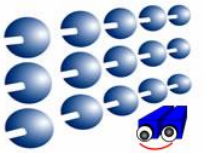


Lessons learned from developing the Gödel Deep Learning library

Robert Varga

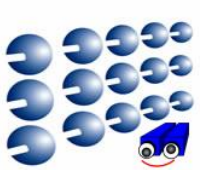
Research Center for Image Processing and Pattern Recognition

Technical University of Cluj-Napoca

2017 IEEE International Conference on Intelligent Computer Communication and Processing

September 7-9, Cluj-Napoca, Romania

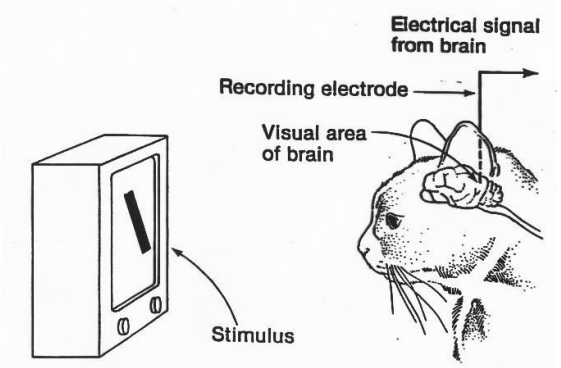
Contents



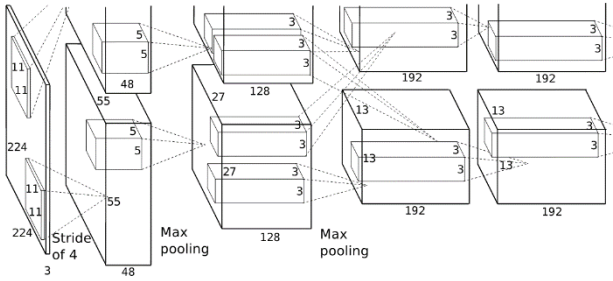
- Introduction
- Other libraries
- Overview
- Case study:
  - First network – regression
  - Second network – classification
  - Third network – deep classification
  - Fourth network – convolutional classification
- Implementation details
- Conclusions

Introduction - history

- Early years
  - Response to visual stimuli in cat cortex [Hubel and Wiesel 1950]
  - Perceptron learning [Rosenblatt 1957]



