Neural Network and Deep Learning Overview Using Julia's Mocha and Lua's Torch

Paulito P. Palmes
IBM Dublin Research Lab

> "My dear, here we must run as fast as we can, just to stay in place. And if you wish to go anywhere you must run twice as fast as that."
> — Lewis Carroll, *Alice in Wonderland*

November 14, 2015

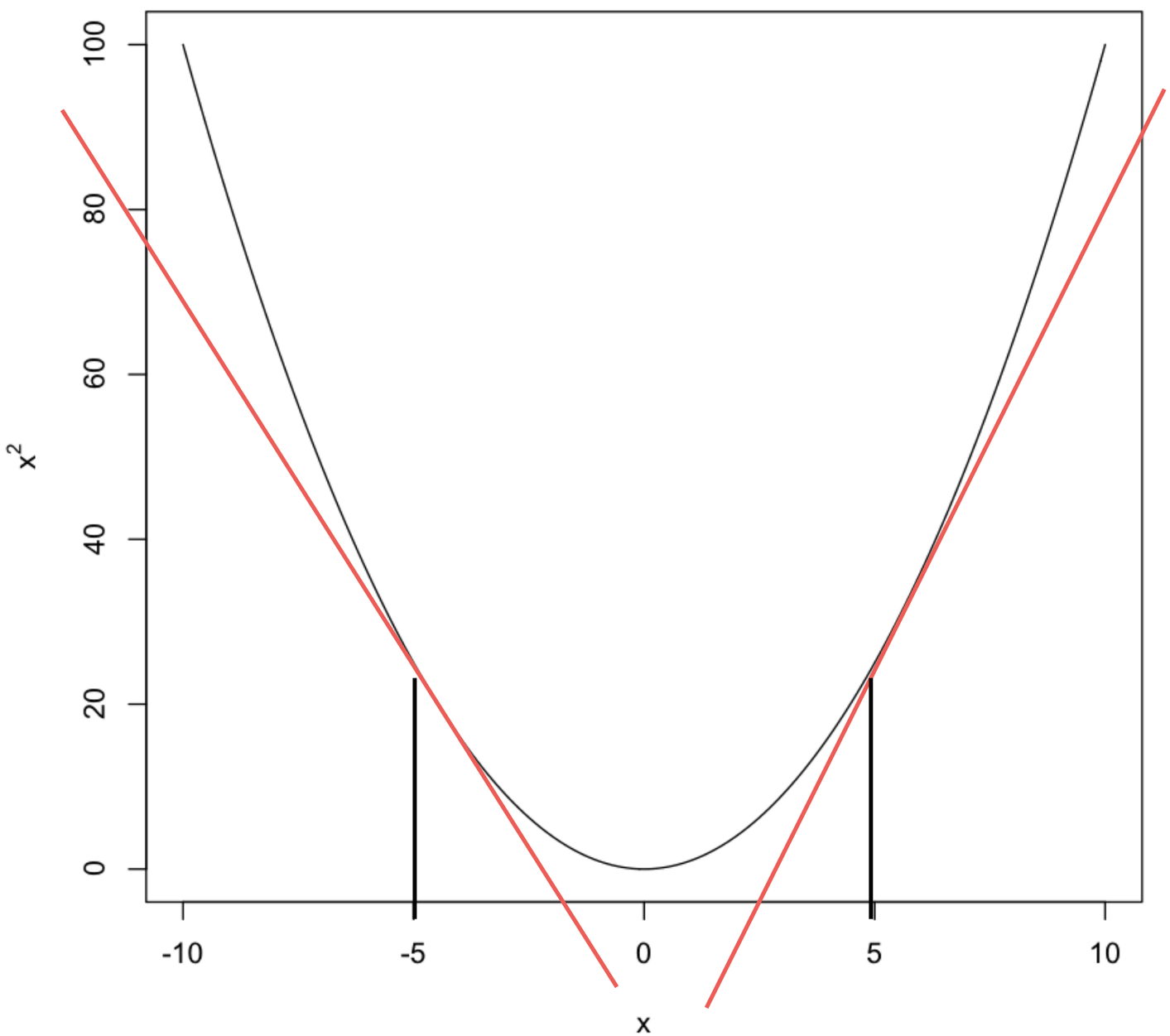
Outline

- **Preliminaries**
- **Historical Tidbits**
- **ANN Feedforward Computation and Error Backpropagation**
- **The Deep Learning Architecture**
- **Sample Implementations (MNIST, CIFAR, XOR)**

Modeling

- **Model selection**
- **Feature selection and extraction**
- **Parameter adaptation**
- **Testing and Validation**
- **Quick R demo of Modeling Workflow:**
  SVM vs RF vs Neural Network

Gradient Descent



x = -5

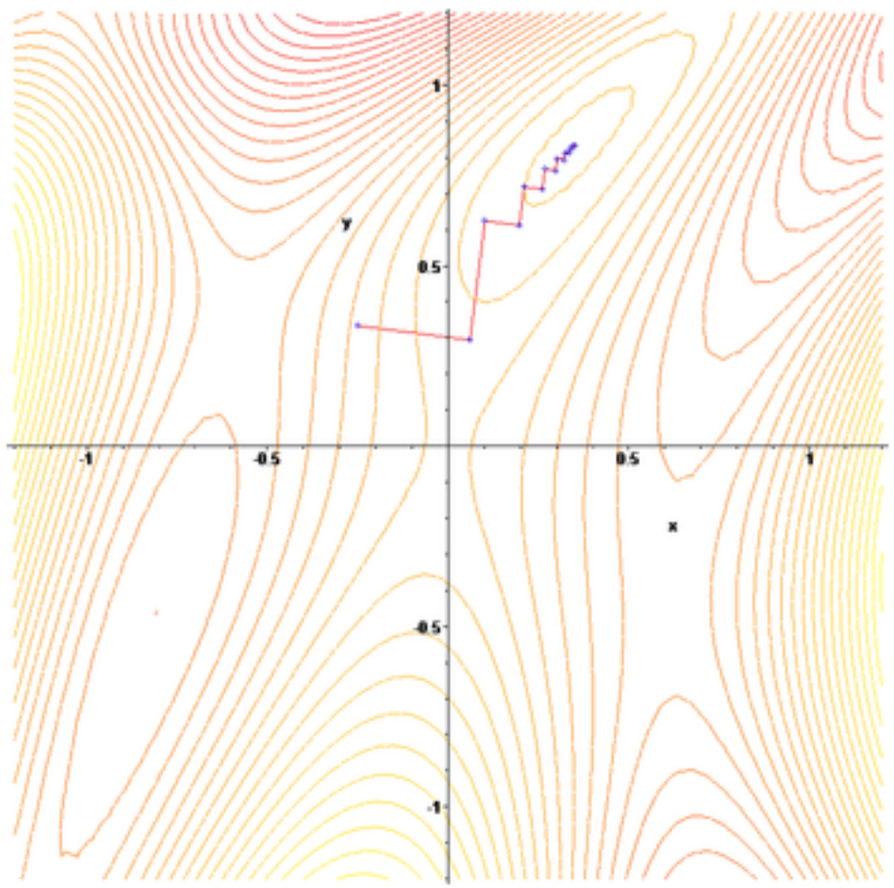
Finding the minimum point in a function y=f(x):

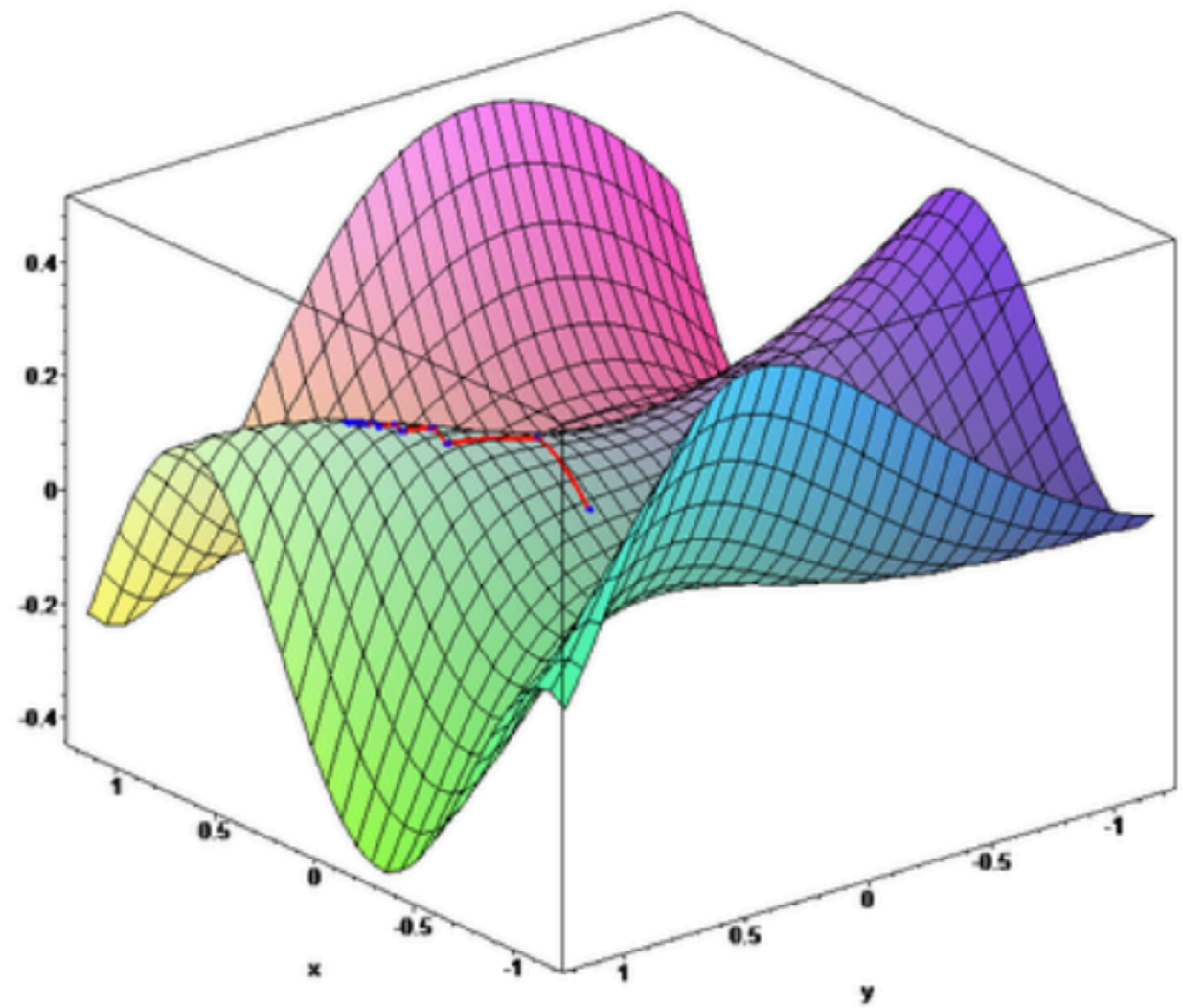
Assumption: f(x) is differentiable (continuous)

1. start x somewhere
2. compute the slope (gradient)
3. move x in the direction negative gradient

f(x1) > f(x2) > f(x3) > … > f(xn)

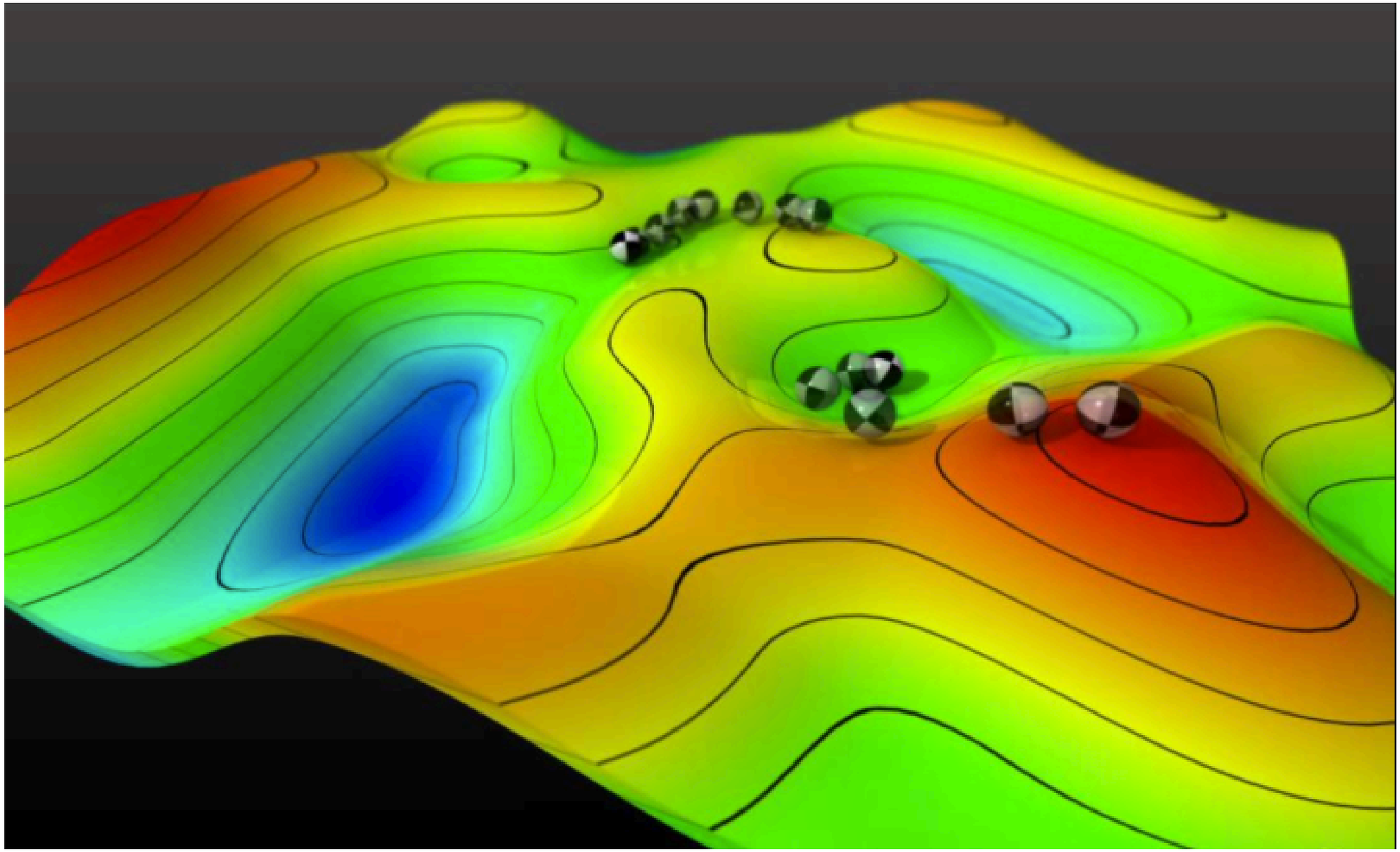
Gradient Descent





local optima

Gradient Descent



source: https://www.youtube.com/watch?v=vWFjqgb-ylQ

Outline

- **Preliminaries (modeling workflow, gradient descent)**
- **Historical Tidbits**
- ANN Feedforward Computation and Error Backpropagation
- The Deep Learning Architecture
- Sample Implementations (digit recognition, iris/xor)

Historical Tidbits

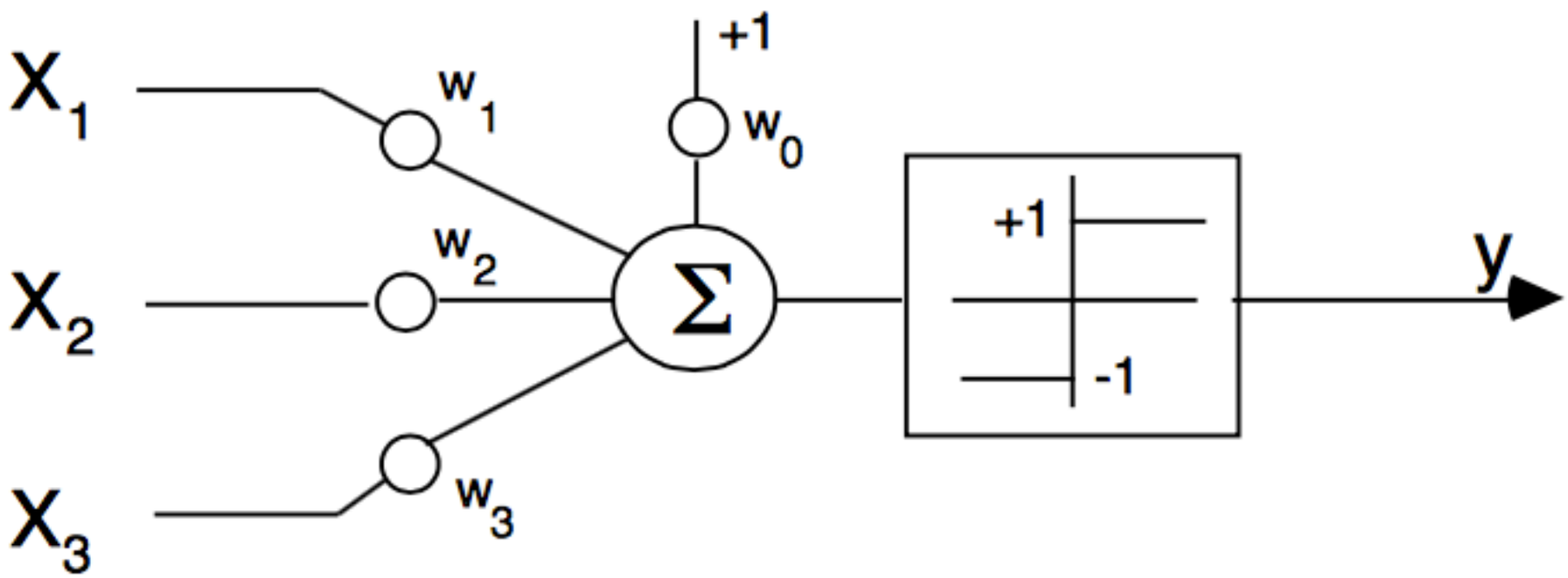
- **1943: McCulloch and Pitts** introduced a simple neuron model with two inputs and one output. The output is binary and remains zero unless the summed inputs reach a certain threshold (threshold logic algorithm)
- **1958: Rosenblatt** invented **Perceptron** which is a 2-layer neural network that uses simple addition and subtraction. It is equivalent to linear discriminant.
- **1969: Marvin Minsky and Papert** wrote a paper criticizing the shortcomings of 2-layer neural networks: can't process exclusive-or (XOR) problem; no enough computation power to process large networks. **This is the first AI winter.**
- **1975: Paul Werbos** solved the exclusive-or problem by backpropagation.
- **1980: Kunihiko Fukushima** invented **neocognitron** which is based on the vision architecture.
- **1986: Rumelhart and McClelland** wrote a textbook about connectionism techniques for simulation of neural processes.
- **1986: Yann LeCun and Geoff Hinton** reintroduced backpropagation with more hidden layers
- **1987: Terry Sejnowski** invented NET-talk for text reading
- **1995: Vladimir Vapnik** introduced SVM. **Second AI Winter.**
- **2006: Hinton Group** won ImageNet large scale visual recognition challenge — significantly better error rate than previous best models.

