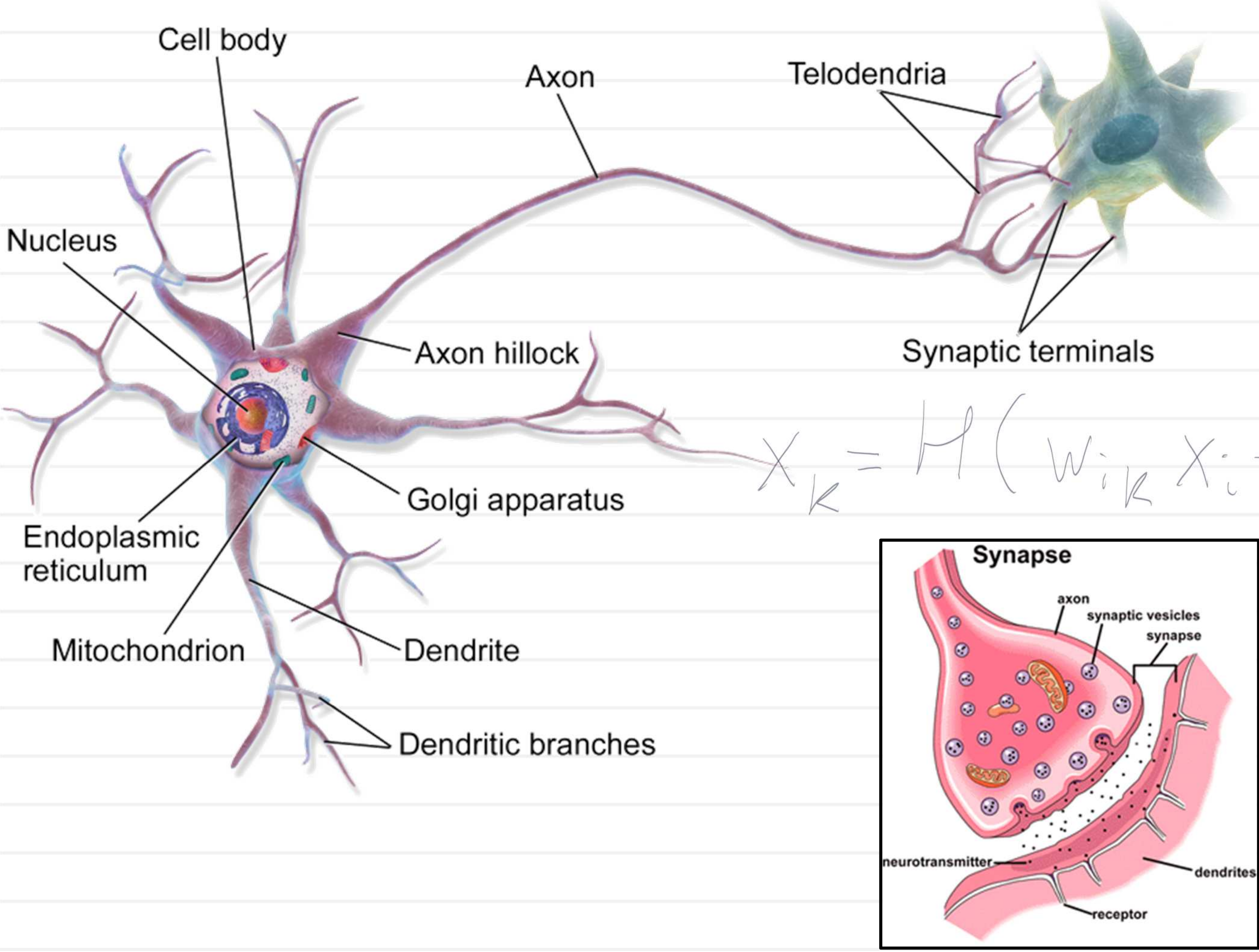
Lecture 3: Deep feed-forward neural networks

Neuron model



$$x_k = H(w_{ik} x_i - \tau)$$

Brain statistics

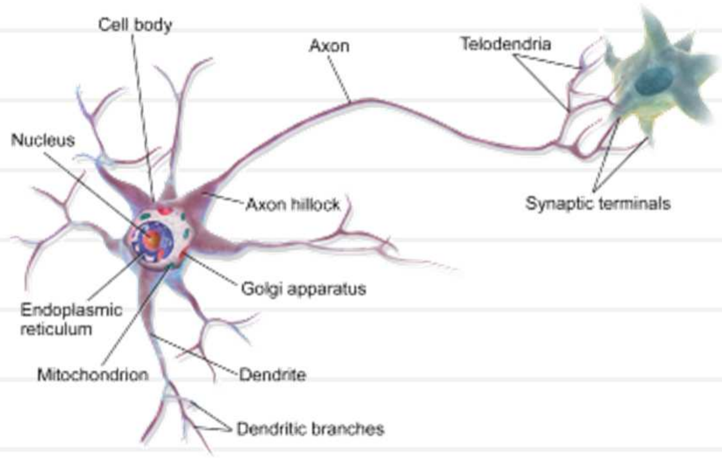
Human brain:

- 100 billion neurons
- average neuron is connected to 1000-10000 other neurons
- 100 trillion synapses
- 10-25% is in visual cortex

Perceptron

[Rosenblatt 1957]: an "artificial neuron"

$y = H(w^T x)$



loop over examples

$y = H(w^T x_i);$

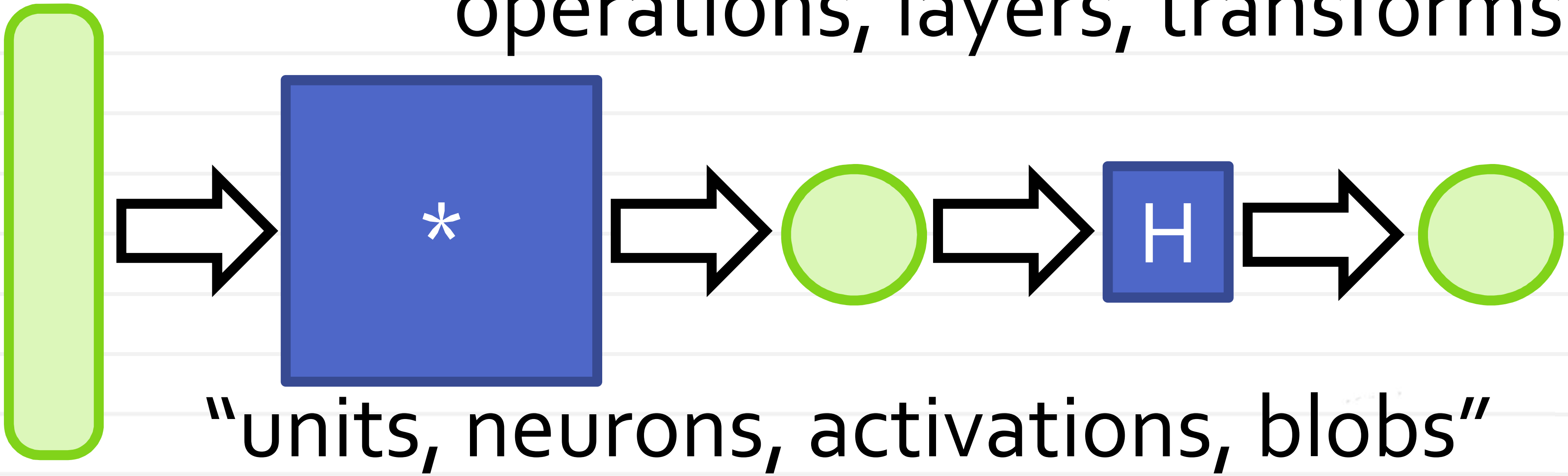
$w = w + 1/2\ x_i * (y_i - y);$

end

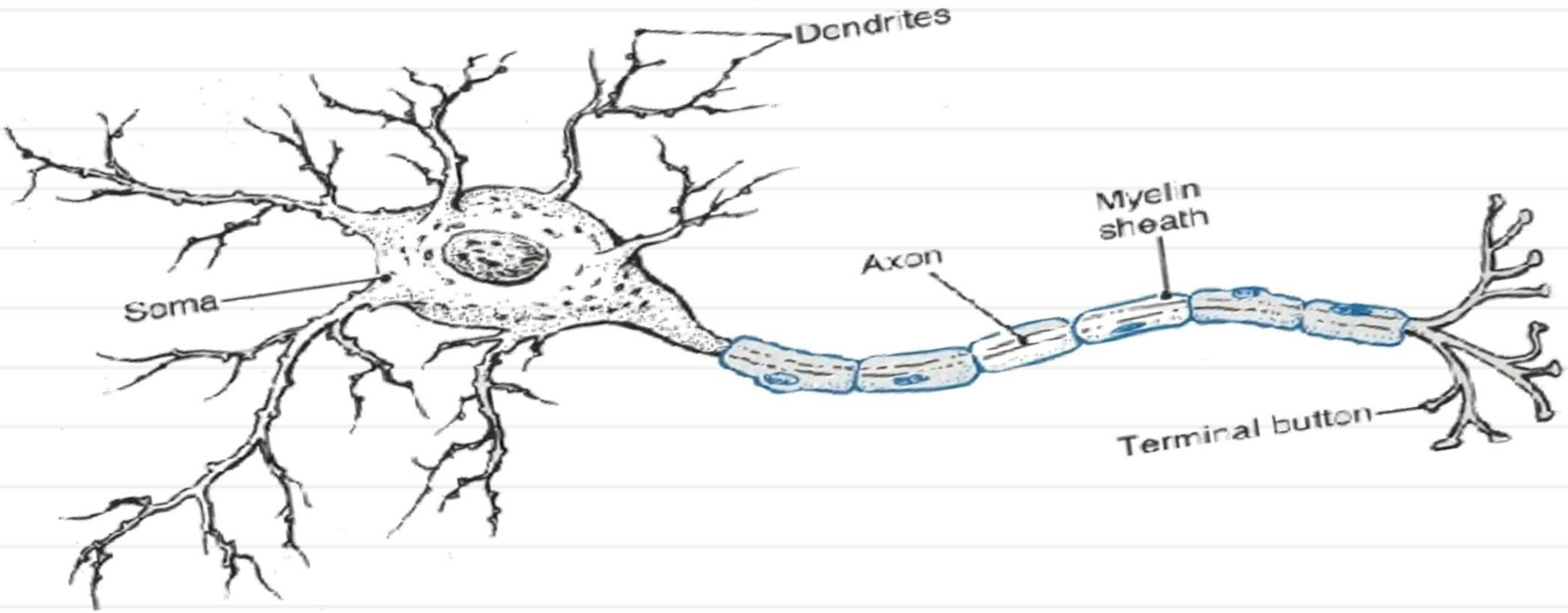
Converges to linear separator of the training data if it exists.

