# MLHEP 2017

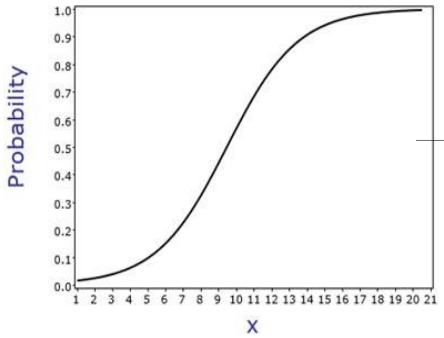
day 4.1

# Neural networks 101

Maxim Borisyak, Alexander Panin, Andrey Ustyuzhanin

# Recap: logistic regression

X ⟶ Wx +b ⟶




⟶ P(y)

# Gradient descent

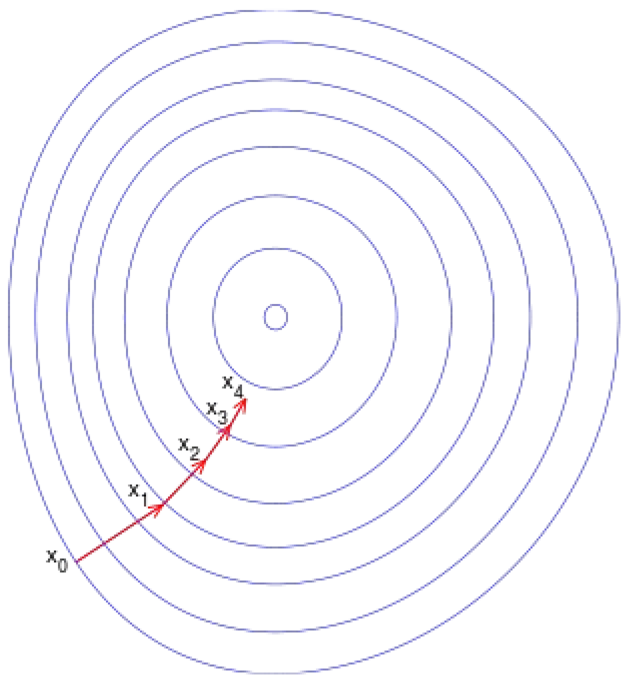
$$P(y|x) = \sigma(w \cdot x + b)$$

$$L = -\sum_i y_i \log P(y|x_i) + (1 - y_i) \log(1 - P(y|x_i))$$

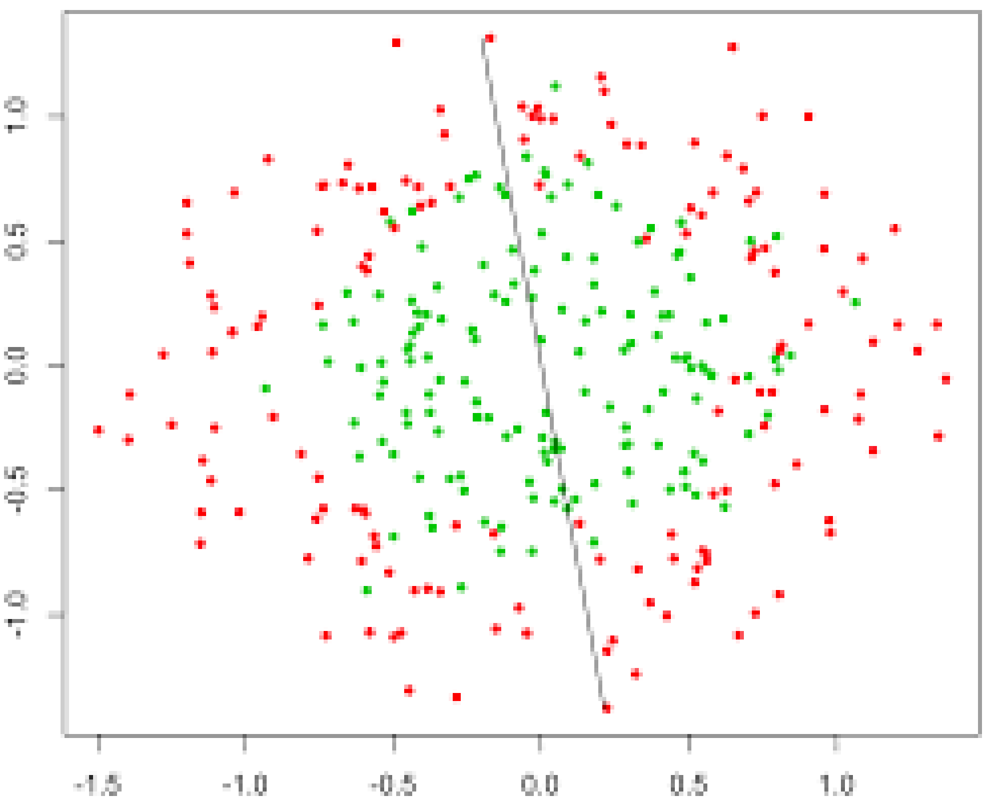
Repeat until convergence

$$\theta_j := \theta_j - \alpha \cdot \frac{\partial L(y, y_{pred})}{\partial \theta_j}$$