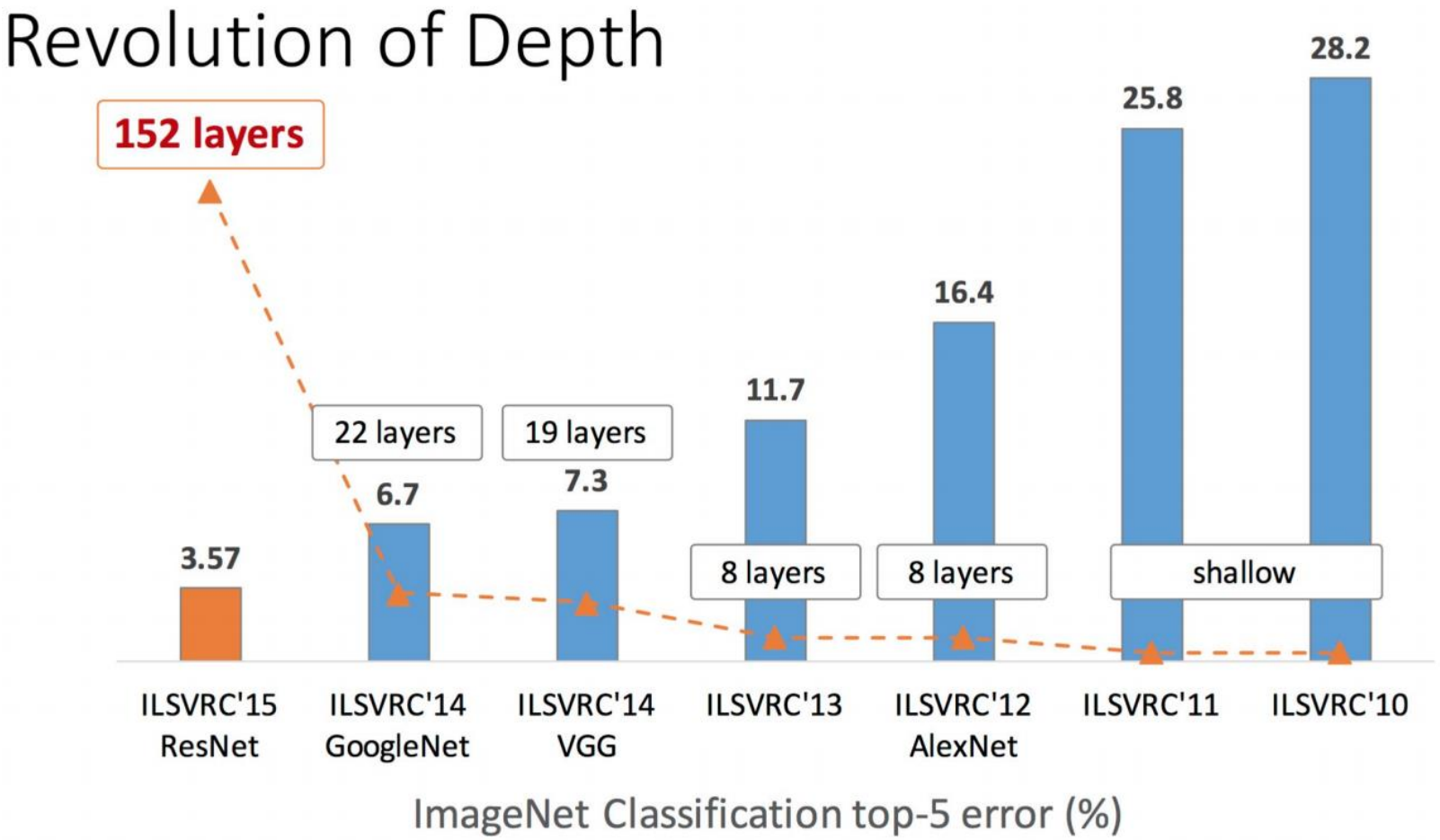
- Decline
  - Limited representation capability [Minsky and Papert 1969]
  - Requires a lot of training data and computational resources
  - No method to train multilayer network
- Backpropagation [Werbos 1974]
- Reemergence
  - Zip code recognition [LeCun 1989]
  - ImageNet winner [Krizhevsky and Hinton 2012]



Introduction - present

- ILSVRC - ImageNet Large Scale Visual Recognition Challenge
- Present
  - LeNet [LeCun 1998]
  - AlexNet [Krizhevsky and Hinton 2012] - ILSVRC winner
  - ZFNet [Zeiler and Fergus 2013] - ILSVRC winner
  - VGG [Simonyan and Zisserman 2014] - runner up
  - GoogLeNet [Google 2014] - ILSVRC winner
  - ResNet [Microsoft 2015] - ILSVRC winner
  - Faster RCNN [Girshick 2015]
  - YOLO9000 [Redmon 2016]
- Massive amounts of training data available
- Computation can be carried out on powerful GPUs

Introduction - trend



From Kris Kitani - 16-385 Computer Vision, CNNs and Deep Learning

Other libraries

- Caffe (Berkeley)
  - Forward/backward API for layers
  - Models available
- TensorFlow (Google)
  - Uses Tensors as underlying data-type
  - Implements computational graphs
- Torch (Facebook)
- Theano, Keras, Darknet
- Most libraries are developed for and maintained under linux

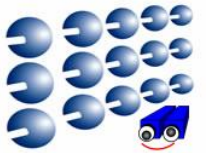
Overview

- What is Gödel Deep Learning?
  - Library for Deep Learning
  - Requires OpenCV for image manipulation
  - Developed in C++
  - Developed for Windows
  - Permits high level definition, training and prediction of feed-forward neural networks
  - Uses multiple CPU cores OpenMP
  - Uses optimized implementations from OpenCV

Overview

- Stats
  - https://gitlab.com/mr.varga.robert/Godel_DL.git
  - Started 19. june 2017
  - 13 commits
  - About 2k lines of code
  - Tested on custom datasets and MNIST
- Goal
  - Provide high level API to easily define network architecture
  - Enable research and development of new types of layers

