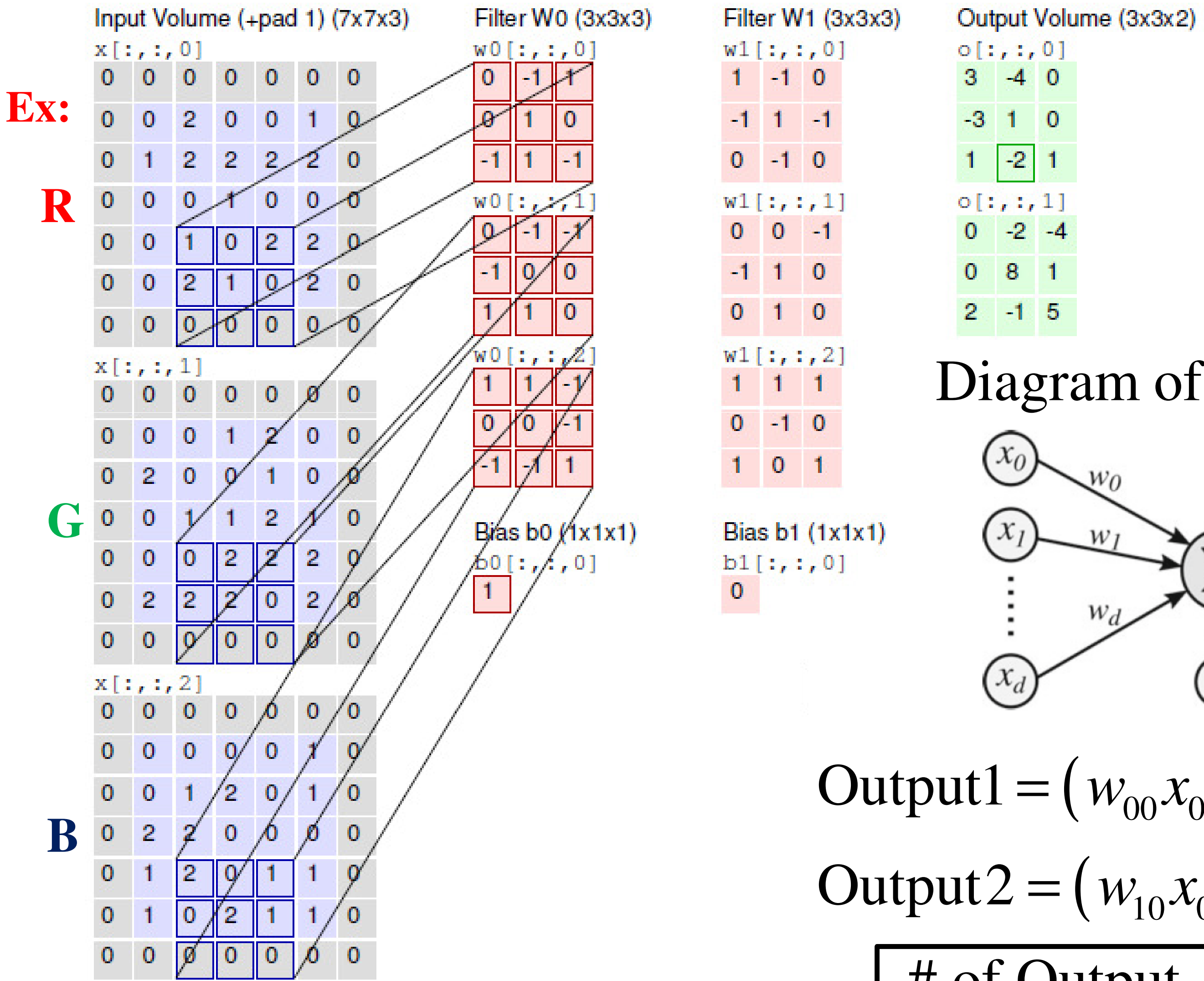
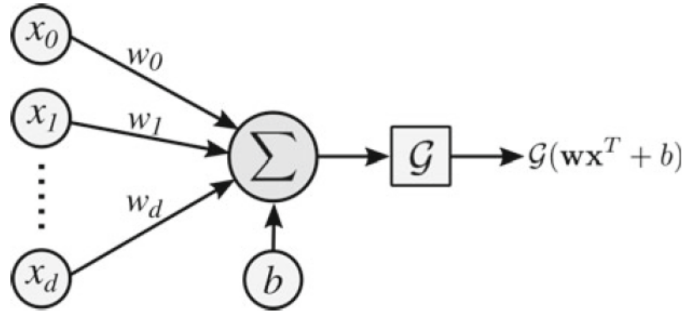


Diagram of an artificial neuron



$$\text{Output1} = \left(w_{00}x_0 + w_{01}x_1 + w_{02}x_2 \right) + b_0$$

$$\text{Output2} = \left(w_{10}x_0 + w_{11}x_1 + w_{12}x_2 \right) + b_1$$

$$\boxed{\text{\# of Output} = \text{\# of Filters}}$$

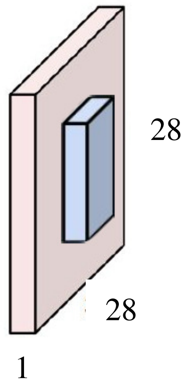
http://cs231n.github.io/convolutional-networks/#architectures