Terminology and graphical language

“operations, layers, transforms”



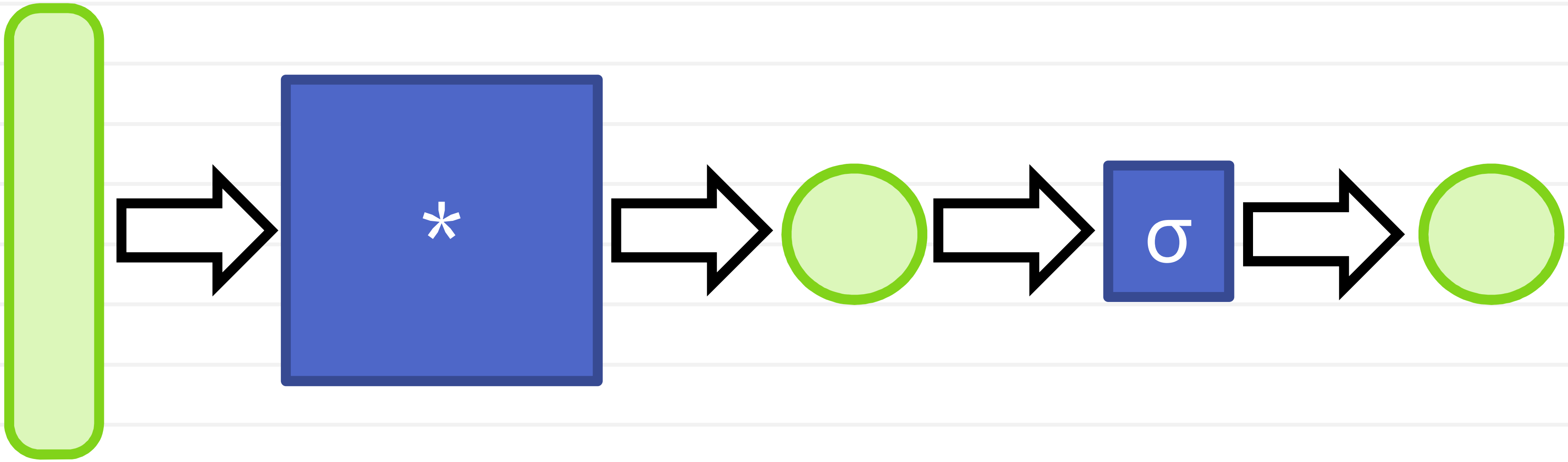
“units, neurons, activations, blobs”



Logistic regression

$$P(y(x) = y_i \mid \omega) = \frac{1}{1 + e^{-y_i \omega^T x_i}} = \sigma(y_i \omega^T x_i)$$

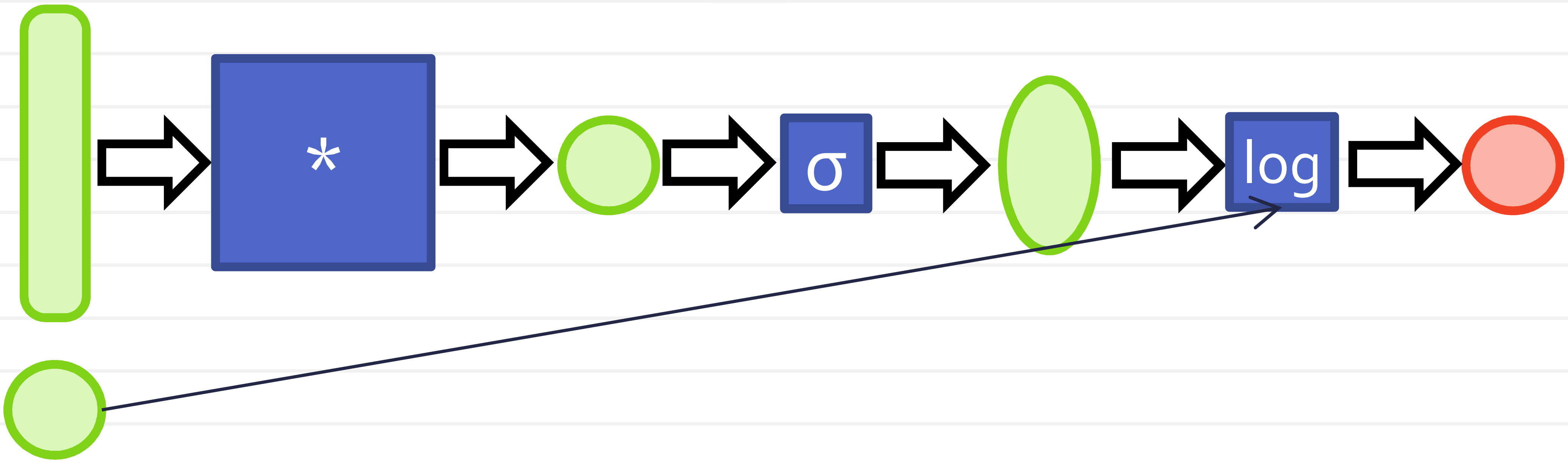
Same diagram/network:



Training logistic regression

$$E(w) = -\sum_{i=1}^{N} \log P(y(x)=y_i \mid w) =$$

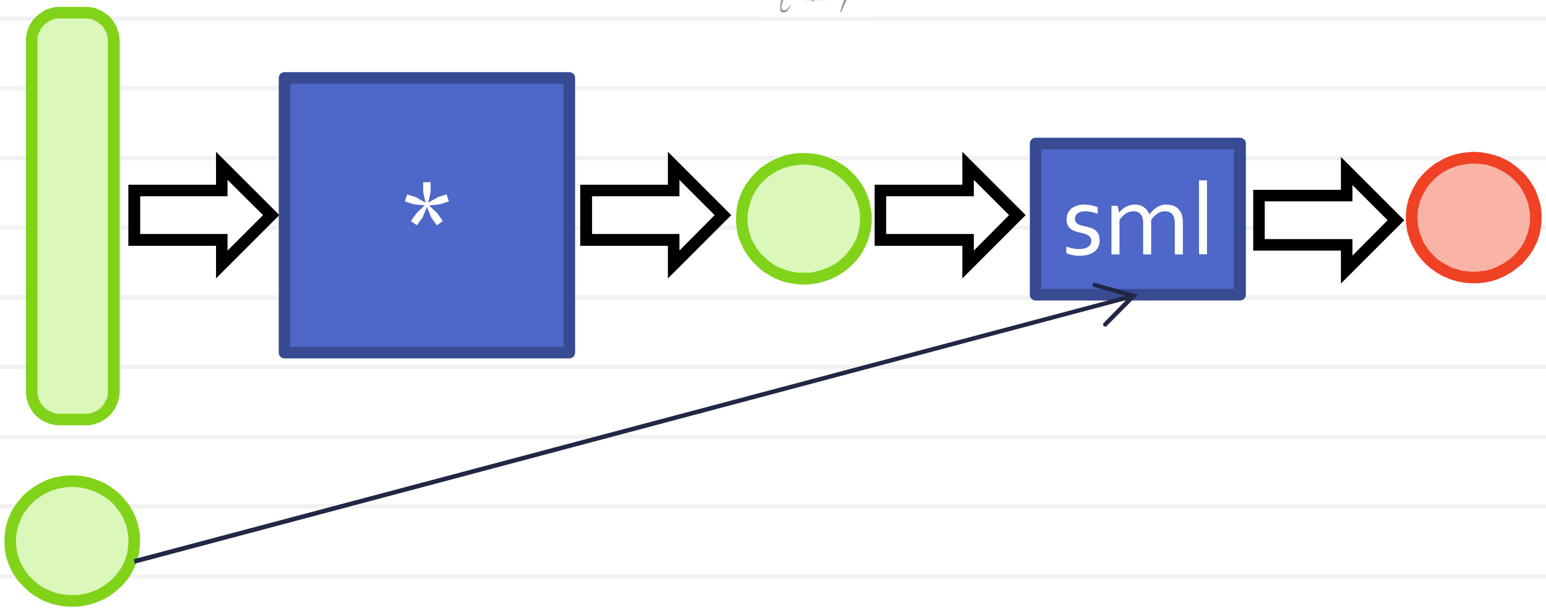
$$= \sum_{i=1}^{N} \log\left(1 + e^{-y_i w^T x_i}\right)$$



Logistic regression: simplifying training

$$E(\omega) = -\sum_{i=1}^{N} \log P(y(x)=y_i \mid \omega) =$$

$$= \sum_{i=1}^{N} \log\left(1 + e^{-y_i \omega^T x_i}\right)$$



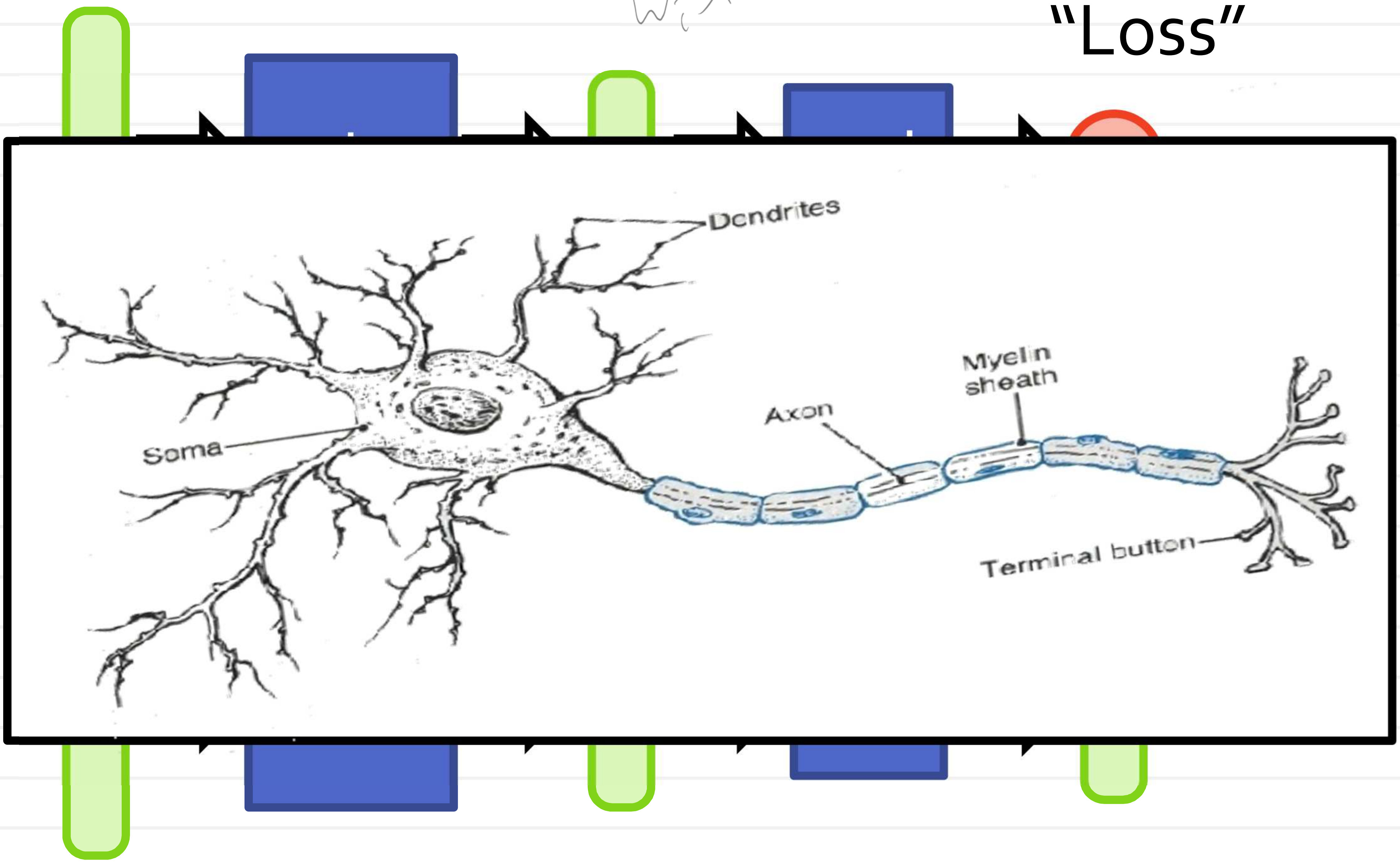
Softmax loss = log loss over softmax/logistic

Multinomial logistic regression

Training:

$W_i^T x$

"Loss"



Biological neuron layers

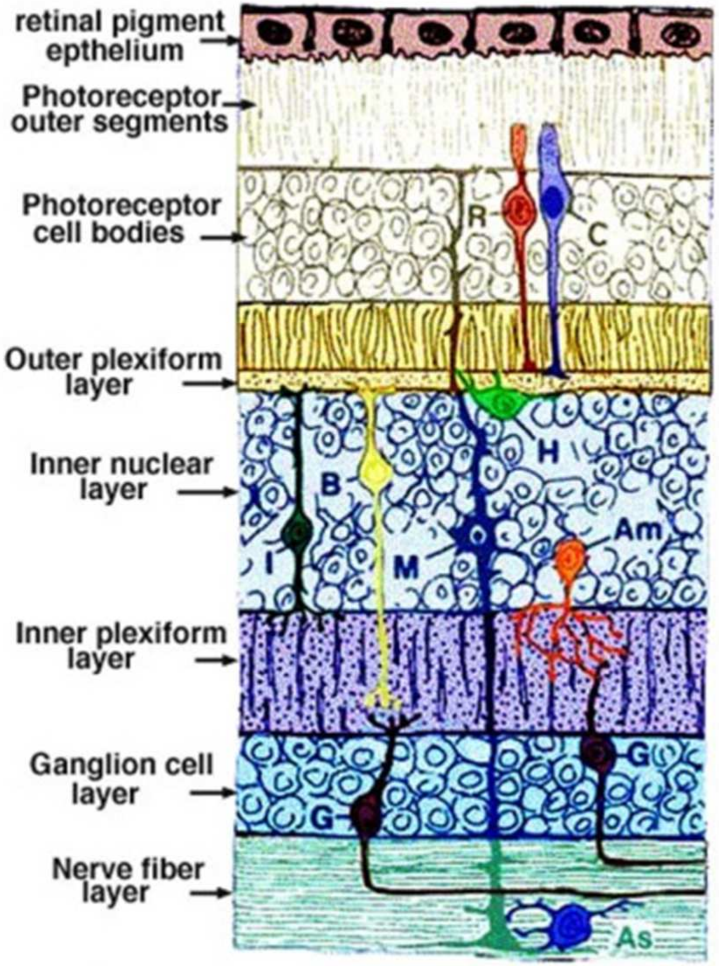


Fig. 5. Scheme of the layers of the developing retina around 5 months' gestation (Modified from Odgen,1989).

