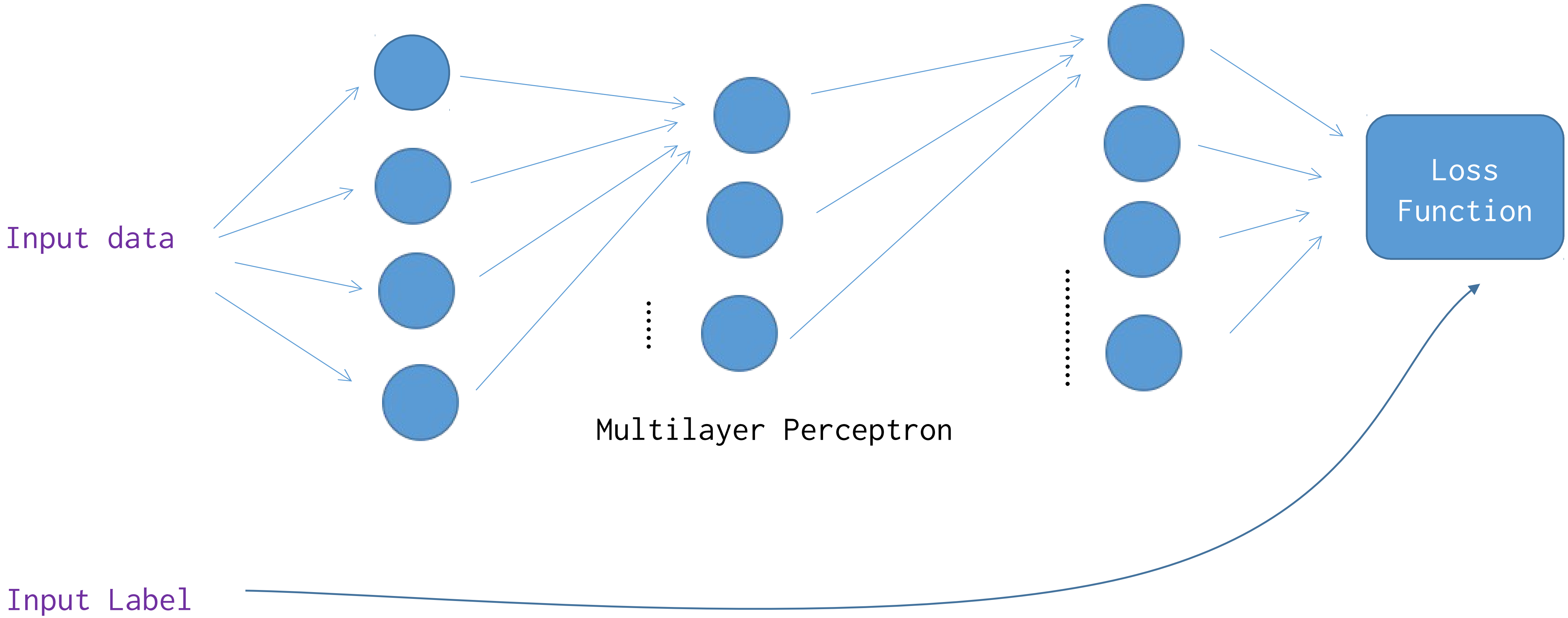
Deep Learning Framework from Scratch

Mo Zhou

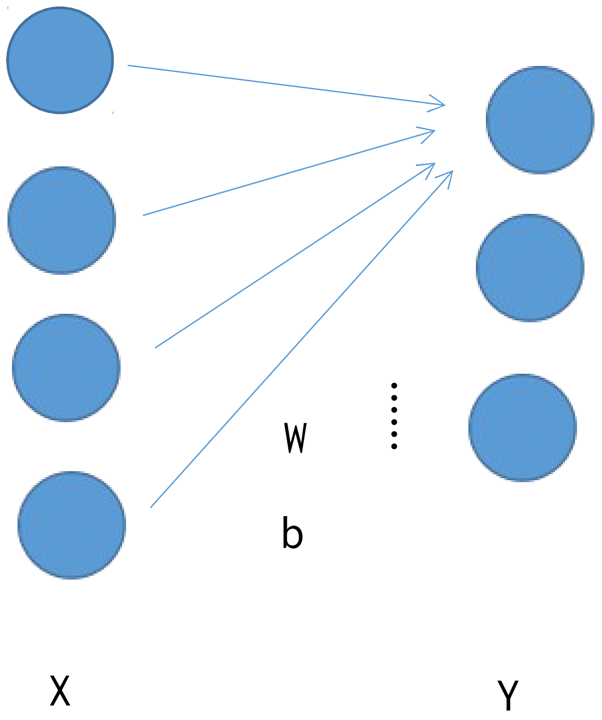
April. 2018

Review the Task



(1) Neural Net as A Mapping; (2) Parameter estimation into Optimization problem;
(3) Non-Convex hence GD. (4) Gradient-based Optimization hence Back-Prop. (diff-able)