Convolutional Layer (1/2)

LeNet on MNIST

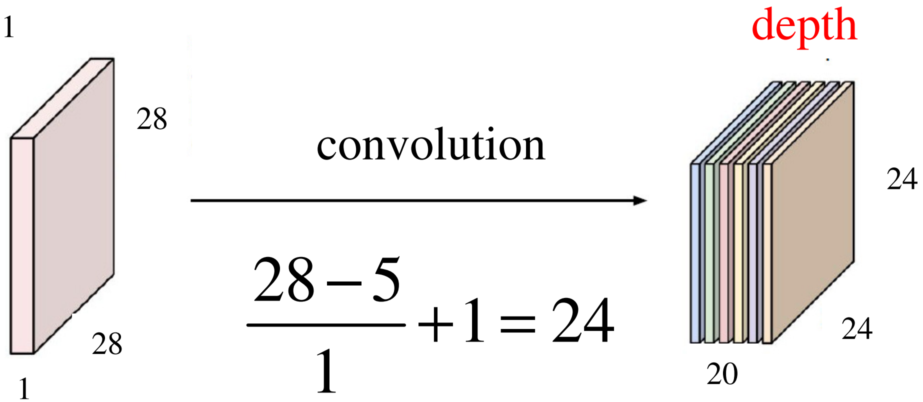
train.prototxt

```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 20
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
```



Input
-- size = 1X28X28

filter
-- filter num is 20, size is 5X5
-- stride = 1



$$\frac{28-5}{1}+1=24$$

	Output volume	Memory (MB)
Batch	64	0.000244
Input	64*1*28*28	0.191406
conv1	64*20*24*24	2.8125

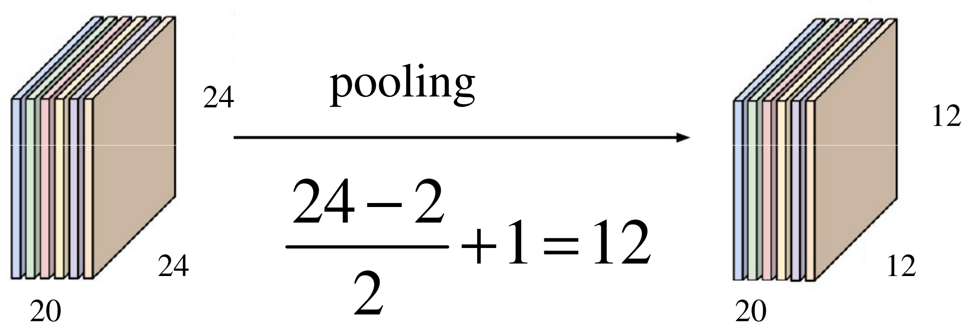
Pooling Layer (1/2)

LeNet on MNIST

train.prototxt

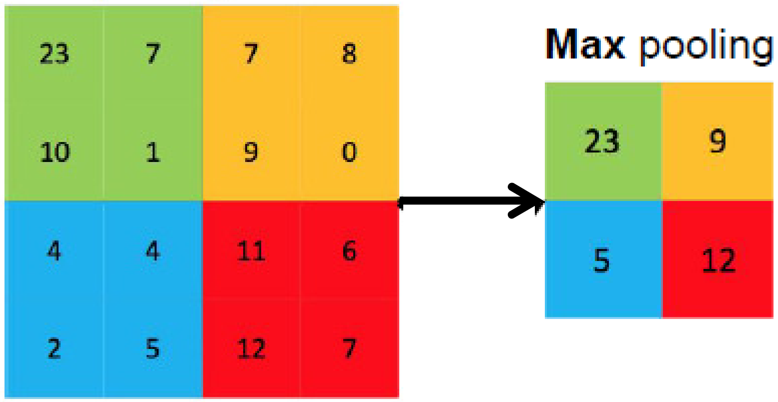
```
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}
```

pooling window
--2X2 max pooling
-- stride = 2



$$\frac{24-2}{2}+1=12$$

2x2 pooling, stride 2



	Output volume	Memory (MB)
Batch	64	0.000244
Input	64*1*28*28	0.191406
conv1	64*20*24*24	2.8125
pool1	64*20*12*12	0.703125

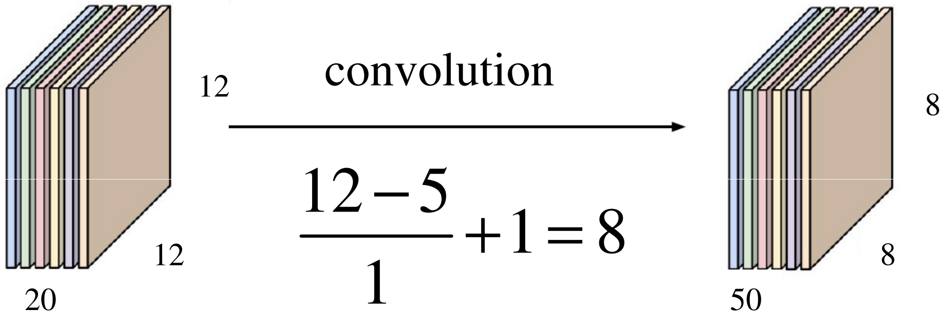
Convolutional Layer (2/2)