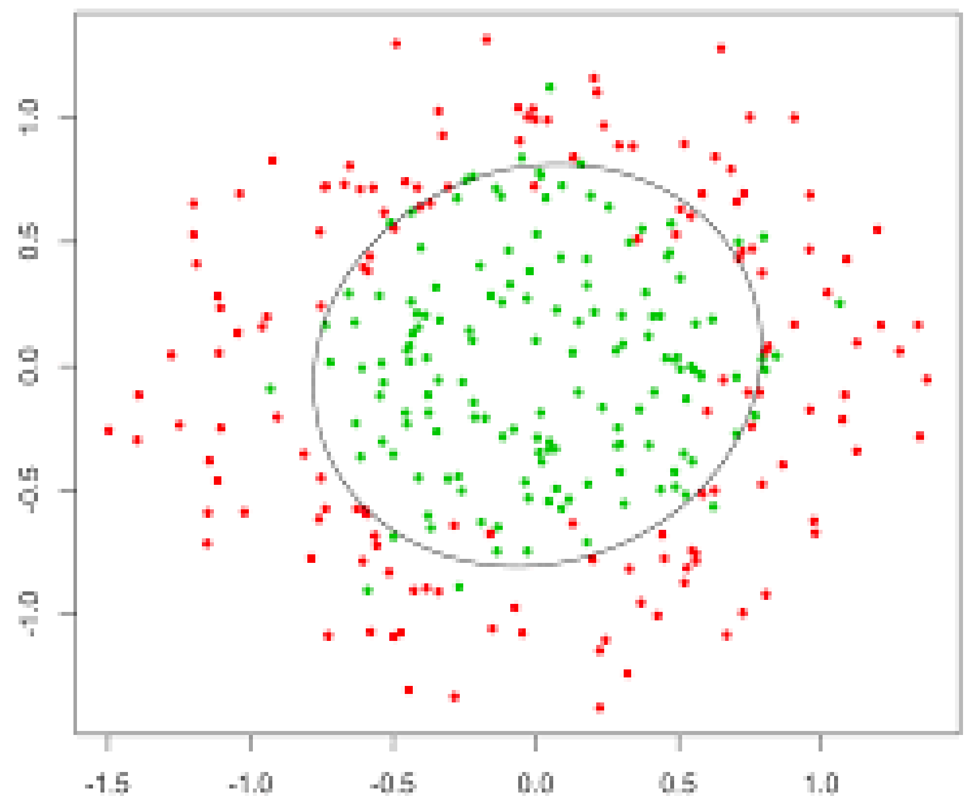
ϴ~{W,b}



# Nonlinear dependencies




What we have




What we want



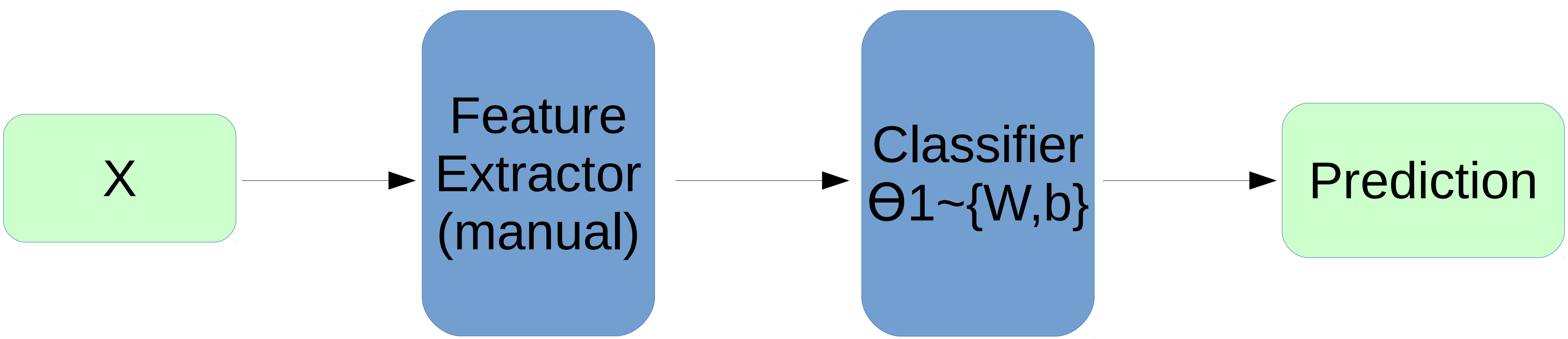
- How to get that?

# Feature extraction

Loss, for example:

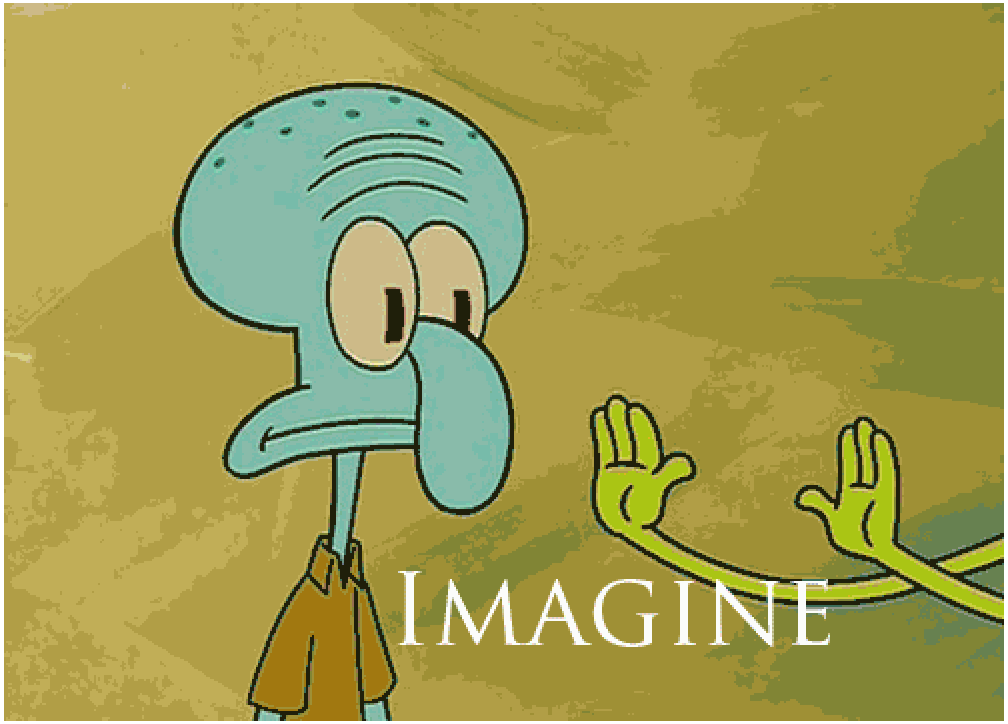
$$L = -\sum_{i} y_i \log P(y|x_i) + (1 - y_i) \log(1 - P(y|x_i))$$

Model:




Training:

$$\underset{\theta_1}{argmin}\, L(y, P(y|x))$$



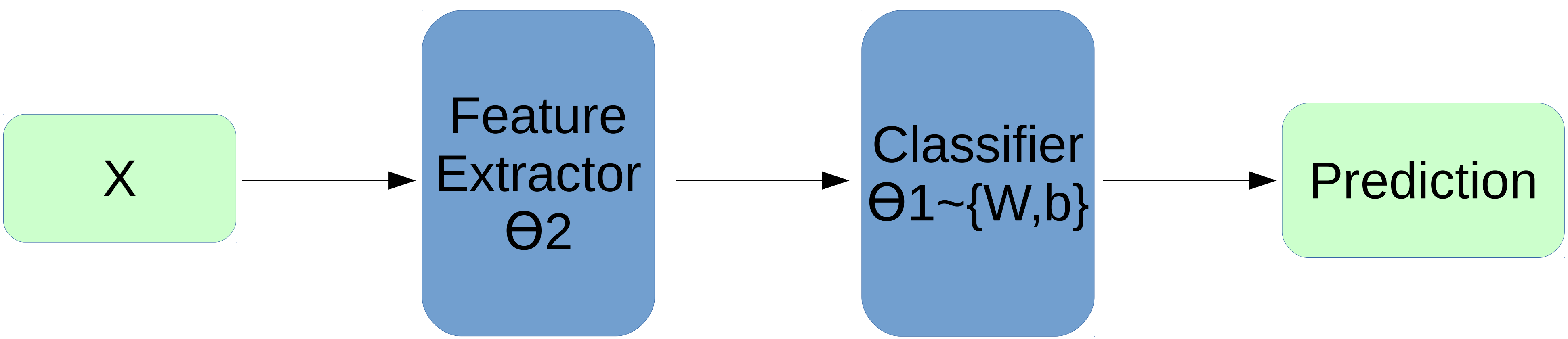

Features would tune to your problem automatically!

# What do we want, exactly?

Loss, for example:

$$L = -\sum_i y_i \log P(y|x_i) + (1 - y_i) \log(1 - P(y|x_i))$$

Model:




Training: ?

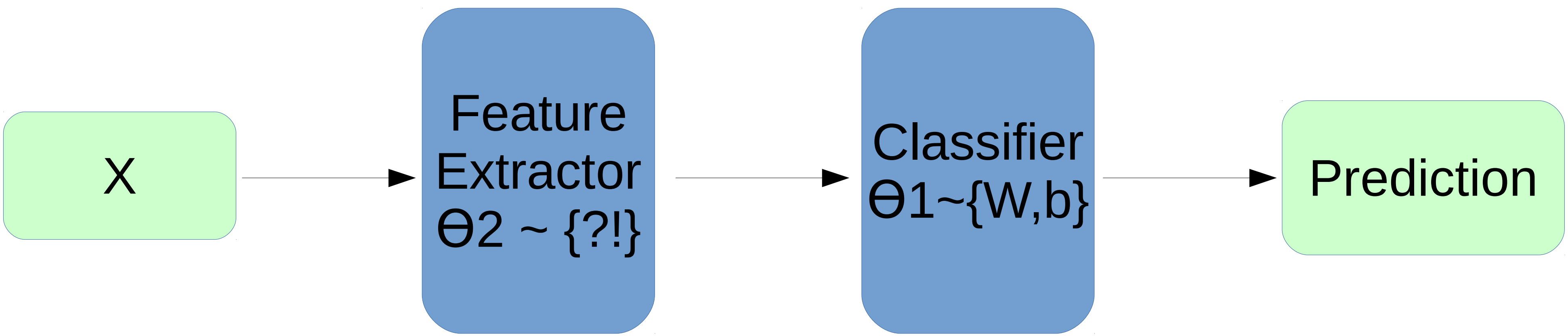
$$\underset{\theta_1}{argmin}\, L(y, P(y|x))$$