Historical Tidbits

- **2010: IBM, Google, Microsoft** started to incorporate Deep Learning in their speech recognition systems.
- **2012: Google** uses Deep Learning to improve error rate by 25% in speech recognition
- **2013: Google** hired **Hinton** and acquired DNN Research started working in Google
- **2013: FB** hired **Yann LeCun**
- **2014: FB** started to use Deep Learning in photo tagging etc.
- **2014: Baidu** hired **Andrew Ng** (from Stanford Univ) to become the head of their research lab in Deep Learning research. **Andrew Ng** is the founder of **Google Brain** that uses **16,000 CPU cores**.
- **2014: Microsoft** established the Deep Learning Tech Center
- **2015: Introduction of Google Photos** that uses Deep Learning to search photos matching descriptions.
- **2015: IBM** starts to collaborate with **Yoshua Bengio** (Univ. of Montreal, Canada)

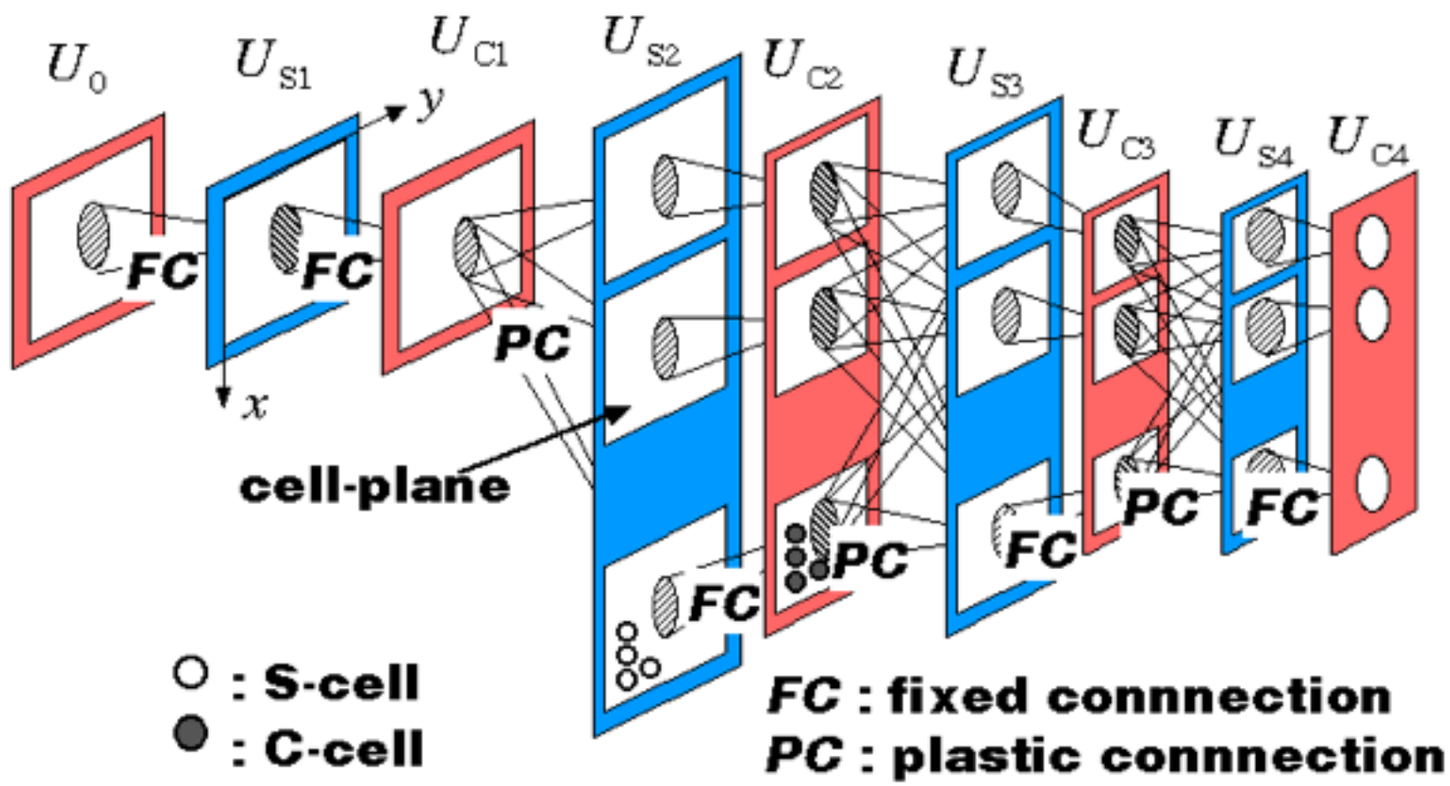
http://www.technologyreview.com/featuredstory/540001/teaching-machines-to-understand-us/

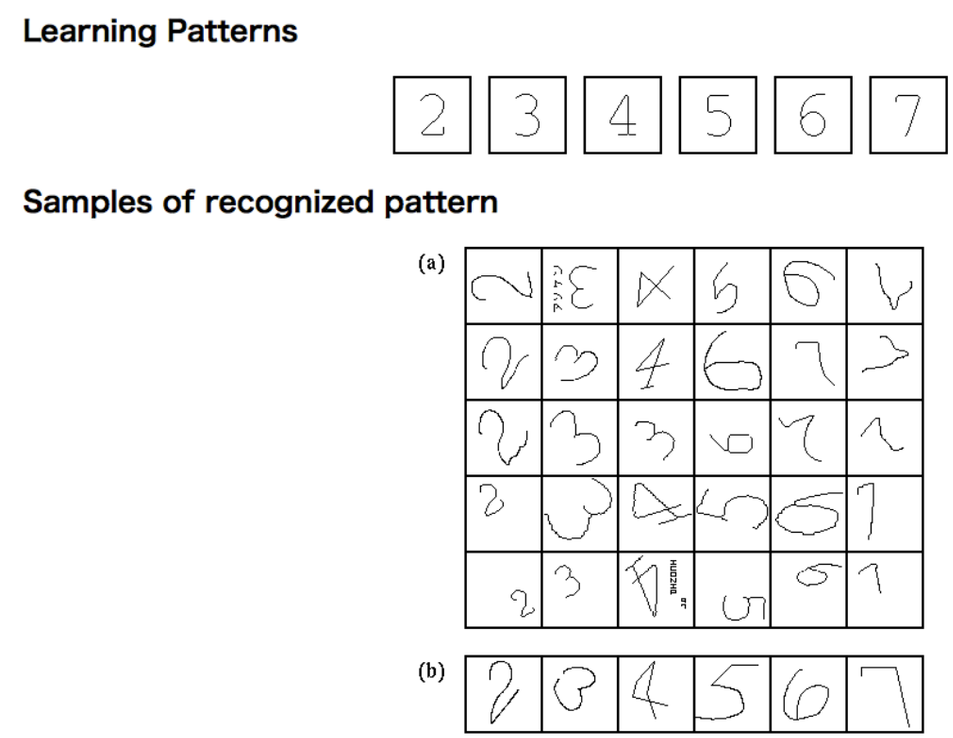
Perceptron



source: http://scholar.lib.vt.edu/theses/available/etd-111597-81423/unrestricted/chap2.pdf

Neocognitron



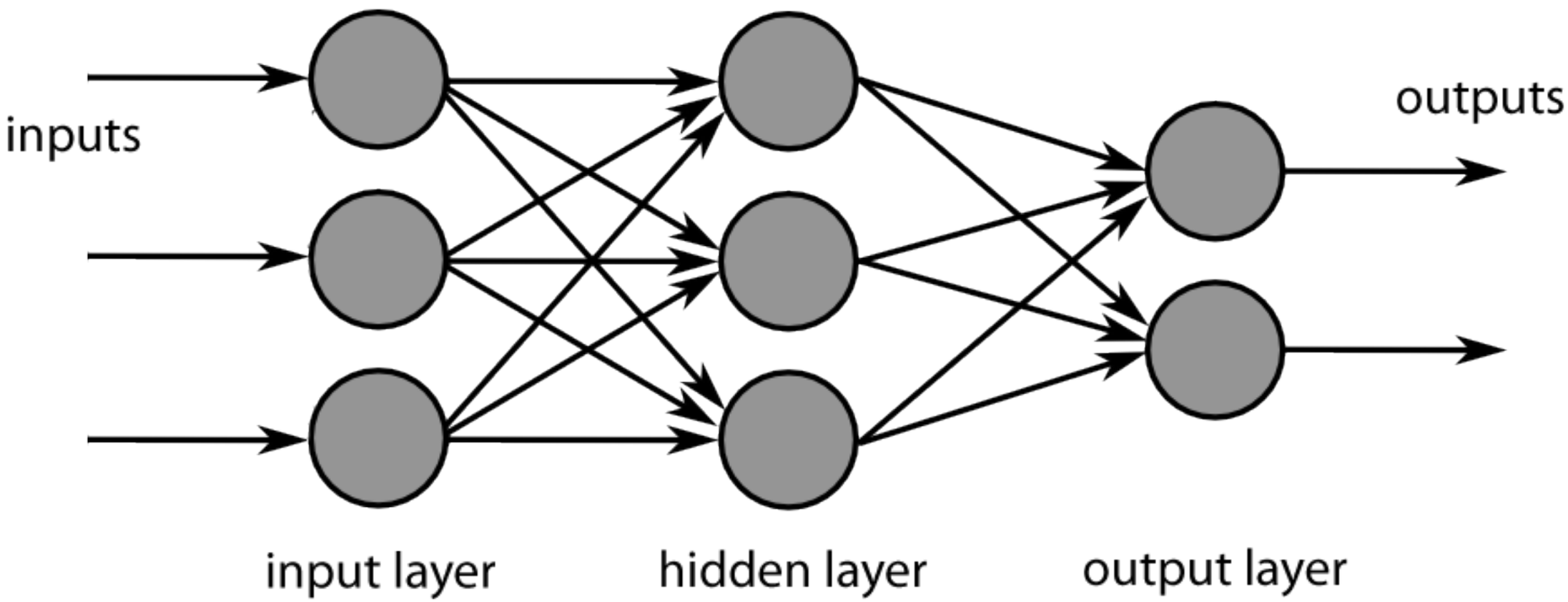


source: http://www.aso.ecei.tohoku.ac.jp/~shun/RNC-e.html

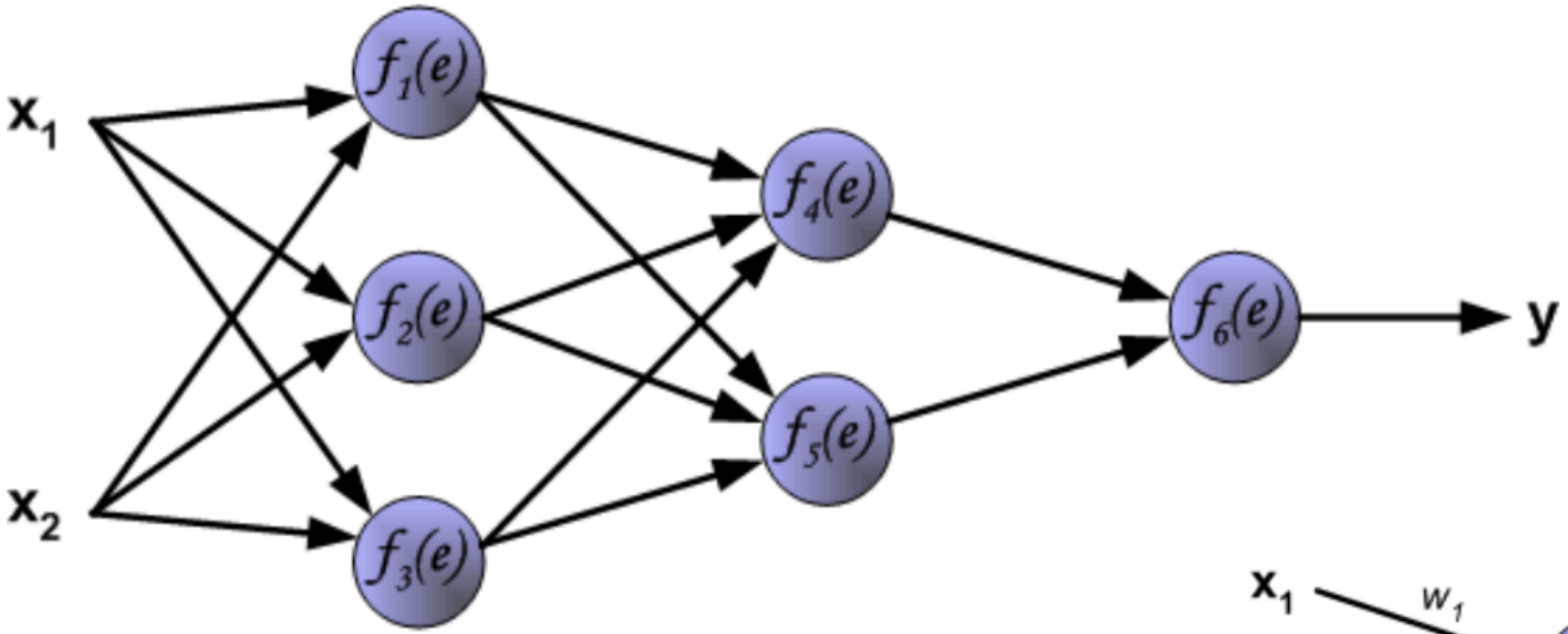
Outline

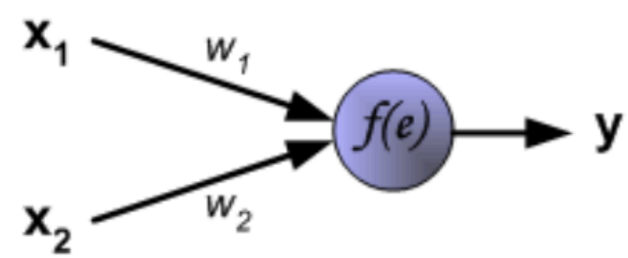
- Preliminaries (modeling workflow)
- Historical Tidbits
- **ANN Feedforward Computation and Error Backpropagation**
- The Deep Learning Architecture
- Sample Implementations (digit recognition, iris/xor)