LeNet on MNIST

train.prototxt

```
layer {
  name: "conv2"
  type: "Convolution"
  bottom: "pool1"
  top: "conv2"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 50
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
```

filter
-- filter num is 50, size = 5X5
-- stride = 1



$$\frac{12-5}{1}+1=8$$

	Output volume	Memory (MB)
Batch	64	0.000244
Input	64*1*28*28	0.191406
conv1	64*20*24*24	2.8125
pool1	64*20*12*12	0.703125
conv2	64*50*8*8	0.78125

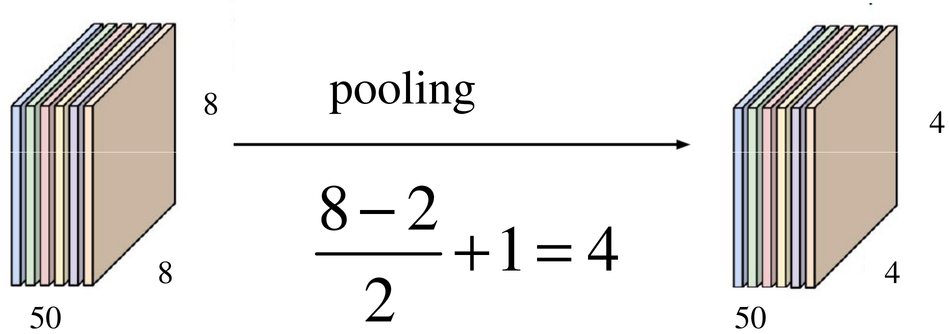
Pooling Layer (2/2)

LeNet on MNIST

train.prototxt

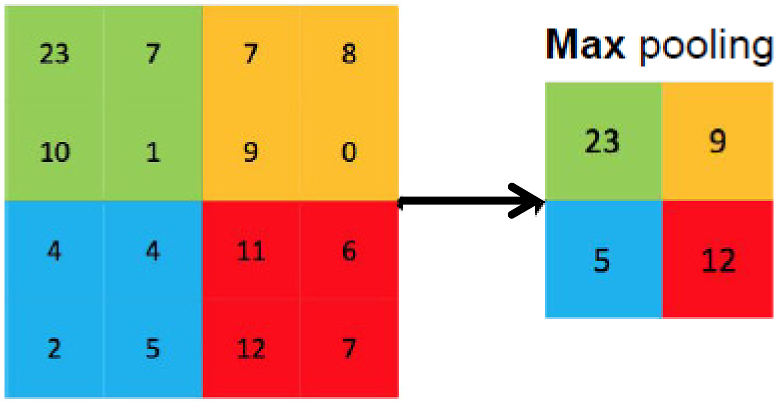
```
layer {
  name: "pool2"
  type: "Pooling"
  bottom: "conv2"
  top: "pool2"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}
```

pooling window
--2X2 max pooling
-- stride = 2



$$\frac{8-2}{2}+1=4$$

2x2 pooling, stride 2



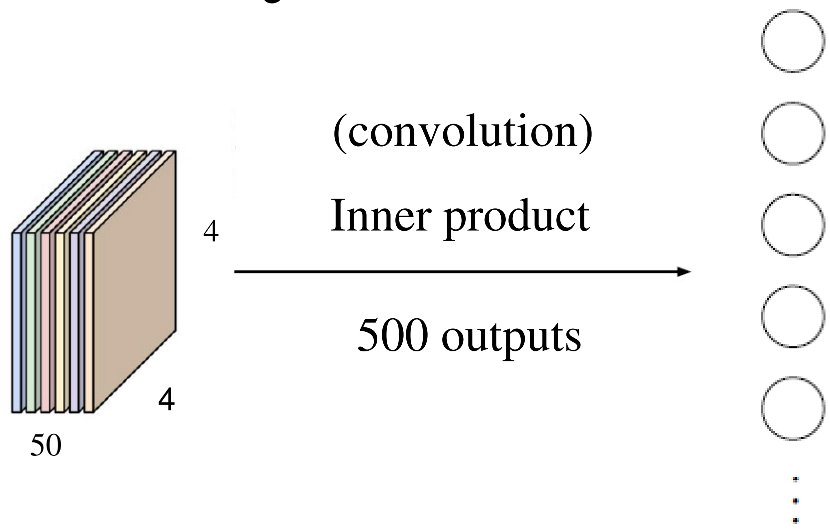
	Output volume	Memory (MB)
Batch	64	0.000244
Input	64*1*28*28	0.191406
conv1	64*20*24*24	2.8125
pool1	64*20*12*12	0.703125
conv2	64*50*8*8	0.78125
pool2	64*50*4*4	0.195313

Inner Product Layer (1/2)

LeNet on MNIST

train.prototxt

```
layer {
  name: "ip1"
  type: "InnerProduct"
  bottom: "pool2"
  top: "ip1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 500
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
```