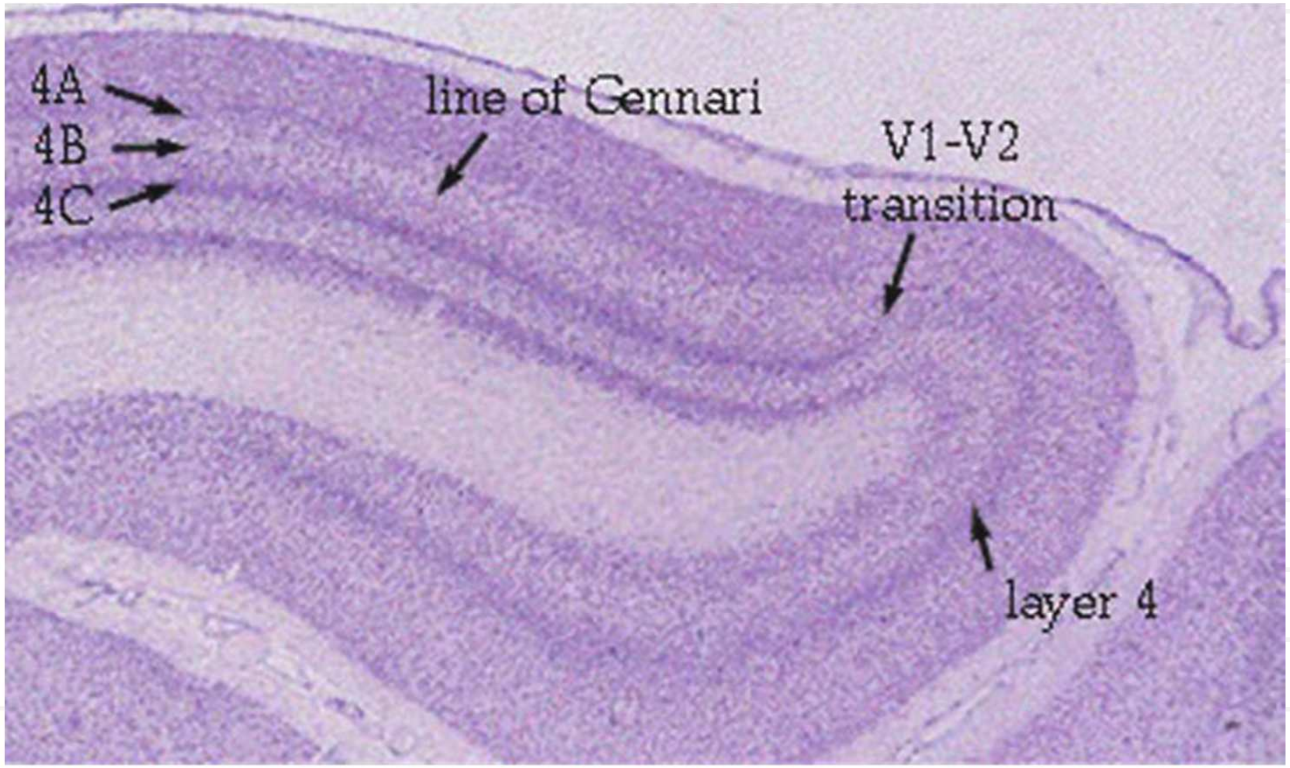
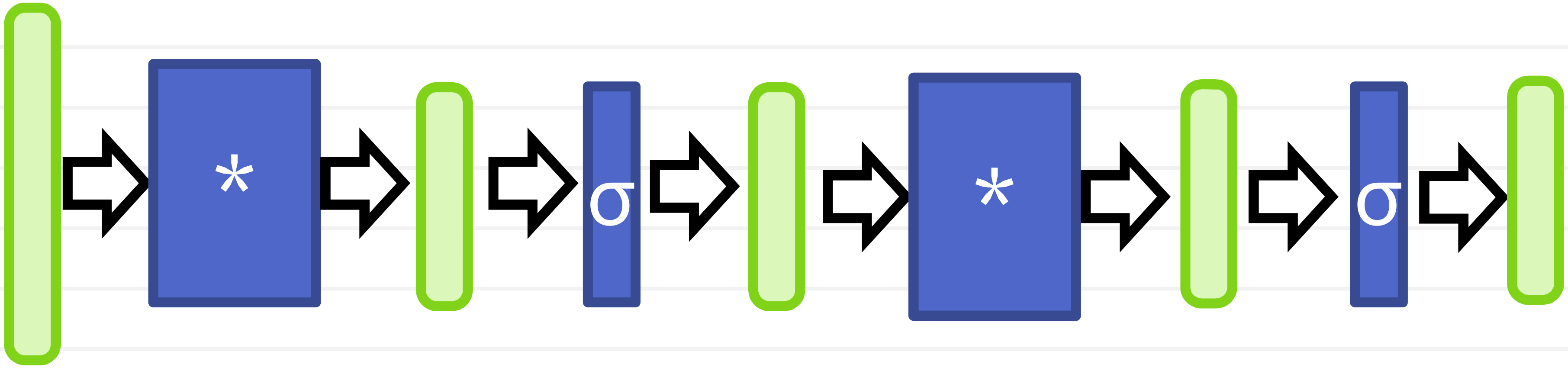


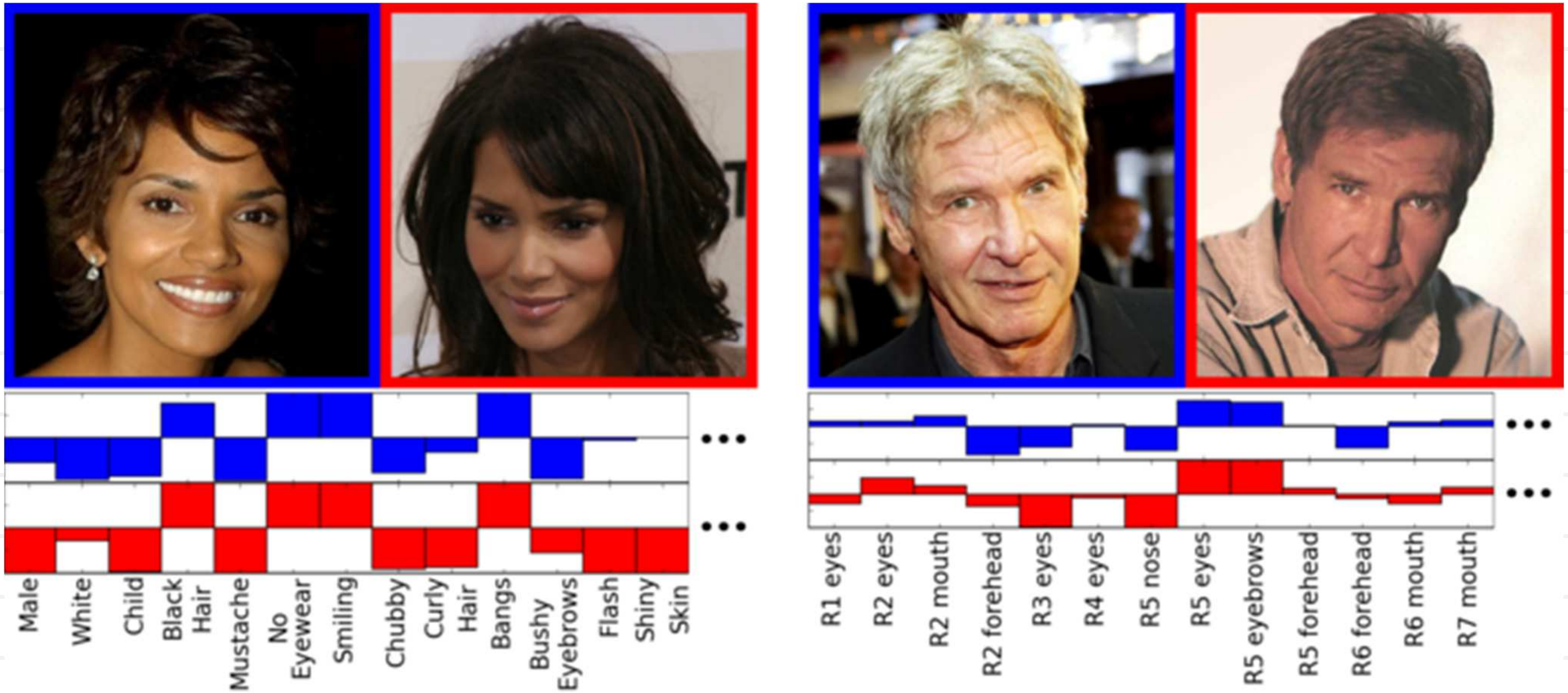
Figure 9. Nissl stained section of the visual cortex to show the border between area 17 (V1) and area 18 (V2).

Multi-layer perceptron idea



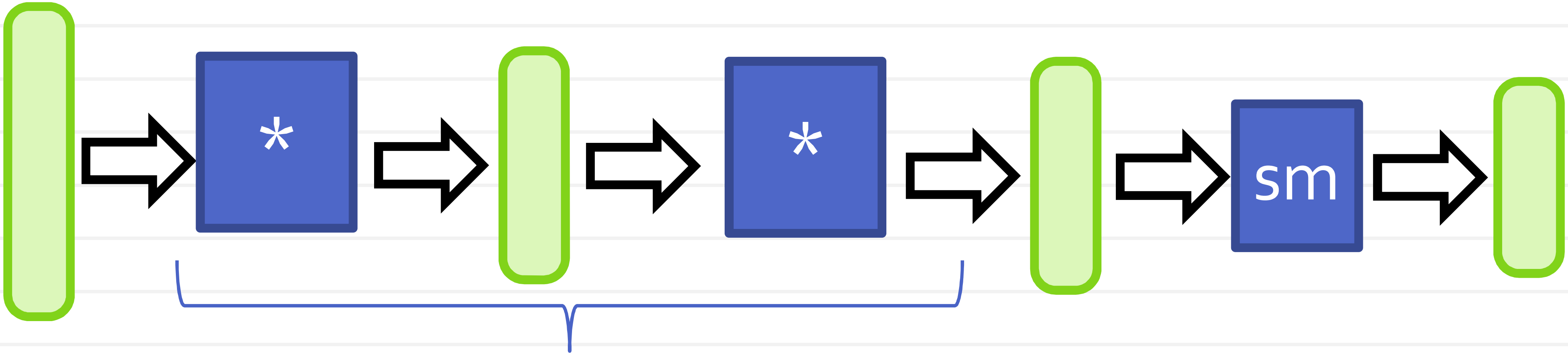
- First layer: parallel logistic regression
- Each predicts presence of some feature in the input
- Second layer is a logistic regression that "weighs" the input of the first layer

Classifier output as features



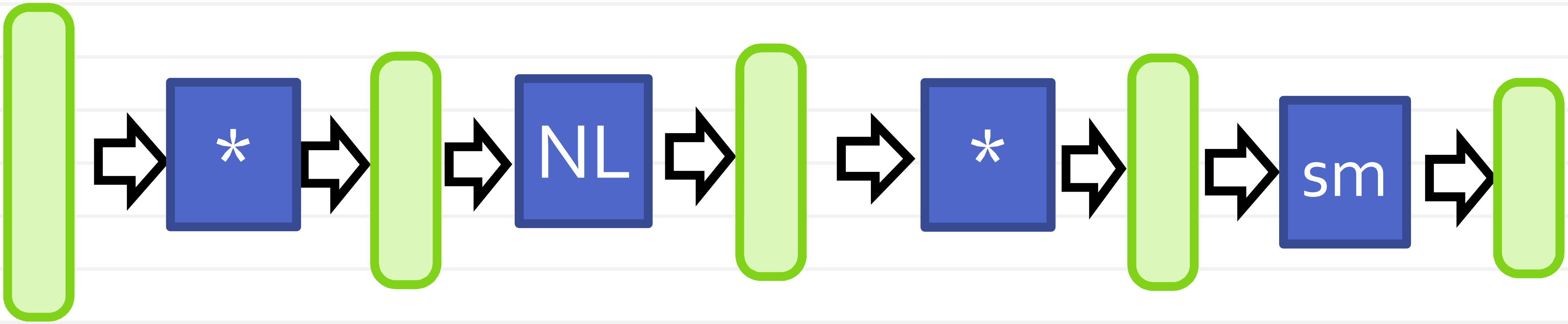
[Kumar et al. **Attribute and Simile Classifiers for Face Verification. ICCV 2009]**