# What do we want, exactly?

Loss, for example:

$$L = -\sum_i y_i \log P(y|x_i) + (1 - y_i)\log(1 - P(y|x_i))$$

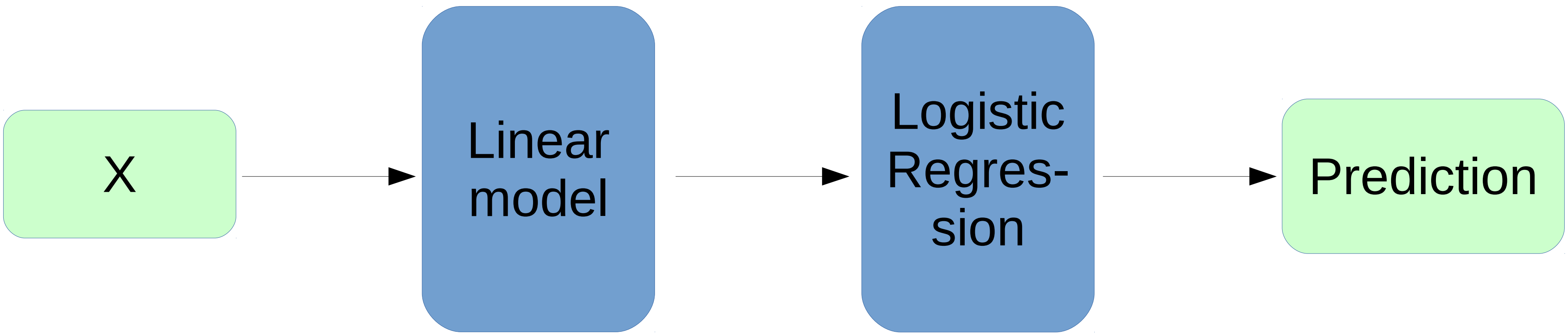
Model:




Gradients: $\underset{\theta_2}{argmin}\, L(y, P(y|x))$ $\underset{\theta_1}{argmin}\, L(y, P(y|x))$

# Try linear

Model:

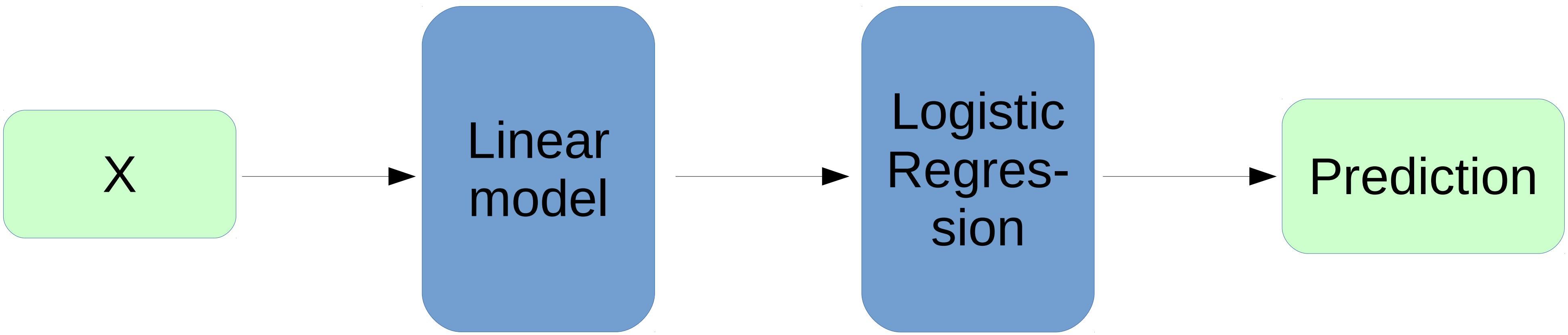



$$h_j = \sum_i w_{ij}^h x_i + b_j^h \quad j \in \{1, 2, \ldots, n\}$$

$$y_{pred} = \sigma\left(\sum_j w_j^o h_j + b^o\right)$$

# Try linear

Model:




$$h_j = \sum_i w_{ij}^h x_i + b_j^h$$

$$j \in \{1,2,\ldots,n\}$$

$$y_{pred} = \sigma\left(\sum_j w_j^o h_j + b^o\right)$$

Output:

$$P(y|x) = \sigma\left(\sum_j w_j^o \left(\sum_i w_{ij}^h x_i + b_j^h\right) + b^o\right)$$



Is it any better than logistic regression?

# Try linear

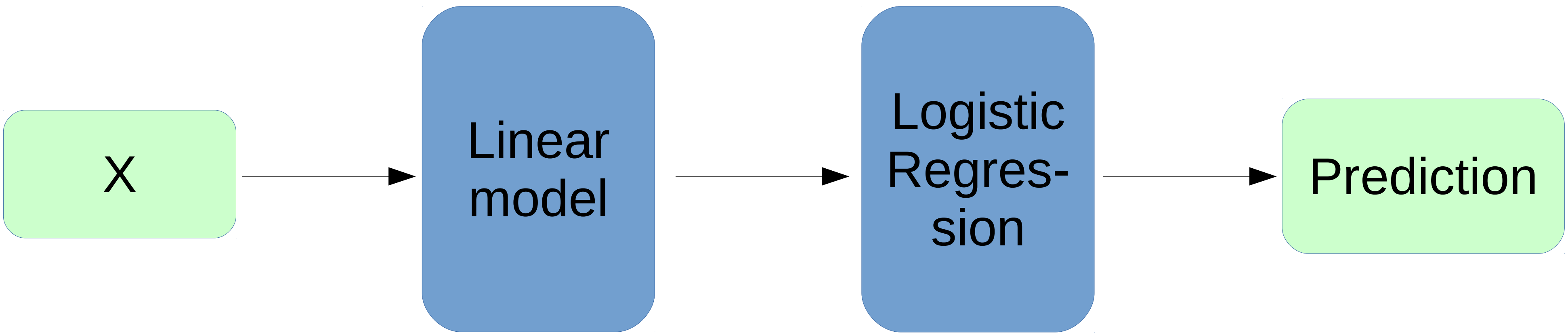
$$P(y|x)=\sigma\left(\sum_j w_j^o\left(\sum_i w_{ij}^h x_i+b_j^h\right)+b^o\right)$$

$$w'_i=\sum_j w_j^o w_{ij}^h \qquad b'=\sum_j w_j^o b_j^h+b^o$$

$$P(y|x)=\sigma\left(\sum_i w'_i x_i+b'\right)$$

# Try linear

Model:




$$h_j = \sum_{\substack{i \\ j \in \{1,2,\ldots,n\}}} w_{ij}^h x_i + b_j^h \qquad y_{pred} = \sigma\left(\sum_j w_j^o h_j + b^o\right)$$

Output:

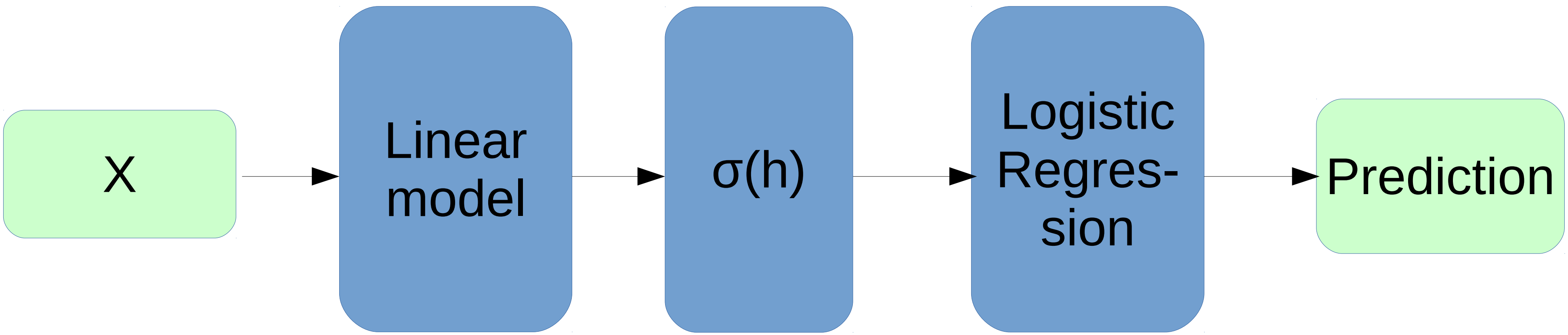
$$P(y|x) = \sigma\left(\sum_j w_j^o \left(\sum_i w_{ij}^h x_i + b_j^h\right) + b^o\right)$$



Is it any better than logistic regression?