	Output volume	Memory (MB)
Batch	64	0.000244
Input	64*1*28*28	0.191406
conv1	64*20*24*24	2.8125
pool1	64*20*12*12	0.703125
conv2	64*50*8*8	0.78125
pool2	64*50*4*4	0.195313
ip1	64*500	0.12207

ReLU Layer

LeNet on MNIST

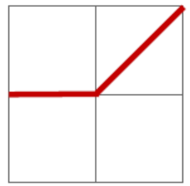
train.prototxt

```
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "ip1"
  top: "ip1"
}
```

Non-linearity: deepen the representation

ReLU

$$x' = \max(0, x)$$



Sigmoid

$$x' = 1/(1 + e^{-x})$$

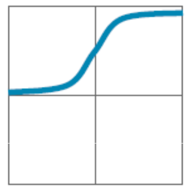
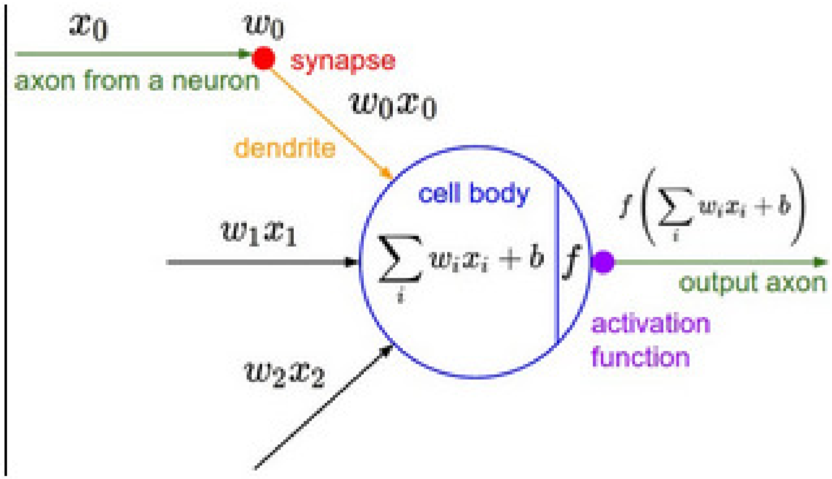


Diagram of an artificial neuron



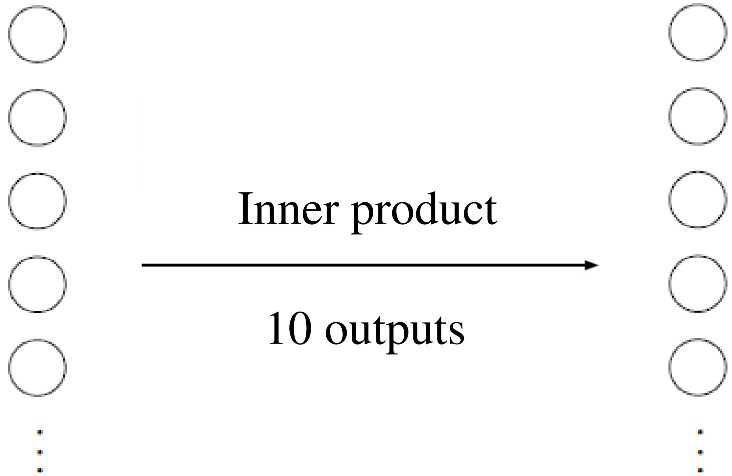
	Output volume	Memory (MB)
Batch	64	0.000244
Input	64*1*28*28	0.191406
conv1	64*20*24*24	2.8125
pool1	64*20*12*12	0.703125
conv2	64*50*8*8	0.78125
pool2	64*50*4*4	0.195313
ip1	64*500	0.12207
ReLU	64*500	0.12207

Inner Product Layer (2/2)

LeNet on MNIST

train.prototxt

```
layer {
  name: "ip2"
  type: "InnerProduct"
  bottom: "ip1"
  top: "ip2"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 10
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
```



	Output volume	Memory (MB)
Batch	64	0.000244
Input	64*1*28*28	0.191406
conv1	64*20*24*24	2.8125
pool1	64*20*12*12	0.703125
conv2	64*50*8*8	0.78125
pool2	64*50*4*4	0.195313
ip1	64*500	0.12207
ReLU	64*500	0.12207
ip2	64*10	0.002441

Loss Layer (TRAIN phase)

LeNet on MNIST

train.prototxt

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

Cross-entropy as loss function

$$H(p,q) = -\sum_x p(x)\log\left[q(x)\right]$$

$$= -\sum_x \underbrace{p(x)}_{\text{true}} \log \underbrace{\left[\frac{e^{z_{y_i}}}{\sum_j e^{z_j}}\right]}_{\text{estimated}}$$

where $p = [0,\ldots,1,\ldots 0]$

contains a single 1 at the y_i-th position

http://cs231n.github.io/linear-classify/

Softmax classifier

$$f_j(z) = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

provides "probabilities" for each class

	Output volume	Memory (MB)
Batch	64	0.000244
Input	64*1*28*28	0.191406
conv1	64*20*24*24	2.8125
pool1	64*20*12*12	0.703125
conv2	64*50*8*8	0.78125
pool2	64*50*4*4	0.195313
ip1	64*500	0.12207
ReLU	64*500	0.12207
ip2	64*10	0.002441
loss	1	0.00000381
total		4.930424