Artificial multilayer networks



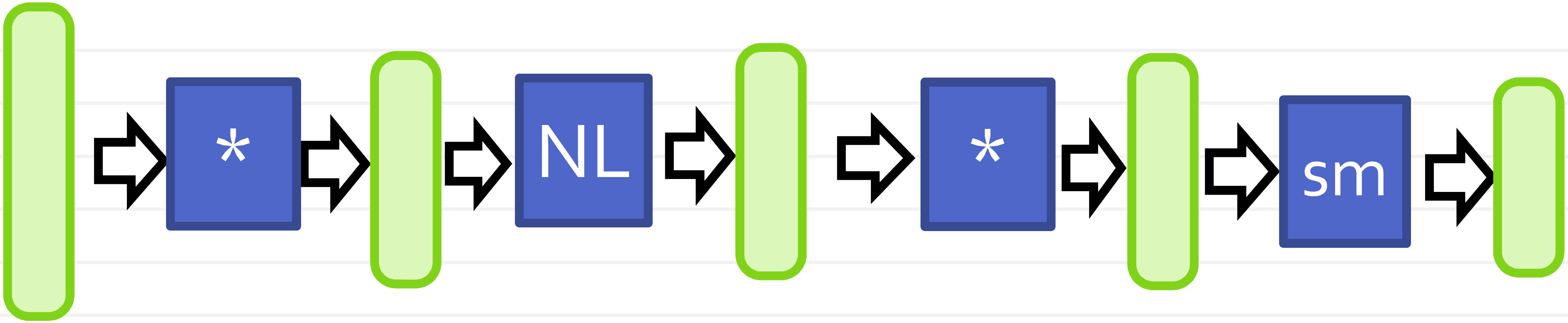
still single matrix multiplication

To get more powerful model need non-linearity:



Adding non-linearities

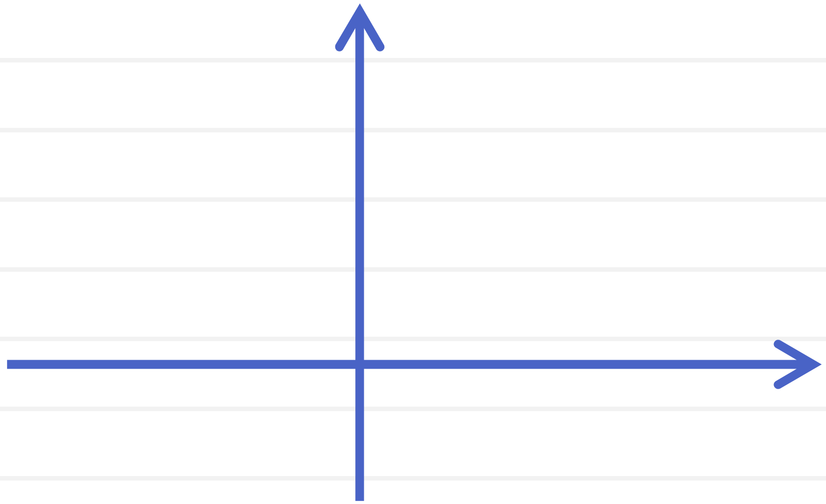
To get more powerful model need non-linearity:



Possible *elementwise* non-linearities:

- Heaviside
- Sigmoid(logistic)/tanh
- More recently:

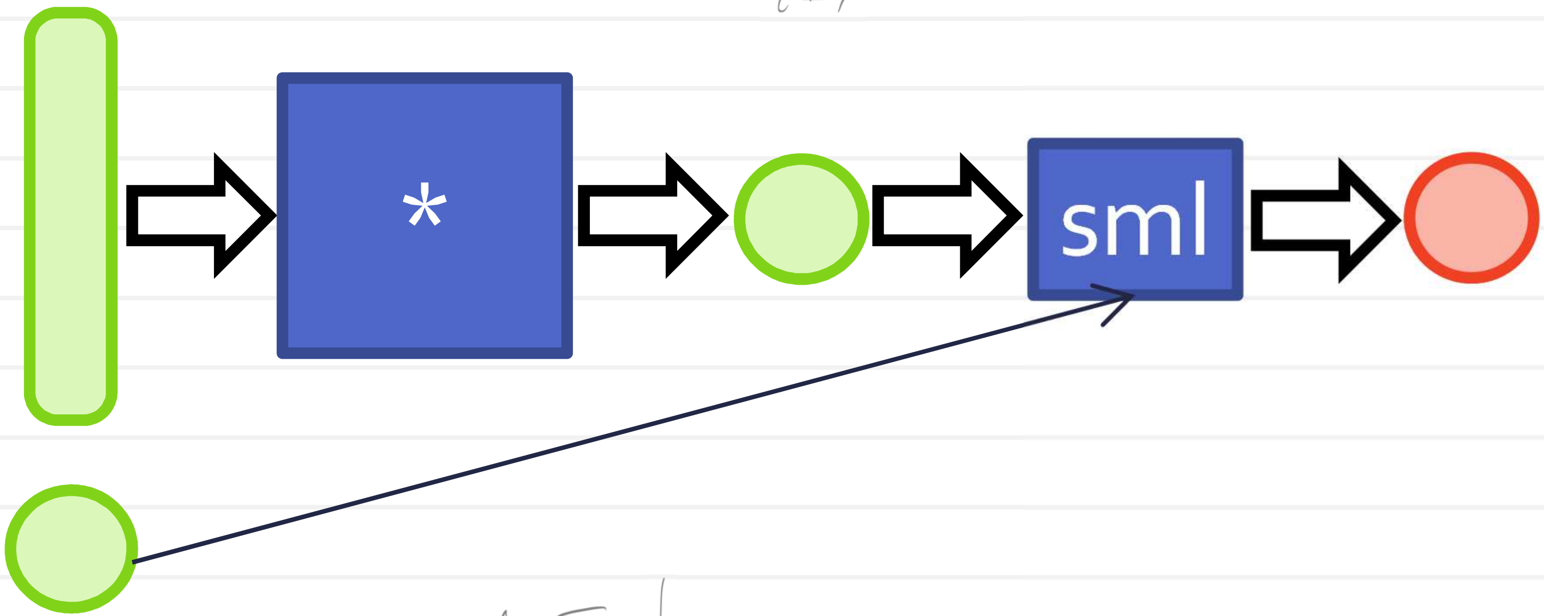
  ReLu(x) = max(0,x)



Training logistic regression

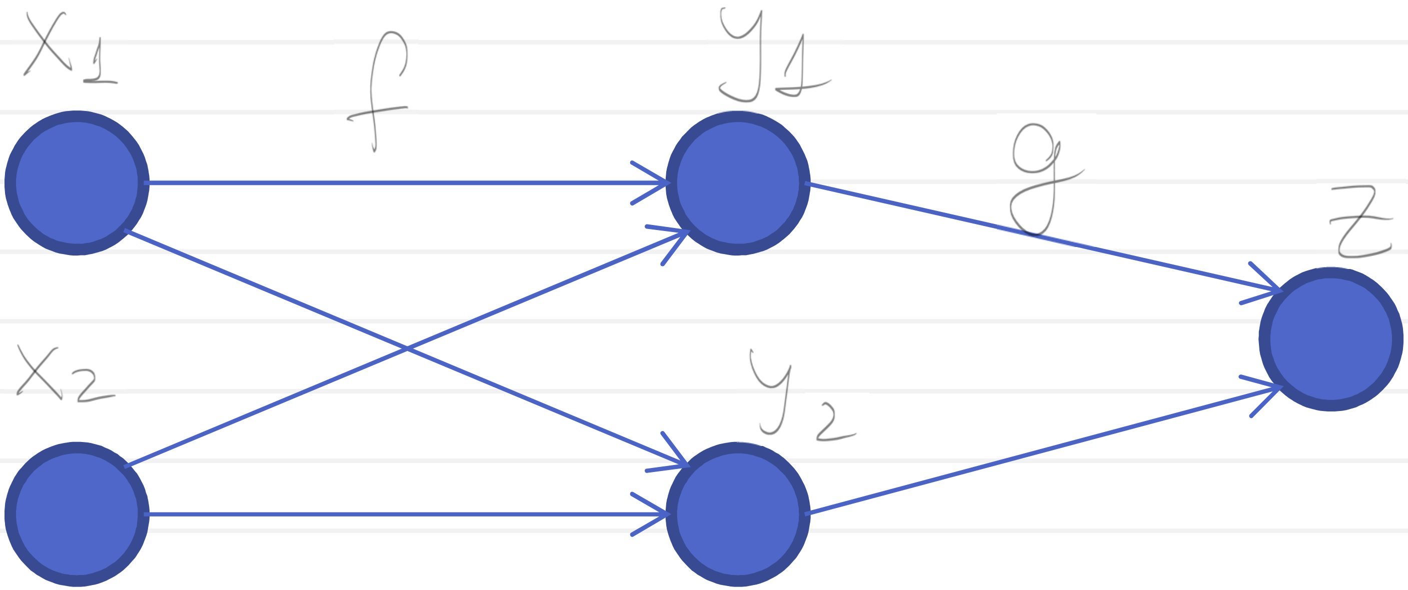
$$E(w) = -\sum_{i=1}^{N} \log P(y(x)=y_i | w) =$$

$$= \sum_{i=1}^{N} \log\left(1 + e^{-y_i w^T x_i}\right)$$



$$\left.\frac{dE}{dw}\right|_{x_i} = \left(\sigma(y_i w^T x_i) - 1\right) y_i x_i$$

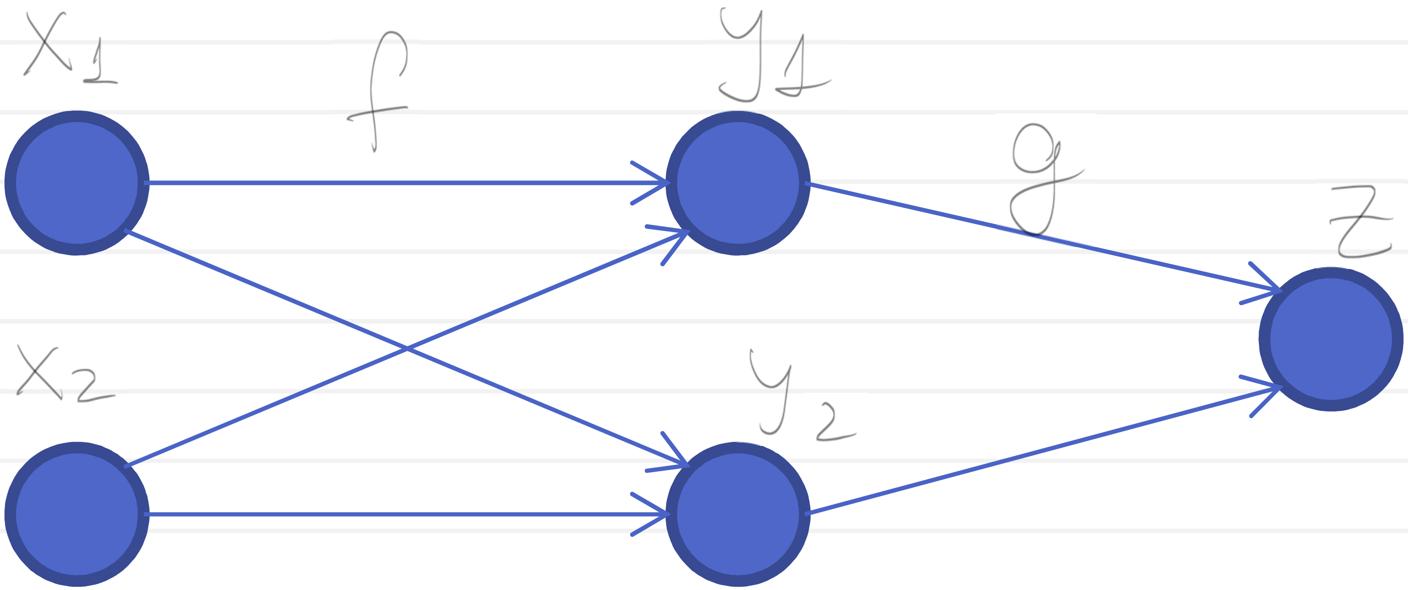
Recap: chainrule



$$\frac{dz}{dy} = \begin{bmatrix} \frac{\partial g}{\partial y_1}(y_1, y_2) \\ \frac{\partial g}{\partial y_2}(y_1, y_2) \end{bmatrix}$$

$$\frac{\partial z}{\partial x_1} = \frac{\partial z}{\partial y_1} \cdot \frac{\partial y_1}{\partial x_1} + \frac{\partial z}{\partial y_2} \cdot \frac{\partial y_2}{\partial x_1}$$