# Nonlinearity

## Model:

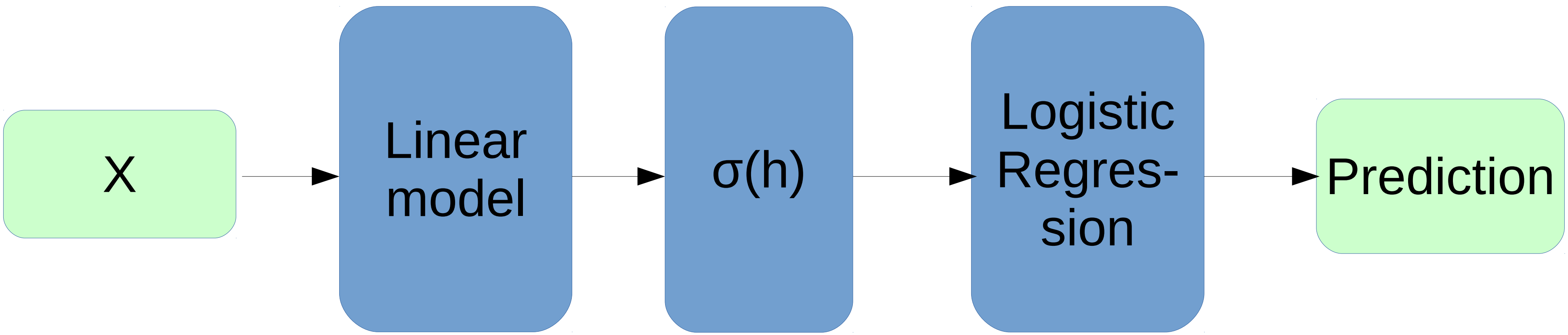



$$h_j = \sigma\left(\sum_{i} w_{ij}^{h} x_i + b_j^{h}\right) \qquad y_{pred} = \sigma\left(\sum_{j} w_j^{o} h_j + b^{o}\right)$$

$$j \in \{1, 2, \dots, n\}$$

# Nonlinearity

Model:




$$h_j = \sigma\left(\sum_i w_{ij}^h x_i + b_j^h\right) \qquad y_{pred} = \sigma\left(\sum_j w_j^o h_j + b^o\right)$$

$$j \in \{1, 2, \dots, n\}$$

Output:

$$P(y|x) = \sigma\left(\sum_j w_j^o \sigma\left(\sum_i w_{ij}^h x_i + b_j^h\right) + b^o\right)$$

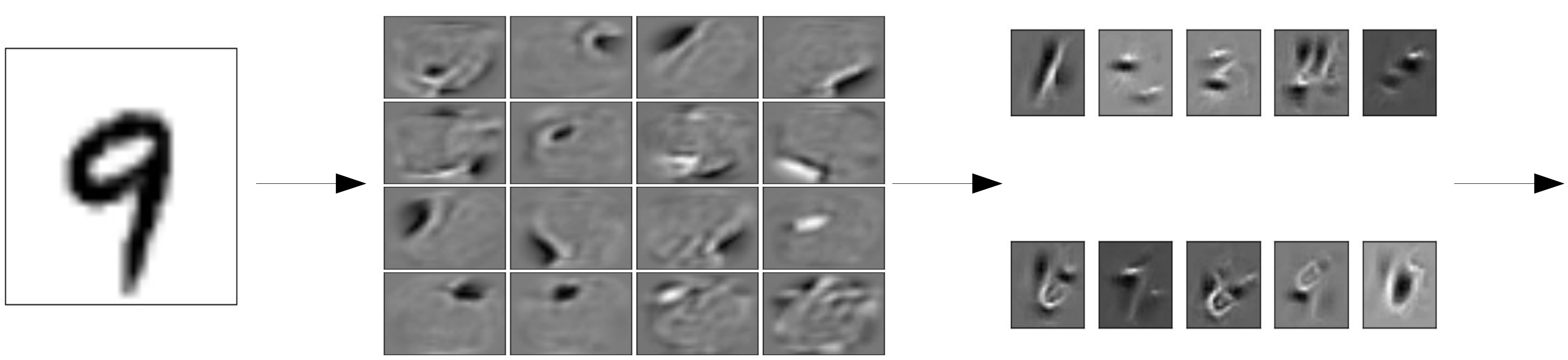
# Nonlinearity

Model:

X → Linear model → σ(h) → Logistic Regres-sion → Prediction

$$h_j=\sigma\left(\sum_{i\in\{1,2,\dots,n\}} w_{ij}^h x_i+b_j^h\right)$$

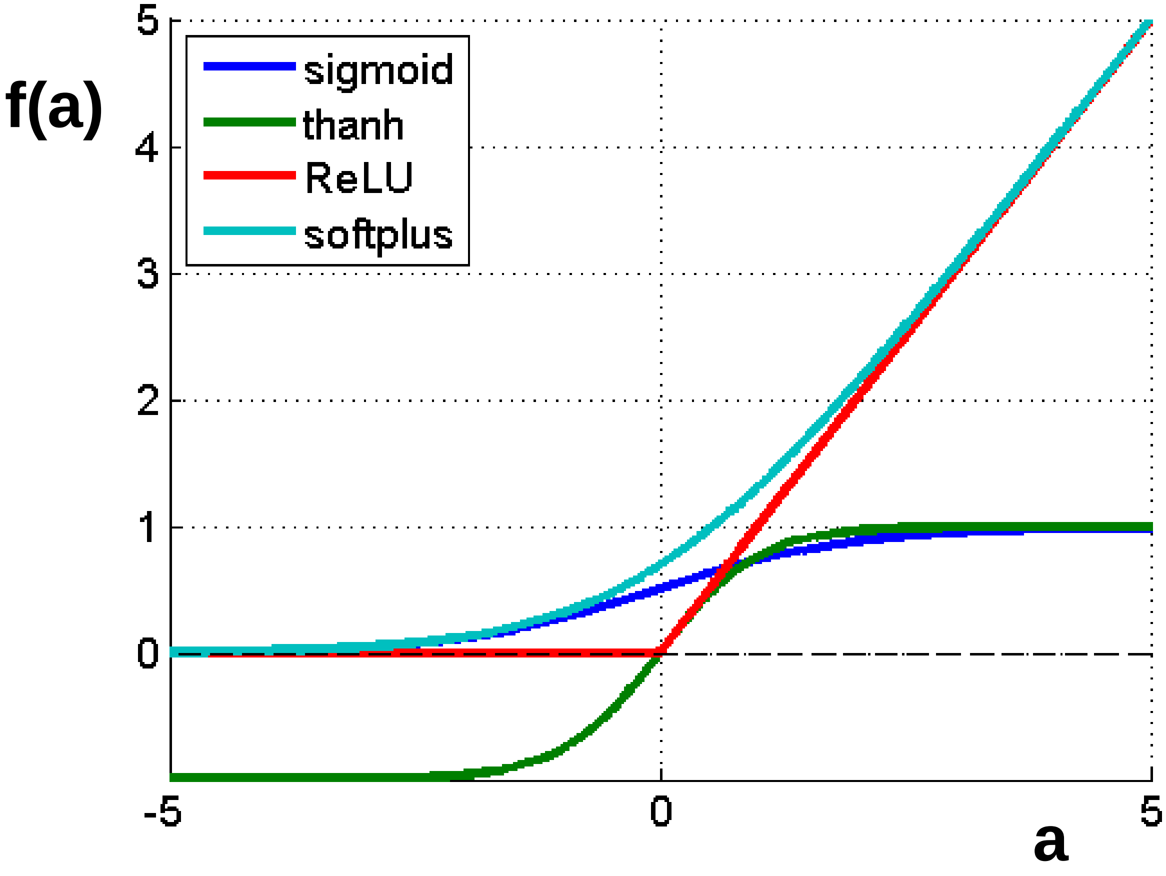
$$y_{pred}=\sigma\left(\sum_j w_j^o h_j+b^o\right)$$