Closer View to the Linear Layer

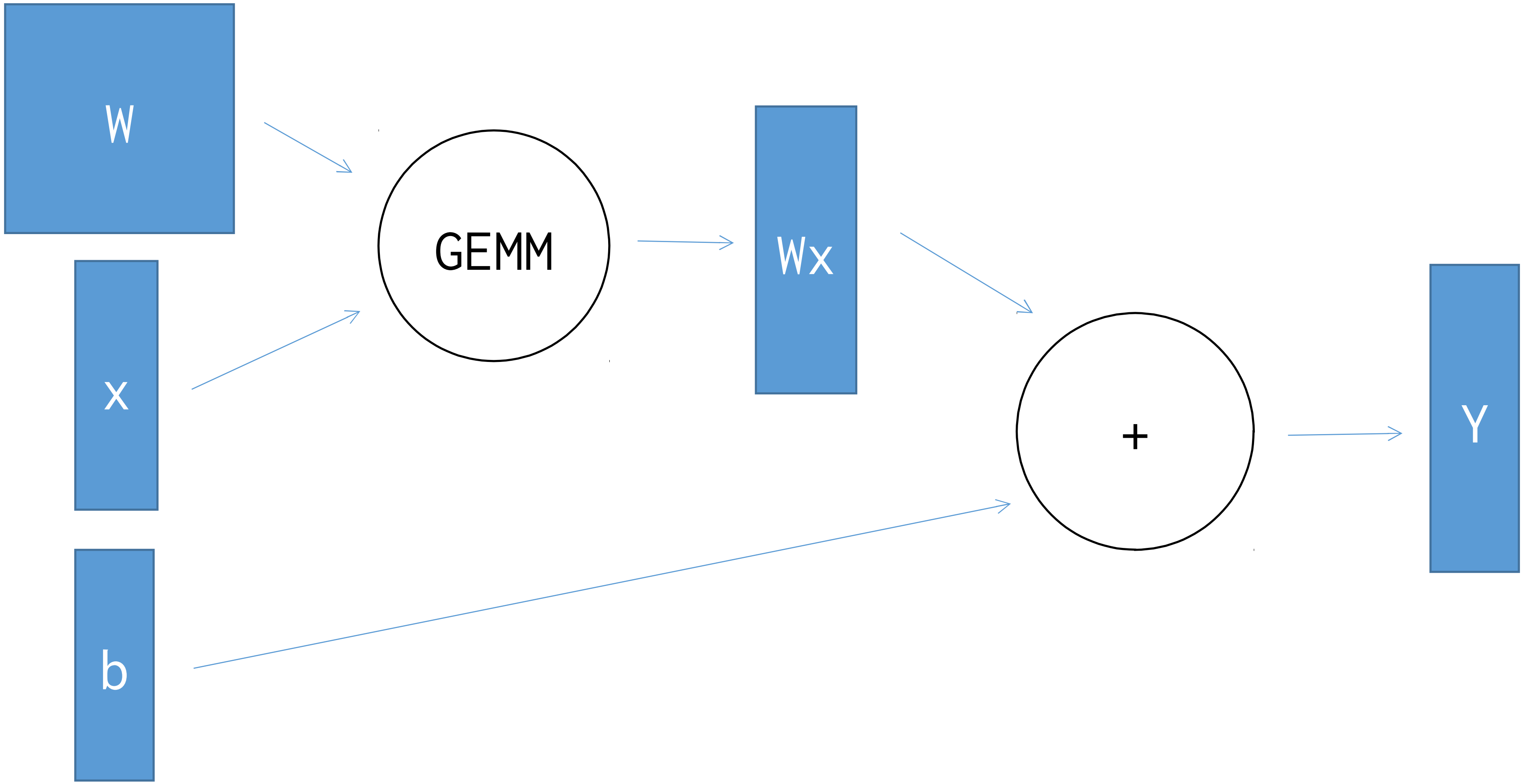


Matrix Equation

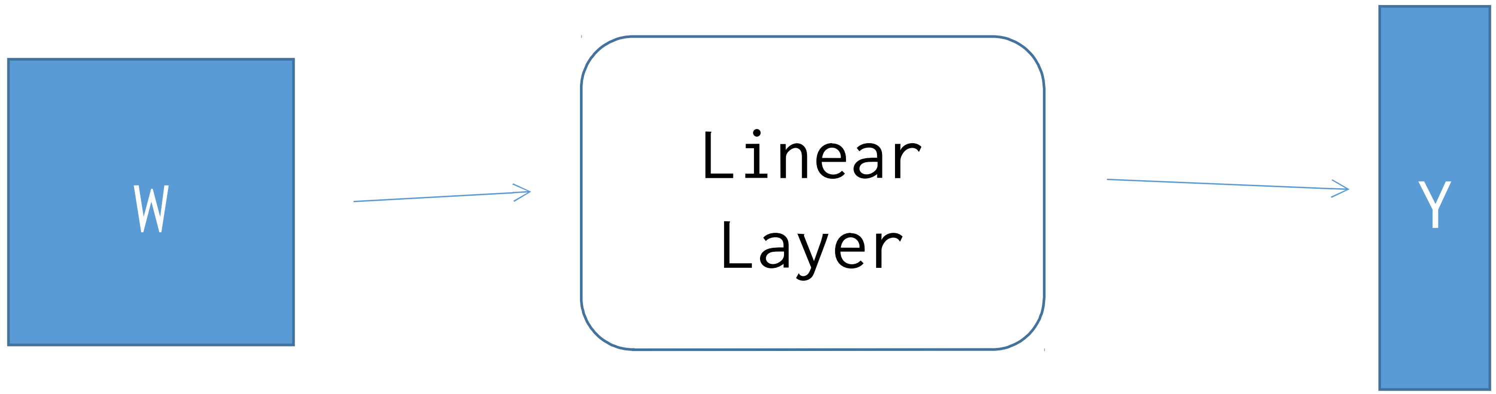
$$Y = WX + b$$

Interpreting the Linear Layer

With (Atomic) Math Operators, into Computation Graph.