Feedforward Neural Network

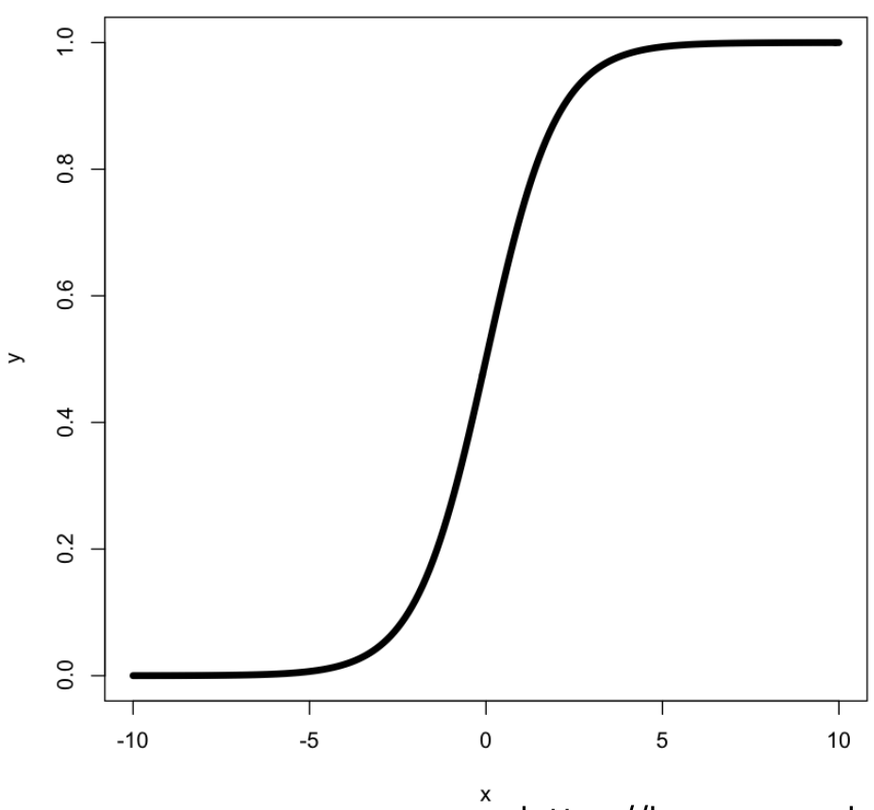


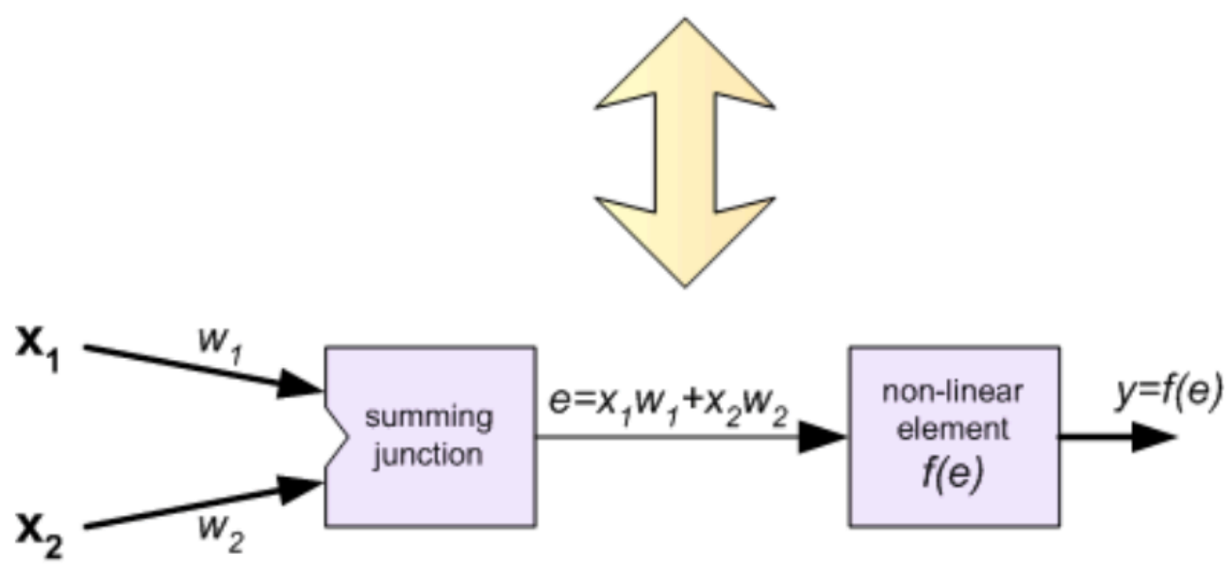
Feedforward Neural Network





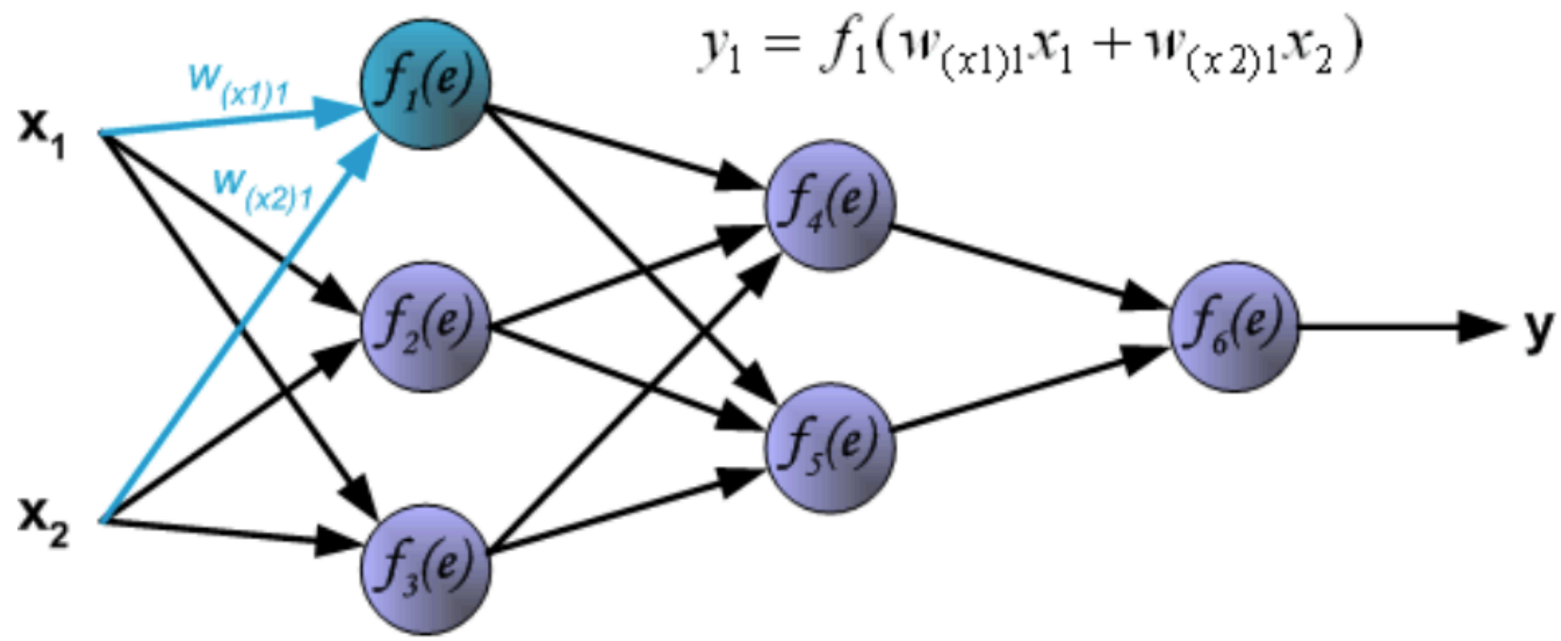
$1/(1+e^{-x})$

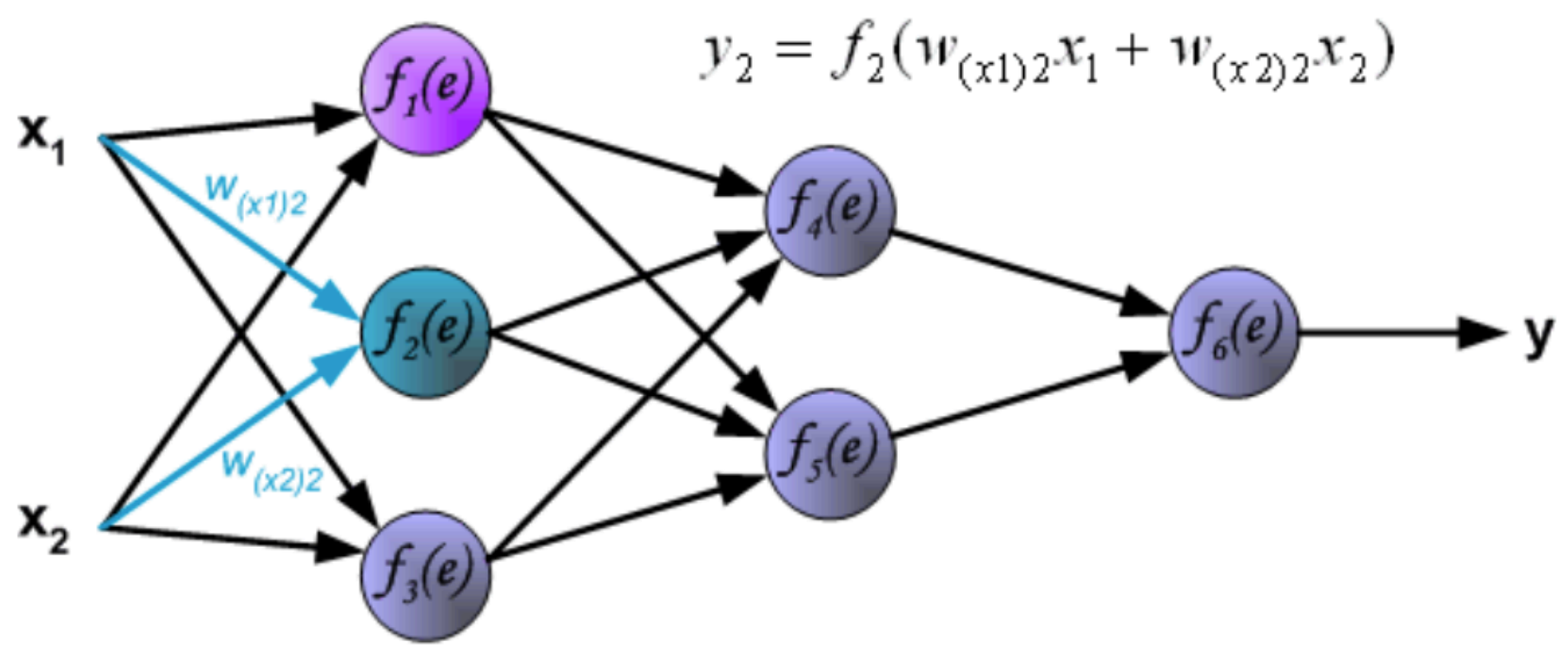




http://home.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html

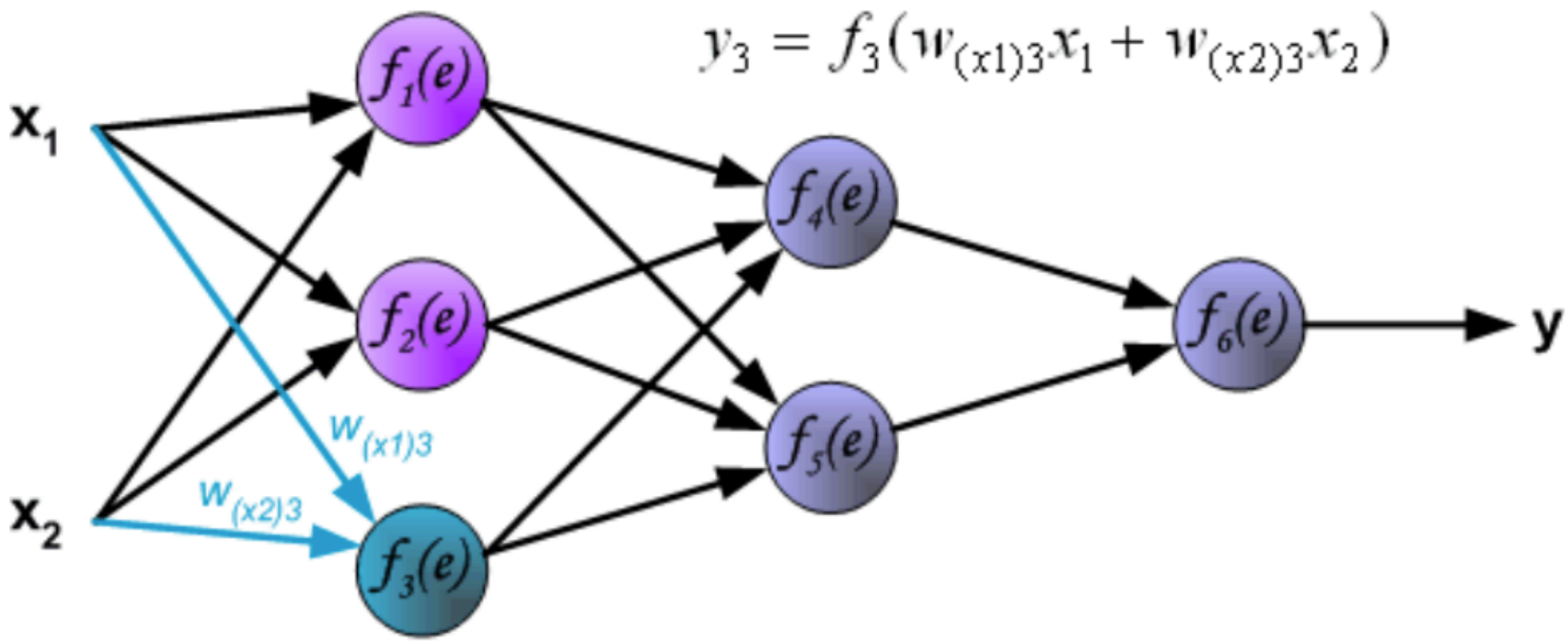
Feedforward Neural Network





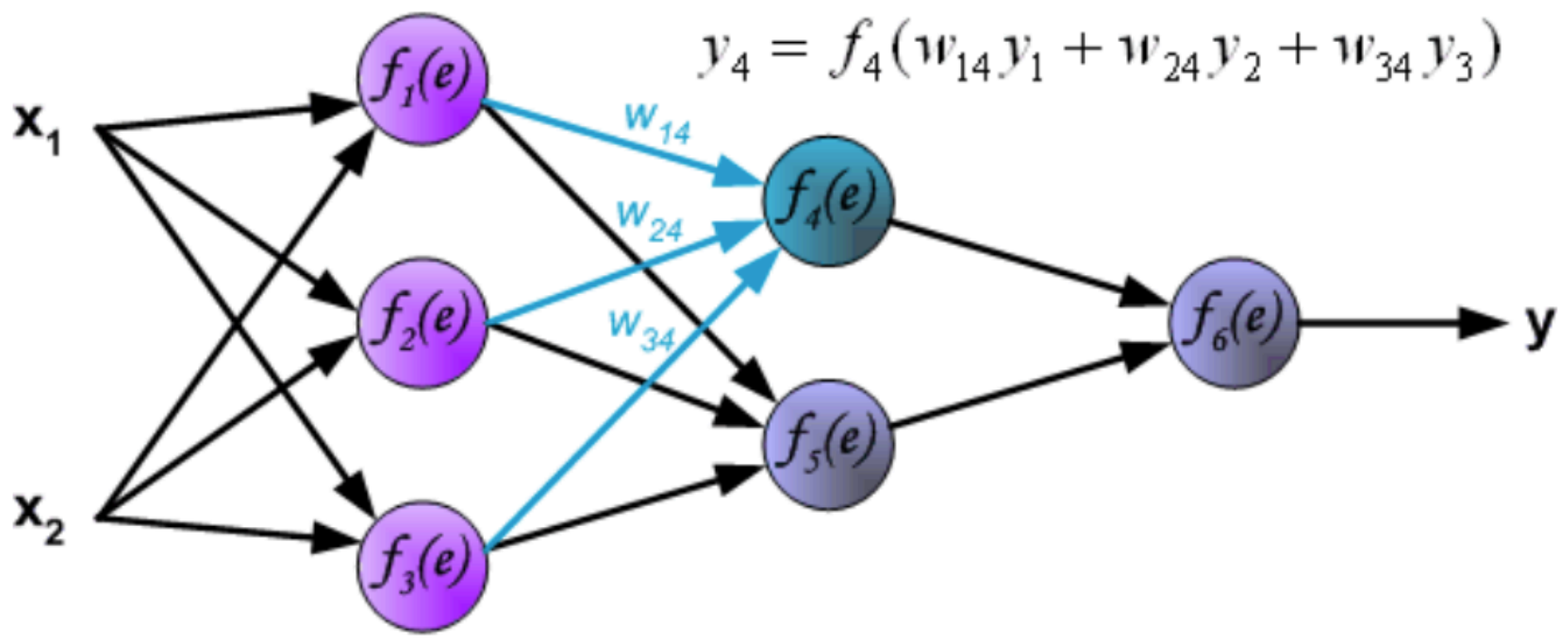
http://home.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html

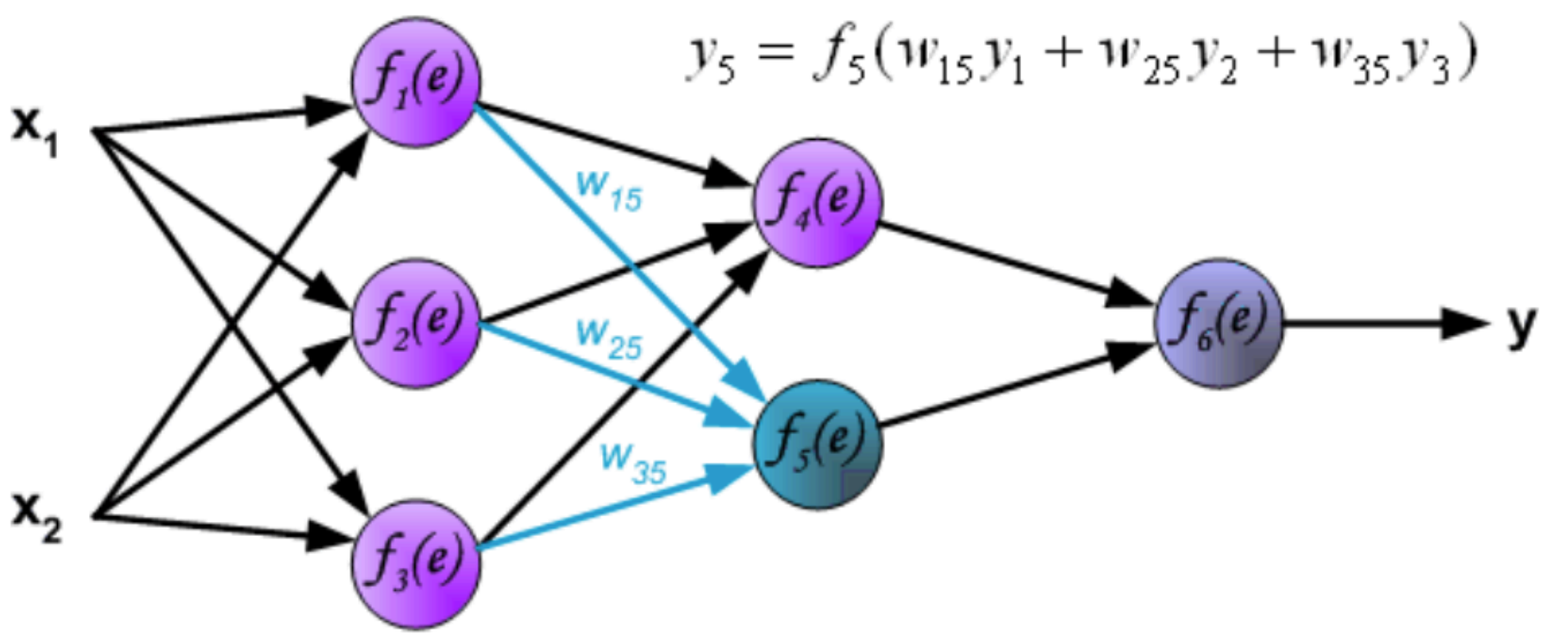
Feedforward Neural Network



http://home.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html

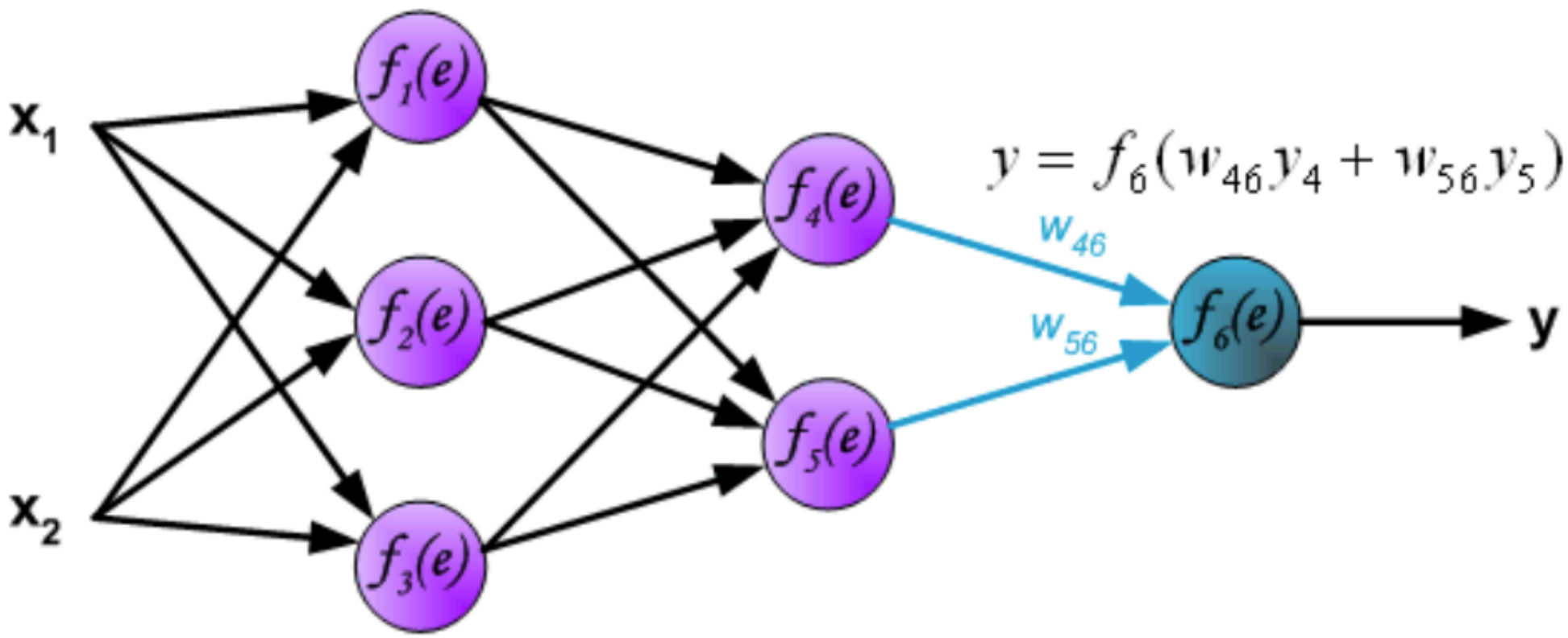
Feedforward Neural Network





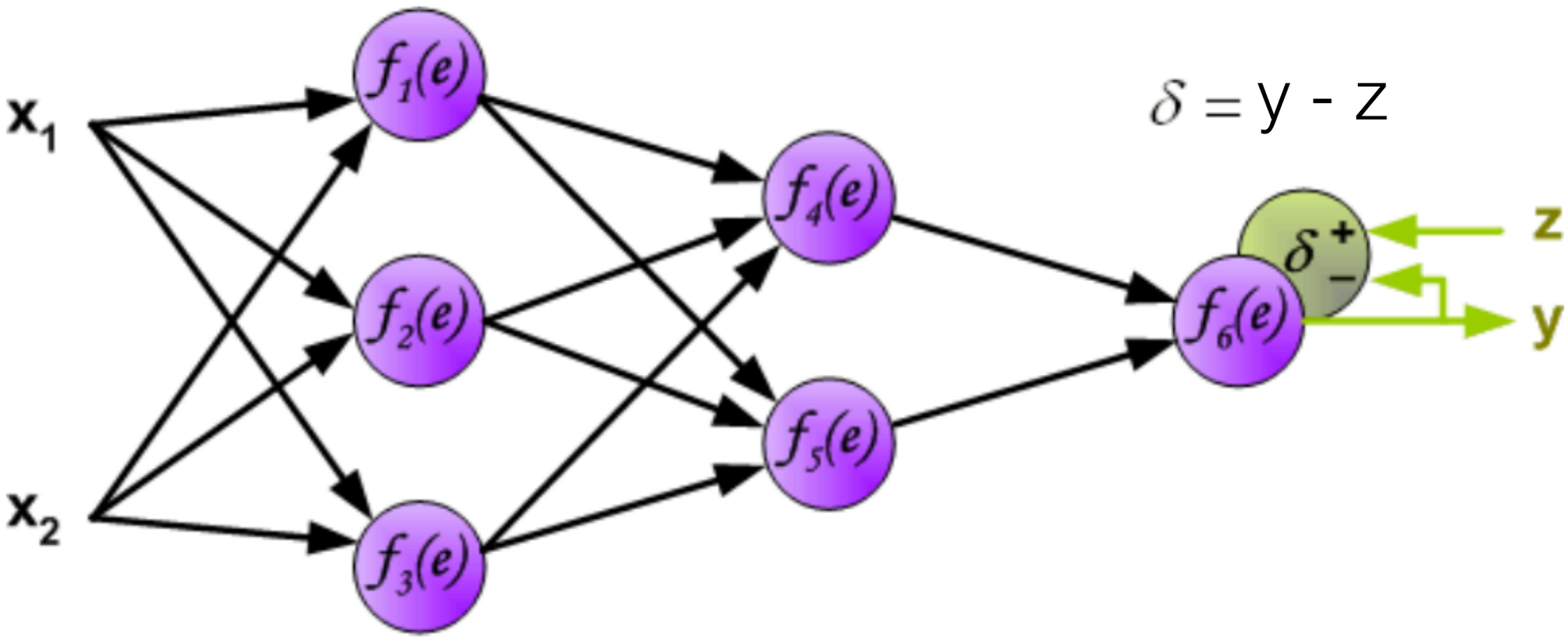
http://home.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html