Recap: chainrule



$$\frac{\partial z}{\partial x_1} = \frac{\partial z}{\partial y_1} \cdot \frac{\partial y_1}{\partial x_1} + \frac{\partial z}{\partial y_2} \cdot \frac{\partial y_2}{\partial x_1}$$

$$\frac{\partial z}{\partial x_2} = \frac{\partial z}{\partial y_1} \cdot \frac{\partial y_1}{\partial x_2} + \frac{\partial z}{\partial y_2} \cdot \frac{\partial y_2}{\partial x_2}$$

Recap: chainrule

$$\frac{\partial z}{\partial x_1} = \frac{\partial z}{\partial y_1} \cdot \frac{\partial y_1}{\partial x_1} + \frac{\partial z}{\partial y_2} \cdot \frac{\partial y_2}{\partial x_1} \qquad \frac{\partial z}{\partial x_2} = \frac{\partial z}{\partial y_1} \cdot \frac{\partial y_1}{\partial x_2} + \frac{\partial z}{\partial y_2} \cdot \frac{\partial y_2}{\partial x_2}$$

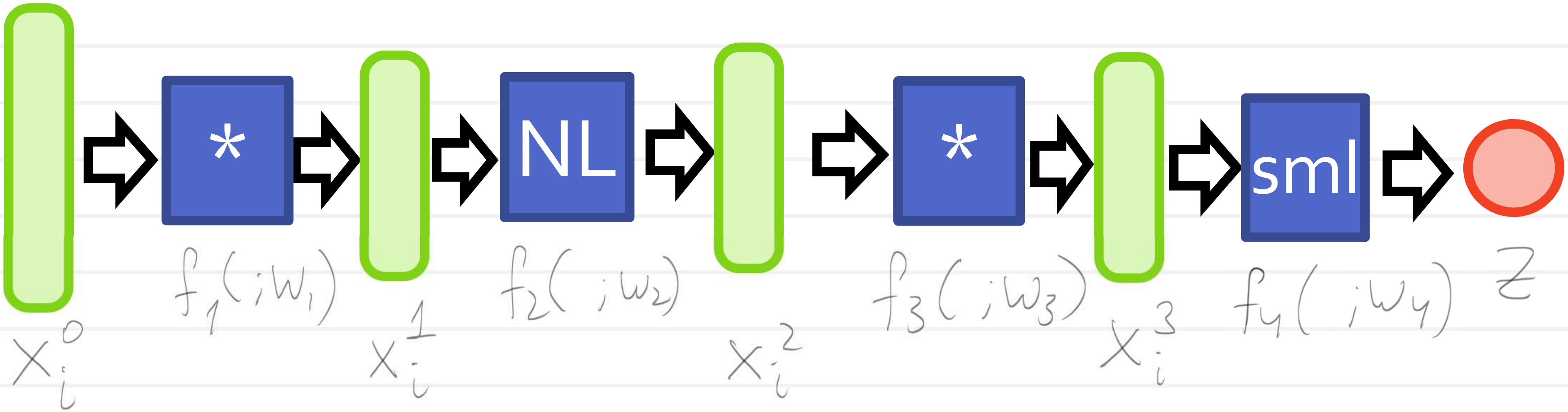
$$\frac{dz}{dx} = \begin{pmatrix} \frac{\partial z}{\partial x_1} \\ \frac{\partial z}{\partial x_2} \end{pmatrix} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} \end{bmatrix} \begin{bmatrix} \frac{\partial z}{\partial y_1} \\ \frac{\partial z}{\partial y_2} \end{bmatrix}$$

Recap: chainrule

$$\frac{dz}{dx} = \begin{bmatrix} \frac{\partial z}{\partial x_1} \\ \frac{\partial z}{\partial x_2} \end{bmatrix} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} \end{bmatrix} \begin{bmatrix} \frac{\partial z}{\partial y_1} \\ \frac{\partial z}{\partial y_2} \end{bmatrix}$$

$$\frac{dz}{dx} = \left(\frac{dy}{dx} \right)^T \frac{dz}{dy}$$

Computing deeper derivatives



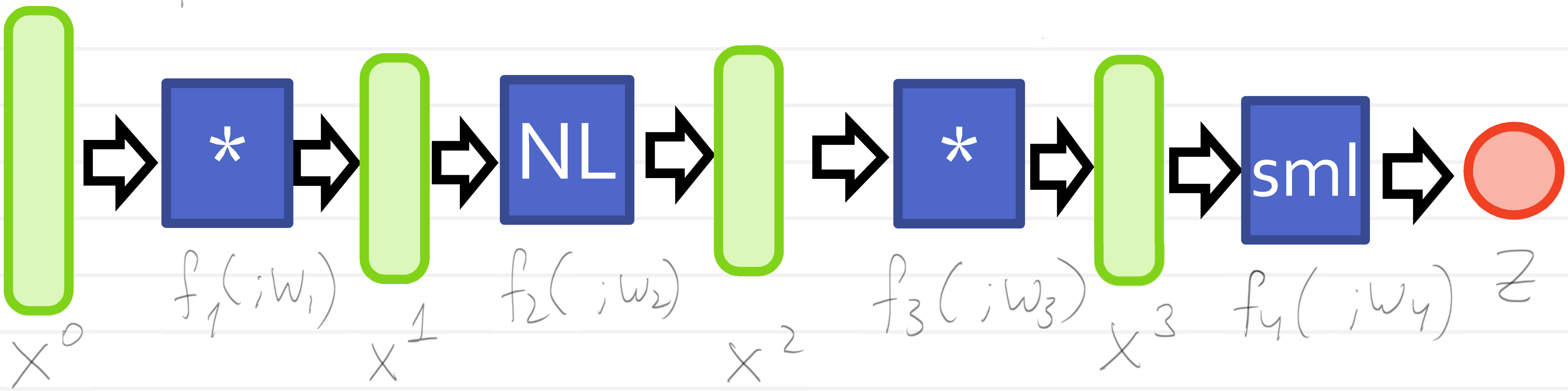
$$z = f_4(f_3(f_2(f_1(x;w_1);w_2);w_3);w_4)$$

Sequential computation: *backpropagation*

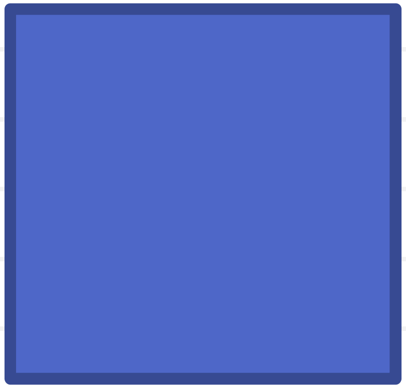
$$\frac{dz}{dw_3} = \frac{dx^3}{dw_3}^T \cdot \frac{dz}{dx^3} \qquad \frac{dz}{dx^2} = \frac{dx^3}{dx^2}^T \frac{dz}{dx^3}$$

$$\frac{dz}{dw_2} = \frac{dx^2}{dw_2}^T \cdot \frac{dz}{dx_2} \qquad \frac{dz}{dx^1} = \frac{dx^2}{dx^1}^T \frac{dz}{dx^2}$$

$$\frac{dz}{dw_1} = \frac{dx^1}{dw_1}^T \cdot \frac{dz}{dx_1} \qquad \frac{dz}{dx^0} = \frac{dx^1}{dx^0}^T \frac{dz}{dx^1}$$



Layer abstraction



Each layer is defined by:

- forward performance: $y = f(x)$
- backward performance:

$$z(x) = z(f(x; w)) \qquad y = f(x; w)$$

$$\frac{dz}{dx} = \frac{dy}{dx}^T \cdot \frac{dz}{dy} \qquad \frac{dz}{dw} = \frac{dy}{dw}^T \cdot \frac{dz}{dy}$$

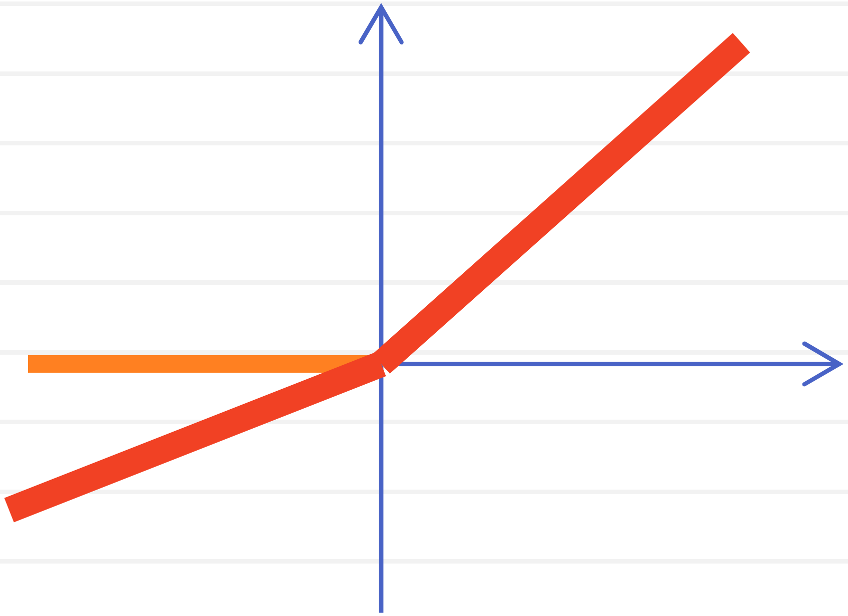
OOP pseudocode of deep learning

```
abstract class Layer {
        params w,dzdw;
        virtual y = forward(x);
        virtual dzdx = backward(dzdy,x,y);
        // should compute dzdw as well

        void update (tau) {
                w = w+tau*dzdw;
        }
};
```

Efficient implementations have to use vector/matrix instructions and work efficiently for minibatches!

Example: “leaky ReLu”



$$f(x;\alpha) = \max(x, \alpha x)$$

$$\frac{dz}{d\alpha} = [\alpha x > x]\, x^T \cdot \frac{dz}{dy}$$

Cornell University Library

arXiv.org > cs > arXiv:1502.01852

Computer Science > Computer Vision and Pattern Recognition

Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification

Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun

(Submitted on 6 Feb 2015)

Computing the partial derivatives

$$\frac{dz}{dx} = \frac{dy}{dx}^T \cdot \frac{dz}{dy} \qquad \frac{dz}{dw} = \frac{dy}{dw}^T \cdot \frac{dz}{dy}$$

Options for partial derivatives:

- Finite differences (bad idea)
- Analytic gradients (good idea)

Debugging is hard

Gradient checking is a good idea!

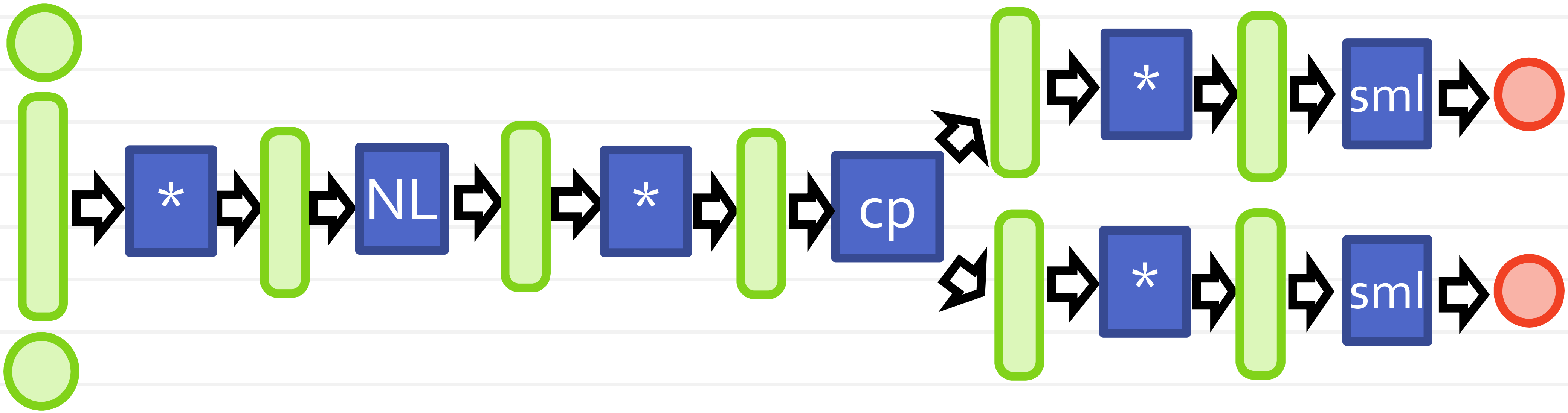
Recap

Deep learning:

- Define each layer
- Assemble a chain of layers
- Loop over minibatches
- For each minibatch find the stochastic gradient and update the parameters (use momentum, etc.)