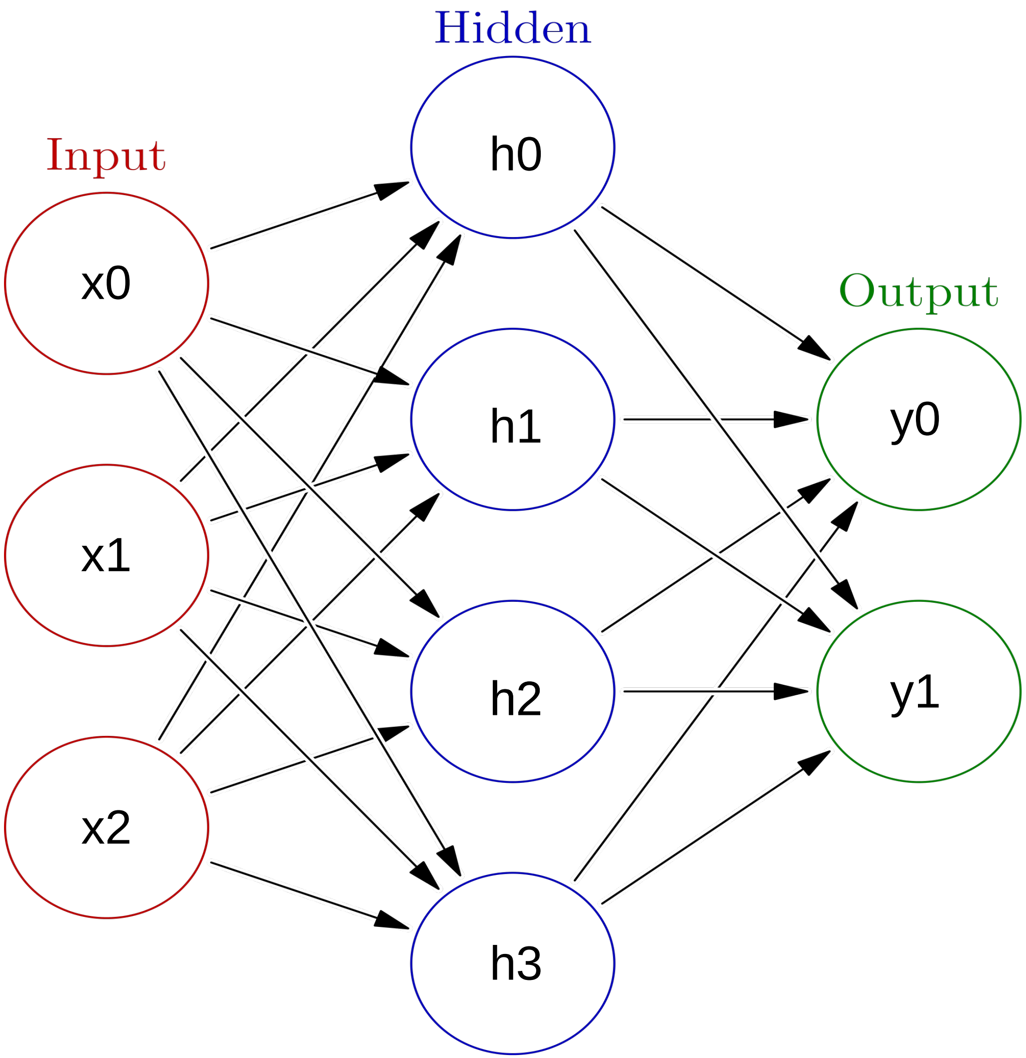
P(“1”)
P(“2”)
...
P(“9”)

# Nonlinearity

- f(a) = 1/(1+e^a)
- f(a) = tanh(a)
- f(a) = max(0,a)
- f(a) = log(1+e^a)