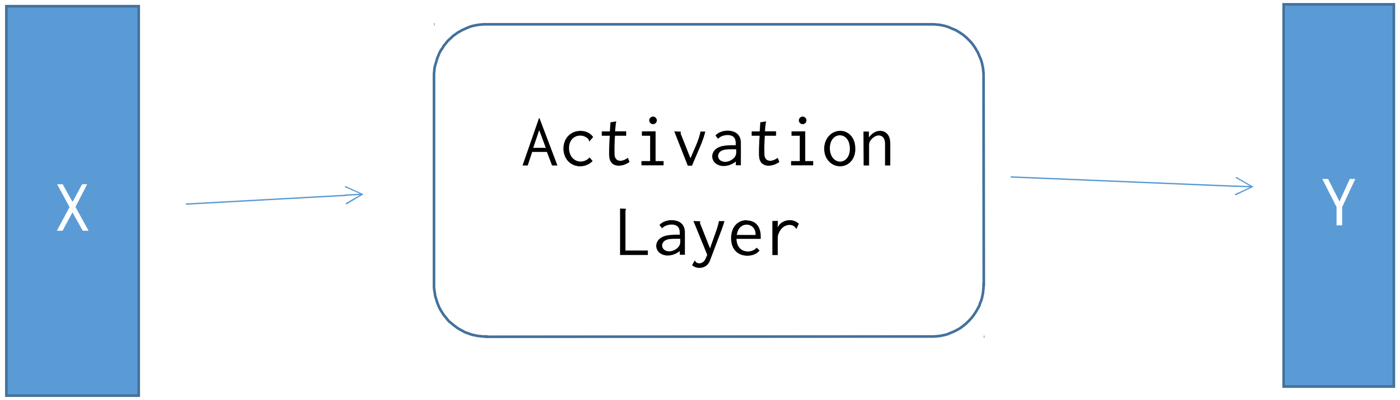
Linear Layer As Building Block



PyTorch:

```
Fc1 = torch.nn.Linear(4096, 512)
```

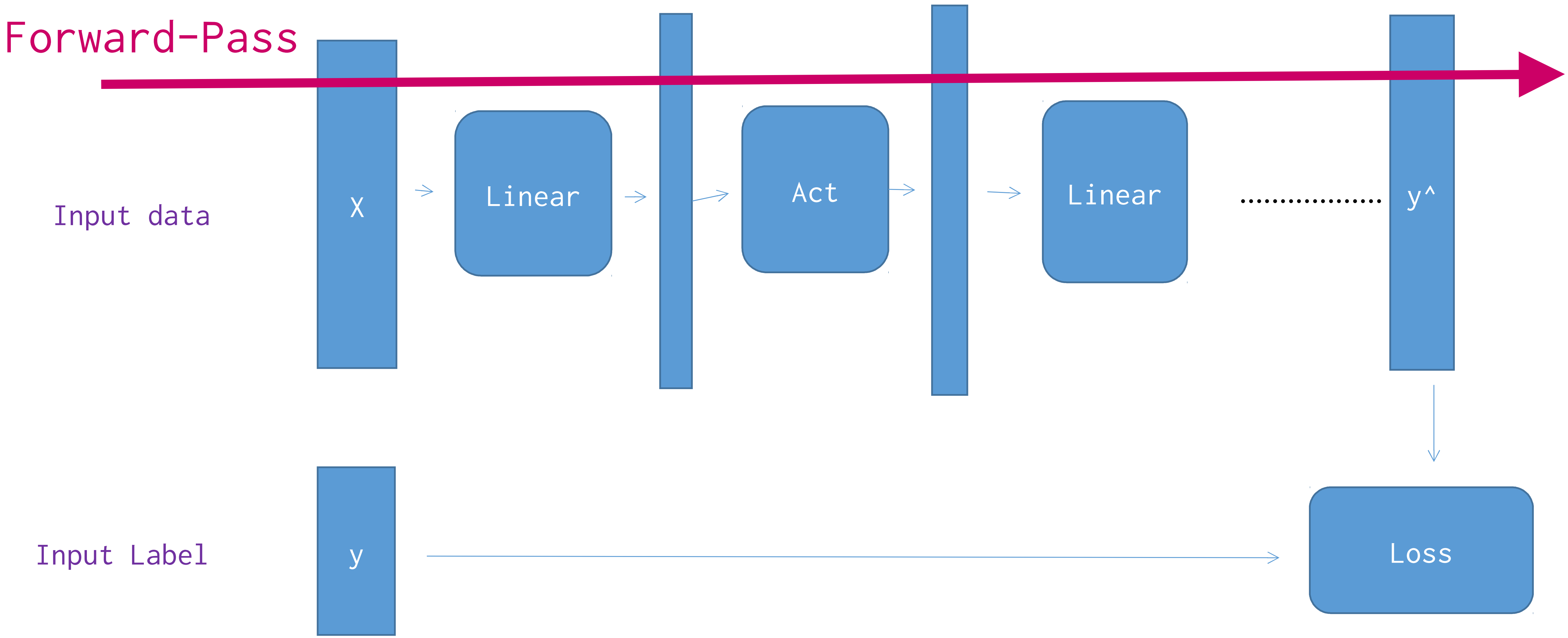
Similar to Activation Function



PyTorch:

```
Relu1 = torch.nn.ReLU()
```

The Original MLP

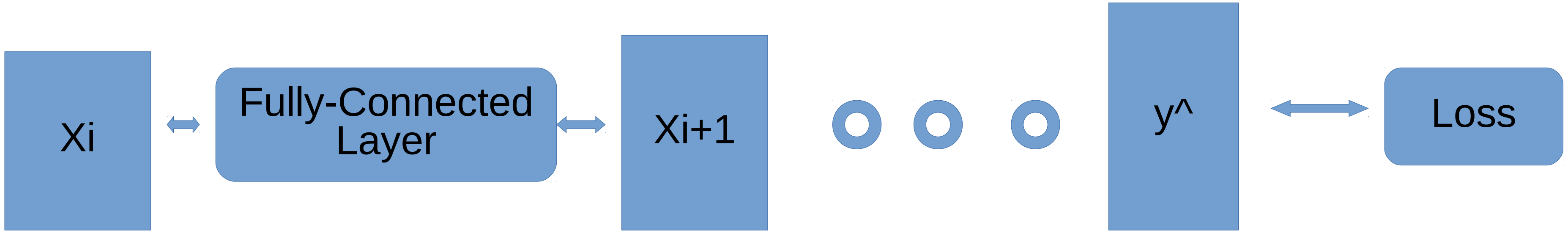


Backward Pass

Automatic Differentiation / Reverse Mode, e.g. Back-propagation Algorithm

$$\frac{\partial L(\hat{y},y)}{\partial x} = \frac{\partial L(\hat{y},y)}{\partial z(x)} \cdot \frac{\partial z(x)}{\partial x}$$

A Neural Network Can Be Implemented in a Modularized Manner.



$$\frac{\partial L(\hat{y},y)}{\partial x_i} = \frac{\partial L(\hat{y},y)}{\partial x_{i+1}} \cdot \frac{\partial x_{i+1}}{\partial x_i}$$

Example of Automatic Differentiation in Reserse Mode

$$y = x_3 e^{x_1 + x_2}$$

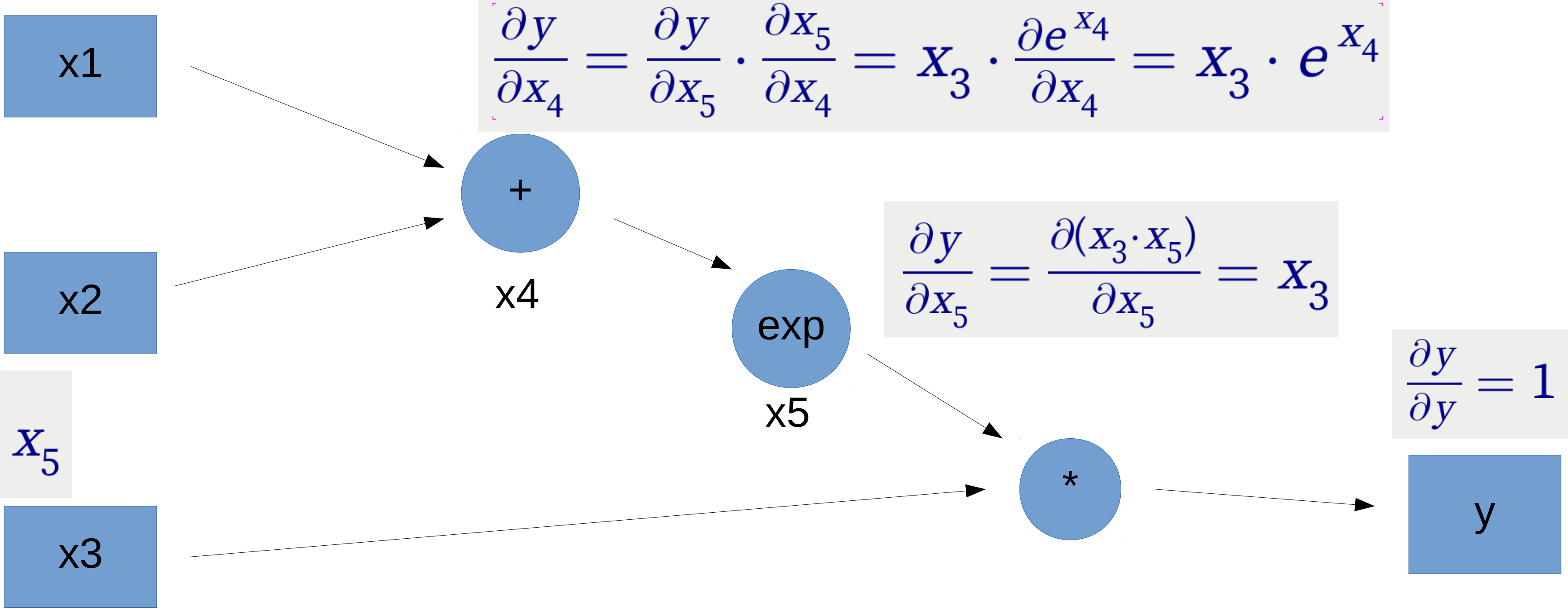
$$\frac{\partial y}{\partial x_1} = \frac{\partial y}{\partial x_4} \cdot \frac{\partial (x_1 + x_2)}{\partial x_1} = \frac{\partial y}{\partial x_4}$$

$$\frac{\partial y}{\partial x_4} = \frac{\partial y}{\partial x_5} \cdot \frac{\partial x_5}{\partial x_4} = x_3 \cdot \frac{\partial e^{x_4}}{\partial x_4} = x_3 \cdot e^{x_4}$$

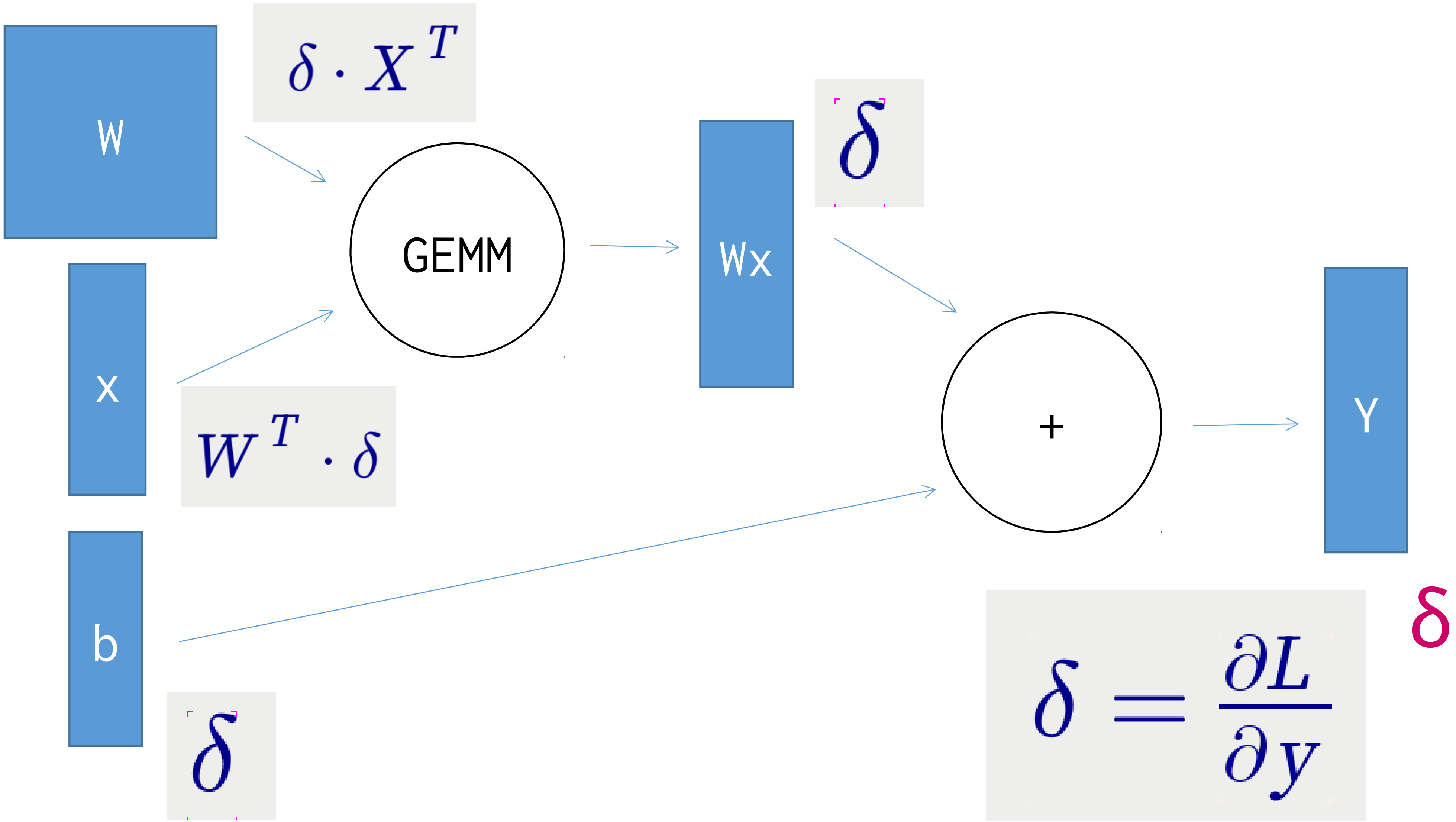
$$\frac{\partial y}{\partial x_5} = \frac{\partial (x_3 \cdot x_5)}{\partial x_5} = x_3$$