Overview - components

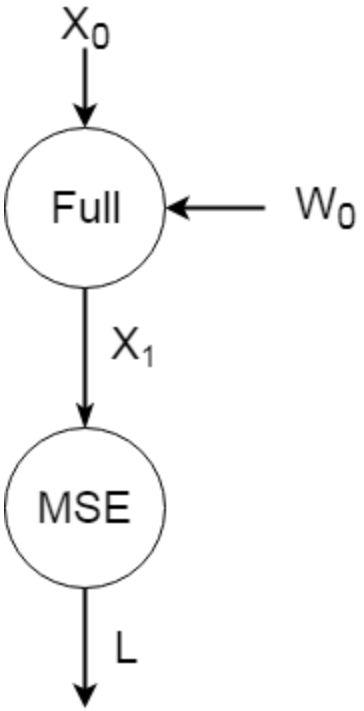


- Network
  - Defines the architecture of the feed forward network
  - A bundle of multiple layers
  - Convenient for training, predicting and evaluation
- Layer
  - Transforms the input data to another representation
- Solver
  - Used for learning the weight parameters of each layer
  - Decoupled from architecture and layers
- Tensor – underlying data type used by all classes

First network - regression

- Fully connected layer + MSE output layer

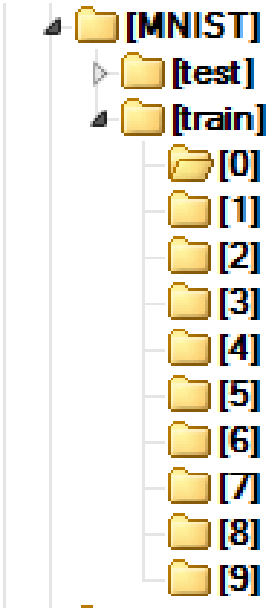
```
Params params;
params.set("learning_rate", 1e-2);
params.set("target_loss", 1e-7);
params.set("max_iter", 10000);

Data data;
data.loadFromFile("..\\data\\regression1.txt");

Network nn(&params);
nn.add(new FullLayer(data.getDim(), 1));
nn.outlayer = new MeanSquaredError();

GDSolver solver(&params);
solver.learn(&data, &nn);
cout << ((FullLayer*)nn.layers[0])->W << endl;
```



Data input-layer

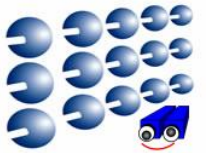
- Loads dataset from disk
  - Single file containing all features
  - Images grouped in folders signifying classes
  - regression1.txt:

```
10 4
15.5984   0.8147   0.1576   0.6557   0.7060
 8.9517   0.9058   0.9706   0.0357   0.0318
13.4720   0.1270   0.9572   0.8491   0.2769
14.3497   0.9134   0.4854   0.9340   0.0462
12.5699   0.6324   0.8003   0.6787   0.0971
13.7048   0.0975   0.1419   0.7577   0.8235
14.5832   0.2785   0.4218   0.7431   0.6948
12.0720   0.5469   0.9157   0.3922   0.3171
20.8020   0.9575   0.7922   0.6555   0.9502
10.3075   0.9649   0.9595   0.1712   0.0344
```



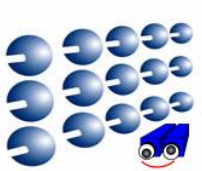
equation: y = 5 x1 + 4 x2 + 8 x3 + 8 x4

Tensor class



- All layers operate on a tensor and output a different tensor
- A tensor is a multidimensional generalization of a vector (1D) or a matrix (2D)
- Of special interest are
  - 2-dimensional tensors (matrices) – height, width (rows, cols)
  - 4-dimensional tensors:
    - Number of instances, number of channels (depth), height, width
- All 4D tensors can be transformed to matrices via linearization

Tensor class – representation



- Matrix form (2D)
  - Use and OpenCV Mat of floats/doubles

```
Mat_<T> mform;
```

  - Each row contains all features for a single instance
- Tensor form (4D)
  - Possible solution
    - Linearize and store `X(i,j,k,l)` at position

```
l + width*(k + height*(j + depth*i)))
i * d1 + j * d2 + k * d3 + l
```