Forward + Backward = 9.86 MB

Accuracy Layer (TEST phase)

LeNet on MNIST

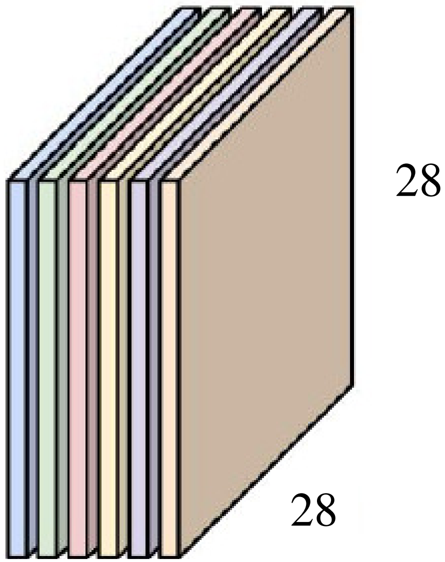
train.prototxt

```
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TEST
  }
  transform_param {
    scale: 0.00390625
  }
  data_param {
    source: "examples/mnist/mnist_test_lmdb"
    batch_size: 100
    backend: LMDB
  }
}
```

1) # of test samples = 10000
2) test phase execute every 500 iterations of train phase (500*64 train samples)
3) test phase has 10000/100 = 100 iterations

Compute accuracy on test samples

```
layer {
  name: "accuracy"
  type: "Accuracy"
  bottom: "ip2"
  bottom: "label"
  top: "accuracy"
  include {
    phase: TEST
  }
```



Batch size = 100 (per iteration)
Memory = 7.71 MB (does not need backward)

Define the MNIST Solver

solver.prototxt

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

Caffe solver

-- coordinate the network's forward inference and backward gradients.

-- optimization algorithms

Stochastic Gradient Descent (SGD)

AdaDelta (AdaDelta)

Adaptive Gradient (AdaGrad)

Adam (Adam)

Nesterov's Accelerated Gradient (Nesterov)

RMSprop (RMSProp)

Parametric Learning

How do we find the label-prediction function f?

Parametric answer: pick it from a family determined by a set of *parameters* θ:

$$f(x) = f(x; \theta)$$

E.g. $f(x; \theta) = \overset{\text{matrix}}{\theta}\overset{\text{vector}}{x}$ "linear prediction"

For us: f is a *network*, θ is a set of *weights*

Parametric Supervised Learning

Altogether: our goal is to find θ in order to

loss   true label   predicted label

$$\text{minimize } L(\theta) = \sum_n \ell(y_n, \hat{y}_n) = \sum_n \ell(y_n, f(x_n; \theta))$$

sum over data   model (network)   parameters (weights)

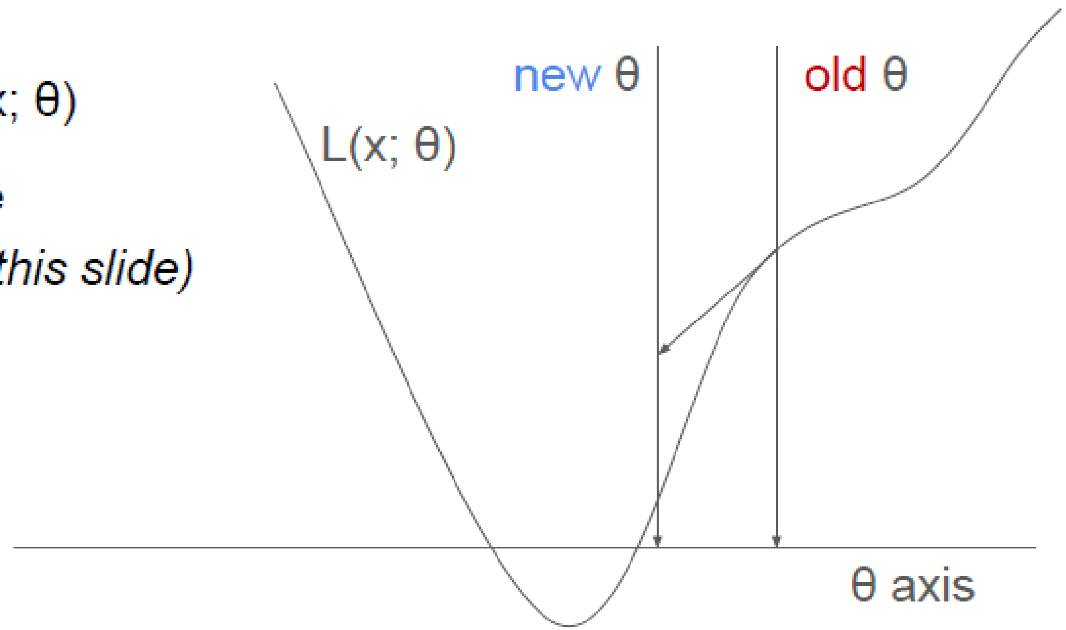
Caffe Tutorial Slides

Gradient Descent: Intuition

Want to minimize "loss" function L(x; θ)