$$\frac{\partial y}{\partial y} = 1$$

$$\frac{\partial y}{\partial x_3} = \frac{\partial (x_3 \cdot x_5)}{\partial x_3} = x_5$$

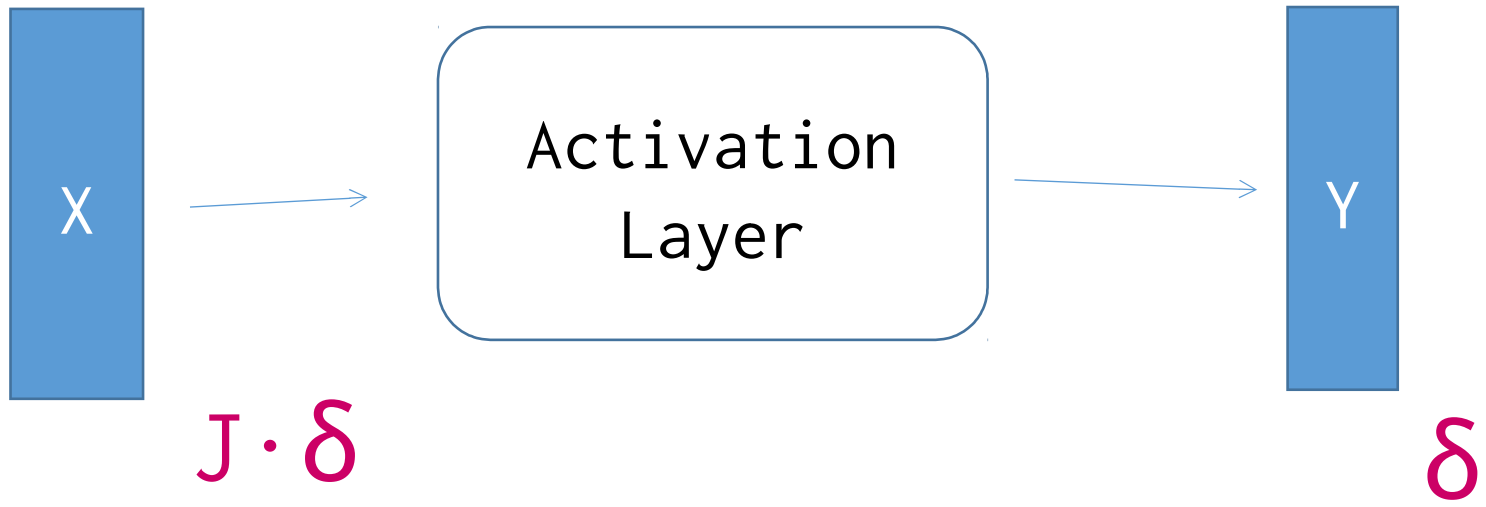


Backward Pass of Linear Layer



Backward Pass of Activation Layer

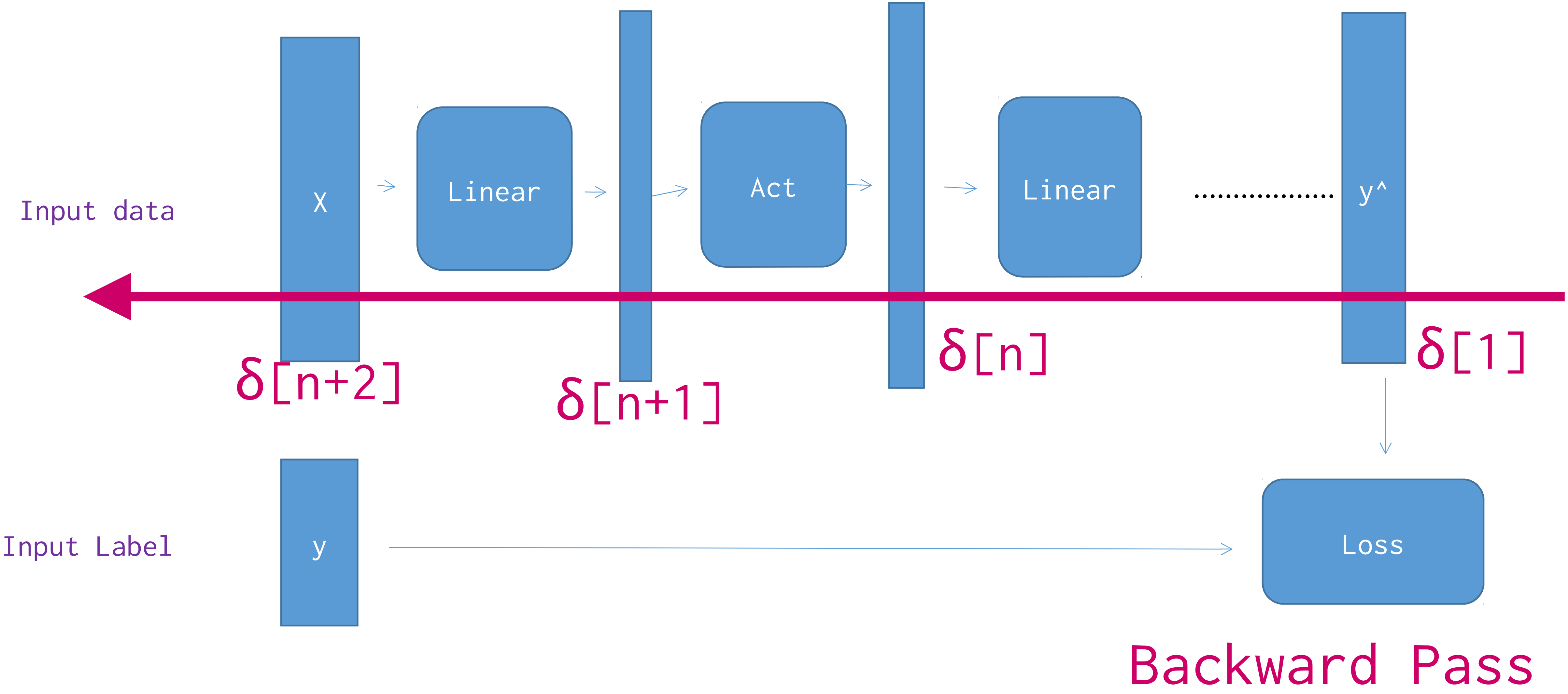
Element-wise Activation Function, e.g. Sigmod, Tanh, ReLU



J is the Jacobian Matrix of y w.r.t x

The Jacobian matrix is a diagonal matrix. That being said, the actual implementation involves no matrix multiplication for sake of memory efficiency.