Feedforward Neural Network



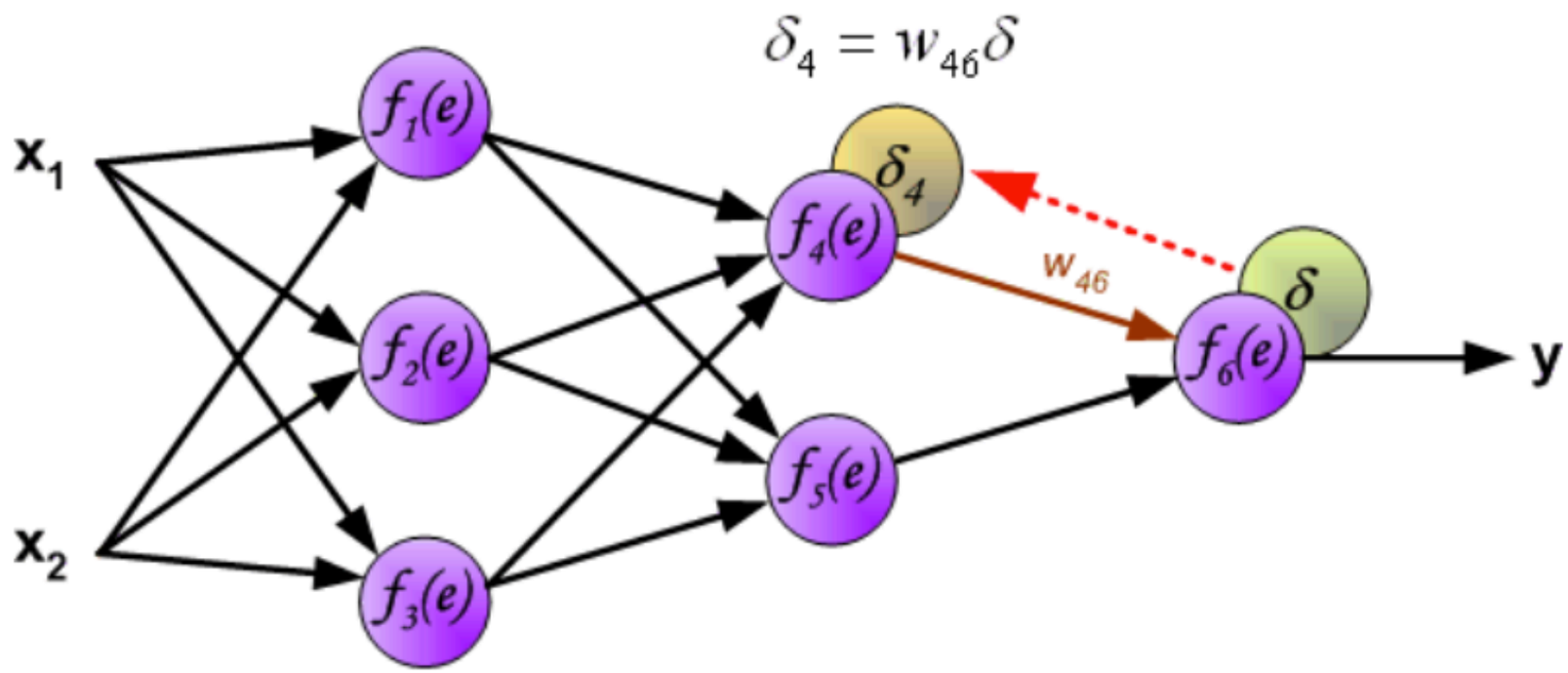
http://home.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html

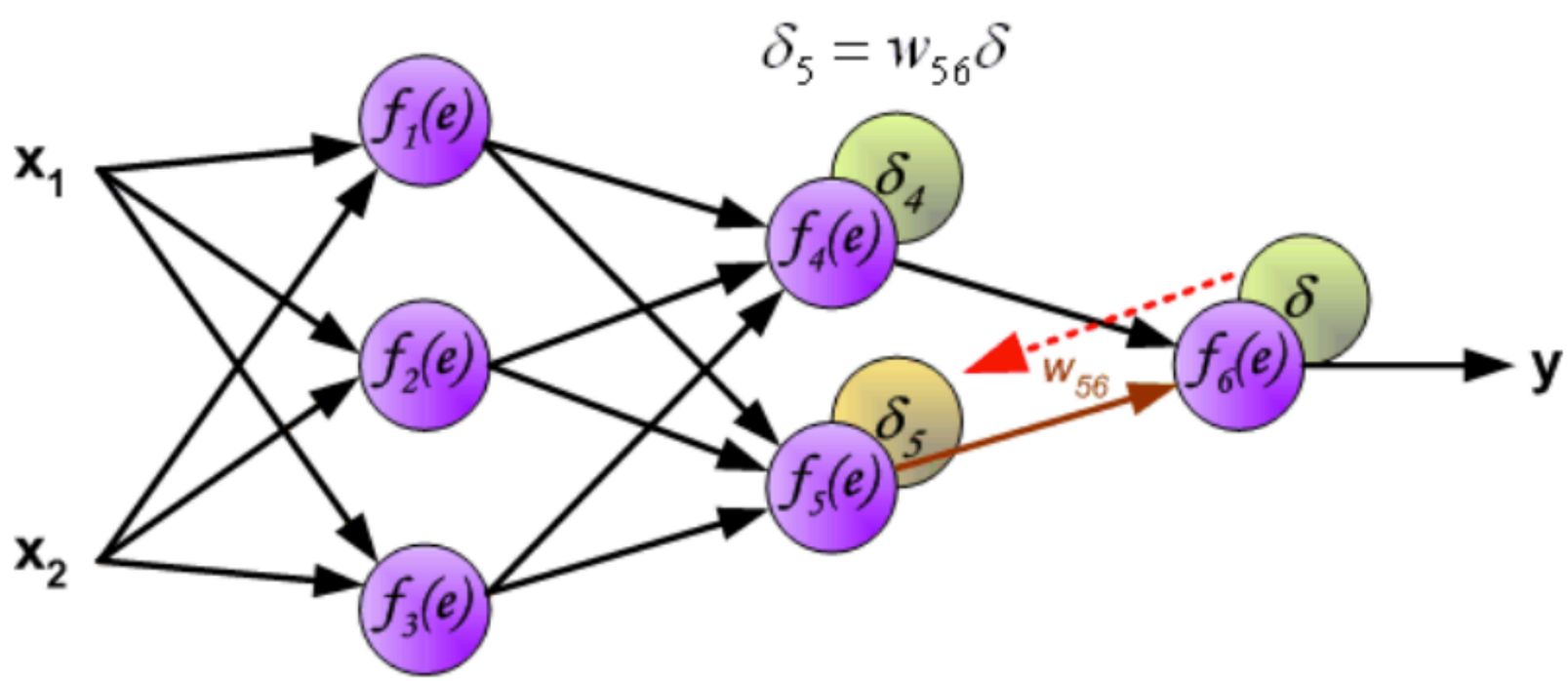
Error Backpropagation



http://home.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html

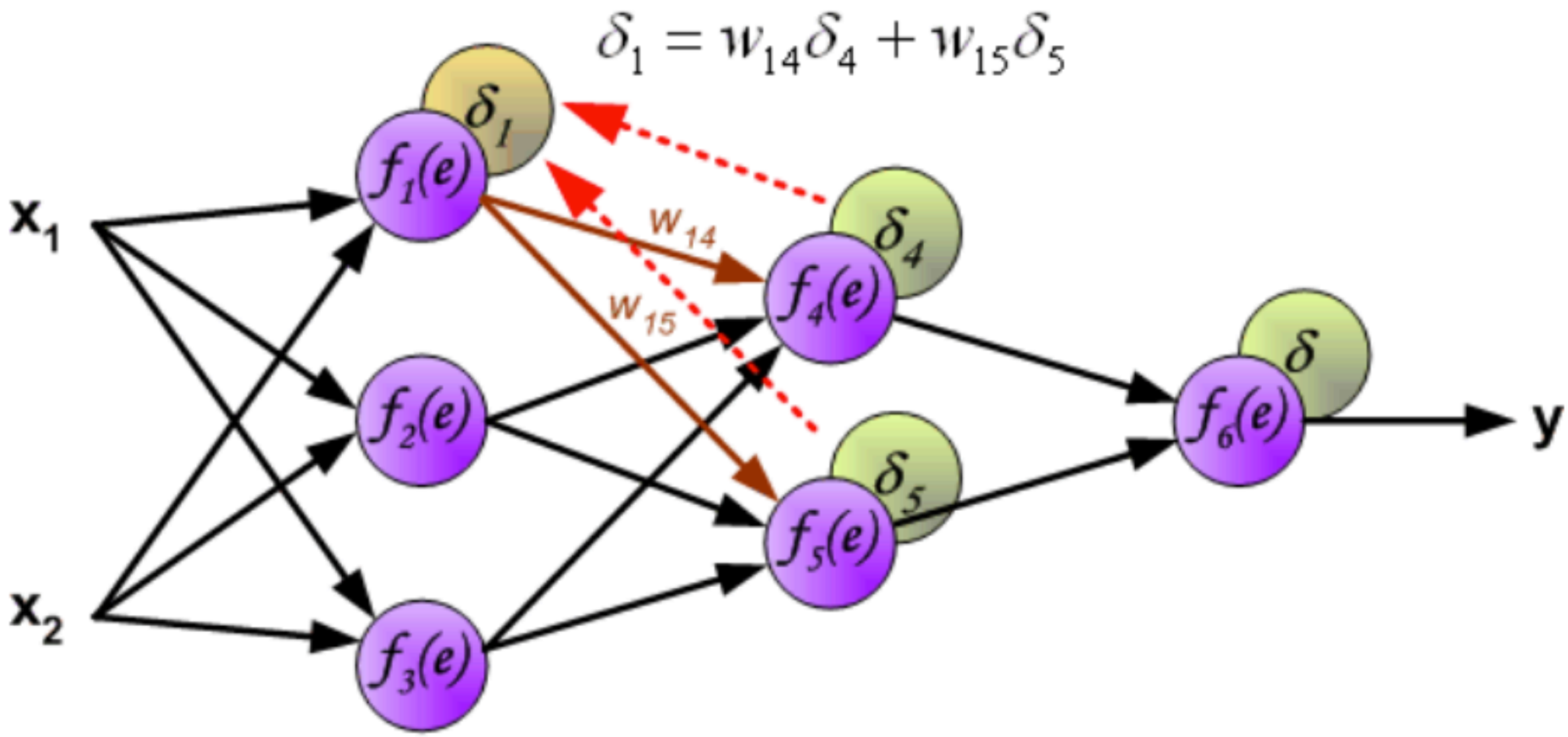
Error Backpropagation





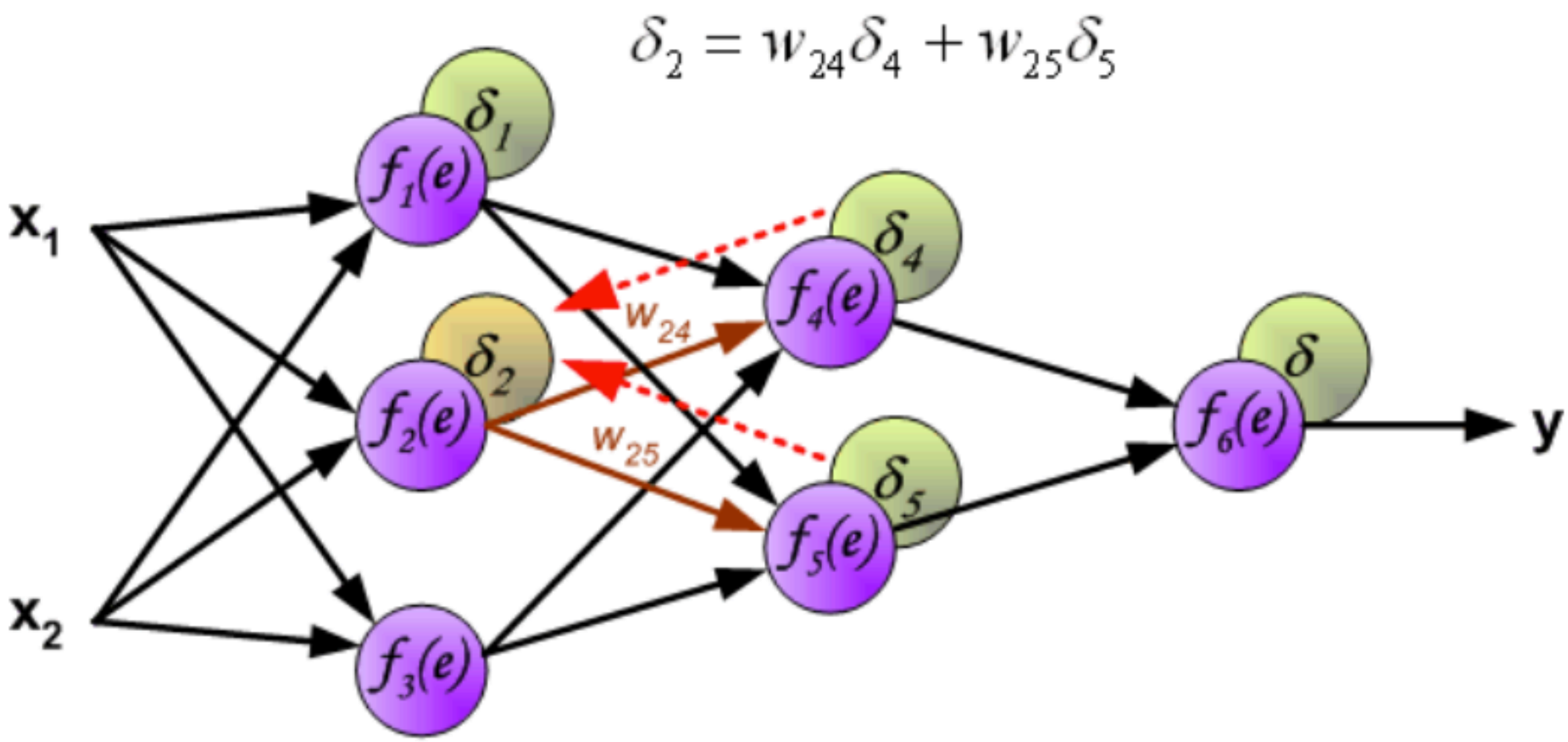
http://home.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html

Error Backpropagation



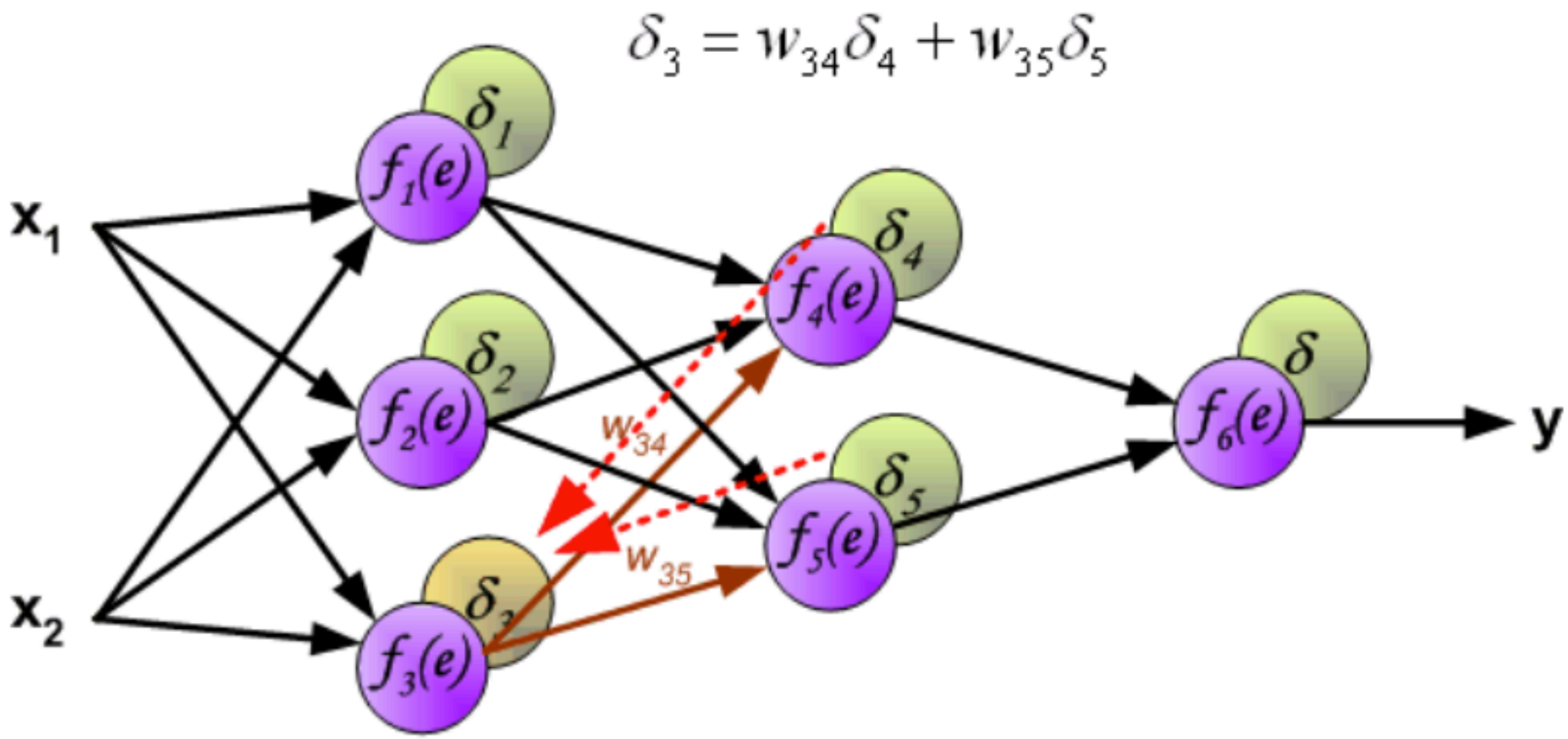
http://home.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html