In fact, chain can easily be replaced with DAG

Example: multitask learning



Typical usecase:

- Two related tasks
- Limited labeled data for the main task
- Lots of labeled data for auxiliary task

Zoo of layers

Multiplicative layer
Convolutional layer

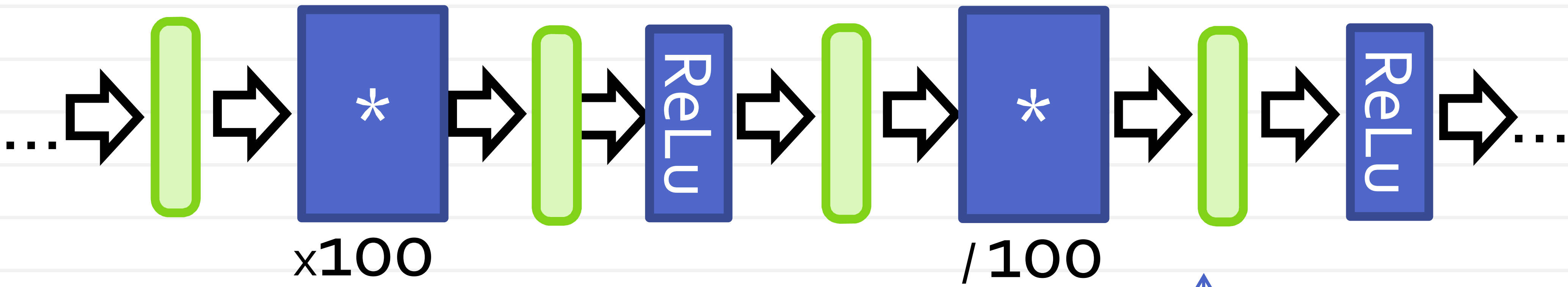
Copy layer
Split layer
Cat layer
Merge layer

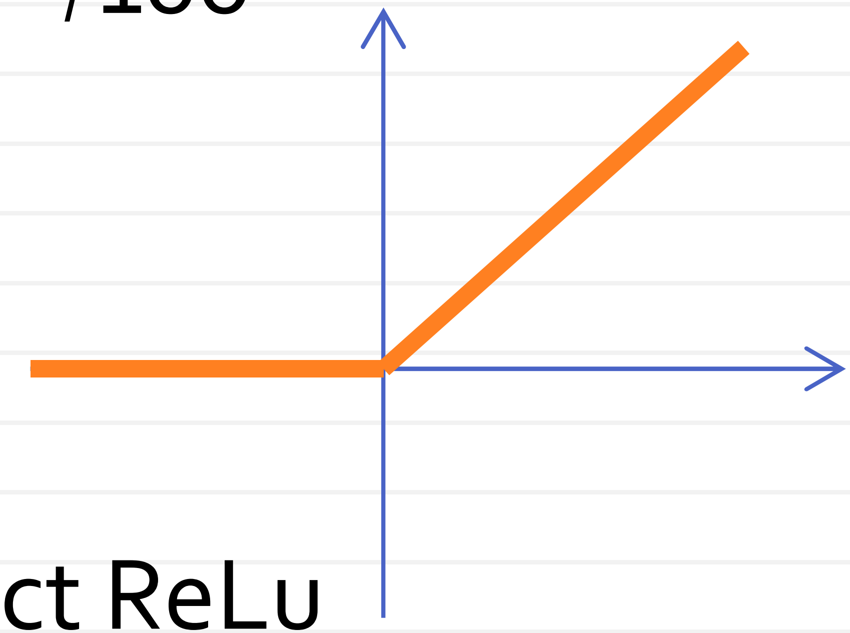
ReLu layer
Sigmoid layer
Softmax layer
Normalization layer
Max-pooling layer

Data providers

Log-loss layer
Softmax loss layer
Hinge loss layer
L2-loss layer
Contrastive loss layer

Reparameterization in ReLu Networks



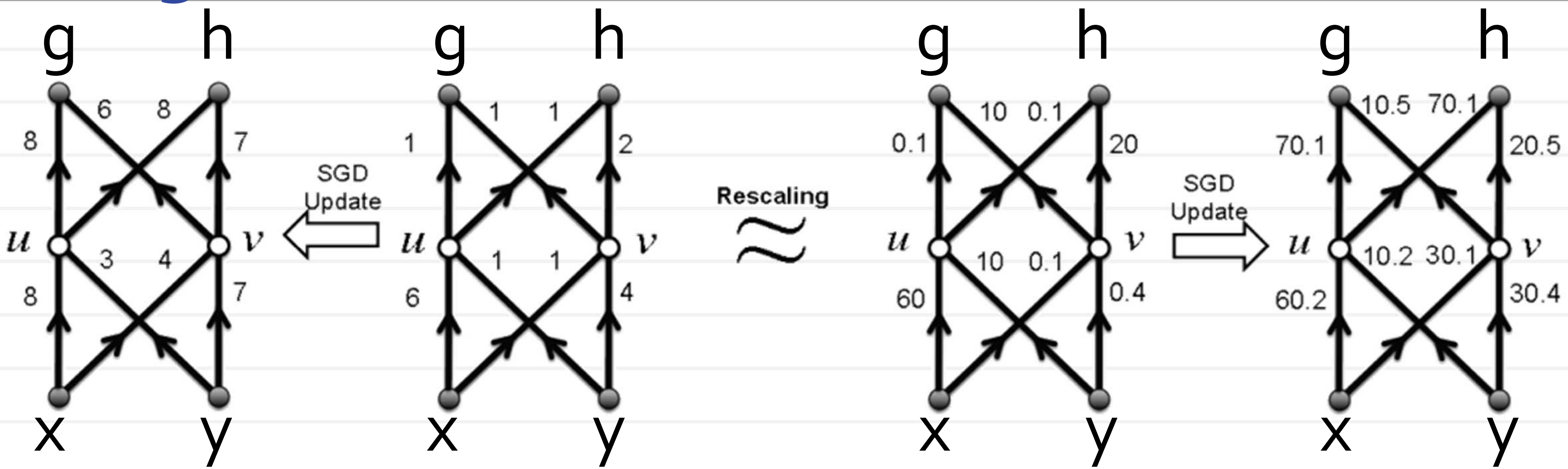


$\alpha > 0$

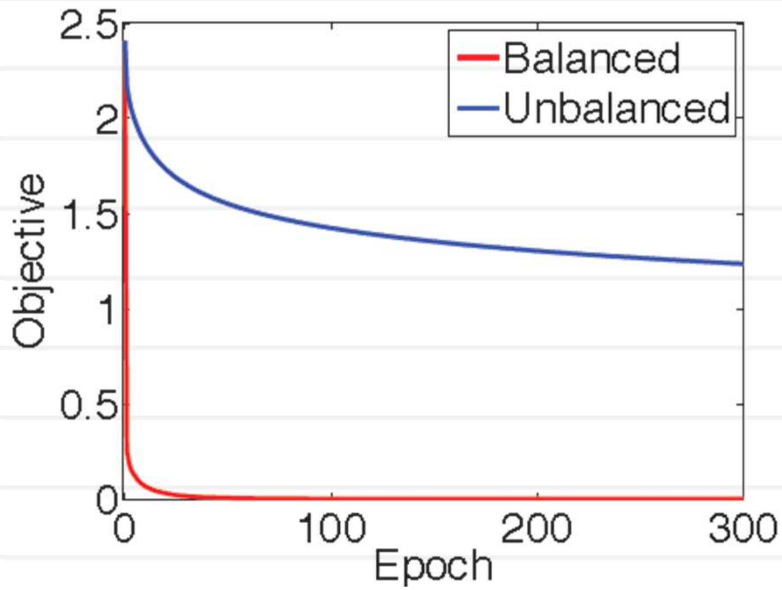
$$1/\alpha \; \mathrm{ReLu}(\alpha\, x) = \mathrm{ReLu}\,(x)$$

Thus: we can easily construct ReLu networks with **different** weights implementing the **same** function

Gauge freedom and SGD

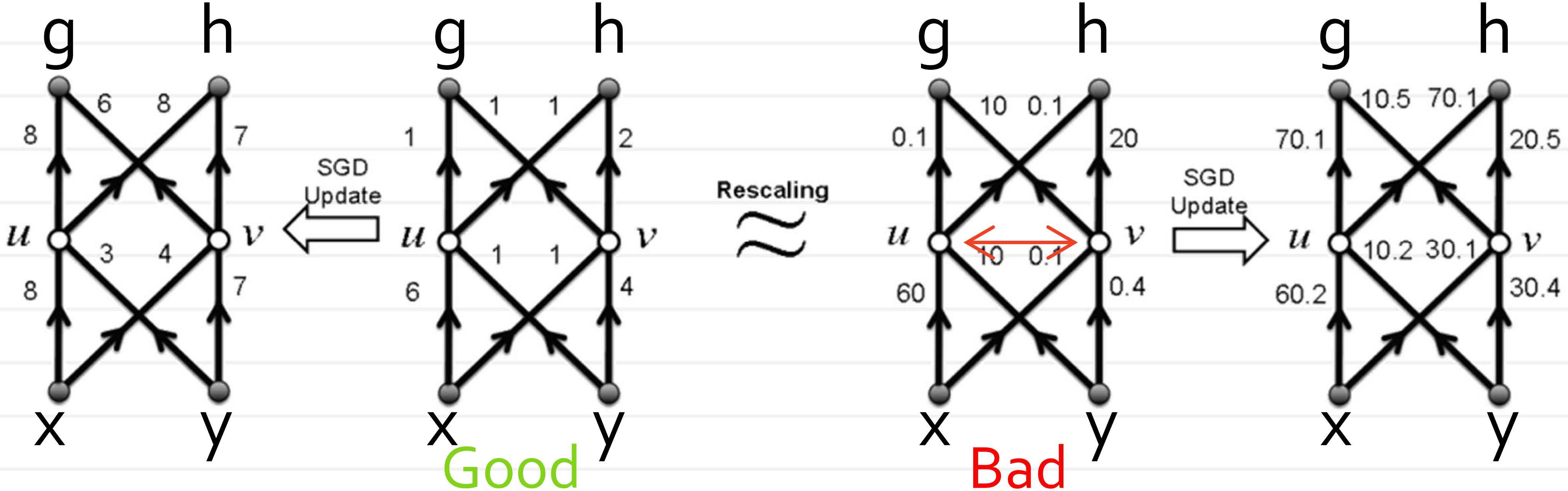


1 SGD step for (x,y = 1,1) and L = g+h



[Neyshabur, Salakhutdinov, Srebro, Path-SGD: Path-Normalized Optimization in Deep Neural Networks, NIPS2015]

Initialization schemes



- **Basic idea 1:** units should have comparable total input weights
- **Basic idea 2:** use layers which keep magnitude
- E.g. [Glorot&Bengio 2010] aka "Xavier-initialization":