θ (vector): parameter to update

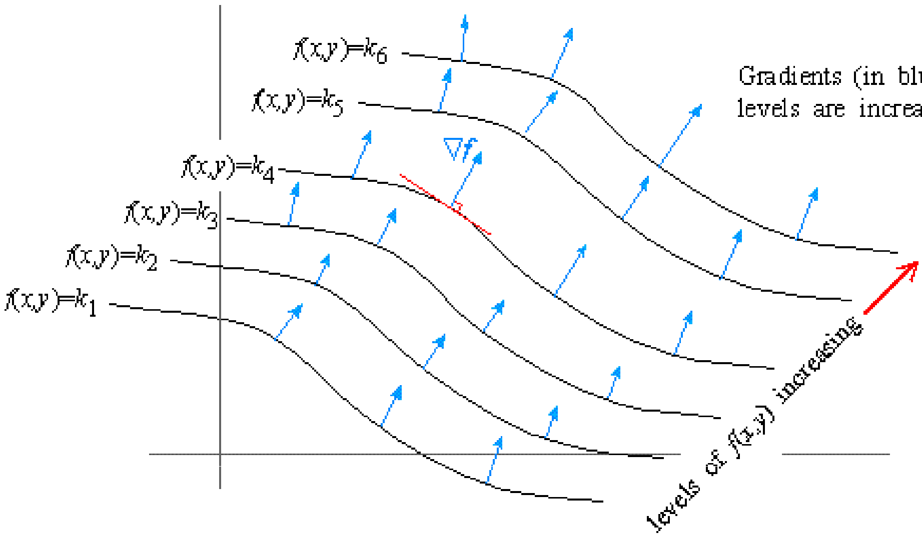
x (vector): input data *(fixed on this slide)*



Move in the direction of the gradient

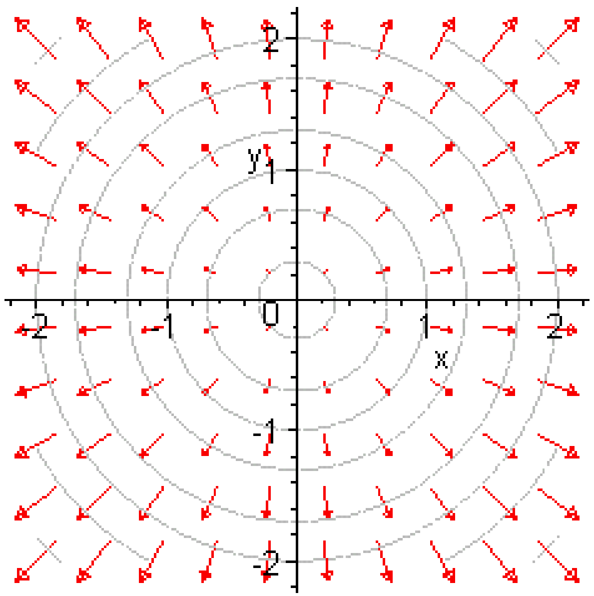
The gradient tells you, for each element of the network parameters,
how the loss changes in response to a change in that parameter.

Steepest descent



For $f = f(x, y)$

gradient field $\nabla f \equiv \dfrac{\partial f}{\partial x}\hat{i} + \dfrac{\partial f}{\partial y}\hat{j}$



Ex: $U(x, y) = x^2 + y^2$

gradient field $\nabla U = 2x\hat{i} + 2y\hat{j}$

The levels of U in direction

$$= \begin{cases} \nabla U : \text{ increasing most quickly (in red)} \\ -\nabla U : \text{ decreasing most quickly} \\ \qquad\qquad \text{(steepest descent)} \end{cases}$$

Stochastic Gradient Descent (SGD)

Want to minimize "loss" function L(x; θ)

1. Pick input datum x

$$\overbrace{\theta \leftarrow \theta -}^{\textbf{4. }\text{Update parameters }\theta} \alpha \underbrace{\nabla_{\theta} L}_{\textbf{2. }\text{Compute parameter gradient}}(x;\theta)$$

3. Multiply by learning rate

Caffe Tutorial Slides

Why “Stochastic”?

The gradient depends on the choice of input datum x

Choose x *randomly* (or just cycle through all data in a fixed order)