Error Backpropagation



http://home.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html

Error Backpropagation

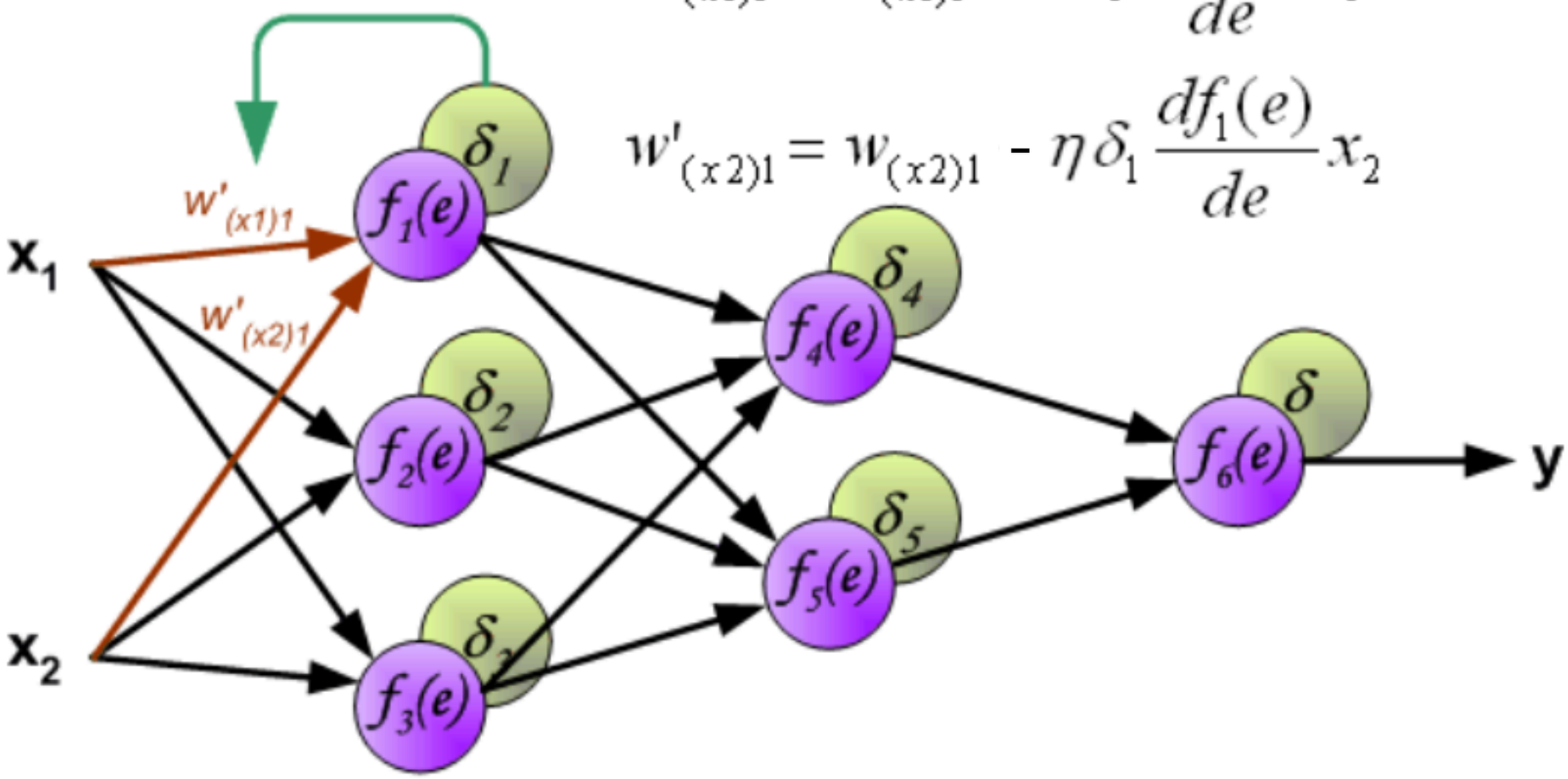


http://home.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html

Error Backpropagation

$$w'_{(x1)1} = w_{(x1)1} - \eta \delta_1 \frac{df_1(e)}{de} x_1$$

$$w'_{(x2)1} = w_{(x2)1} - \eta \delta_1 \frac{df_1(e)}{de} x_2$$

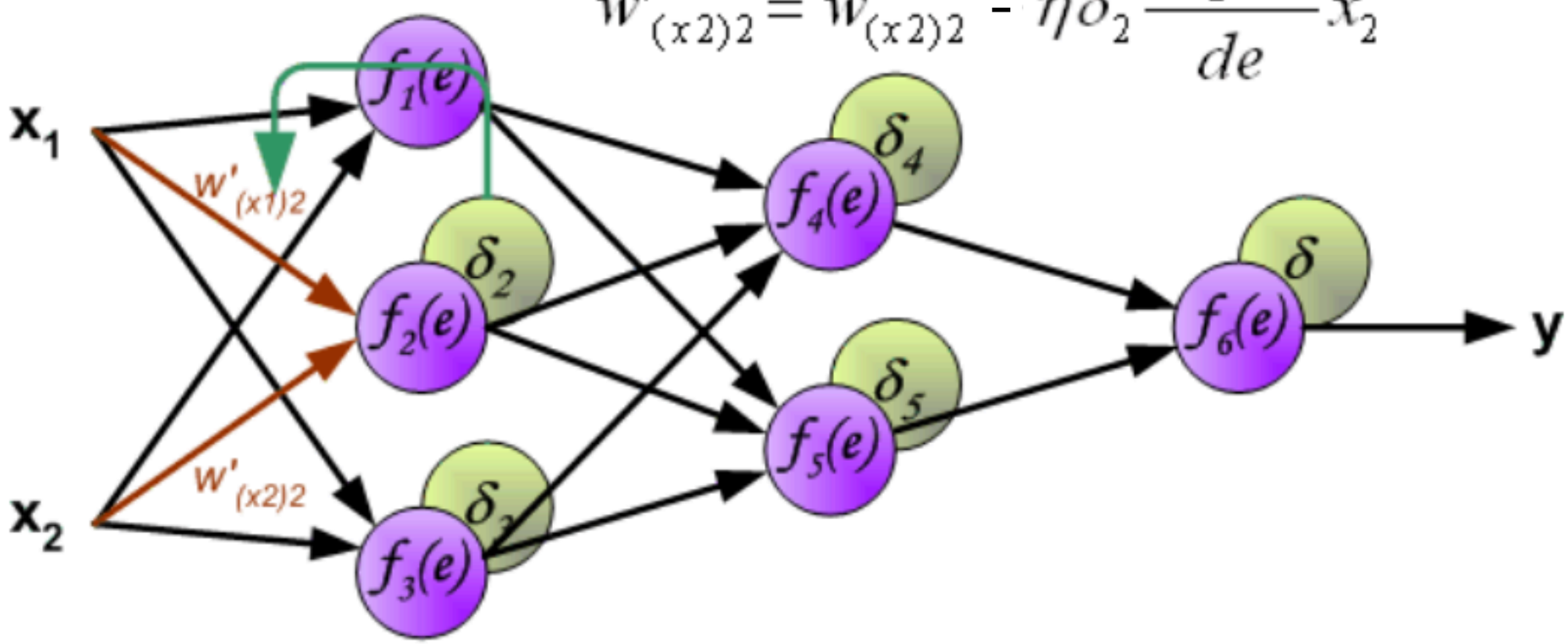


http://home.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html

Error Backpropagation

$$w'_{(x1)2} = w_{(x1)2} - \eta \delta_2 \frac{df_2(e)}{de} x_1$$

$$w'_{(x2)2} = w_{(x2)2} - \eta \delta_2 \frac{df_2(e)}{de} x_2$$

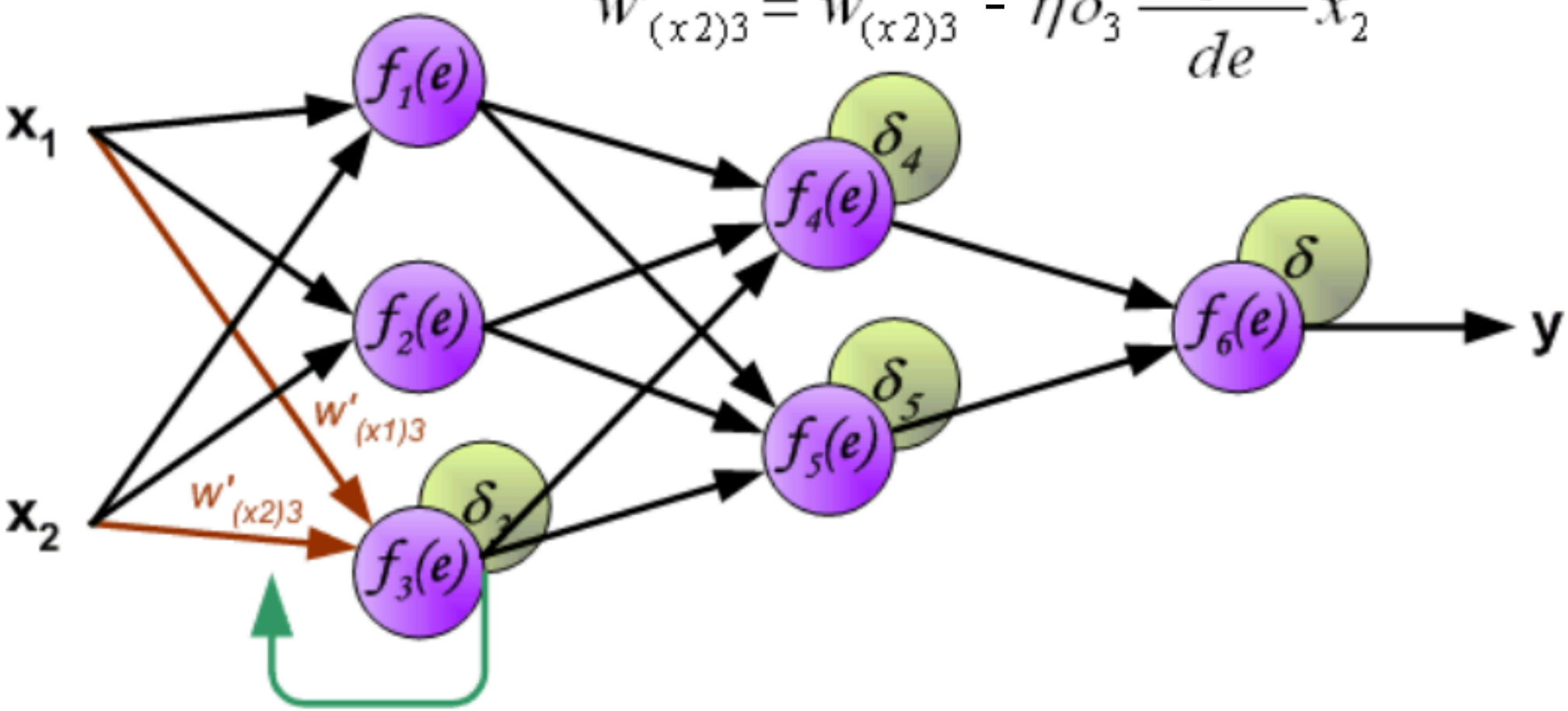


http://home.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html

Error Backpropagation

$$w'_{(x1)3} = w_{(x1)3} - \eta \delta_3 \frac{df_3(e)}{de} x_1$$

$$w'_{(x2)3} = w_{(x2)3} - \eta \delta_3 \frac{df_3(e)}{de} x_2$$



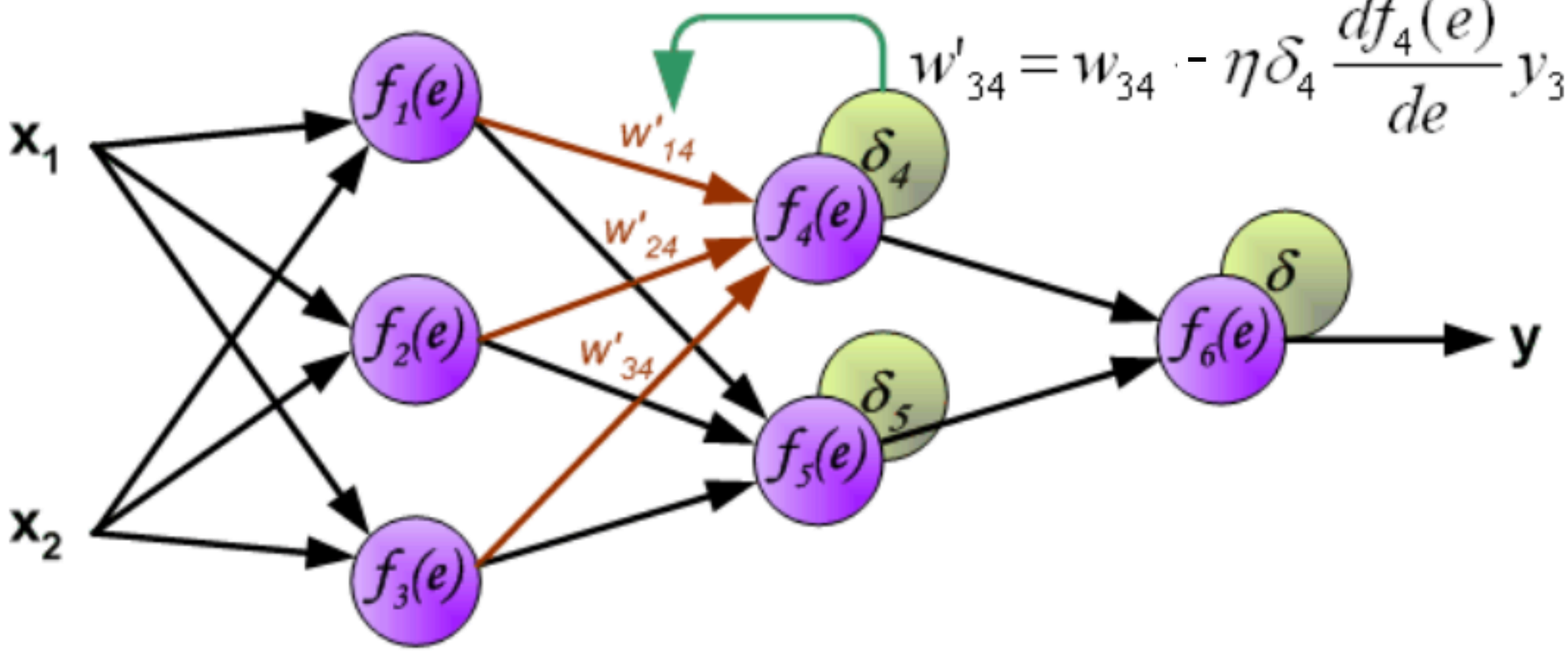
http://home.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html

Error Backpropagation

$$w'_{14} = w_{14} - \eta\delta_4 \frac{df_4(e)}{de} y_1$$

$$w'_{24} = w_{24} - \eta\delta_4 \frac{df_4(e)}{de} y_2$$

$$w'_{34} = w_{34} - \eta\delta_4 \frac{df_4(e)}{de} y_3$$



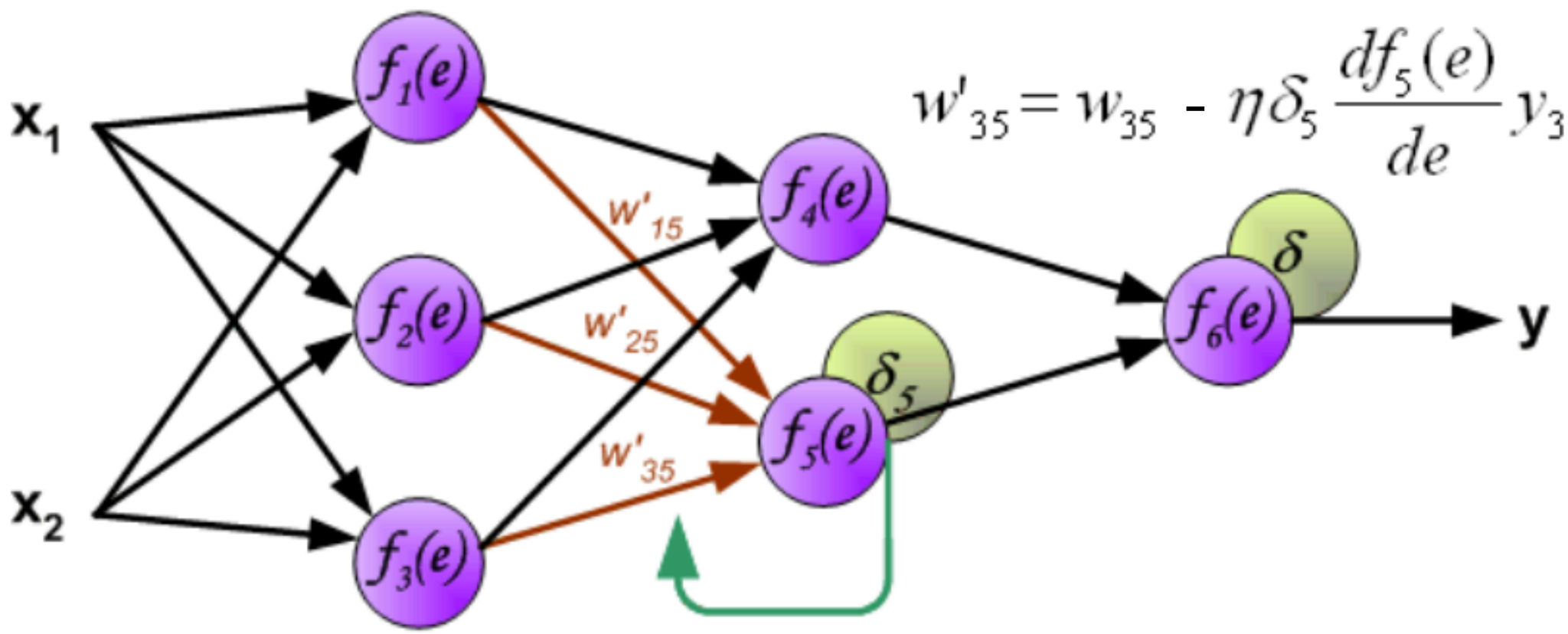
http://home.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html