# Initialization, symmetry problem

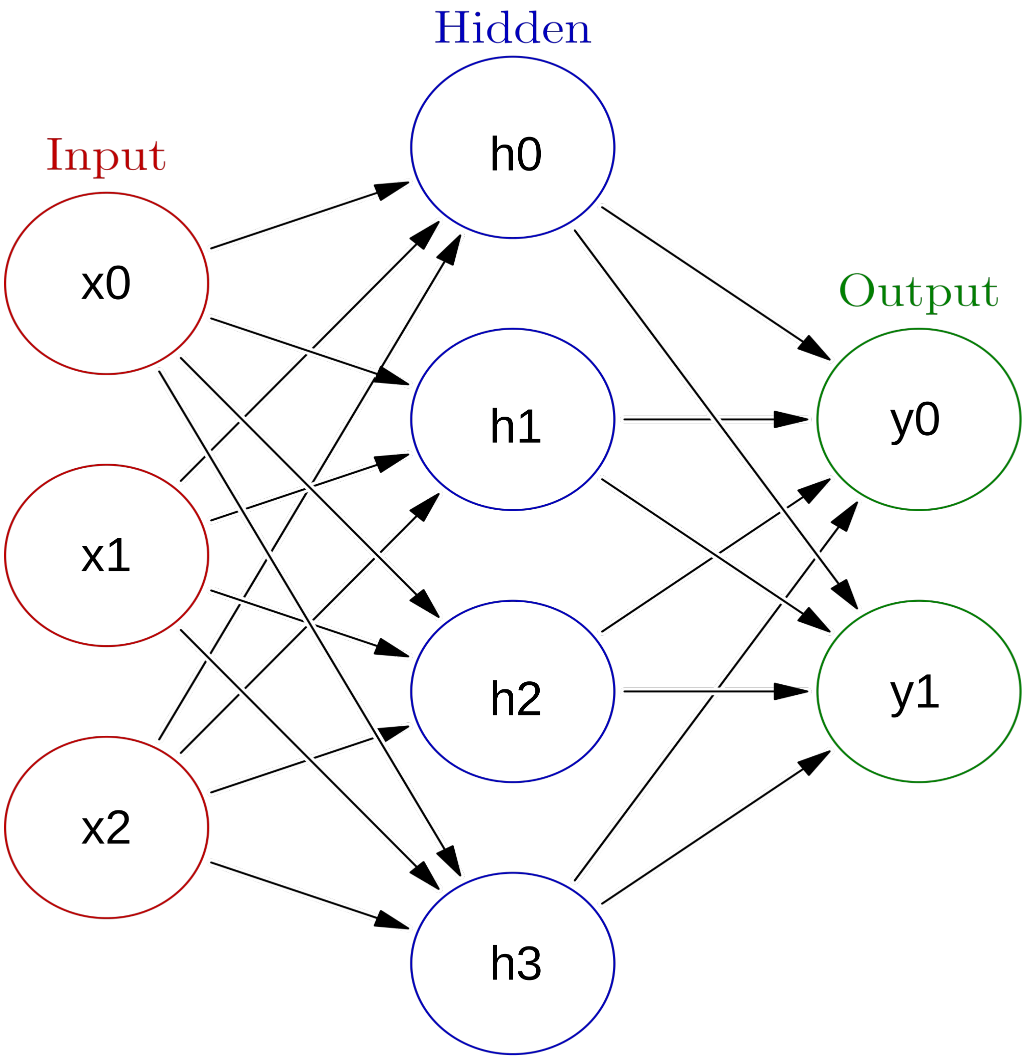



- Initialize with zeros

  $W \leftarrow 0$

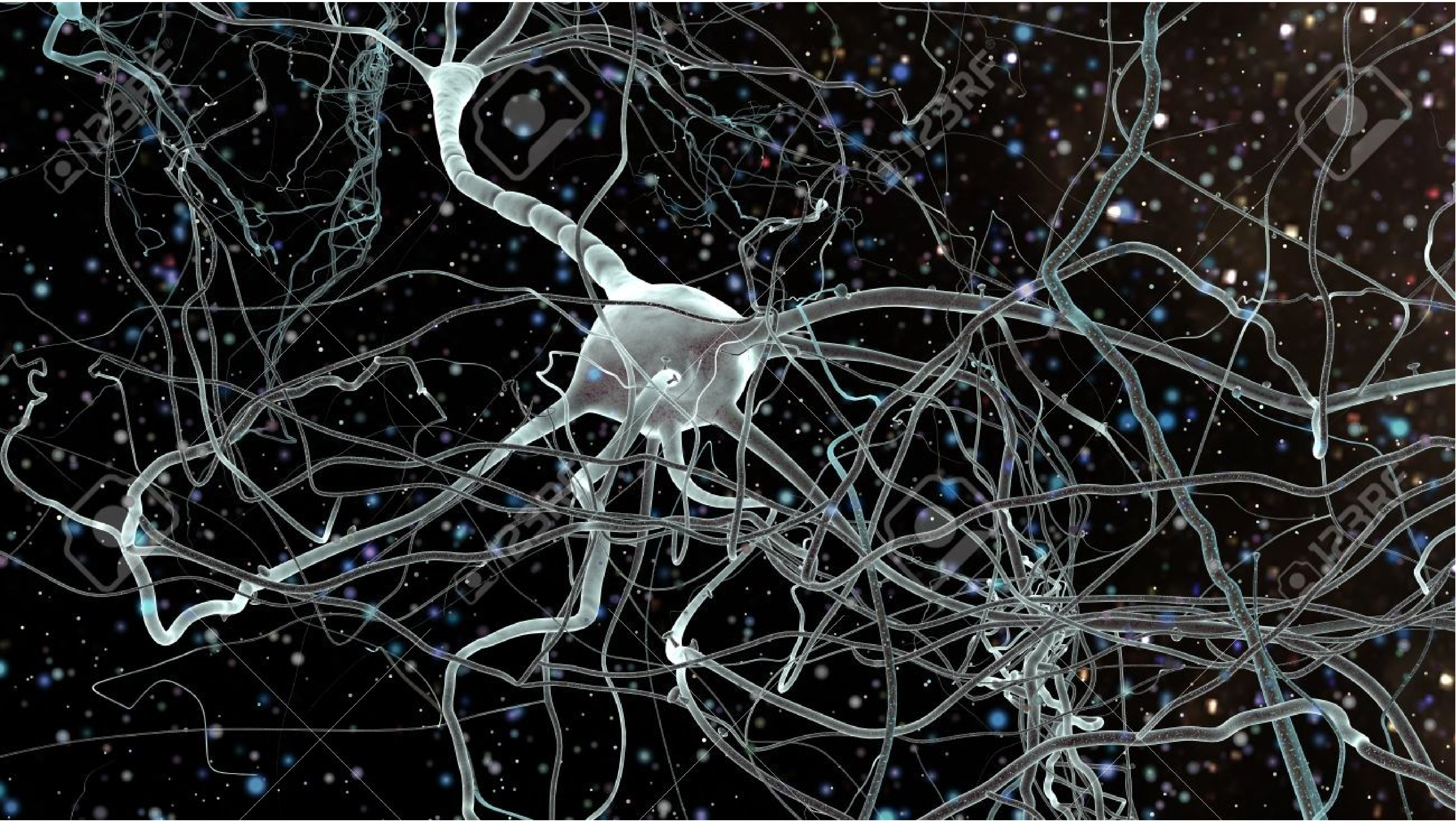
- What will the first step look like?

# Initialization, symmetry problem

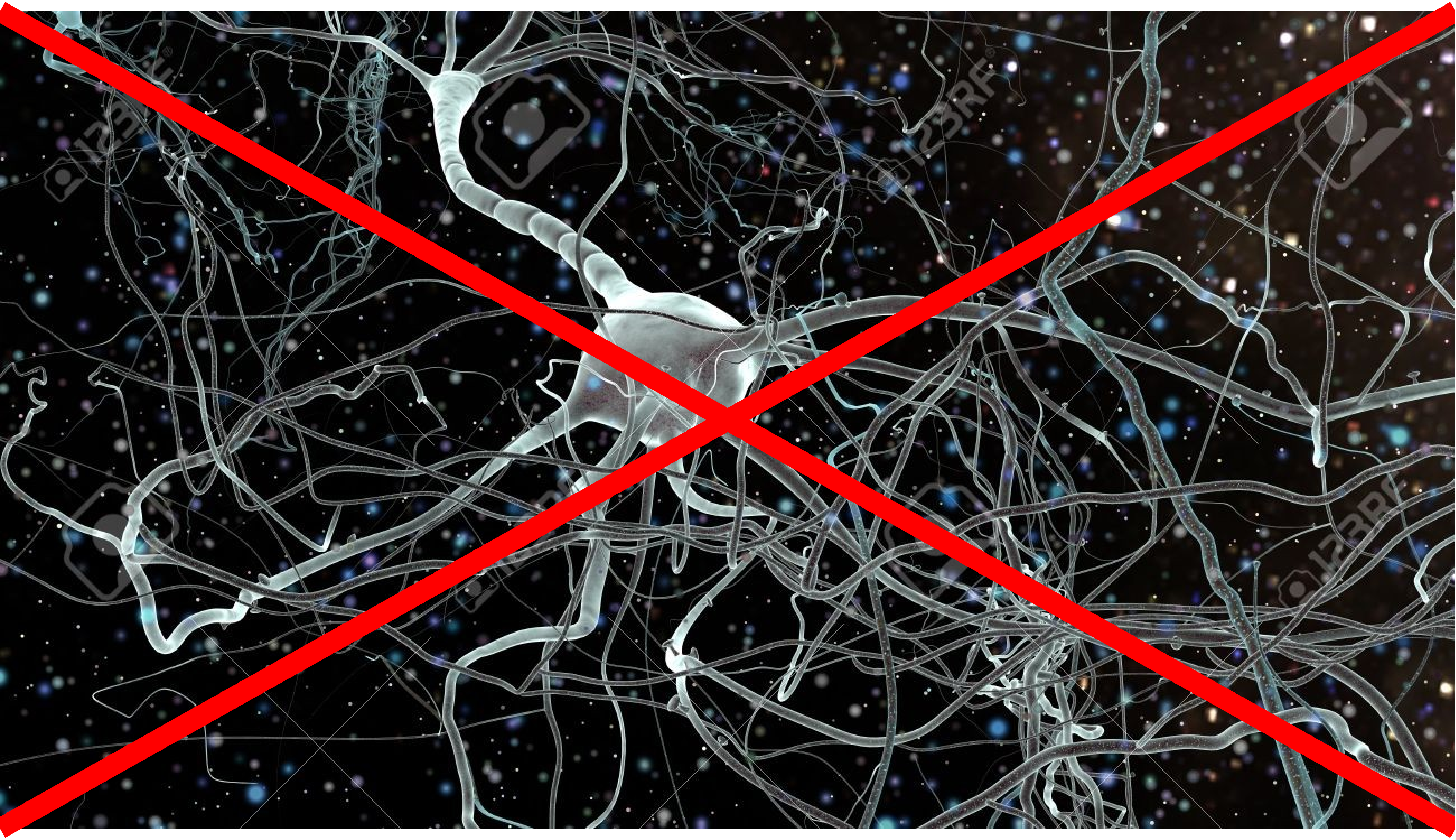



- Break the symmetry!
- Initialize with random numbers!
  $W \leftarrow N(0,0.01)$?
  $W \leftarrow U(0,0.1)$?
- Can get a bit better for deep NNs