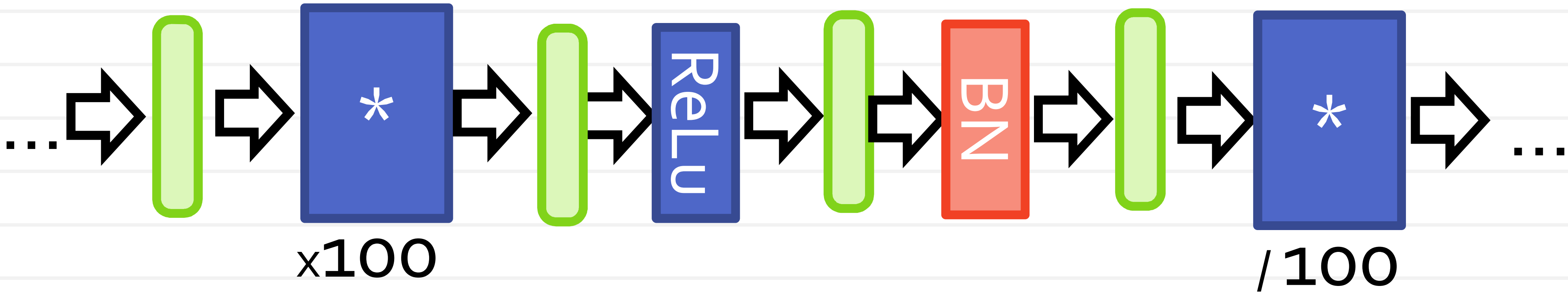
$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right]$$

- E.g. [He et al, Arxiv15] for ReLu networks:

$$W \sim \mathcal{N}(0, \sqrt{2/n_i})$$

Batch normalization

[Szegedy and Ioffe 2015]



- Makes the training process invariant to some reparameterizations
- Use mini-batch statistics at training time
- Use population statistics at test time
- At test time can be "incorporated" into adjacent multiplicative layer

Batch normalization layer

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

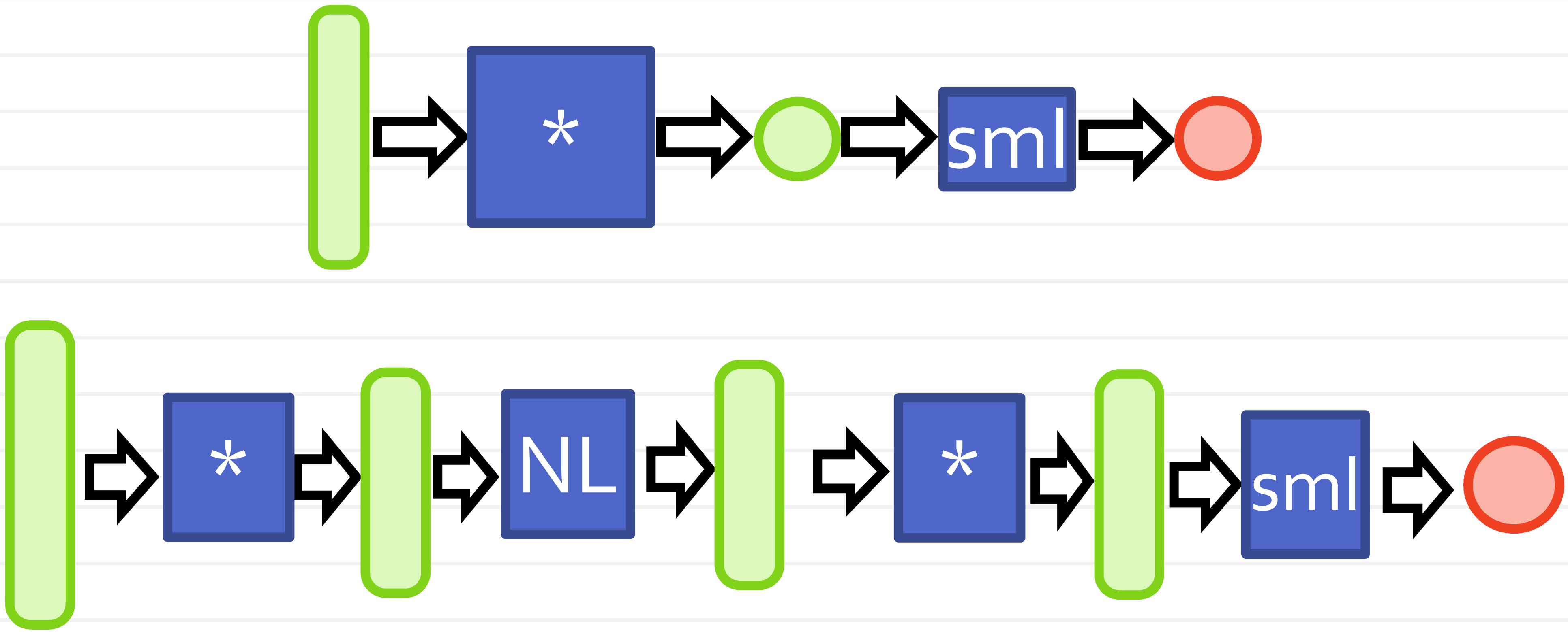
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

covariant to reparameterization

learnable by SGD

[Szegedy and Ioffe 2015]

Back to regularization



- Overfitting is severe for deep models (why?)
- The progress on deep learning was "delayed" till huge amount of data

Recap: regularization

4 strategies to avoid overfitting (aka *regularize learning*):

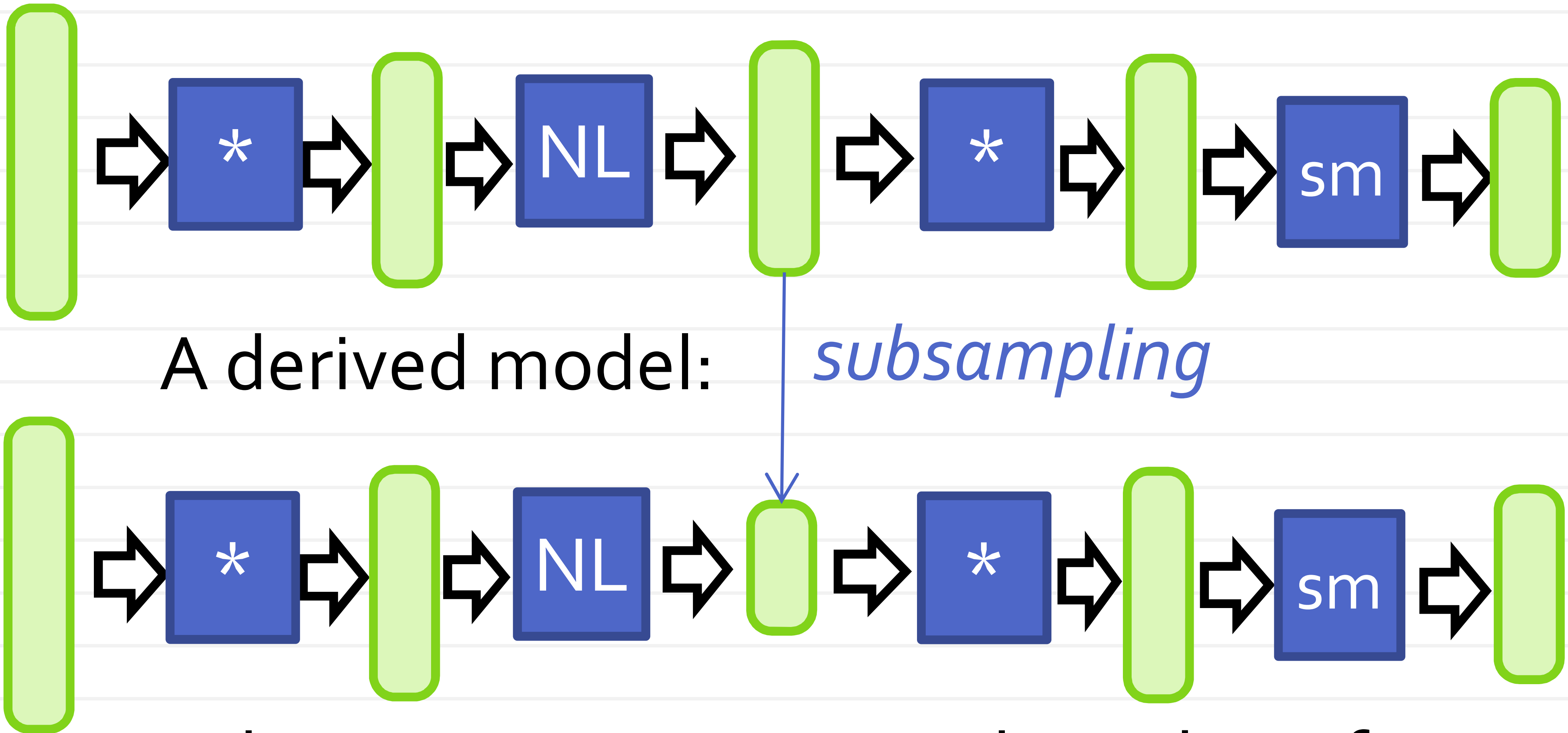
- Pick a "simpler" model (e.g. conv nets)
- Stop optimization early (always keep checking progress on)
- Impose smoothness (weight decay)
- Bag multiple models

Bagging multiple NN

- Different local minima help
- Diversifying architectures helps even more
- Unit weights are often prefered to tuned weights
- (Almost) all classification competitions are won by ensembles of deep models

Dropout idea

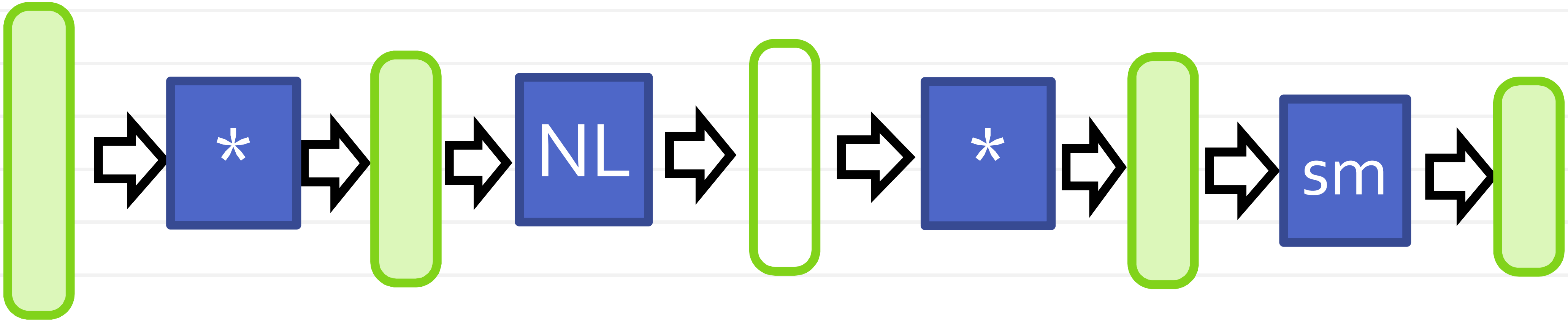
Pseudo-ensemble training:



Goal: “train” an exponential number of such reduced models [Srivastava et al.]

Dropout idea

Pseudo-ensemble training:



- At training time, define which units are active at random (*mask*)
- At test time, let them all be active but multiply by dropout probability

How to implement dropout

Define it as a layer!