Error Backpropagation

$$w'_{15} = w_{15} - \eta\delta_5 \frac{df_5(e)}{de} y_1$$

$$w'_{25} = w_{25} - \eta\delta_5 \frac{df_5(e)}{de} y_2$$

$$w'_{35} = w_{35} - \eta\delta_5 \frac{df_5(e)}{de} y_3$$

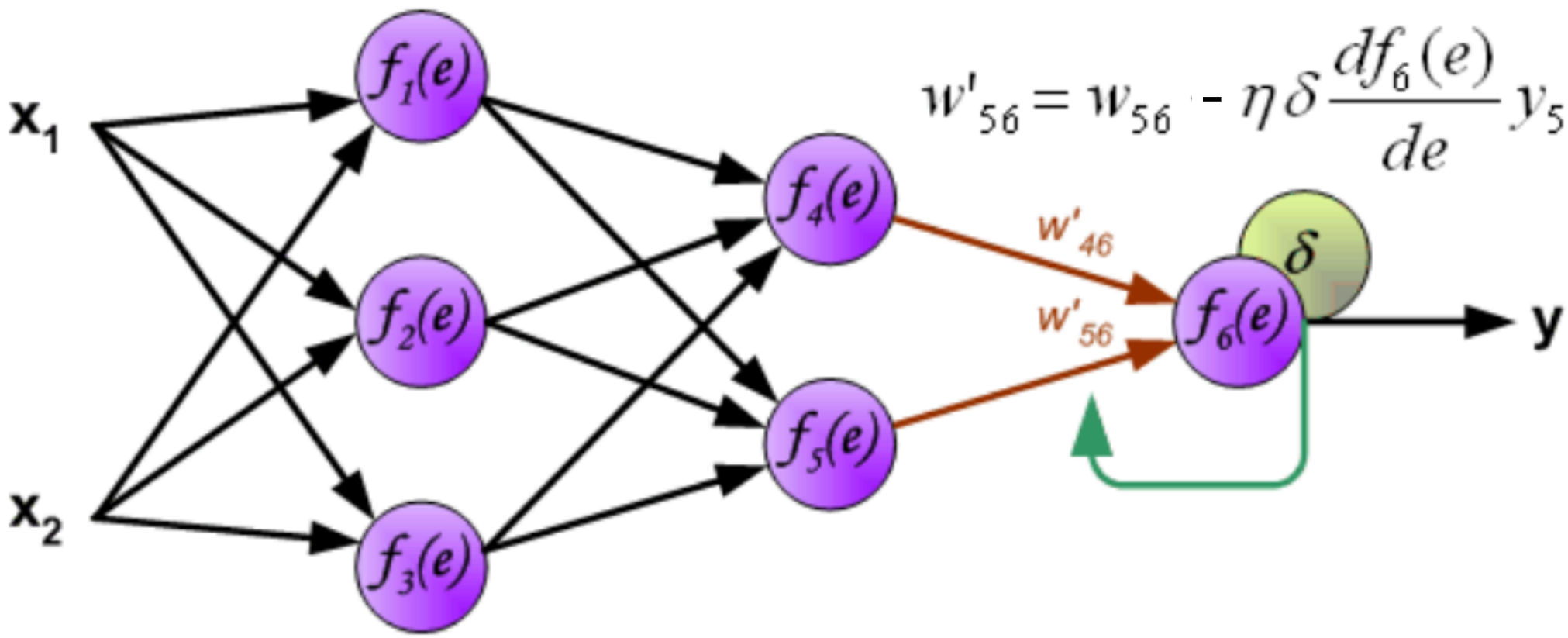


http://home.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html

Error Backpropagation

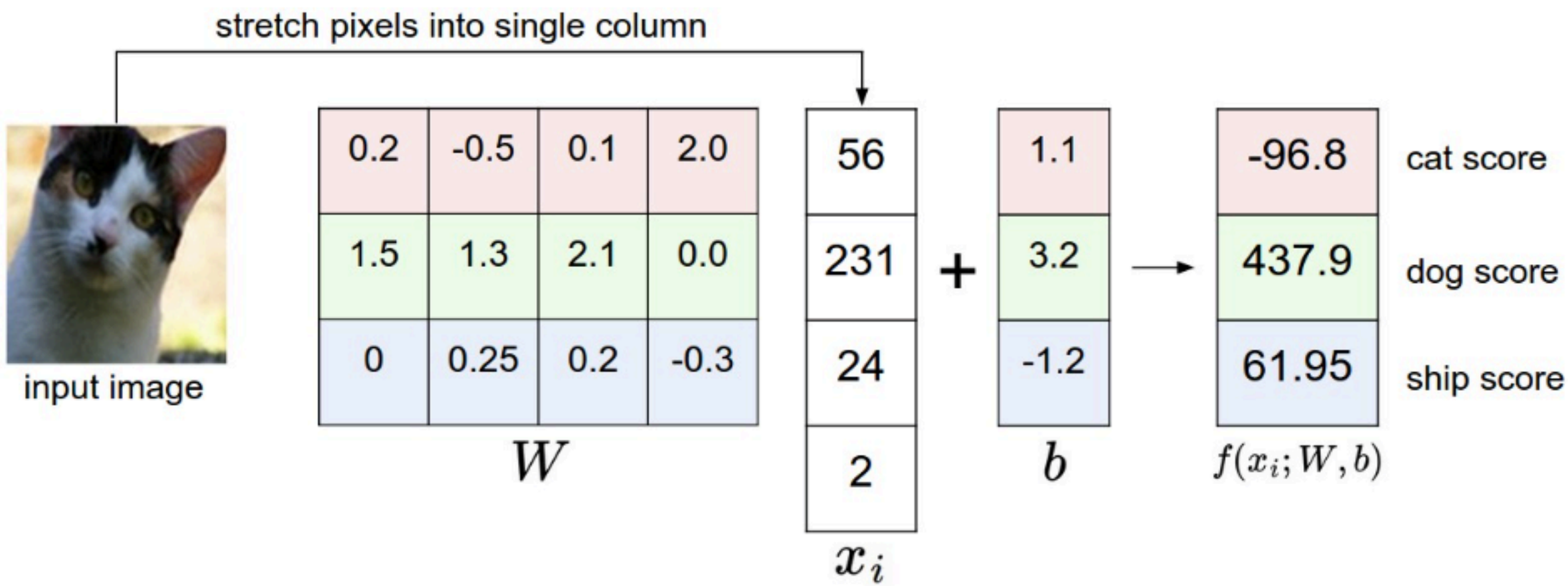
$$w'_{46} = w_{46} - \eta\delta\frac{df_6(e)}{de}y_4$$

$$w'_{56} = w_{56} - \eta\delta\frac{df_6(e)}{de}y_5$$



http://home.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html

Relevance of Error Backpropagation



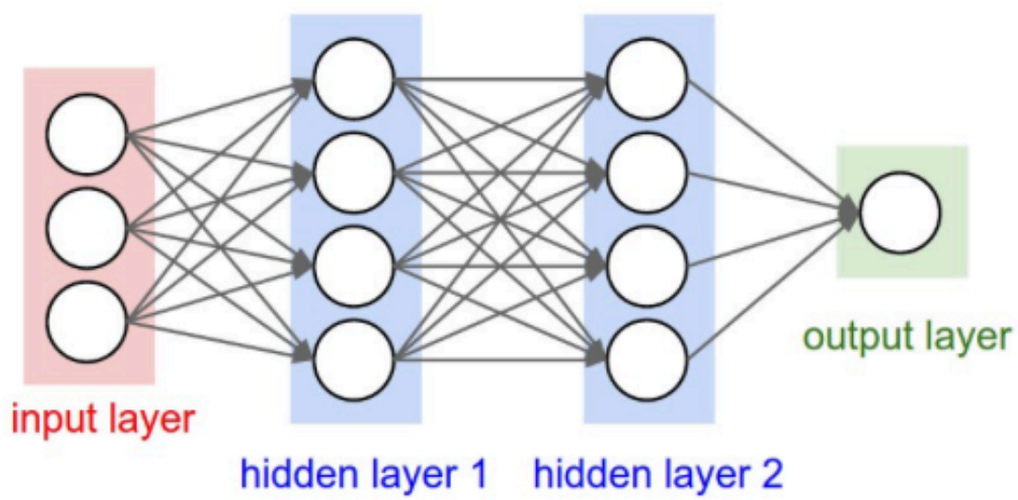
Template Matching

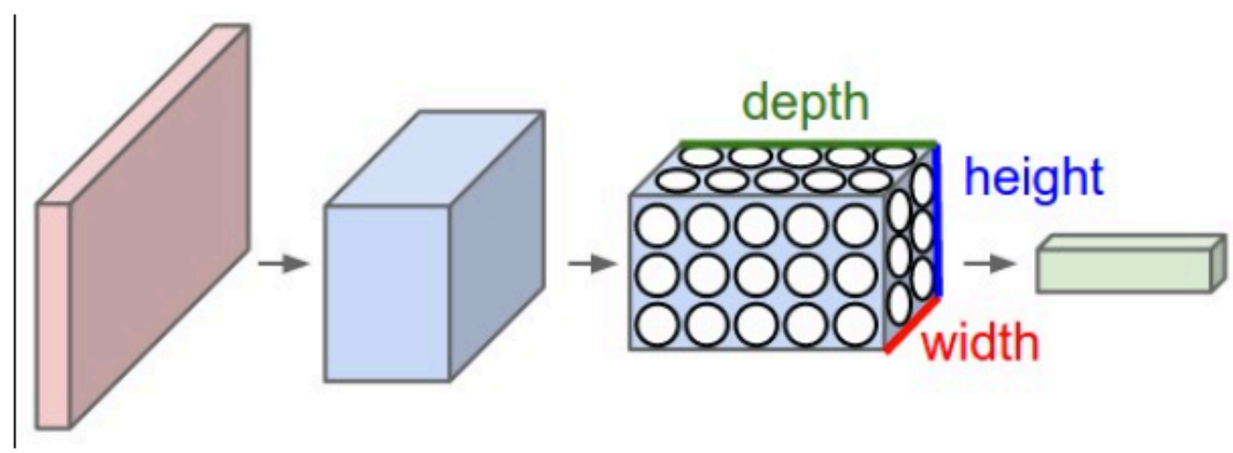
http://home.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html

Outline

- **Preliminaries (modeling workflow, gradient descent)**
- **Historical Tidbits**
- **ANN Feedforward Computation and Error Backpropagation**
- **The ANN and Deep Learning Architecture**
- Sample Implementations (digit recognition, iris/xor)

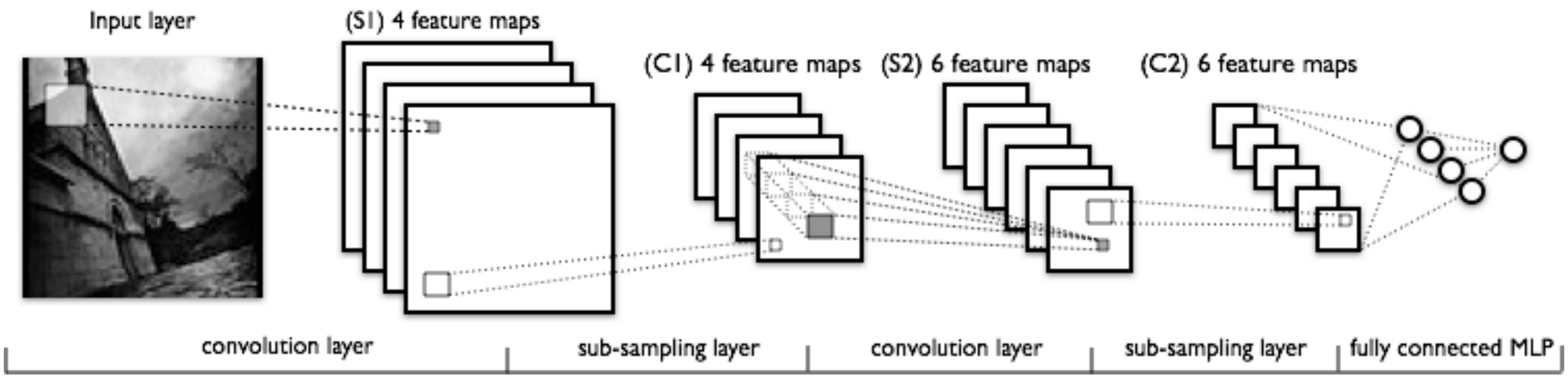
ANN vs Deep Learning





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

LeNET



- Convolutional Layer
- Pooling Layer
- Fully-connected Layer

http://deeplearning.net/tutorial/lenet.html

Convolution

- Common technique to change pixel intensities on the intensities of surrounding pixels
- A common usage is to create digital image effects such as blur, sharpen, edge detection, etc. used by many Photo editing softwares

Convolution

Original

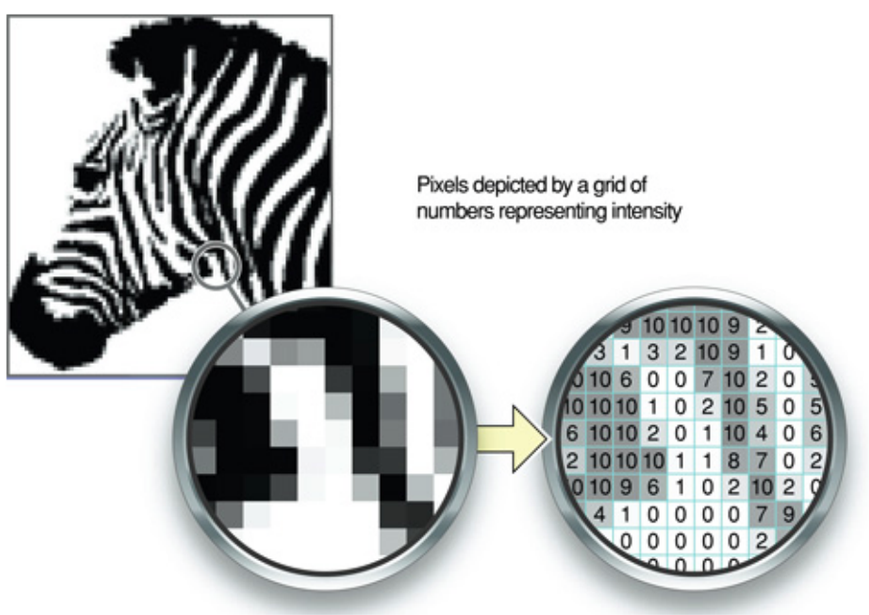
Emboss

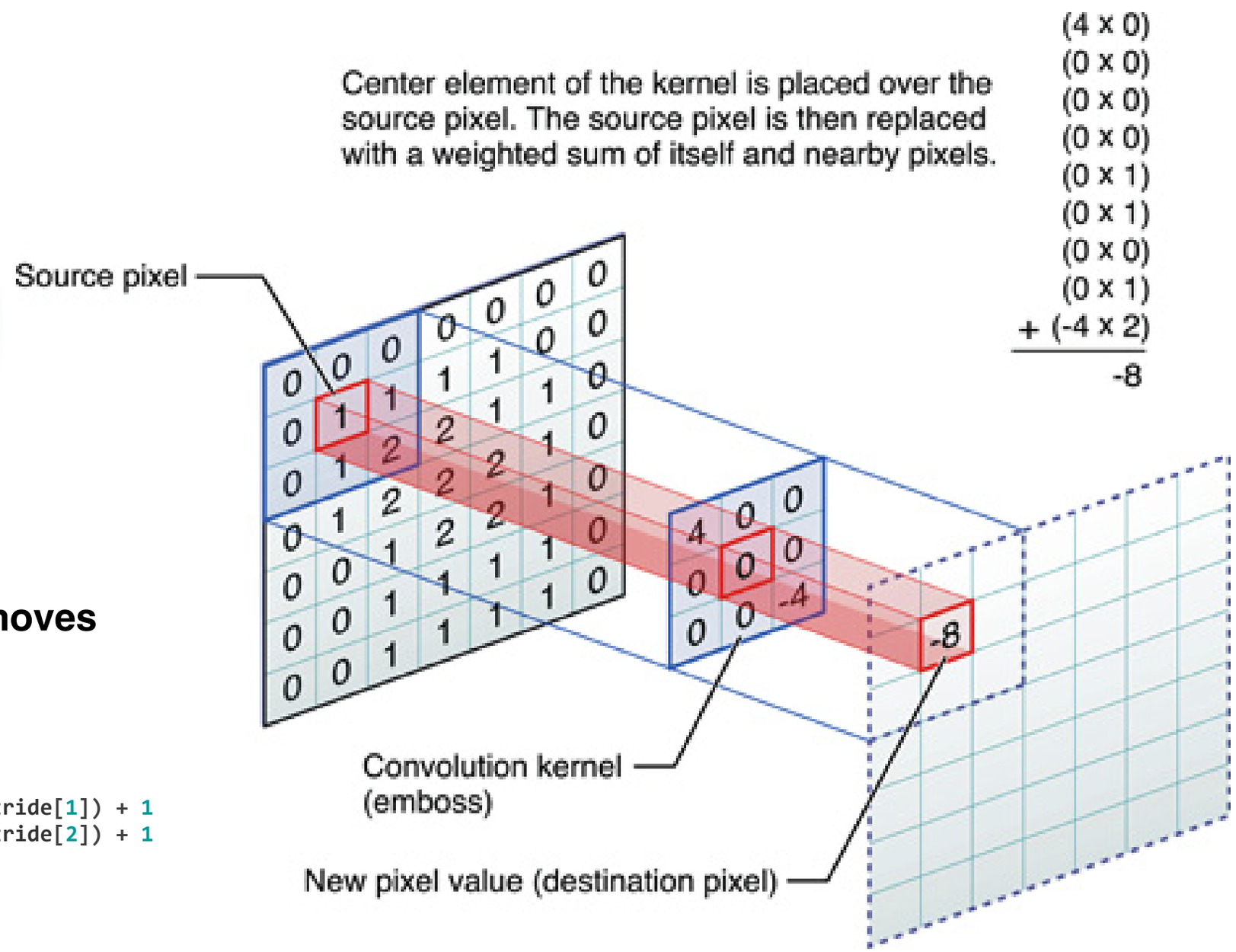
3x3 Kernel

-2	-2	0
-2	6	0
0	0	0

https://developer.apple.com/library/ios/documentation/Performance/Conceptual/vImage/ConvolutionOperations/ConvolutionOperations.html