# Biological inspiration



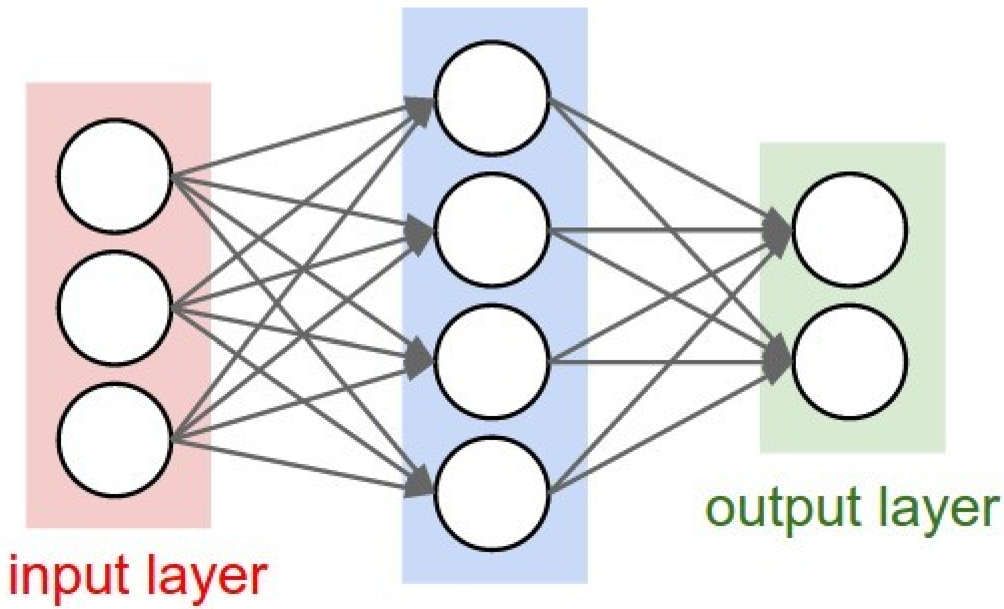
# Biological inspiration



# Connectionist phrasebook

- Layer – a building block for NNs :
  - “Dense layer”: $f(x) = Wx+b$
  - “Nonlinearity layer”: $f(x) = \sigma(x)$
  - Input layer, output layer
  - A few more we gonna cover later
- Activation – layer output
  - i.e. some intermediate signal in the NN
- Backpropagation – a fancy word for “chain rule”

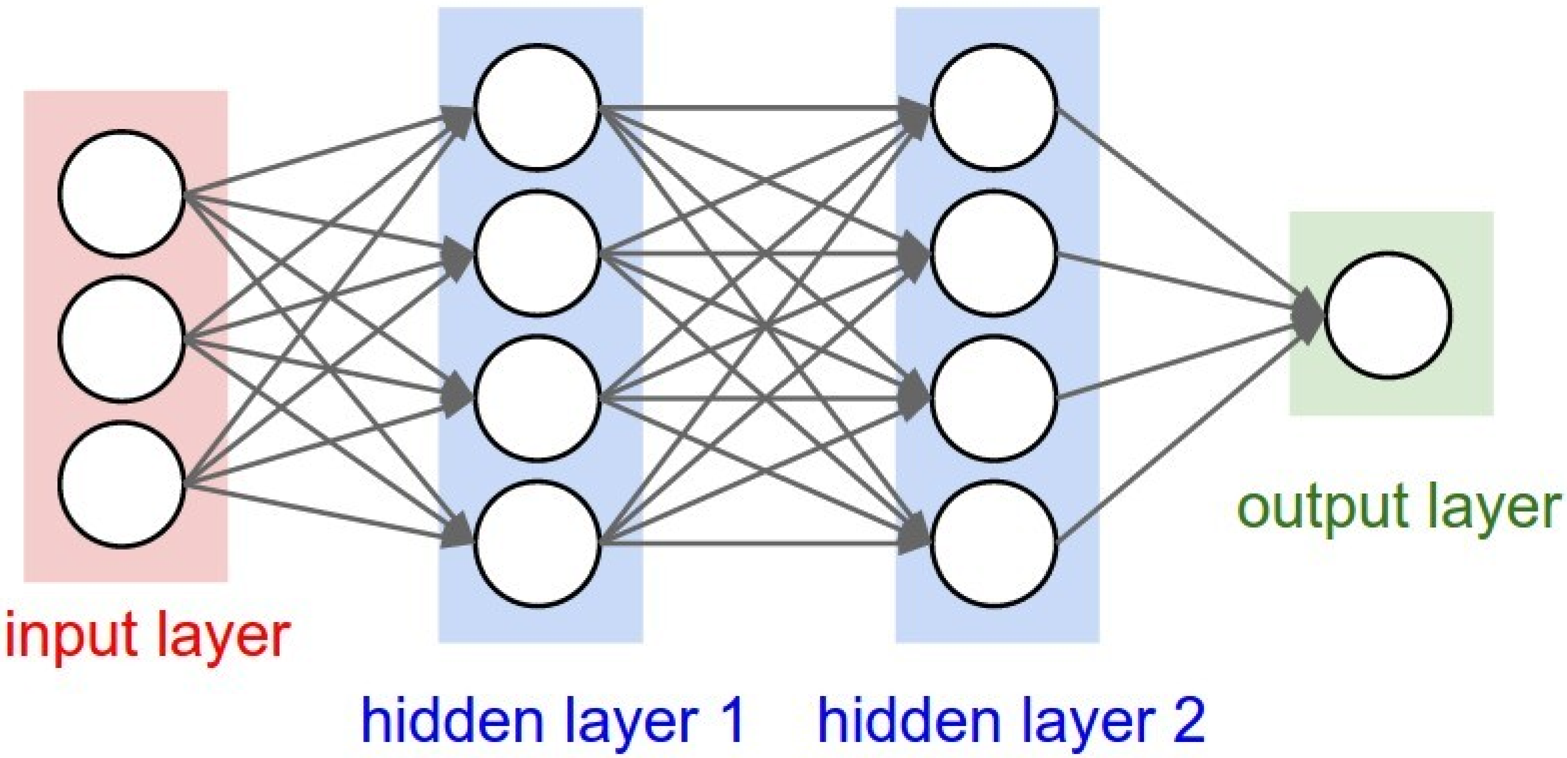
# Connectionist phrasebook



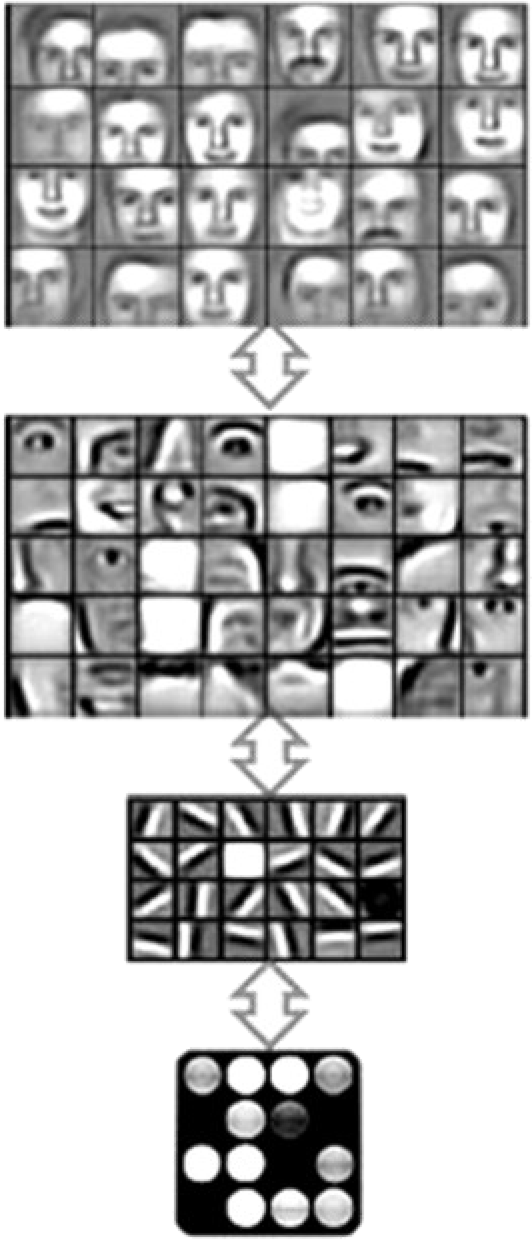

Dense layer
with a nonlinearity

- “Train it via backprop!”

# Connectionist phrasebook




How do we train it?