Back-Pass of the Whole Network

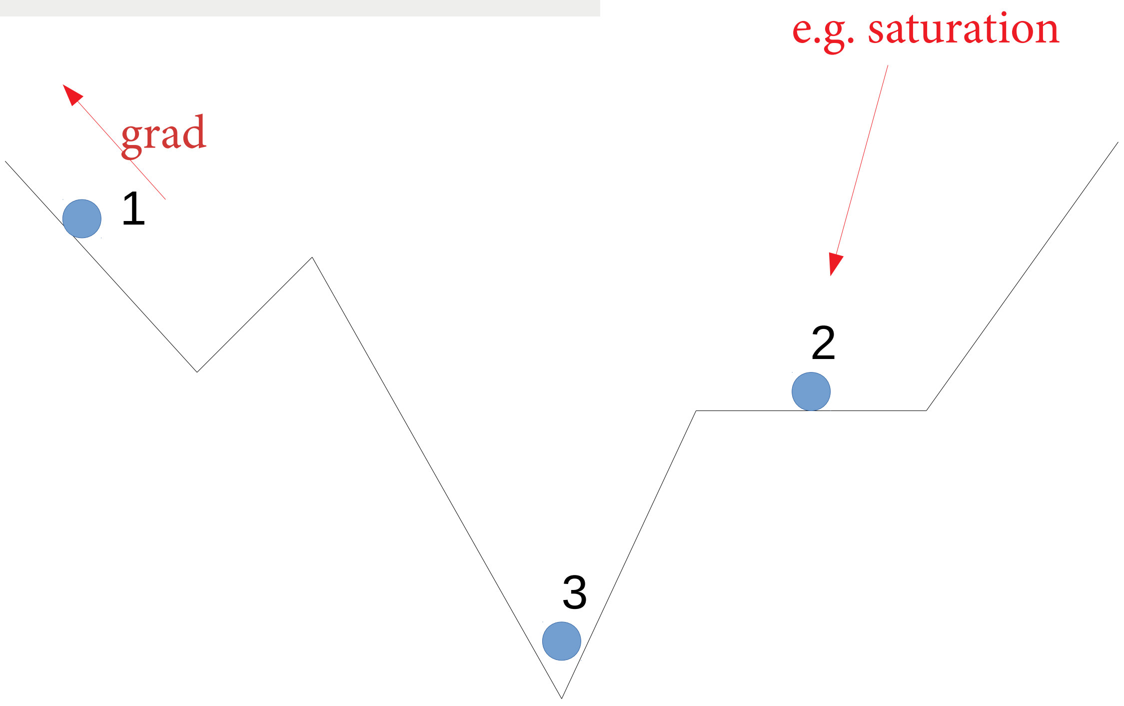


Optimization

First-Order Methods, e.g. SGD, SGDM, Adagrad, RMSProp, Adam.

$$W \leftarrow W - \eta \cdot \Delta W$$

Second-Order Methods requires Hessian Matrix which would cost an insane amount of memory. Algorithms for approximated Hessian matrices are available but currently the first-order methods are most widely used and have demonstrated its effectiveness.



Main Procedure of Neural Network Training

```
for epoch in range(maxepoch):

    for xbatch, ybatch in trainingset:

        yhat = forward(xbatch)
        loss(yhat, ybatch)
        backward()
        update()

    for xbatch, ybatch in valset:

        yhat = forward(xbatch)
        performance(yhat, ybatch)
```

State-of-the-Art Implementations

- Caffe (C++, Fast, Module, Static Graph, not Flexible)
- LuaTorch (C/Lua, Fast, Module, Imperative, Flexible)
- PyTorch (Py/C/C++, Fast, Dynamic Graph, UltraFlexible)
- Theano (Py/?, Static Symbolic Graph, EOL)
- TensorFlow/early (Py/C++, Static Graph)
- Keras (TF Abstraction)
- MXNet, CNTK, Chainer, NEON, …….
- ...

My Naive, but Simple Approach

- Tensor Class,   Abstraction of Vectors, Matrices, ...
- Blob Class,   Combination of 2 Tensors : (Value, Gradient)
- Layer Class,   Forward/Backward with Blobs
- Graph Class,   Put things above into a Computation Graph

Reference: source of Caffe and (Lua)Torch

Overview of My DL Framework

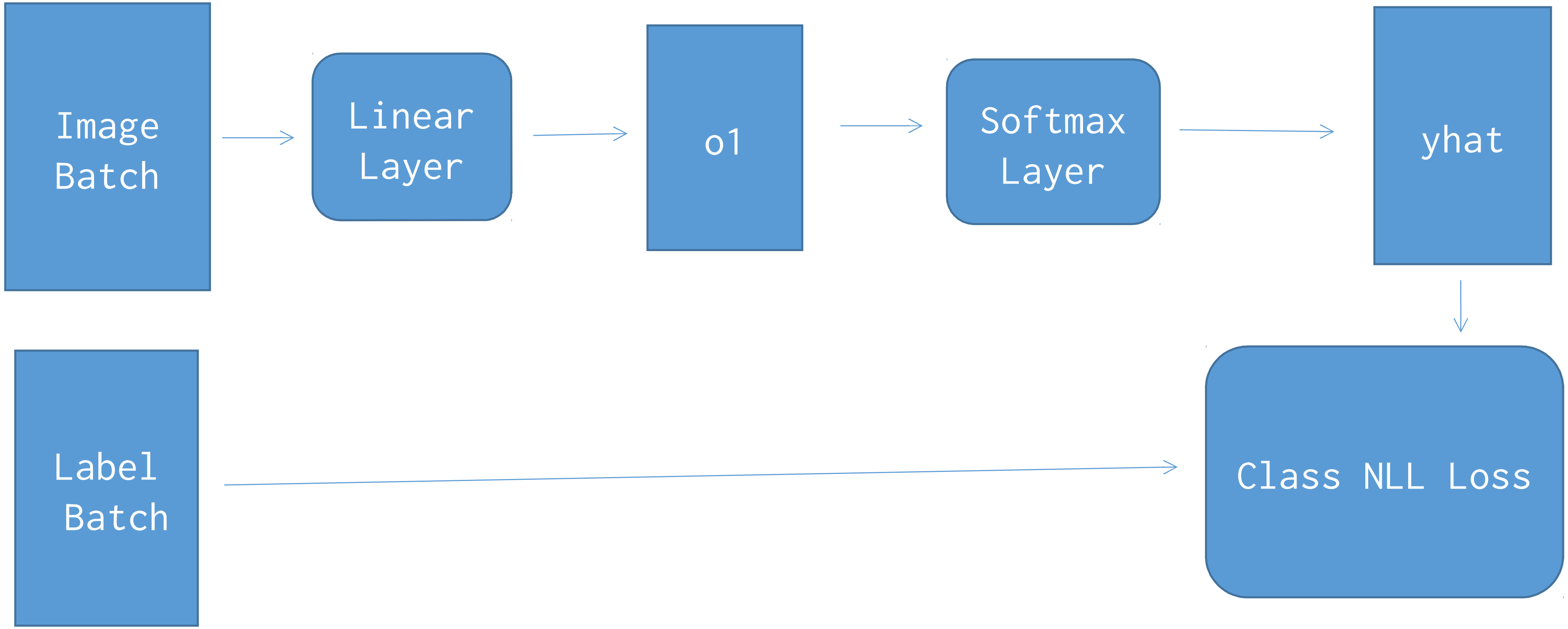
Abstractions

* Tensor (Scalar, Vector, Matrix, n-D Array, ...)
* Atomic Operations (GEMM, +, -, *, /, exp, ...)
* Layers (Linear, ReLU, Conv2d, SoftMax, ...)
* Loss Functions (NLLLoss, MSELoss, Triplet, ...)