Convolution



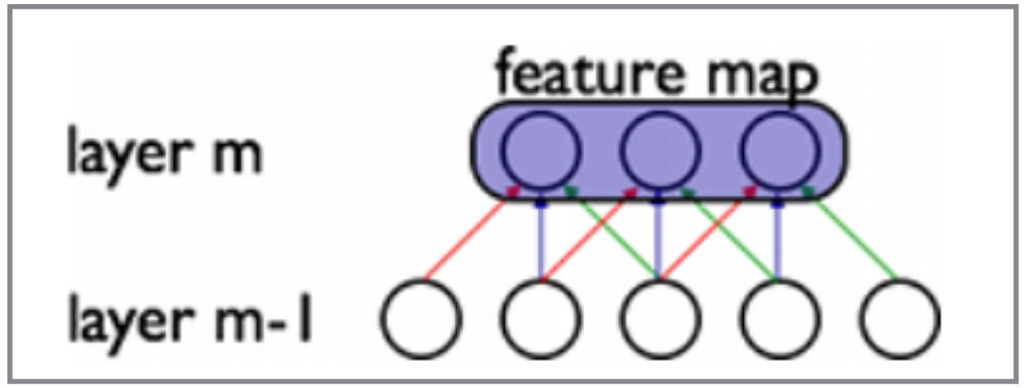


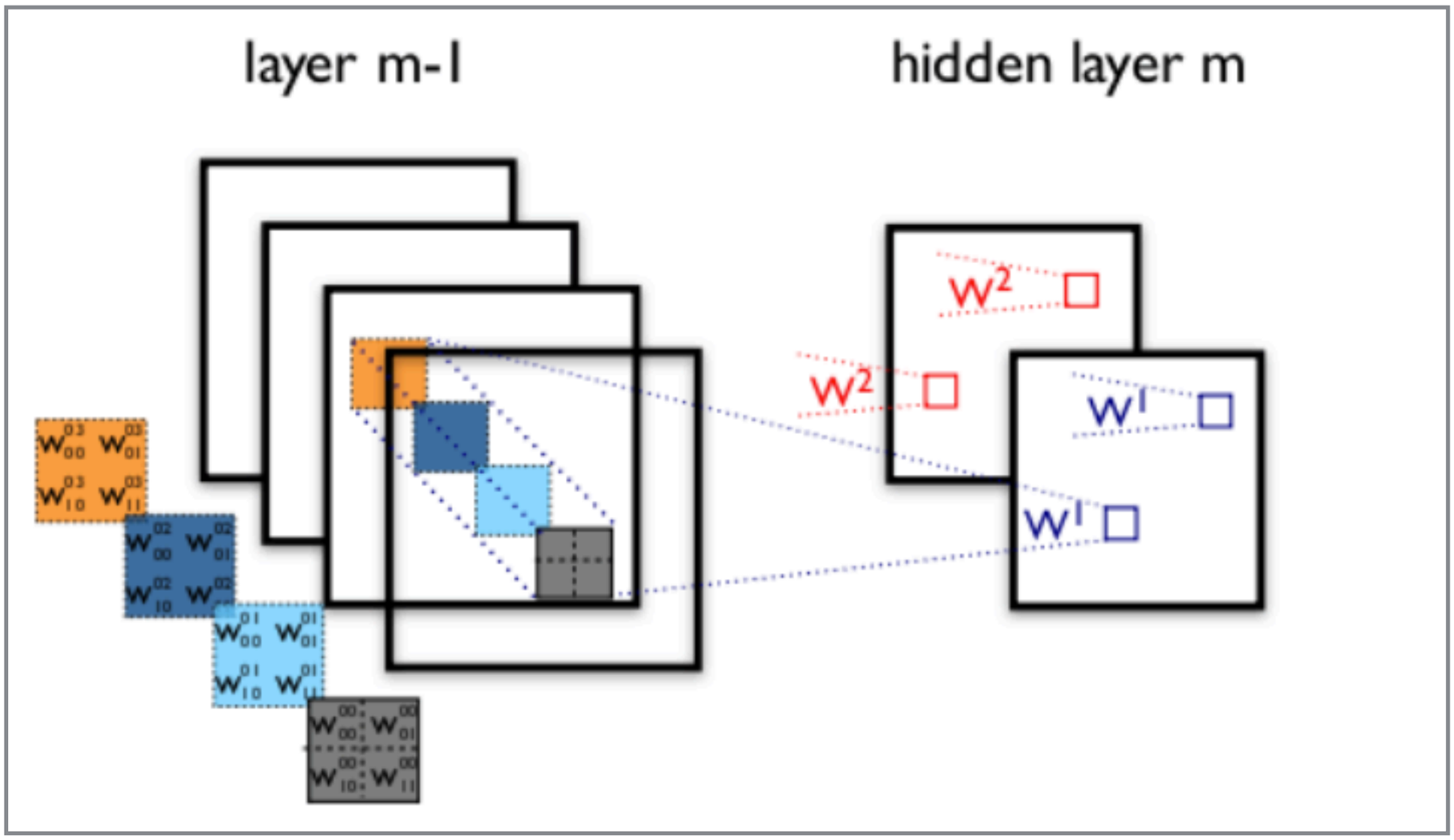
Kernel: size of receptive field
Stride: kernel step size as kernel moves
Pad: optional

```
width_out  = div(width_in  + 2*pad[1]-kernel[1], stride[1]) + 1
height_out = div(height_in + 2*pad[2]-kernel[2], stride[2]) + 1
```

https://developer.apple.com/library/ios/documentation/Performance/Conceptual/vImage/ConvolutionOperations/ConvolutionOperations.html

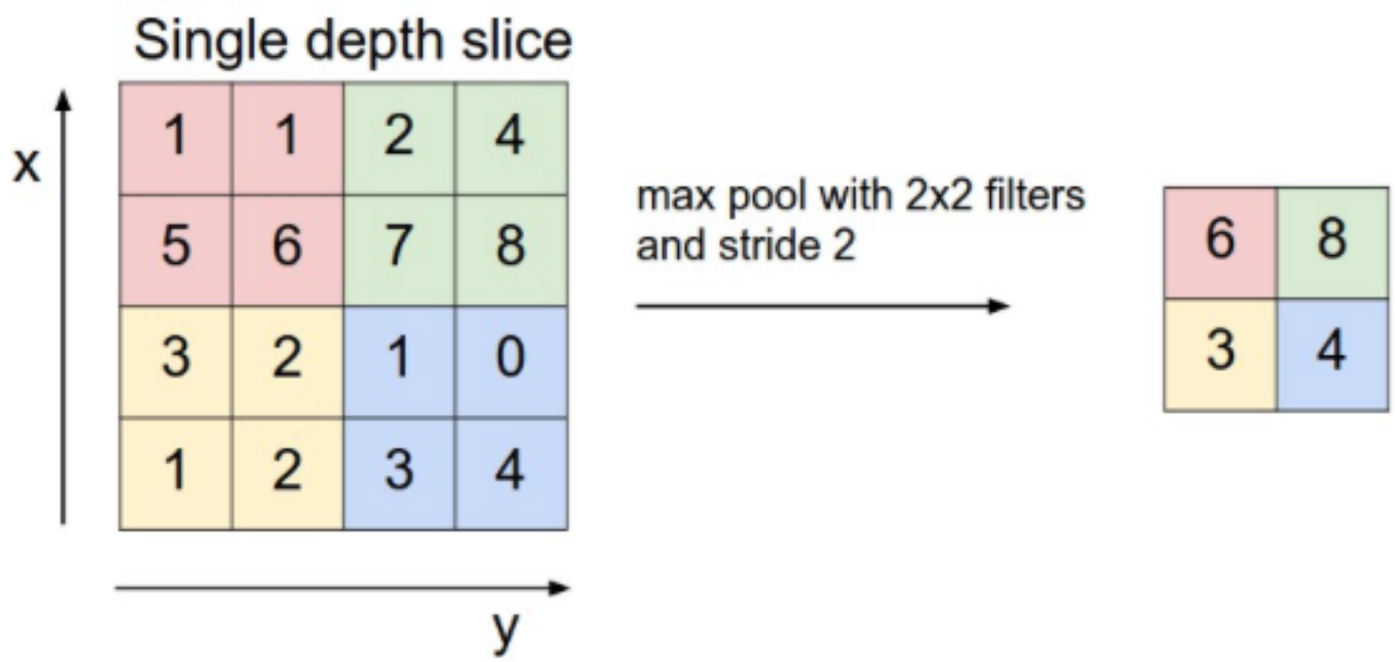
Convolution





http://deeplearning.net/tutorial/lenet.html#lenet

Pooling



- Max pooling
- L2-Norm pooling
- Average Pooling

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

Outline

- **Preliminaries (modeling workflow, gradient descent)**
- **Historical Tidbits**
- **ANN Feedforward Computation and Error Backpropagation**
- **The ANN and Deep Learning Architecture**
- **Sample Implementations (MNIS, CIFAR, IRIS)**

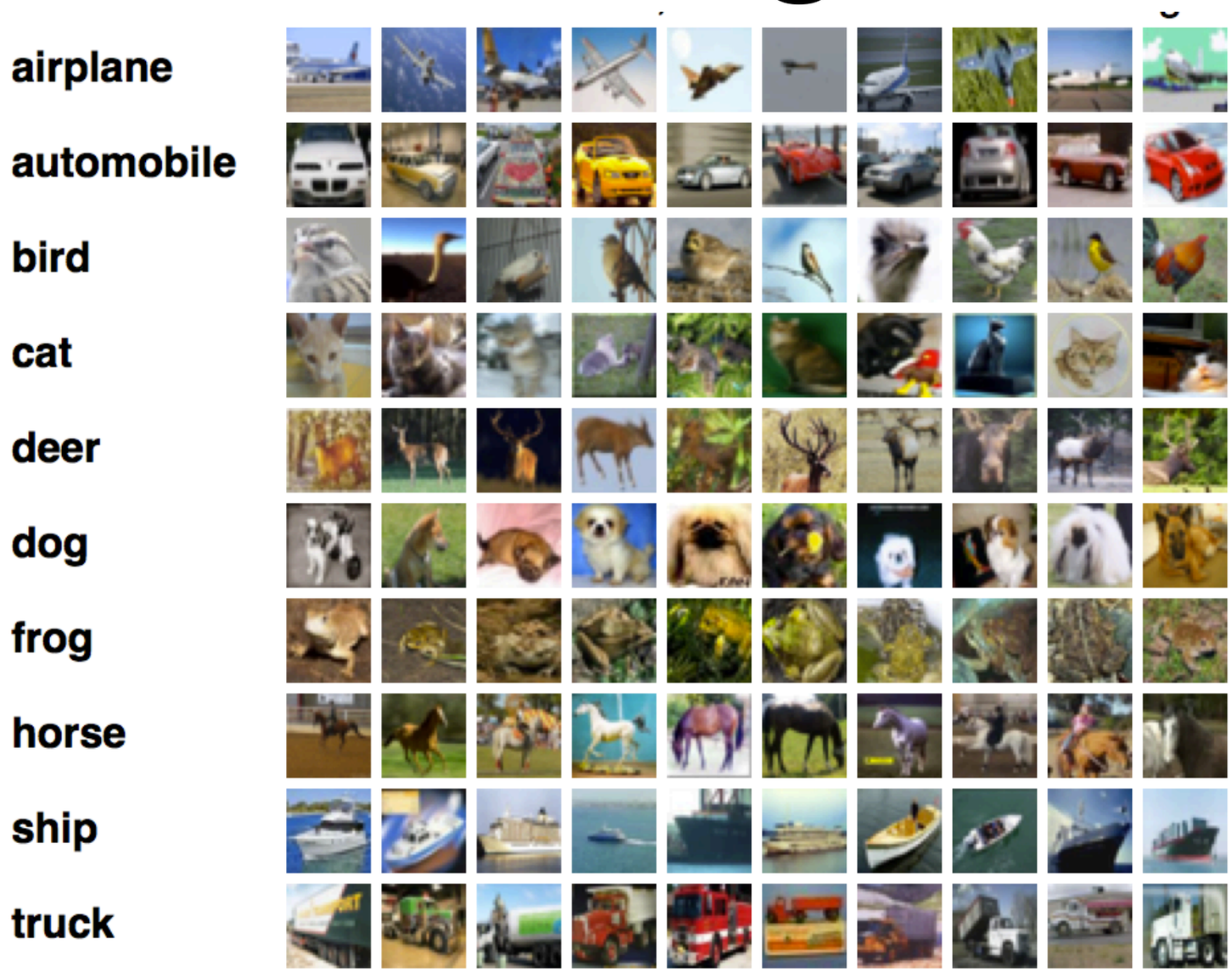
MNIST Using LeNET

- LeNet is a deep convolutional neural network
  - Lecun, Y.; Bottou, L.; Bengio, Y.; Haffner, P., Gradient-based learning applied to document recognition, Proceedings of the IEEE, vol.86, no.11, pp.2278-2324, Nov 1998.
- MNIST
  - Handwritten digit classification tasks
  - Training size: 60,000
  - Testing size: 10,000
  - image: 28(length) x 28(width) x 1(channel)

MNIST

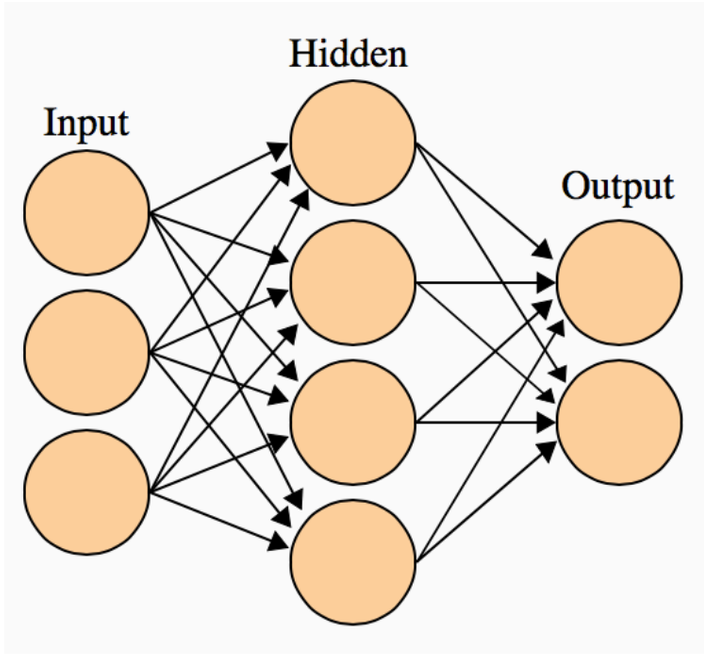
- **Image Size: 28 (length) x 28 (width) x 1 (channels)**
- **Training Size: 60,000**
- **Testing Size: 10,000**
- **No. of classes: 10**

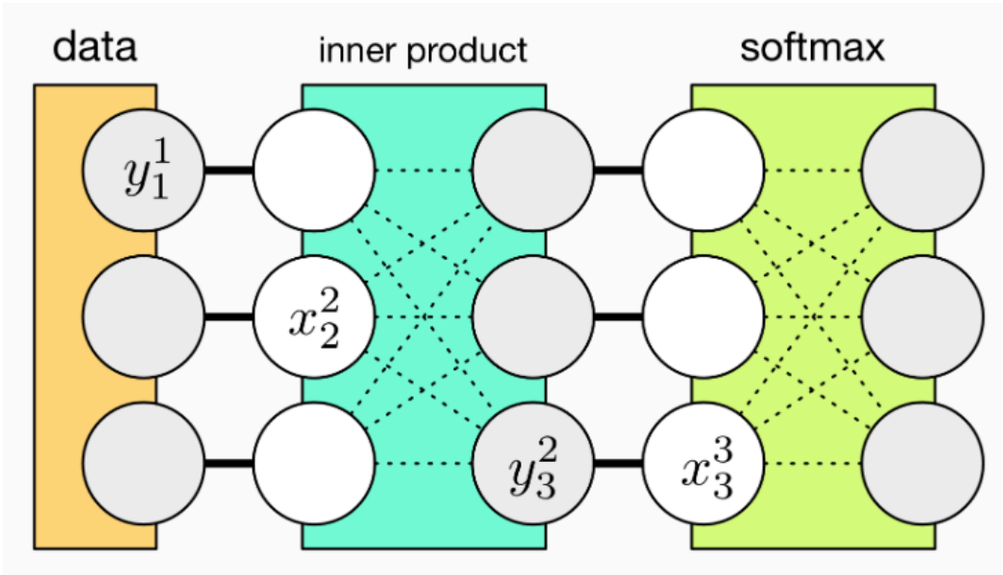
CIFAR-10 using ConvNet



- Image Size: 32 (length) x 32 (width) x 3 (channels)
- Training Size: 50,000
- Testing Size: 10,000
- No. of classes: 10

Julia's Mocha Network Representation

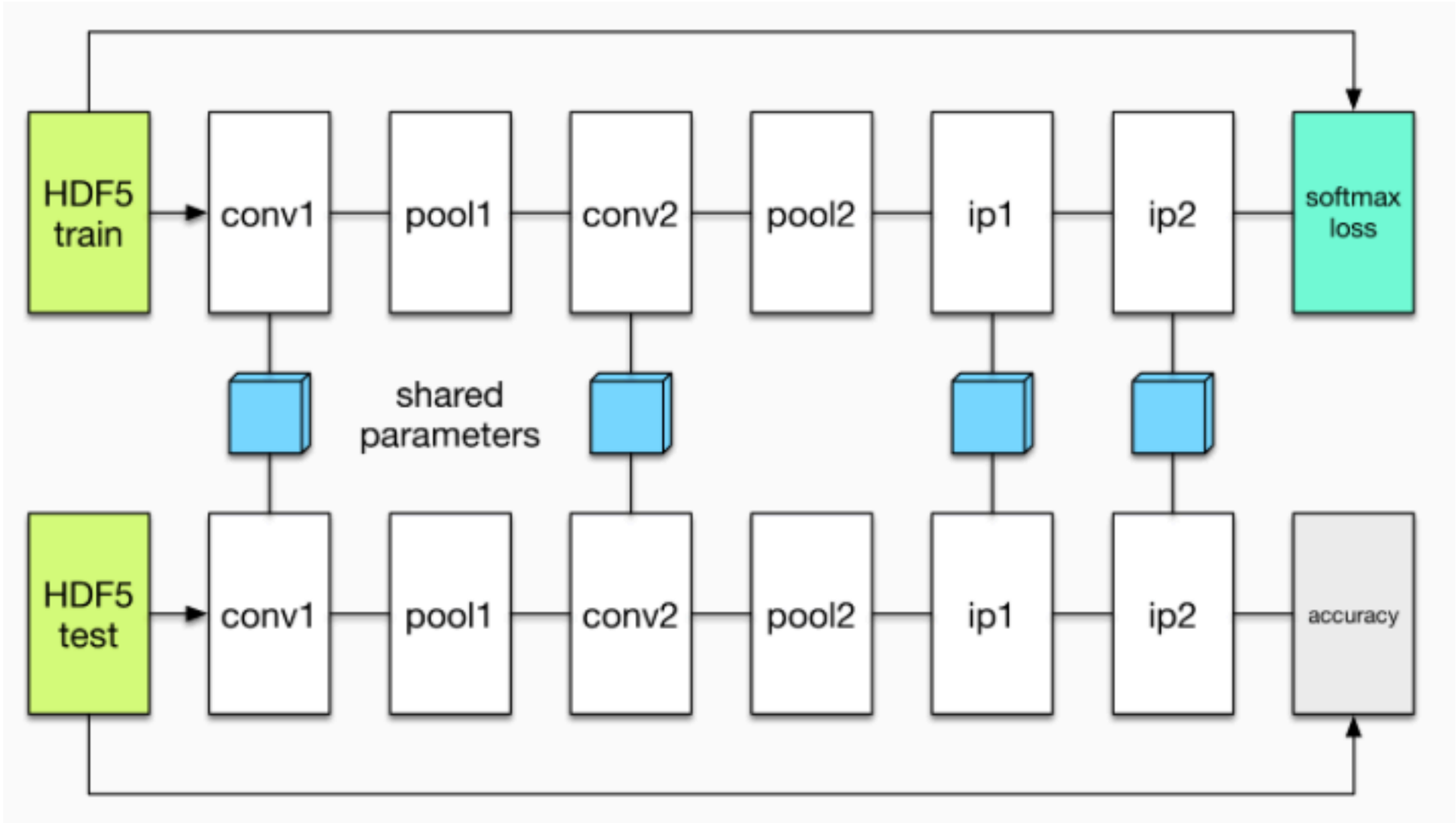




- **Data Layer:** Read data from source and feed them to top layers.
- **Computation Layers:** Take input stream from bottom layers, carry out computations and feed the computed results to top layers.
- **Loss Layer:** Take computed results (and ground truth labels) from bottom layers, compute a scalar loss value. Loss values from all the loss layers and regularizers in a net are added together to define the final loss function of the net. The loss function is used to train the net parameters in back propagation.
- **Statistics Layers:** Take input from bottom layers and compute useful statistics like classification accuracy. Statistics are accumulated throughout multiple iterations.
- **Utility Layers**
- **Other layers**