  - Proposed solution
    - Store as a 2D vector of Mats (2D x 2D = 4D)

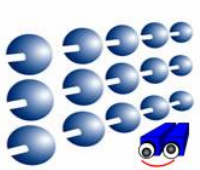
```
vector<vector<Mat_<T> > > tform;
```

    - Enables fast convolution and visualization as images

Tensor class
- efficient representations

Matrix-form	Both	Tensor-form
FullLayer	ReLU	Convolution
Soft-max	Dropout	Maxpool
Outlayers		

Layer abstract class



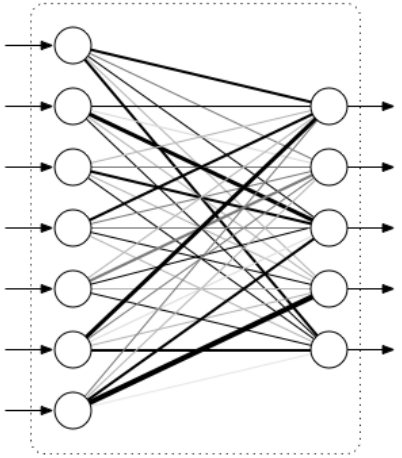
- Every layer has:
  - `void forward(Tensor& Xin, Tensor& Xout)`
    - Forward propagate (or predict)
    - Calculate output tensor from input tensor
    - Store input tensor (required for backpropagation)
  - `void backward(Tensor& dXin, Tensor& dXout);`
    - Backpropagate derivatives from output to input
    - `dXin` and `dXout` are tensors representing the derivative of the loss function
    - Calculate and store partial derivatives of weights

Fully connected layer

- Performs a matrix multiplication on the matrix-form of a tensor
- Forward:

$$X_{out} = \bar{X}_{in} \cdot W$$

  - Use padded input: added column of ones



- Backward:

$$\frac{\partial L}{\partial W} = \bar{X}_{in}^{t} \cdot dX_{out}$$

$$dX_{in} = \frac{\partial L}{\partial X_{in}} = dX_{out} \cdot \breve{W}^{t}$$

  - remove padding from weight matrix

Output layers

- Every output layer has:
  - `void forward(Tensor& Xin, Mat_<T>& Target, T& out);`
    - Calculates the loss function (scalar) out
    - Requires target values (ground truth)
  - `void backward(Tensor& dXin, Mat_<T>& Target, T dout);`
    - Calculates partial derivative of the loss function for each input
    - Requires target values (ground truth)
- Each network contains a single output layer
- Last layer in the network
- Produces a scalar loss value
  - indicates the performance of the network

Mean Squared Error out-layer

- Average mean squared difference between prediction and target (ground-truth)
  - Forward:

$$L_{MSE} = \frac{1}{2n}\sum_{i}\left(X_{in_i} - T_i\right)^2$$

  - Backward:

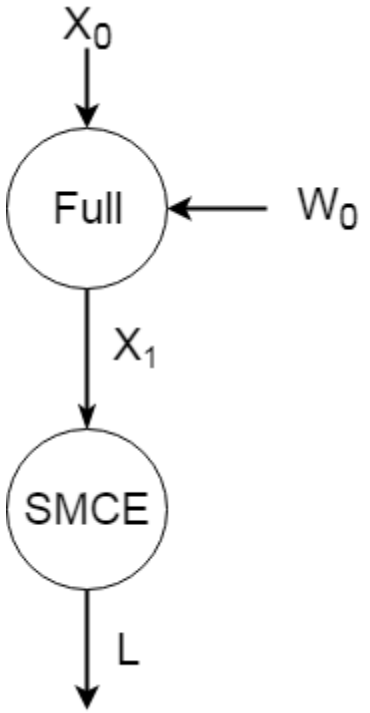
$$dX_{in_i} = \frac{1}{n}\left(X_{in_i} - T_i\right)$$

Experimental results – network 1

- Dataset: regression1.txt
- Running time: 0.9 sec
- End loss: 0
- Training error: 0
- Output:
  - [4.9991; 3.9985; 7.9979; 7.9996; 0.0030]
- Ground truth:
  - [5; 4; 8; 8; 0]