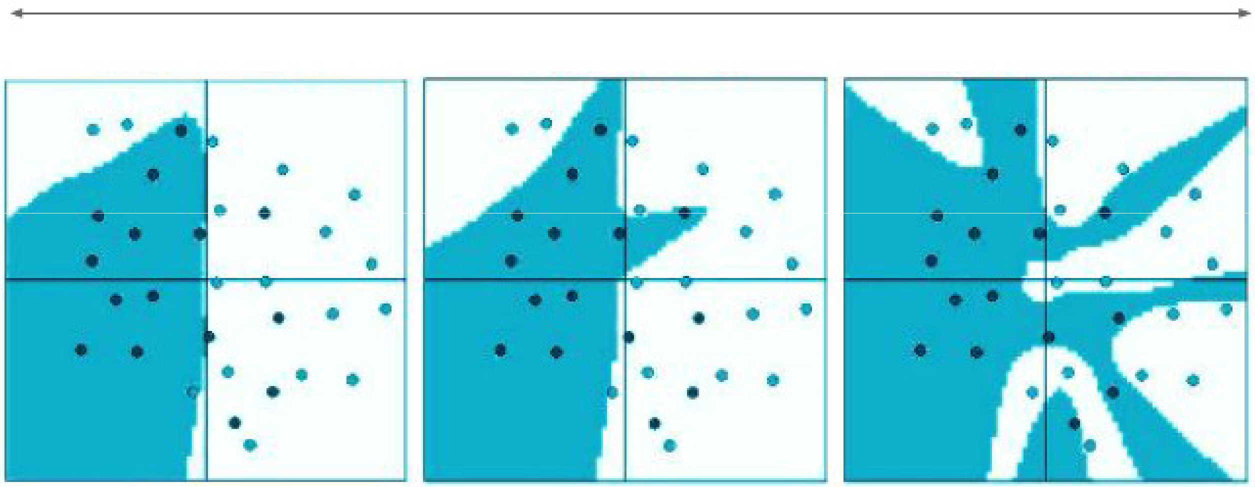
(The alternative is to average the gradient over all available data, “batch gradient descent”:

$$\theta \leftarrow \theta - \alpha \sum_i \nabla_\theta L(x_i; \theta)$$

That’s too slow for big data!)

Underfitting and Overfitting

fewer parameters more parameters



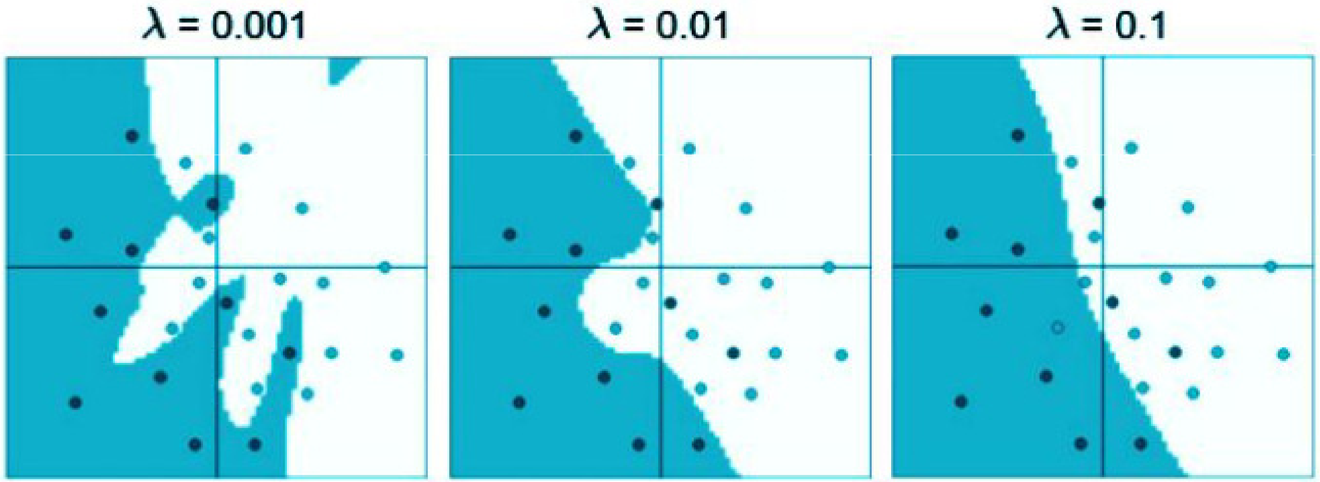
underfitting:
not enough parameters
to model the data

overfitting:
enough parameters to
memorize the training
set without generalizing

Caffe Tutorial Slides

Regularization

How can we prevent overfitting without reducing the number of parameters?



Add a *regularization penalty* to our loss: “complicated” solutions are worse

Caffe Tutorial Slides

Regularization: Weight Decay and Dropout

Weight Decay: minimize $L(\theta) + \lambda\|\theta\|^2$ to pull weights toward zero

λ (scalar) is an optimization setting… pick it empirically

aka "L^2 regularization"

Dropout: during training, randomly set a fraction p of activations to zero

p is an optimization setting (often 0.5)

forces model to be robust to noise

SGD with Weight Decay and Momentum

$$\theta \leftarrow \theta - \alpha \ \nabla_\theta L(x;\theta)$$

SGD with Weight Decay and Momentum

$$\theta \leftarrow \theta - \alpha \left(\nabla_\theta L(x;\theta) - \lambda\theta \right)$$

weight decay
(*regularization*)

Regularization term:

1) regularization term makes the weights smaller.

2) smaller weights → lower complexity → provide a simpler and more powerful explanation for the data.