Steps of Network Training

* Forward Pass
* Backward Pass
* Parameter Update

Example: Classification on MNIST



```
cout << ">> Initialize Network" << endl;

Graph<double> net (784, 1, 100);
net.name = "test net";
net.addLayer("fc1", "Linear", "entryDataBlob", "fc1", 10);
net.addLayer("sm1", "Softmax", "fc1", "sm1");
net.addLayer("cls1", "ClassNLLLoss", "sm1", "cls1", "entryLabelBlob");
net.addLayer("acc1", "ClassAccuracy", "sm1", "acc1", "entryLabelBlob");
net.dump();
```

```
Name: fc1
Type: Linear
input: entryDataBlob
output: fc1
output_dim: 10
```

```
Name: sm1
Type: Softmax
input: fc1
output: sm1
```

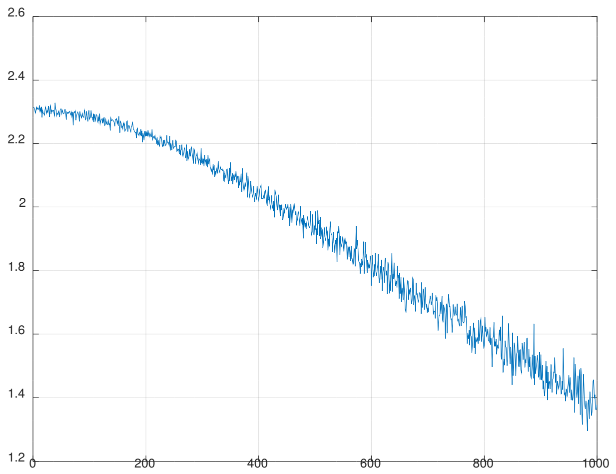
```
Name: cls1
Type: ClassNLLLoss
input: sm1
output: cls1
input_label: entryLabelBlob
```

```
Name: acc1
Type: ClassAccuracy
input: sm1
output: acc1
input_label: entryLabelBlob
```

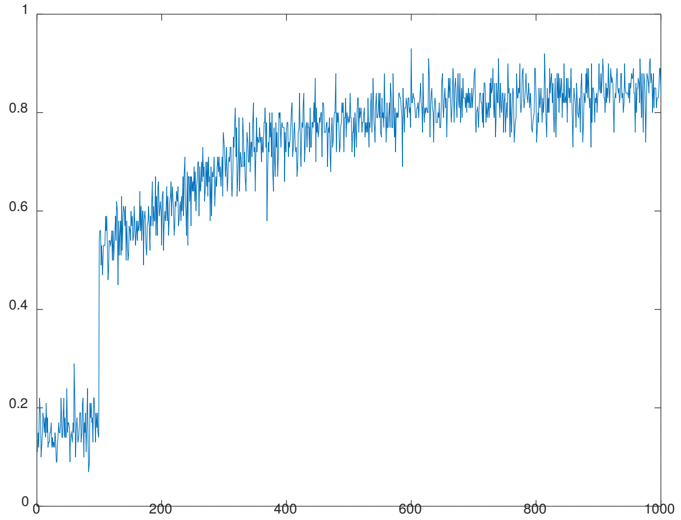
```
cout << ">> Start training" << endl;
for (int iteration = 0; iteration < maxiter; iteration++) {
    leicht_bar_train(iteration);
    // -- get batch
    Tensor<double>* batchIm = new Tensor<double> (100, 784);
    batchIm->copy(trainImages.data + (iteration%iepoch)*batchsize*784, batchsize*784);
    batchIm->transpose_();
    batchIm->scal_(1./255.);
    net.getBlob("entryDataBlob", true)->value.copy(batchIm->data, 784*batchsize);
    net.getBlob("entryLabelBlob", true)->value.copy(
                trainLabels.data + (iteration%iepoch)*batchsize*1, batchsize*1);
    delete batchIm;

    // -- forward
    net.forward();
    // -- zerograd
    net.zeroGrad();
    // -- backward
    net.backward();
    // -- report
    net.report();
    // -- update
    net.update(1e-3, "Adam");
```

```
// -- test every
if ((iteration+1)%testevery==0) {
    leicht_bar_val(iteration);
    vector<double> accuracy;
    vector<double> l;
    for (int t = 0; t < 42; t++) {
        // -- get batch
        Tensor<double>* tbatchIm = new Tensor<double> (100, 784);
        tbatchIm->copy(valImages.data + t*batchsize*784, batchsize*784);
        tbatchIm->transpose_();
        tbatchIm->scal_(1./255.);
        net.getBlob("entryDataBlob", true)->value.copy(tbatchIm->data, 784*batchsize);
        net.getBlob("entryLabelBlob", true)→value.copy(
                    valLabels.data + t*batchsize*1, batchsize*1);
        delete tbatchIm;

        net.forward(); net.report();
    }
}
```

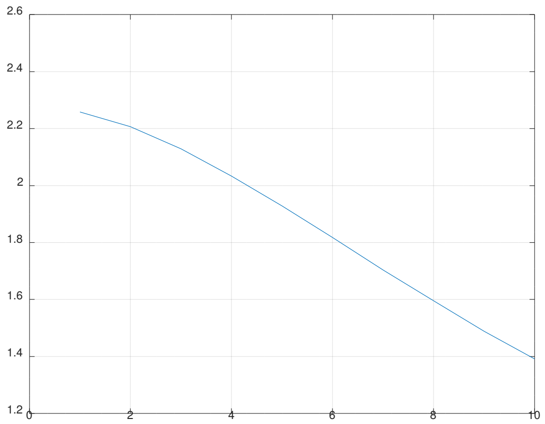
Training/Validation Curves



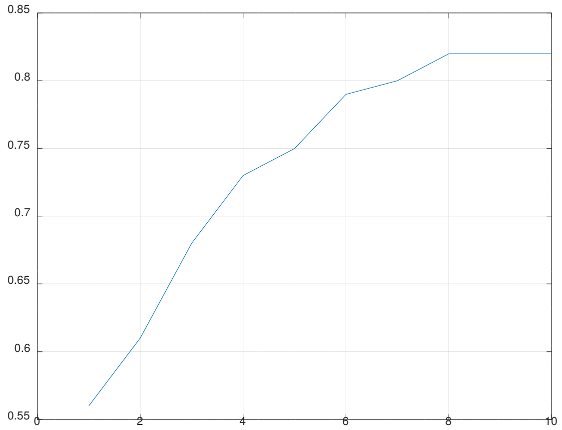
Training Loss



Training Accuracy



Validation Loss



Validation Accuracy

Time Cost & Compiler Black Magic

- I5-2520M: 2.1 Sec. (OpenMP, -O2, -march=native)
- I7-6900K: 1.0 Sec. (OpenMP, -O2, -march=native)

500 Iterations, Clang slightly faster than GCC

- I5-2520M: 32.3 Sec. (no OpenMP, -O0)
- I5-2520M: 17.4 Sec. (OpenMP, -O0)
- I5-2520M: 2.4 Sec. (no OpenMP, -O2)

- cuDNN: faster ...