Second network - classification

```
Params params, outparams;
params.set("learning_rate", 1e-2);
params.set("max_iter", 200);

Data data;
data.loadImageDataset("d:/imgdb/MNIST/train");

Network nn(&params);
nn.add(new FullLayer(data.getDim(), data.getNoClasses()));
nn.outlayer = new SMCE();

GDSolver solver(&params);
T end_loss = solver.learn(&data, &nn);
outparams.set("final loss", end_loss);
outparams.set("training time", t.toc());

Evaluation eval;
outparams.set("training error", eval.classification_error(&data, &nn));

Data test_data;
test_data.loadImageDataset("d:/imgdb/MNIST/test");
outparams.set("test error", eval.classification_error(&test_data, &nn));
```



Softmax layer

- Produces a probability distribution
- Assume input values are unnormalized log-likelihoods
- Forward:

$$X_{out_{ij}} = \frac{\exp(X_{in_{ij}})}{\sum_k \exp(X_{in_{ik}})}$$

- Backward:

$$dX_{in_{ij}} = X_{out_{ij}} \left(dX_{out_{ij}} - \sum_k dX_{out_{ik}} X_{out_{ik}} \right)$$

Cross-entropy out-layer

- The negative average of the log-probabilities of the correct classes
- Forward:

$$L_{CE} = -\frac{1}{n}\sum_i \log(X_{in_{iT_i}})$$

- Backward:

$$dX_{in_{ij}} = -1/(nX_{in_{ij}})[j = T_i]$$

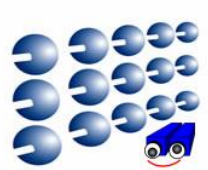
SMCE out-layer

- Combine Softmax and Cross-entropy into a single layer
- More numerically stable
  - Avoid overflow in *exp*

$$X_{out_{ij}} = \frac{\exp(X_{in_{ij}} - \max(X_{in_{ik}}))}{\sum_k \exp(X_{in_{ij}} - \max(X_{in_{ik}}))}$$

  - Avoid division in backward step
  - Avoid some log operations

Solver



- Gradient descent
  - Perform forward pass
    - Input tensors are stored by each layer for later
  - Perform backward pass
    - Each layer knows how to calculate the gradient
  - Perform parameter update

$$W \leftarrow W - \alpha \frac{\partial L}{\partial W}$$

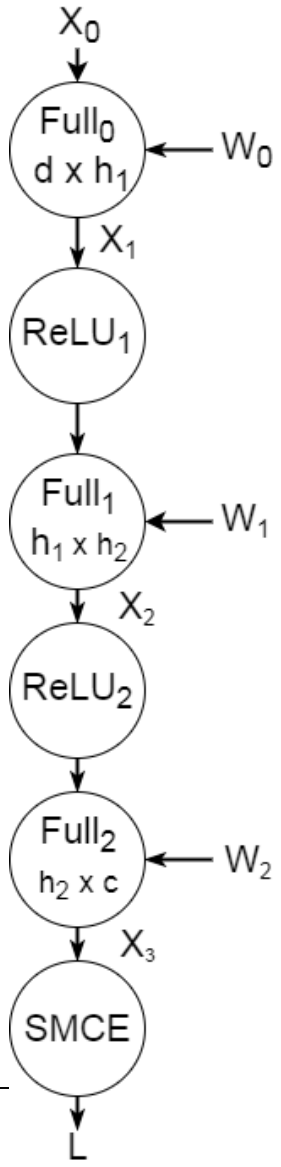
- Stochastic gradient descent
  - Random permutation of input instances
  - Select a small batch and perform gradient descent
  - Required for large networks (memory constraints)

Experimental results – network 2

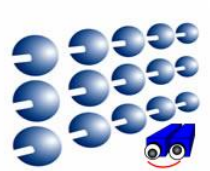
- Dataset: MNIST
- Running time: 56 sec
- End loss: 0.26
- Training error: 7.3%
- Test error: 7.6%
- Issues
  - Model is simple