SGD with Weight Decay and Momentum

$$\theta \leftarrow \theta - \alpha\left(\nabla_\theta L(x;\theta) - \underset{\substack{\uparrow\\ \text{weight decay}\\ (\textit{regularization})}}{\lambda\theta}\right) + \underbrace{p[\text{last update}]}_{\substack{\text{momentum}\\ \text{(p is a number less than 1)}}}$$

Dropout (momentum) term:

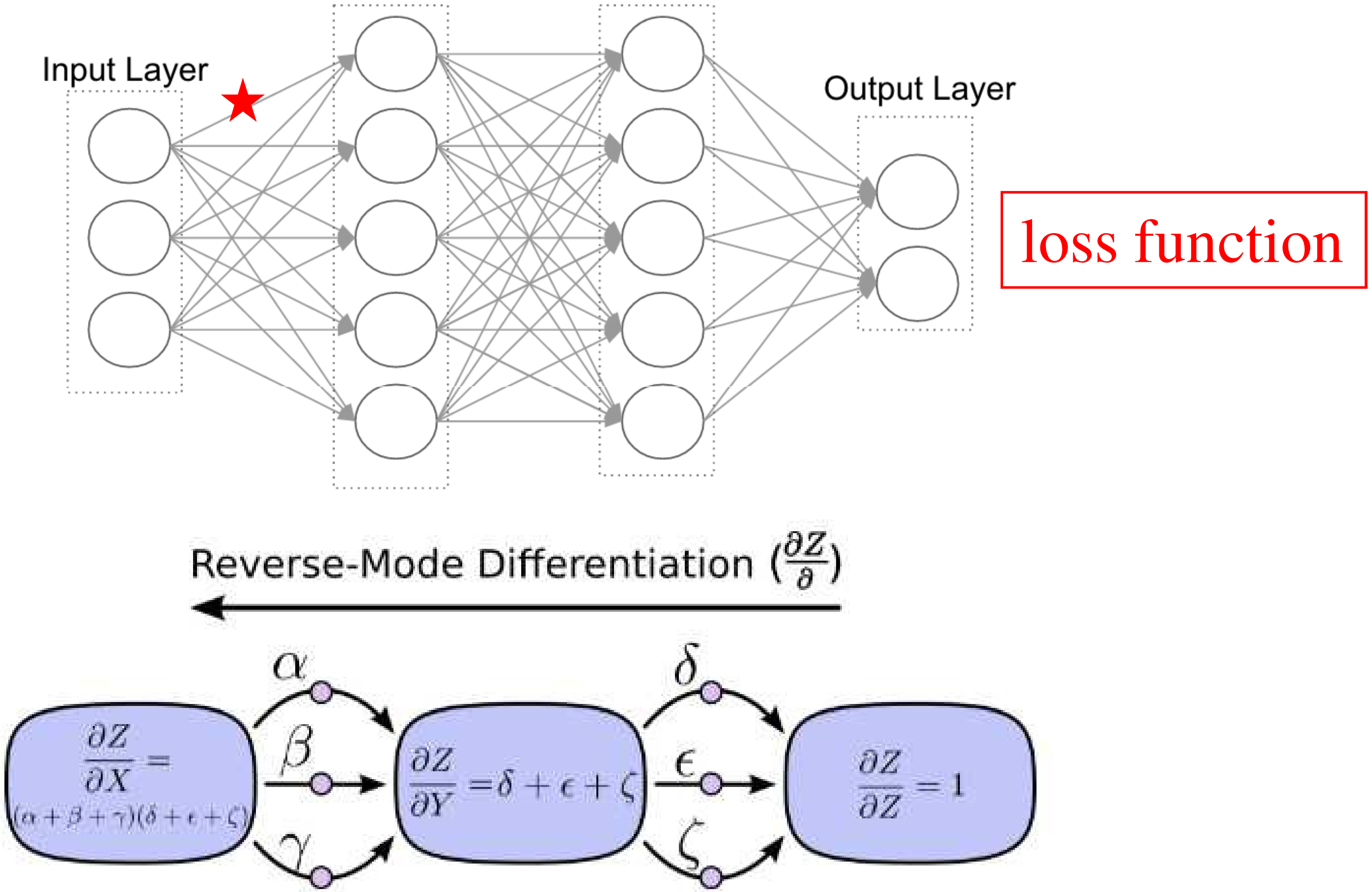
1)Before dropout: output of a neuron = x.

2)With dropout: the expected output = $px+(1-p)0 = px$.

(because output = 0 with probability $1-p$)

3)At test time: adjust $x \rightarrow px$ to keep the same expected output.

Dealing with gradients: Back-propagation



Define the MNIST Solver

solver.prototxt

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

test_iter: 100

-- how many test iterations should occur per test_interval.

test_interval: 500

-- how often the test phase of the network will be executed.

lr_policy: "inv "

base_lr: 0.01

gamma: 0.0001

power: 0.75

--learning rate =

base_lr * (1 + gamma * iter) ^ (- power)

momentum: 0.9

-- how much of the previous weight will be retained in the new calculation.

weight_decay: 0.0005

-- the factor of (regularization) penalization of large weights.

snapshot: 5000

-- how often caffe should output a model and solverstate.

Conclusion

- Optimization methods of deep learning are very tricky.
- MNIST data is our starting point to test new algorithms.