Another Example: LeNet on MNIST for Classification

```
int
main(void)
{
    leicht_threads(2);
    cout << ">> Reading MNIST training dataset" << endl;

    Tensor<double> trainImages(37800, 784); trainImages.setName("trainImages");
    leicht_hdf5_read("mnist.th.h5", "/train/images", 0, 0, 37800, 784, trainImages.data);
    Tensor<double> trainLabels(37800, 1);   trainLabels.setName("trainLabels");
    leicht_hdf5_read("mnist.th.h5", "/train/labels", 0, 0, 37800, 1, trainLabels.data);

    cout << ">> Reading MNIST validation dataset" << endl;

    Tensor<double> valImages(4200, 784); valImages.setName("valImages");
    leicht_hdf5_read("mnist.th.h5", "/val/images", 0, 0, 4200, 784, valImages.data);
    Tensor<double> valLabels(4200, 1);   valLabels.setName("valLabels");
    leicht_hdf5_read("mnist.th.h5", "/val/labels", 0, 0, 4200, 1, valLabels.data);
```

```
cout << ">> Initialize Network" << endl;

// reference: caffe/examples/mnist/lenet
Blob<double> label   (1, batchsize, "label", false);
Blob<double> X       (batchsize, 784, "X", false);
Blob<double> image   (batchsize, 1, 28, 28, "image", false);
Blob<double> conv1   (batchsize, 20, 24, 24);          conv1.setName("conv1");
Blob<double> pool1   (batchsize, 20, 12, 12);          pool1.setName("pool1");
Blob<double> conv2   (batchsize, 50, 8, 8);            conv2.setName("conv2");
Blob<double> pool2   (batchsize, 50, 4, 4);            pool2.setName("pool2");
Blob<double> pool2f  (batchsize, 800);                 pool2f.setName("pool2f");
Blob<double> pool2fT (800, batchsize);                 pool2fT.setName("pool2fT");
Blob<double> ip1     (500, batchsize);                 ip1.setName("ip1");
Blob<double> ip2     (10, batchsize);                  ip2.setName("ip2");
Blob<double> sm1     (10, batchsize);                  sm1.setName("sm1");
Blob<double> loss    (1);                              loss.setName("loss");
Blob<double> acc     (1);                              acc.setName("acc");
```

```
Layer<double>         lid1;  // X->image bs,784->bs,1,28,28
Conv2dLayer<double>   lconv1 (batchsize, 1, 28, 28, 20, 5); // image->conv1 bs,1,28,28->bs,20,24,24
MaxpoolLayer<double>  lpool1 (batchsize, 20, 24, 24, 2, 2); // conv1->pool1 bs,20,24,24->bs,20,12,12
Conv2dLayer<double>   lconv2 (batchsize, 20, 12, 12, 50, 5); // pool1->conv2 bs,20,12,12->bs,50,8,8
MaxpoolLayer<double>  lpool2 (batchsize, 50, 8, 8, 2, 2); // conv2->pool2 bs,50,8,8->bs,50,4,4
Layer<double>         lid2;  // pool2->pool2f(lattened) bs,50,4,4->bs,800
    TransposeLayer<double>lt1;   // pool2f->pool2fT bs,800->800,bs
LinearLayer<double>   lfc1   (500, 800); // pool2fT->ip1 800,bs->500,bs
ReluLayer<double>     lrelu1; // ip1->ip1
LinearLayer<double>   lfc2   (10, 500); // ip1->ip2 500,bs->10,bs
SoftmaxLayer<double>  lsm1;  // ip2->sm1
ClassNLLLoss<double>  lloss; // sm1->loss
ClassAccuracy<double> lacc;  // sm1->acc
```

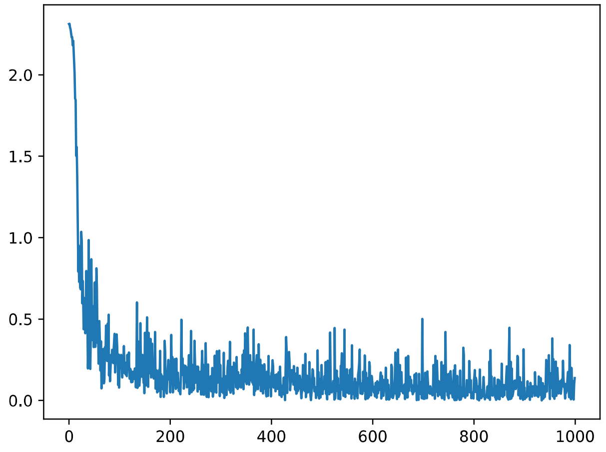
```
    cout << ">> Start training" << endl;
    for (int iteration = 0; iteration < maxiter; iteration++) {
        tic();
        leicht_bar_train(iteration);

        // -- get batch
        X.value.copy(
//trainImages.data + (iteration%overfit)*batchsize*784, batchsize*784);
trainImages.data + (iteration%iepoch)*batchsize*784, batchsize*784);
        label.value.copy(
//trainLabels.data + (iteration%overfit)*batchsize*1, batchsize*1);
trainLabels.data + (iteration%iepoch)*batchsize*1, batchsize*1);
        X.value.scal_(1./255.);
```

```
// -- forward : unfold with vim: BEIGN,ENDs/; /;\r/g
lid1.forward(X, image);             //X.dump(true, false); image.dump(true, false);
lconv1.forward(image, conv1);       //conv1.dump(true, false);
lpool1.forward(conv1, pool1);       //pool1.dump(true, false);
lconv2.forward(pool1, conv2);       //conv2.dump(true, false);
lpool2.forward(conv2, pool2);       //pool2.dump(true, false);
lid2.forward(pool2, pool2f);        //pool2f.dump(true, false);
lt1.forward(pool2f, pool2fT);       //pool2fT.dump(true, false);
//auto p2T = pool2f.value.transpose();
//pool2fT.value.copy(p2T->data, p2T->getSize());
//delete p2T;
lfc1.forward(pool2fT, ip1);         //ip1.dump(true, false);
lrelu1.forward(ip1, ip1);           //ip1.dump(true, false);
lfc2.forward(ip1, ip2);             //ip2.dump(true, false);
lsm1.forward(ip2, sm1);             //sm1.dump(true, false);
lloss.forward(sm1, loss, label);    //loss.dump(true, false);
lacc.forward(sm1, loss, label);     //acc.dump(true, false);
```

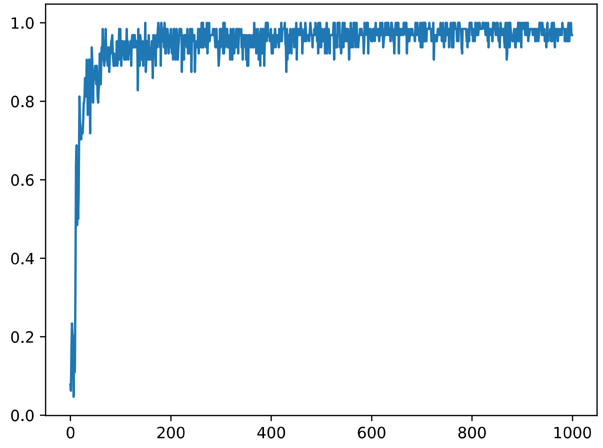
```
// -- zerograd
label.zeroGrad(); X.zeroGrad(); image.zeroGrad();
conv1.zeroGrad(); pool1.zeroGrad(); conv2.zeroGrad();
pool2.zeroGrad(); pool2f.zeroGrad(); pool2fT.zeroGrad();
ip1.zeroGrad(); ip2.zeroGrad(); sm1.zeroGrad();
loss.zeroGrad(); acc.zeroGrad();

lid1.zeroGrad(); lconv1.zeroGrad(); lpool1.zeroGrad();
lconv2.zeroGrad(); lpool2.zeroGrad(); lid2.zeroGrad();
lfc1.zeroGrad(); lrelu1.zeroGrad(); lfc2.zeroGrad();
lsm1.zeroGrad(); lloss.zeroGrad(); lacc.zeroGrad();
```

```
// -- backward : unfold with vim: BEIGN,ENDs/; /;\r/g
lloss.backward(sm1, loss, label);   //sm1.dump();
lsm1.backward(ip2, sm1);            //ip2.dump();
lfc2.backward(ip1, ip2);            //ip1.dump();
lrelu1.backward(ip1, ip1);          //ip1.dump();
lfc1.backward(pool2fT, ip1);        //pool2fT.dump();
        lt1.backward(pool2f, pool2fT);      //pool2f.dump();
//auto p2fT = pool2fT.gradient.transpose();
//pool2f.gradient.copy(p2fT->data, p2fT->getSize());
// delete p2fT;
lid2.backward(pool2, pool2f);       //pool2.dump();
lpool2.backward(conv2, pool2);      //conv2.dump();
lconv2.backward(pool1, conv2);      //pool1.dump();
lpool1.backward(conv1, pool1);      //conv1.dump();
lconv1.backward(image, conv1);      //image.dump();

// regularize
lconv1.regularization(); lconv2.regularization();
lfc1.regularization();   lfc2.regularization();

// -- report
lloss.report(); lacc.report(true);
label.dump(true, false);
lconv1.dumpstat(); lconv2.dumpstat();
lfc1.dumpstat();   lfc2.dumpstat();
//pool1.dump(true, false);

cv_train_loss.append(iteration, lloss.lossval);
cv_train_acc.append(iteration, lacc.accuracy);
```