Third network – deep classification

```
Params params, outparams;
params.set("learning_rate", 1e-5);
params.set("max_iter", 2000);
int hidden_layer_size = 100;
int hidden_layer_size2 = 100;
params.set("hidden1", hidden_layer_size);
params.set("hidden2", hidden_layer_size2);
//params.set("batch_size", 600);

Data data;
data.loadImageDataset("d:/imgdb/MNIST/train");

Network nn(&params);
nn.add(new FullLayer(data.getDim(), hidden_layer_size));
nn.add(new ReLU());
nn.add(new FullLayer(hidden_layer_size, hidden_layer_size2));
nn.add(new ReLU());
nn.add(new FullLayer(hidden_layer_size2, data.getNoClasses()));
nn.outlayer = new SMCE();
```



Weight initialization



- Important and often overlooked
- Monitor loss function values
  - Exploding loss function:
    - Large learning rate
    - Large initial weights
- Sample weights randomly
  - Uniform distribution around 0
  - Normal distribution with a small standard deviation and zero mean
- Std. deviation dependent on the number of instances/dimensionality
- Xavier initialization $\sigma \approx \sqrt{|X|}$

Rectified Linear Unit layer

- For deep learning non-linear layers need to be inserted between linear layers (full layers and convolutional layers)
- Introduced as a better alternative for *sigmoid*, *tanh* and other activation layers
- Solution for vanishing gradients
- Forward:

$$X_{out} = \max(X_{in}, 0) = [X_{in} > 0] X_{in}$$

- Backward:

$$dX_{in} = [X_{in} > 0] dX_{out}$$

Leaky Rectified Linear Unit layer

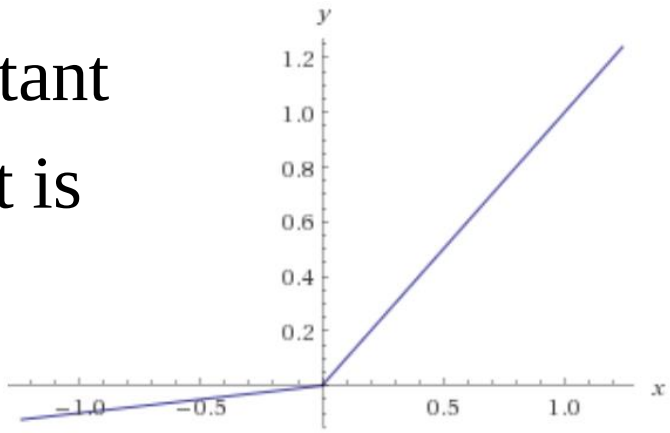
- Forward:

$$X_{out} = ([X_{in} > 0] + b[X_{in} < 0])X_{in}$$

- Backward:

$$dX_{in} = ([X_{in} > 0] + b[X_{in} < 0])dX_{out}$$

- If $b = 0$ equivalent to normal ReLU
- Typically b is a small positive constant
- Propagates derivatives even if input is negative

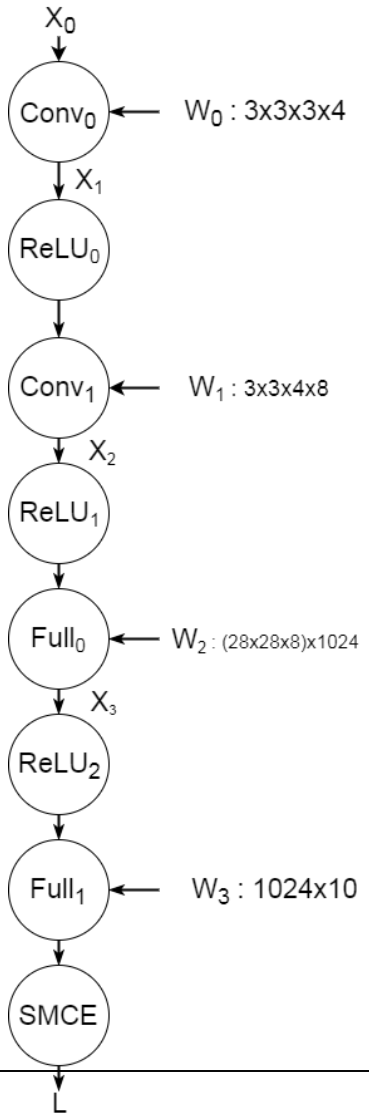


Experimental results – network 3

- Dataset: MNIST
- Running time: 13k sec = 3.6 hrs
- End loss: 0.10
- Training error: 2.9%
- Test error: 4.1%
- Issues
  - Weight initialization is important