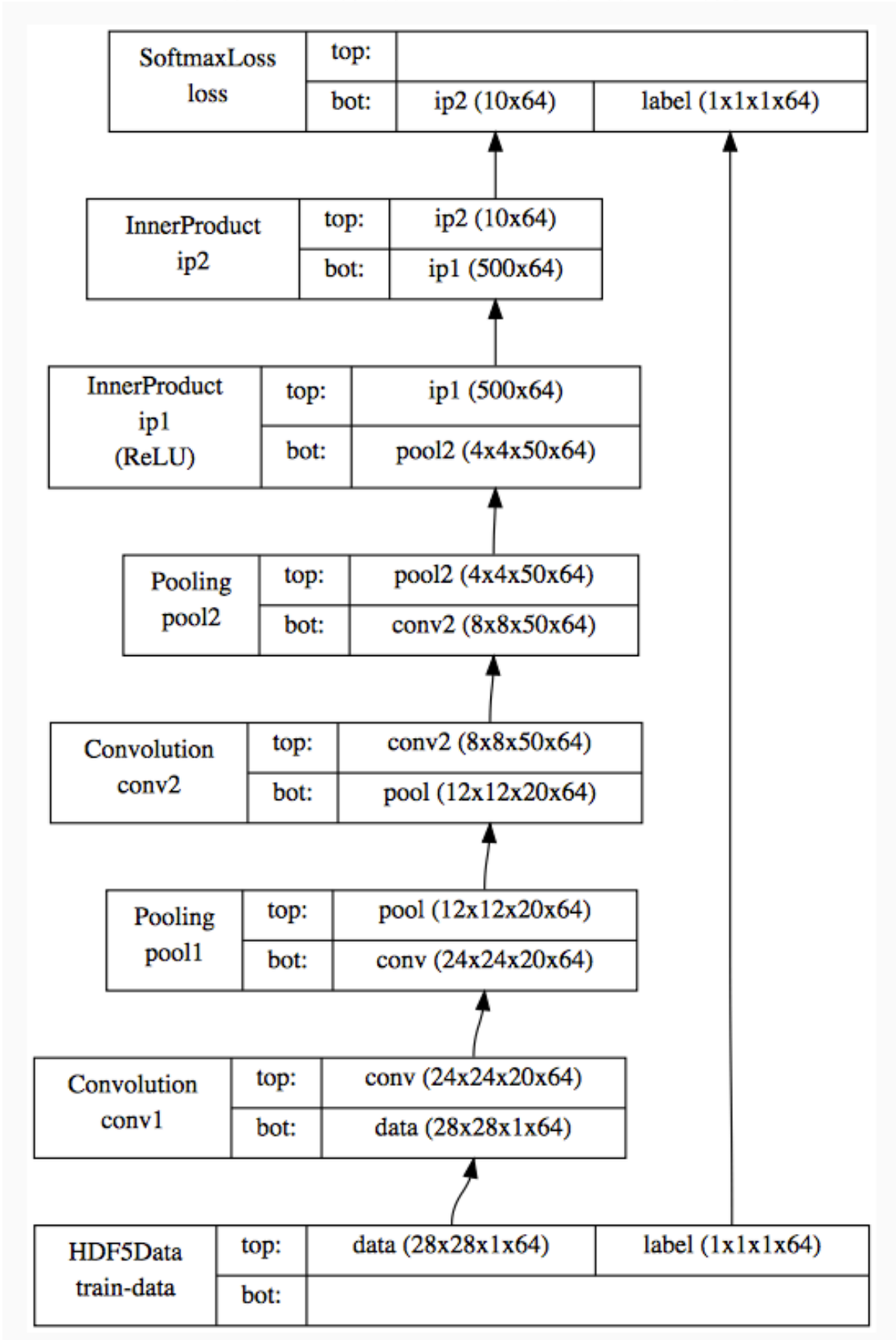
http://mochajl.readthedocs.org/en/latest/tutorial/mnist.html

LeNet+Mocha



http://mochajl.readthedocs.org/en/latest/tutorial/mnist.html

LeNet+Mocha



http://mochajl.readthedocs.org/en/latest/user-guide/network.html#debugging

Torch vs Julia

- explain 4D Tensor data representation
- run the code locally in CPU mode and in amazon cloud in GPU mode

	Fortran	Julia	Python	R	Matlab	Octave	Mathematica	JavaScript	Go	LuaJIT	Java
	gcc 4.8.2	0.3.7	2.7.9	3.1.3	R2014a	3.8.1	10.0	V8 3.14.5.9	go1.2.1	gsl-shell 2.3.1	1.7.0_75
fib	0.57	2.14	95.45	528.85	4258.12	9211.59	166.64	3.68	2.20	2.02	0.96
parse_int	4.67	1.57	20.48	54.30	1525.88	7568.38	17.70	2.29	3.78	6.09	5.43
quicksort	1.10	1.21	46.70	248.28	55.87	1532.54	48.47	2.91	1.09	2.00	1.65
mandel	0.87	0.87	18.83	58.97	60.09	393.91	6.12	1.86	1.17	0.71	0.68
pi_sum	0.83	1.00	21.07	14.45	1.28	260.28	1.27	2.15	1.23	1.00	1.00
rand_mat_stat	0.99	1.74	22.29	16.88	9.82	30.44	6.20	2.81	8.23	3.71	4.01
rand_mat_mul	4.05	1.09	1.08	1.63	1.12	1.06	1.13	14.58	8.45	1.23	2.35

http://julialang.org/

Demo

- **MNIST**
  - **CPU vs GPU**
- **CIFAR-10**
  - **CPU vs GPU**
  - **Torch vs Julia**

MNIST.jl

```
#ENV["MOCHA_USE_CUDA"] = "true"
#ENV["MOCHA_USE_NATIVE_EXT"] = "true"
#ENV["OMP_NUM_THREADS"] = 1
#blas_set_num_threads(1)
using Mocha
srand(12345678)

data_layer  = HDF5DataLayer(name="train-data", source="data/train.txt", batch_size=64, shuffle=@windows ? false : true)
conv_layer  = ConvolutionLayer(name="conv1", n_filter=20, kernel=(5,5), bottoms=[:data], tops=[:conv])
pool_layer  = PoolingLayer(name="pool1", kernel=(2,2), stride=(2,2), bottoms=[:conv], tops=[:pool])
conv2_layer = ConvolutionLayer(name="conv2", n_filter=50, kernel=(5,5), bottoms=[:pool], tops=[:conv2])
pool2_layer = PoolingLayer(name="pool2", kernel=(2,2), stride=(2,2), bottoms=[:conv2], tops=[:pool2])
fc1_layer   = InnerProductLayer(name="ip1", output_dim=500, neuron=Neurons.ReLU(), bottoms=[:pool2], tops=[:ip1])
fc2_layer   = InnerProductLayer(name="ip2", output_dim=10, bottoms=[:ip1], tops=[:ip2])
loss_layer  = SoftmaxLossLayer(name="loss", bottoms=[:ip2,:label])

#backend = GPUBackend()
backend = CPUBackend()
init(backend)

common_layers = [conv_layer, pool_layer, conv2_layer, pool2_layer, fc1_layer, fc2_layer]
net = Net("MNIST-train", backend, [data_layer, common_layers..., loss_layer])
exp_dir = "snapshots"
params = SolverParameters(max_iter=10000, regu_coef=0.0005,
   mom_policy=MomPolicy.Fixed(0.9),
   lr_policy=LRPolicy.Inv(0.01, 0.0001, 0.75),
   load_from=exp_dir)
solver = SGD(params)
setup_coffee_lounge(solver, save_into="$exp_dir/statistics.jld", every_n_iter=1000)
# report training progress every 100 iterations
add_coffee_break(solver, TrainingSummary(), every_n_iter=100)
# save snapshots every 5000 iterations
add_coffee_break(solver, Snapshot(exp_dir), every_n_iter=5000)
# show performance on test data every 1000 iterations
data_layer_test = HDF5DataLayer(name="test-data", source="data/test.txt", batch_size=100)
acc_layer = AccuracyLayer(name="test-accuracy", bottoms=[:ip2, :label])
test_net = Net("MNIST-test", backend, [data_layer_test, common_layers..., acc_layer])
add_coffee_break(solver, ValidationPerformance(test_net), every_n_iter=1000)

solve(solver, net)

destroy(net)
destroy(test_net)
shutdown(backend)
```

http://mochajl.readthedocs.org/en/latest/tutorial/mnist.html

CIFAR10.jl

```
#ENV["MOCHA_USE_CUDA"] = "true"
#ENV["MOCHA_USE_NATIVE_EXT"] = "true"
#ENV["OMP_NUM_THREADS"] = 1
#blas_set_num_threads(1)

using Mocha
data_tr_layer = HDF5DataLayer(name="data-train", source="data/train.txt", batch_size=100, shuffle=@windows ? false : true)
data_tt_layer = HDF5DataLayer(name="data-test", source="data/test.txt", batch_size=100)
conv1_layer = ConvolutionLayer(name="conv1", n_filter=32, kernel=(5,5), pad=(2,2), stride=(1,1), filter_init=GaussianInitializer(std=0.0001), bottoms=[:data], tops=[:conv1])
pool1_layer = PoolingLayer(name="pool1", kernel=(3,3), stride=(2,2), neuron=Neurons.ReLU(), bottoms=[:conv1], tops=[:pool1])
norm1_layer = LRNLayer(name="norm1", kernel=3, scale=5e-5, power=0.75, mode=LRNMode.WithinChannel(), bottoms=[:pool1], tops=[:norm1])
conv2_layer = ConvolutionLayer(name="conv2", n_filter=32, kernel=(5,5), pad=(2,2),
   stride=(1,1), filter_init=GaussianInitializer(std=0.01), bottoms=[:norm1], tops=[:conv2], neuron=Neurons.ReLU())
pool2_layer = PoolingLayer(name="pool2", kernel=(3,3), stride=(2,2), pooling=Pooling.Mean(), bottoms=[:conv2], tops=[:pool2])
norm2_layer = LRNLayer(name="norm2", kernel=3, scale=5e-5, power=0.75, mode=LRNMode.WithinChannel(), bottoms=[:pool2], tops=[:norm2])
conv3_layer = ConvolutionLayer(name="conv3", n_filter=64, kernel=(5,5), pad=(2,2),
    stride=(1,1), filter_init=GaussianInitializer(std=0.01), bottoms=[:norm2], tops=[:conv3], neuron=Neurons.ReLU())
pool3_layer = PoolingLayer(name="pool3", kernel=(3,3), stride=(2,2), pooling=Pooling.Mean(), bottoms=[:conv3], tops=[:pool3])
ip1_layer   = InnerProductLayer(name="ip1", output_dim=10, weight_init=GaussianInitializer(std=0.01), weight_regu=L2Regu(250), bottoms=[:pool3], tops=[:ip1])
loss_layer  = SoftmaxLossLayer(name="softmax", bottoms=[:ip1, :label])
acc_layer   = AccuracyLayer(name="accuracy", bottoms=[:ip1, :label])
common_layers = [conv1_layer, pool1_layer, norm1_layer, conv2_layer, pool2_layer, norm2_layer, conv3_layer, pool3_layer, ip1_layer]

#backend = GPUBackend()
backend = CPUBackend()
init(backend)

net = Net("CIFAR10-train", backend, [data_tr_layer, common_layers..., loss_layer])
lr_policy = LRPolicy.Staged(
  (60000, LRPolicy.Fixed(0.001)),
  (5000, LRPolicy.Fixed(0.0001)),
  (5000, LRPolicy.Fixed(0.00001)),
)
solver_params = SolverParameters(max_iter=70000, regu_coef=0.004, mom_policy=MomPolicy.Fixed(0.9), lr_policy=lr_policy, load_from="snapshots")
solver = SGD(solver_params)
# report training progress every 200 iterations
add_coffee_break(solver, TrainingSummary(), every_n_iter=200)
# save snapshots every 5000 iterations
add_coffee_break(solver, Snapshot("snapshots"), every_n_iter=5000)
# show performance on test data every 1000 iterations
test_net = Net("CIFAR10-test", backend, [data_tt_layer, common_layers..., acc_layer])
add_coffee_break(solver, ValidationPerformance(test_net), every_n_iter=1000)

solve(solver, net)

destroy(net)
destroy(test_net)
shutdown(backend)
```

http://mochajl.readthedocs.org/en/latest/tutorial/cifar10.html

CIFAR10.lua

```
require 'nn'
require 'itorch'

net = nn.Sequential()
net:add(nn.SpatialConvolution(3, 6, 5, 5)) -- 1 input image channel, 6 output channels, 5x5 convolution kernel
net:add(nn.SpatialMaxPooling(2,2,2,2))     -- A max-pooling operation that looks at 2x2 windows and finds the max.
net:add(nn.SpatialConvolution(6, 16, 5, 5))
net:add(nn.SpatialMaxPooling(2,2,2,2))
net:add(nn.View(16*5*5))                    -- reshapes from a 3D tensor of 16x5x5 into 1D tensor of 16*5*5
net:add(nn.Linear(16*5*5, 120))             -- fully connected layer (matrix multiplication between input and weights)
net:add(nn.Linear(120, 84))
net:add(nn.Linear(84, 10))                  -- 10 is the number of outputs of the network (in this case, 10 digits)
net:add(nn.LogSoftMax())                    -- converts the output to a log-probability. Useful for classification problems
criterion = nn.ClassNLLCriterion() -- a negative log-likelihood criterion for multi-class classification

print('Lenet5\n' .. net:__tostring());

--os.execute('wget -c https://s3.amazonaws.com/torch7/data/cifar10torchsmall.zip')
--os.execute('unzip cifar10torchsmall.zip')

trainset = torch.load('cifar10-train.t7')
testset = torch.load('cifar10-test.t7')
classes = {'airplane', 'automobile', 'bird', 'cat','deer', 'dog', 'frog', 'horse', 'ship', 'truck'}

-- ignore setmetatable for now, it is a feature beyond the scope of this tutorial. It sets the index operator.
setmetatable(trainset,
    {__index = function(t, i)
                return {t.data[i], t.label[i]}
            end}
);
trainset.data = trainset.data:double() -- convert the data from a ByteTensor to a DoubleTensor.

function trainset:size()
    return self.data:size(1)
end

mean = {} -- store the mean, to normalize the test set in the future
stdv  = {} -- store the standard-deviation for the future
for i=1,3 do -- over each image channel
    mean[i] = trainset.data[{ {}, {i}, {}, {}  }]:mean() -- mean estimation
    print('Channel ' .. i .. ', Mean: ' .. mean[i])
    trainset.data[{ {}, {i}, {}, {}  }]:add(-mean[i]) -- mean subtraction
    stdv[i] = trainset.data[{ {}, {i}, {}, {}  }]:std() -- std estimation
    print('Channel ' .. i .. ', Standard Deviation: ' .. stdv[i])
    trainset.data[{ {}, {i}, {}, {}  }]:div(stdv[i]) -- std scaling
end

trainer = nn.StochasticGradient(net, criterion)
trainer.learningRate = 0.001
trainer.maxIteration = 5 -- just do 5 epochs of training.
trainer:train(trainset)
```

```
testset.data = testset.data:double()   -- convert from Byte tensor to Double tensor
for i=1,3 do -- over each image channel
    testset.data[{ {}, {i}, {}, {}  }]:add(-mean[i]) -- mean subtraction
    testset.data[{ {}, {i}, {}, {}  }]:div(stdv[i]) -- std scaling
end
predicted = net:forward(testset.data[101])
print(predicted:exp())

for i=1,predicted:size(1) do
    print(classes[i], torch.exp(predicted[i]))
end
correct = 0
for i=1,10000 do
    local groundtruth = testset.label[i]
    local prediction = net:forward(testset.data[i])
    local confidences, indices = torch.sort(prediction, true)  -- true means sort in
descending order
    if groundtruth == indices[1] then
        correct = correct + 1
    end
end

print(correct, 100*correct/10000 .. ' % ')
class_performance = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
total_performance = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
for i=1,10000 do
    local groundtruth = testset.label[i]
    local prediction = net:forward(testset.data[i])
    local confidences, indices = torch.sort(prediction, true)  -- true means sort in
descending order
    total_performance[groundtruth]=total_performance[groundtruth]+1
    if groundtruth == indices[1] then
        class_performance[groundtruth] = class_performance[groundtruth] + 1
    end
end

for i=1,#classes do
    print(classes[i], 100*class_performance[i]/total_performance[i] .. ' %')
end

-- net = net:cuda()
-- criterion = criterion:cuda()
-- trainset.data = trainset.data:cuda()
-- trainer = nn.StochasticGradient(net, criterion)
-- trainer.learningRate = 0.001
-- trainer.maxIteration = 5 -- just do 5 epochs of training.
-- trainer:train(trainset)
```

http://code.madbits.com/wiki/doku.php?id=tutorial_supervised
http://torch.ch/blog/2015/07/30/cifar.html

THANK YOU!

"Imagination is the only weapon in the war against reality."
— *Lewis Carroll, Alice in Wonderland*