Forward propagation:

$$n \sim \text{Bernouli}(p)$$

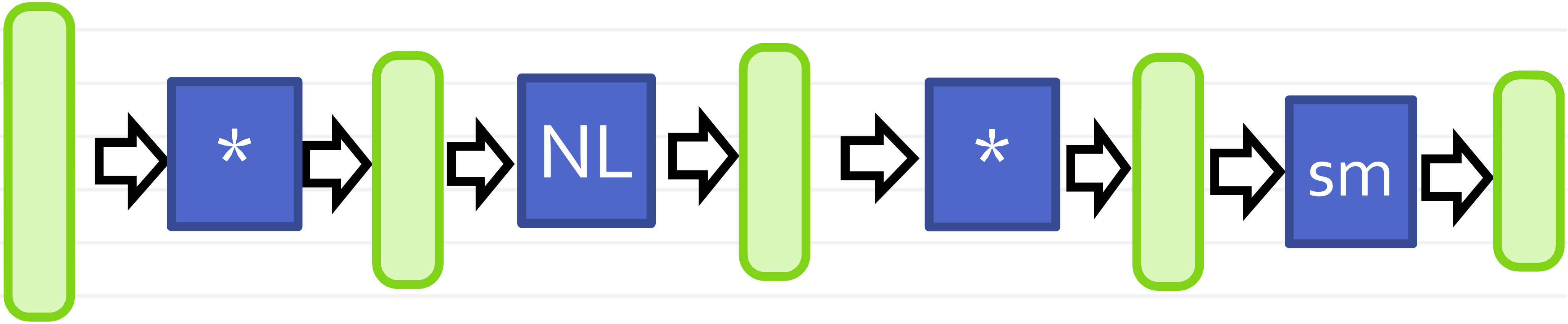
$$y = x \odot n$$

Backward propagation:

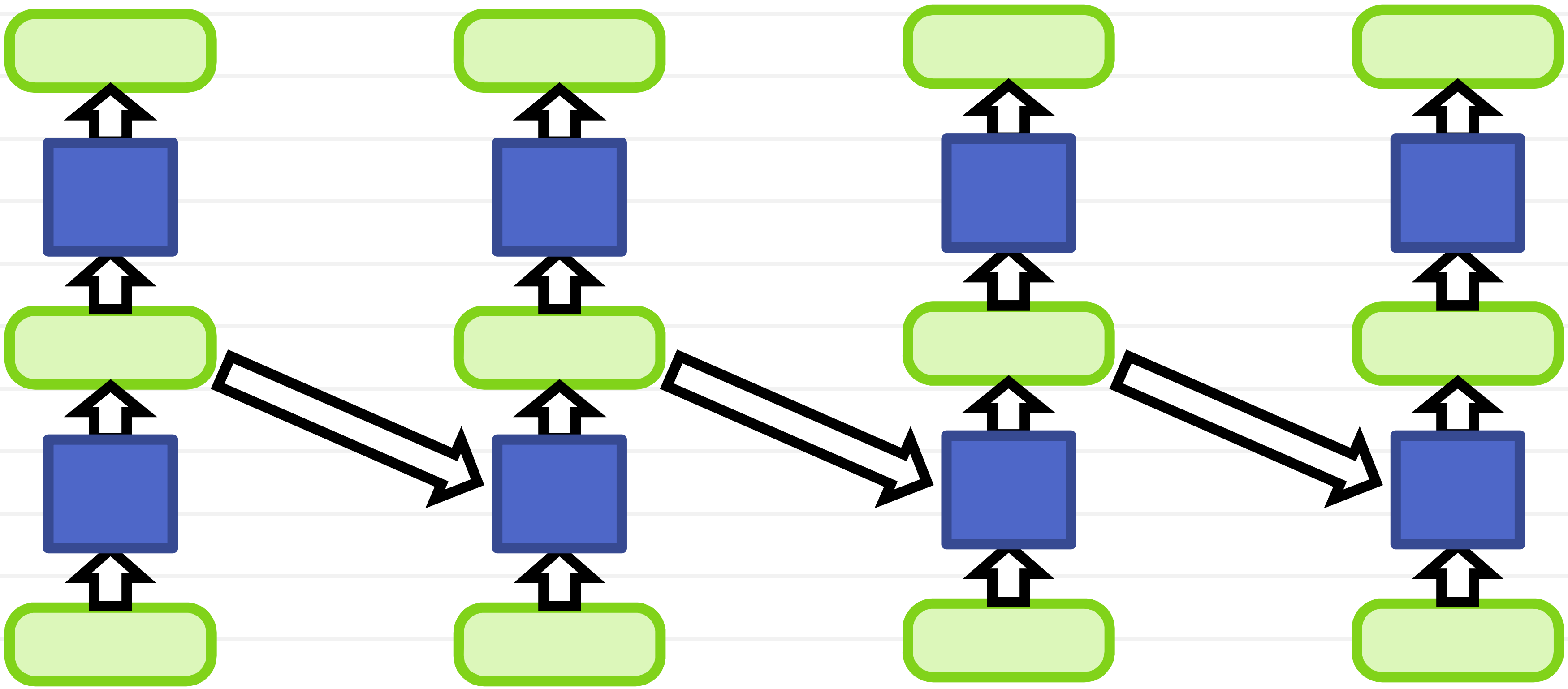
$$\frac{dz}{dx} = \frac{dz}{dy} \odot n$$

At test time: average all models with n=1/p

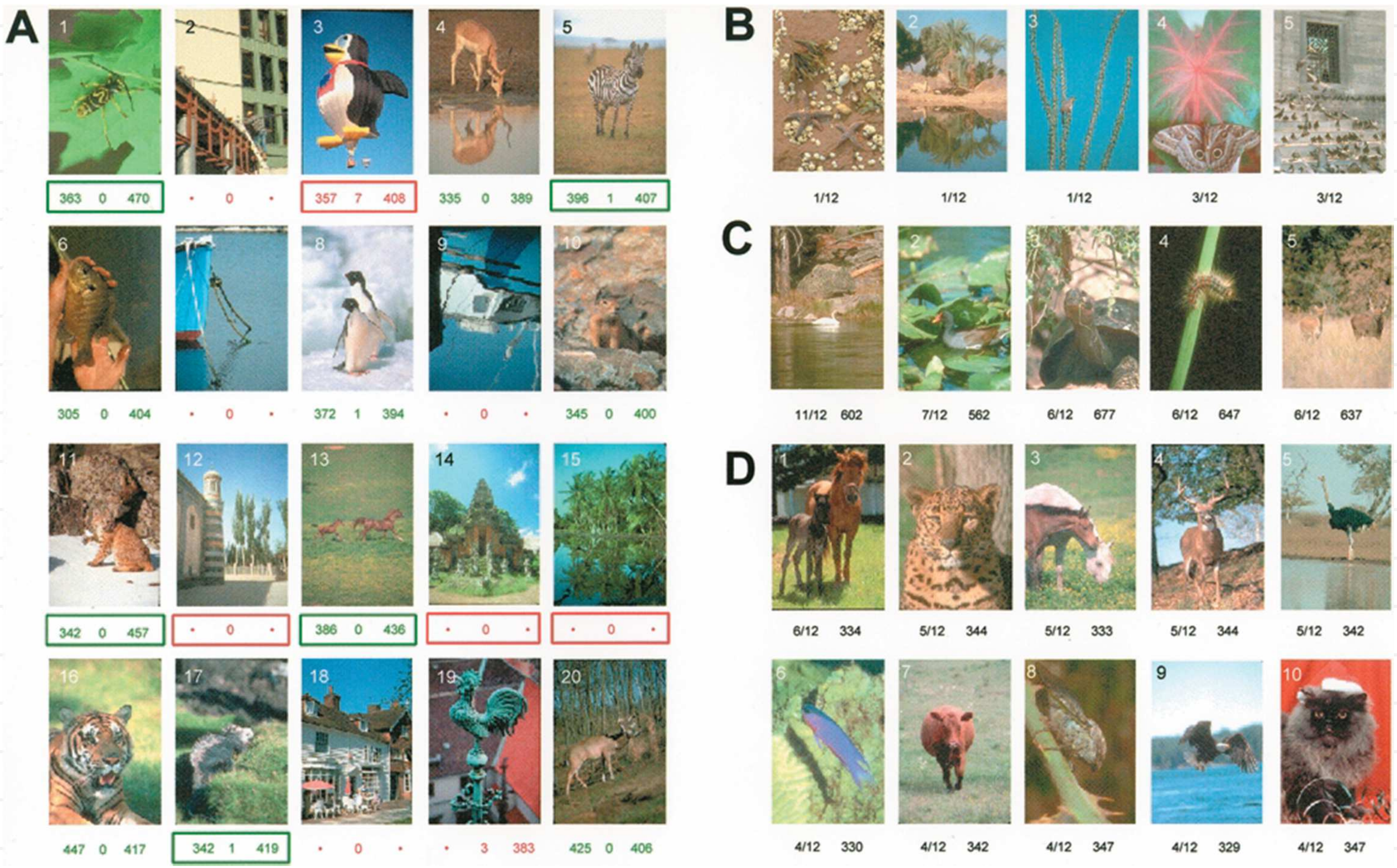
Feedforward net vs. Recurrent net



Typical *recurrent* architecture:



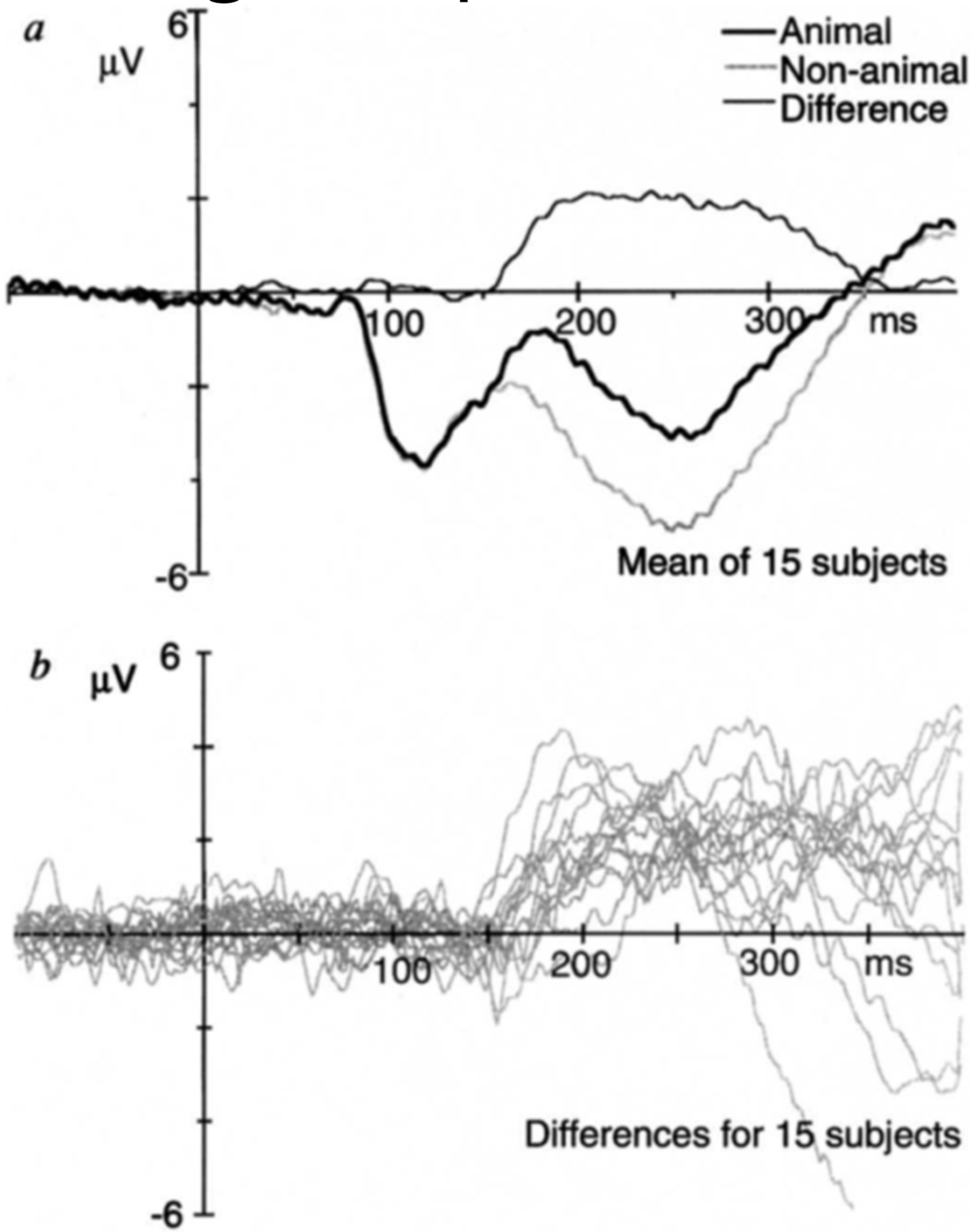
Feedforward nets in the brain



[Simon Thorpe et al.]

Feedforward classification in the brain

Average response of several EEG electrodes



- 94-98% accuracy after 150 ms
- Most likely, only feed-forward processing is possible in 150 ms

[Thorpe et al. Nature'96]

Recap

- Deep learning emerge naturally from shallow models (e.g. logistic regression)
- Loose connection to biological neural networks
- Modular paradigm is important
- Several very good packages for feed-forward models exist

Bibliography

Neeraj Kumar, Alexander C. Berg, Peter N. Belhumeur, Shree K. Nayar: Attribute and simile classifiers for face verification. ICCV 2009: 365-372

Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun: Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. CoRRabs/1502.01852 (2015)

Behnam Neyshabur, Ruslan Salakhutdinov, Nathan Srebro: Path-SGD: Path-Normalized Optimization in Deep Neural Networks. NIPS 2015

Xavier Glorot, Yoshua Bengio: Understanding the difficulty of training deep feedforward neural networks. AISTATS 2010: 249-256

Bibliography

Sergey Ioffe, Christian Szegedy:
Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. ICML2015: 448-456

Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov:
Dropout: a simple way to prevent neural networks from overfitting.
Journal of Machine Learning Research 15(1): 1929-1958 (2014)

Thorpe, Simon, Denis Fize, and Catherine Marlot. "Speed of processing in the human visual system." Nature 381.6582 (1996): 520-522.