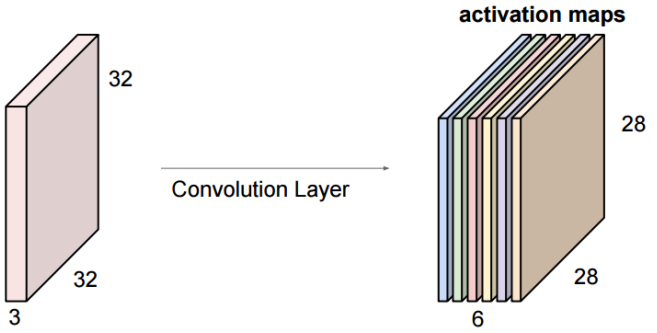
Fourth network – convolutional

```
Params params, outparams;
params.set("learning_rate", 1e-3);
params.set("max_epochs", 100);
params.set("learning_rate_decay", 9e-1);
params.set("learning_rate_epochs", 10);
params.set("batch_size", 100);
params.set("regularization_factor", 0);// 1e-2);

Data data;
data.loadImageDataset("d:/imgdb/MNIST/train");

Network nn(&params);
nn.add(new Convolution(1, 28, 3, 4));//0
nn.add(new ReLU());//1
nn.add(new Convolution(4, 28, 3, 8));//2
nn.add(new ReLU());//3
nn.add(new FullLayer(8*28*28, 1024));//4
nn.add(new ReLU());//5
nn.add(new FullLayer(1024, 10));//6
nn.outlayer = new SMCE();
```



Convolutional layer

- Generalization of 2D image convolution
- Performs inner product with a weight matrix on all input channels
- Use tensor-form of tensors
- Advantages:
  - Number of weights is small and they are reused
  - Takes the spatial layout of features into account



Convolutional layer

- Forward

$$X_{out}(i,j) = \sum_{k=1:d_{in}} W_{kj} \circ X_{in}(i,k)$$

  - For each instance
  - For each output channel
  - Perform convolution on each input channel
  - Sum up values (3D convolution = local inner products)
  - Constant term included (omitted from formulas)

- Backward

$$dX_{in}(i,k) = \sum_{j=1:d_{out}} W_{kj} \circ dX_{out}(i,j)$$

Regularization

- Penalize large weight values
- L2 regularized loss function

$$L_{reg} = L + \frac{\lambda}{2}\|W\|^2$$

- Can be easily implemented in the backwards function or in the update step
- L2 regularization with weight decay λ:

$$W \leftarrow W - \alpha\left(\frac{\partial L}{\partial W} + \lambda W\right)$$

- Avoids overfitting for complex models

Experimental results – network 4

- Dataset: MNIST
- Running time: 117k sec = 32.5 hrs
- End loss: 0.079
- Training error: 1.48%
- Test error: 1.83%
- Issues
  - Stochastic batch gradient descent is required
  - Regularization is required for complex models

Experimental results - summary

- Dataset: MNIST

Network	Training time	Test error
1 Full layer	56 sec	7.60%
3 Full layers	13,000 sec	4.10%
2 Conv + 2 Full layers	117,000 sec	1.83%
LeNet-5 1998	-	0.95%
DropConnect 2013	-	0.21%

Conclusions

- Library contains essential layers
- Development provided insights into
  - Why other libraries use tensors and how they are represented in the memory
  - How backpropagation works
  - Vanishing gradients and weight magnitudes
  - The importance of weight initialization
- Further work
  - Collaboration
  - Further testing and validation
  - Development of other existing layers (batch normalization) and solvers (momentum, AdaGrad, Adam)
  - Research and development of new layers

Acknowledgment:
This work was supported by the EU H2020 project, UP-Drive under grant nr. 688652

Thank you for your attention!

Questions?