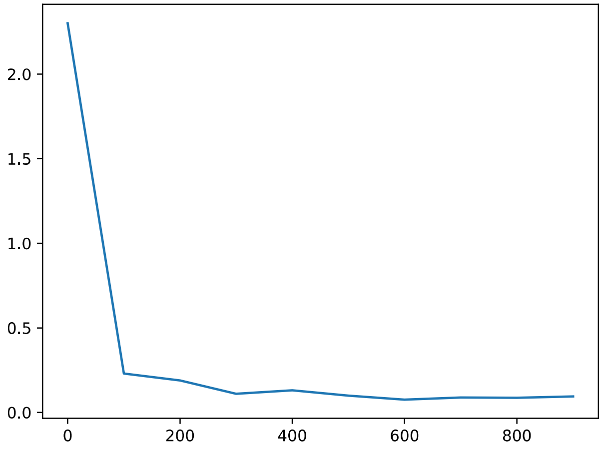
Training/Validation Curves



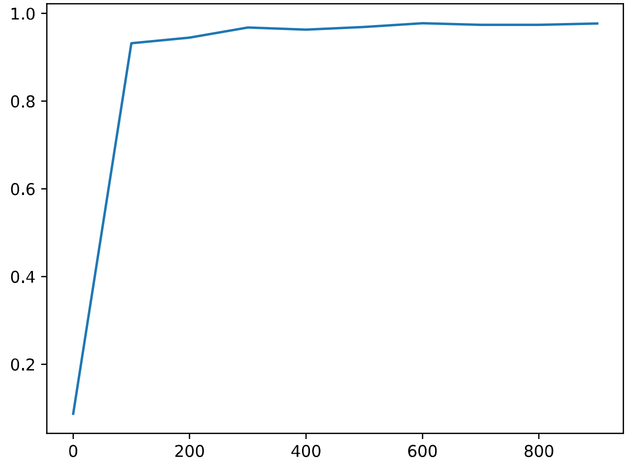
Training Loss



Training Accuracy



Validation Loss



Validation Accuracy

Final Accuracy
0.96

Implementation Detail

Five Core Modules:

1. BLAS and some extra subroutines

2. Tensor Class

3. Blob Class

4. Layer Class

5. (Static) Graph Class

Aux1. Data loading routines
Aux2. Plotting helper
Aux3. Unit test helper
Aux4. Benchmarker

Language: C++, Python

Styling: mixture of (Lua)Torch and Caffe

Lines of Code: 4106

Implementation Detail: BLAS

3 Levels

Level 1 BLAS:
asum, axpy, copy, dot, nrm2, scal

Level 2 BLAS:
gemv (skipped)

Level 3 BLAS:
gemm

Extra:
Conv2d

```
// BLAS-1 function: ASUM
template <typename Dtype> Dtype
asum(size_t n, Dtype* x, int incx)
{
        Dtype ret = 0.;
#if defined(USE_OPENMP)
#pragma omp parallel for reduction (+:ret)
#endif
        for (size_t i = 0; i < n; i++) {
            ret += x[i*incx] > (Dtype)0. ? x[i*incx] : -x[i*incx];
        }
        return ret;
}
```

Implementation Detail: BLAS

3 Levels

Level 1 BLAS:
asum, axpy, copy, dot, nrm2, scal

Level 2 BLAS:
gemv (skipped)

Level 3 BLAS:
gemm

Extra:
Conv2d

```
    if (!transA && !transB) { // A * B
        //#pragma omp parallel for collapse(2)
        //for (size_t i = 0; i < M; i++) {
        //  for (size_t j = 0; j < N; j++) {
        //      Dtype vdot = beta * C[i*ldc+j];
        //      for (size_t k = 0; k < K; k++) {
        //          vdot += alpha * A[i*lda+k] * B[k*ldb+j];
        //      }
        //      C[i*ldc+j] = vdot;
        //  }
#if defined(USE_OPENMP)
#pragma omp parallel for
#endif // USE_OPENMP
        for (size_t i = 0; i < M; i++) {
            if (beta != 1.) for (size_t j = 0; j < N; j++)
                C[i*ldc+j] *= beta;
            for (size_t k = 0; k < K; k++) {
                Dtype temp = alpha * A[i*lda+k];
#if defined(USE_OPENMP)
#pragma omp simd
#endif // USE_OPENMP
                for (size_t j = 0; j < N; j++)
                    C[i*ldc+j] += temp * B[k*ldb+j];
            }
        } // I5-2520M, OMPthread=2, 512x512 double gemm, 10 run, 553 ms.
```

Implementation Detail: Tensor Class

Core Data:

```
std::vector<size_t> shape;
Dtype *data = nullptr;
```

Core Methods:

```
Constructor
Locator,
Copier,
dump,
resize,
expand,
unexpand,
fill_,
+ - * /,
rand,
rot180,
clone,
sign,
transpose_,
save,
load,
overloaded
operators,
BLAS wrapper
```

Implementation Detail: Blob Class

Core Data:

```
Tensor<Dtype> value;
Tensor<Dtype> gradient;
bool requires_grad = true;
```

Core Methods:

```
Constructor
resizer,
transpose,
clone,
zerograd,
dump
```

Implementation Detail: Layer Class

Core Data:

```
std::vector<Blob<Dtype>*> parameters;
```

Core Methods:

```
zeroGrad,
forward(Blob<Dtype>& input, Blob<Dtype>& output)
backward(Blob<Dtype>& input, Blob<Dtype>& output)
update() → SGD(), SGDM(), Adam()
```

```
    // Linear, !row_major ? Wx + b -> y : xW + b -> y
    // XXX: support >2D tensor as long as the row_major attrib is correct.
    void forward(Blob<Dtype>& input, Blob<Dtype>& output) {
        // output += GEMM(W, X)
        Tensor<Dtype>::gemm(false, false, 1., &W.value, &input.value, 0., &output.value);
        // output += expand(b)
        if (use_bias) {
            size_t batchsize = input.value.getSize(1);
            auto bb = b.value.expand(batchsize);
            output.value += *bb;
            delete bb;
        }
    }
```

Implementation Detail: Layer Class

```
    void backward(Blob<Dtype>& input, Blob<Dtype>& output) {
        if (!output.requires_grad) return;
        // grad of W: g x x^T
        Tensor<Dtype>::gemm(false, true, 1., &output.gradient, &input.value, 0., &W.gradient);
        // grad of X: W^T x g
        if (input.requires_grad) {
            Tensor<Dtype>::gemm(true, false, 1., &W.value, &output.gradient, 0., &input.gradient);
        }
        // grad of b: unexpand(g)
        if (use_bias) {
            auto gb = output.gradient.unexpand(1);
            b.gradient += *gb;
            delete gb;
        }
    }
```

Implementation Detail: Graph Class

Core Data:

```
std::vector<Blob<Dtype>*> nodes;
std::vector<Layer<Dtype>*> edges;
```

Core Methods:

```
zerograd,
addLayer,
forward,
backward,
update
```

Trouble You May Encounter

- Reliability (API change, Unit Tests)
- Gradient Error (Check derivative & implementation)
- Memory Leak (C/C++: GDB, Valgrind)
- Training Error (e.g. Numerical Precision Problems)
- Performance (BLAS/GEMM: SIMD, OpenMP, CUDA)
- Compilation (Compiler Flags)
- Don't write your own BLAS unless what you are doing

Thanks

Neural Network:

1. Pattern Recognition and Machine Learning (PRML)
2. Deep Learning (Ian Goodfellow)
3. Stanford CS231n (Feifei Li)
4. Efficient Backprop (Yann LeCun)

Automatic Differentiation:

1. Wikipedia
2. CS231n

Implementation Reference:

1. (Lua)Torch
2. Caffe
3. PyTorch

C++/Coding Reference:

1. C++ Primer Plus
2. OpenMP Guide

BLAS Reference:

1. Netlib BLAS/LAPACK