**Discrete Choices**

⋮

**Layer 2 Features**

**Layer 1 Features**

**Original Data**

# Potential caveats?

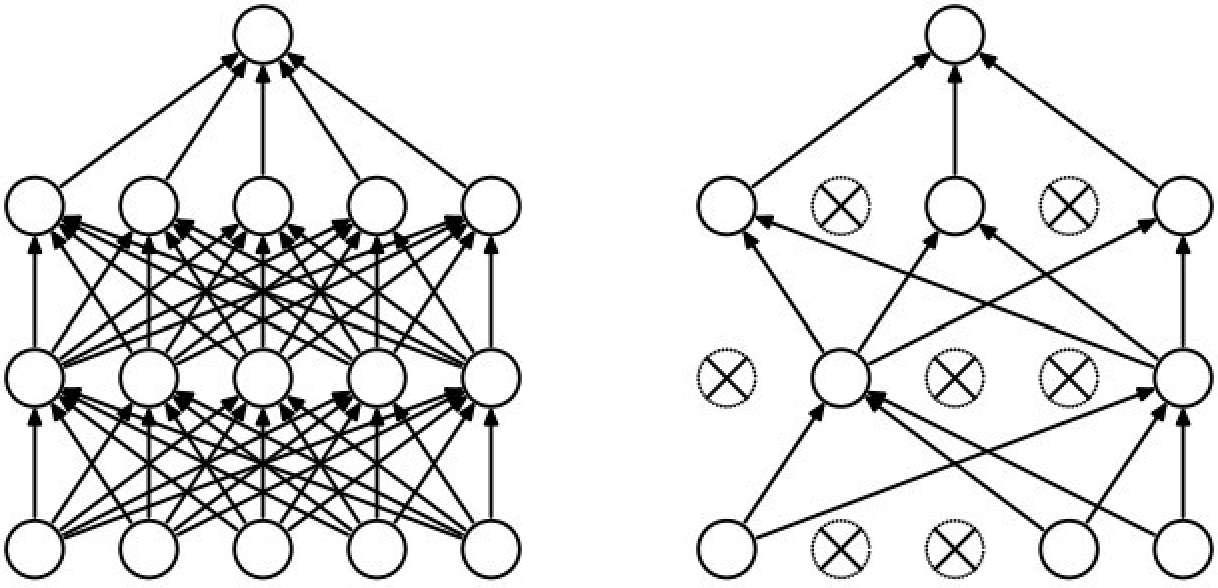
# Potential caveats?

- Hardcore overfitting
- No “golden standard” for architecture
- Computationally heavy

# Regularization

- L1, L2, as usual

- Dropout



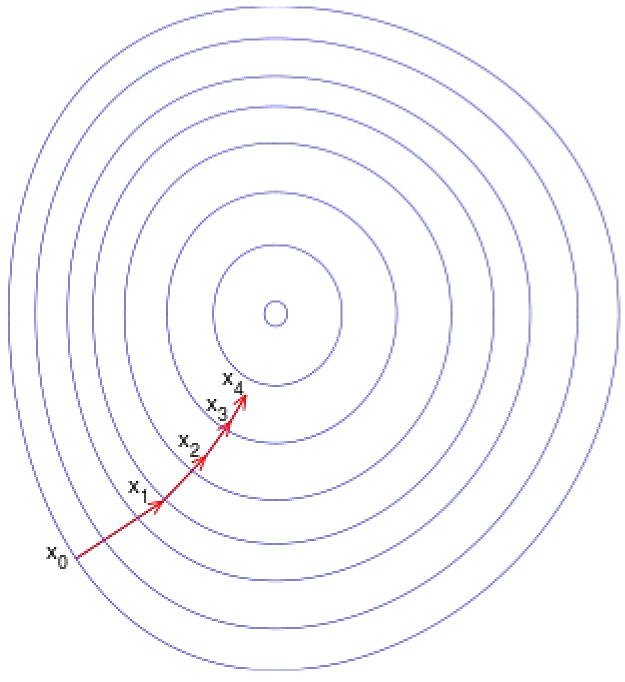
(a) Standard Neural Net (b) After applying dropout.

# Faster than gradient descent

Update:

$$w_{i+1} \leftarrow w_i - \alpha \frac{\partial L}{\partial w}$$

- a – learning rate a<<1
- L – loss function




**Can we do better?**

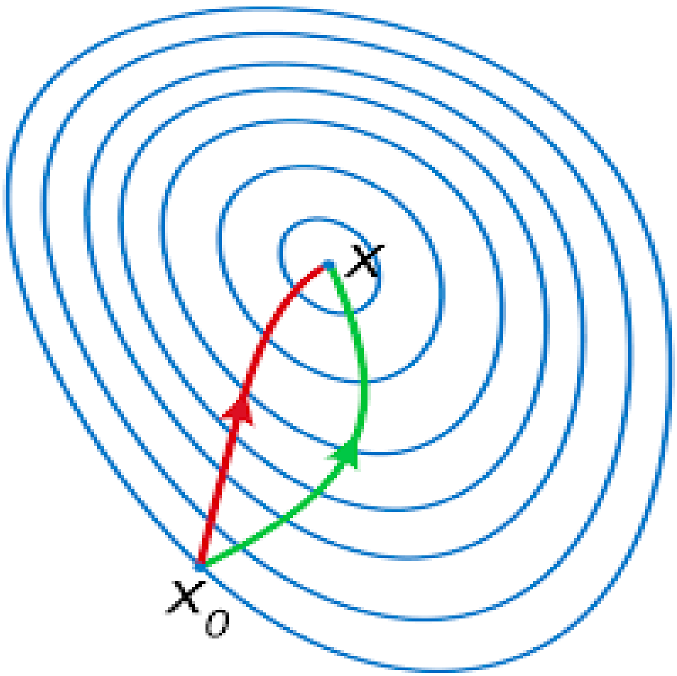
# Newton-Raphson

Parameter update

$$w_{i+1} \leftarrow w_i - \alpha H_L^{-1} \frac{\partial L}{\partial w}$$

Hessian:

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1\,\partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1\,\partial x_n} \\ \frac{\partial^2 f}{\partial x_2\,\partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2\,\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n\,\partial x_1} & \frac{\partial^2 f}{\partial x_n\,\partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$




Red: Newton-Raphson
Green: gradient descent

Any drawbacks?

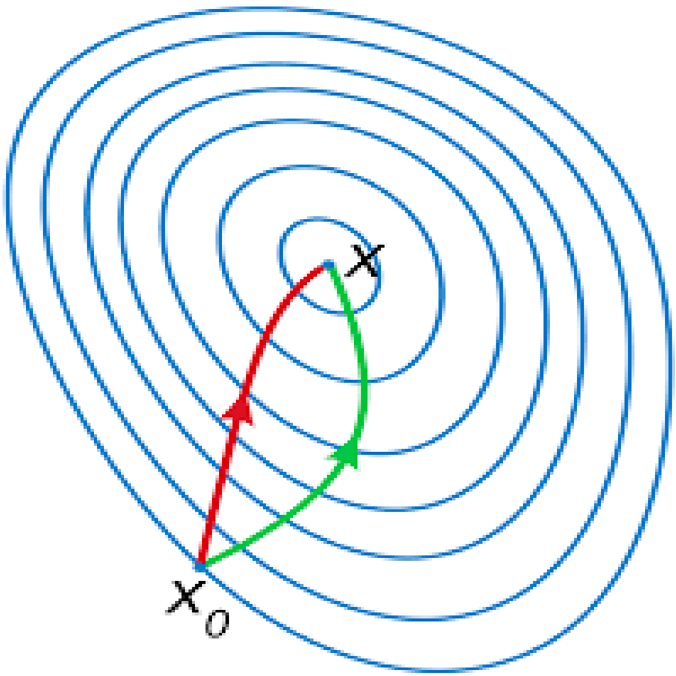
# Newton-Raphson

Parameter update

$$w_{i+1} \leftarrow w_i - \alpha H_L^{-1} \frac{\partial L}{\partial w}$$

Hessian:

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1\,\partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1\,\partial x_n} \\ \frac{\partial^2 f}{\partial x_2\,\partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2\,\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n\,\partial x_1} & \frac{\partial^2 f}{\partial x_n\,\partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$




Red: Newton-Raphson
Green: gradient descent

**LARGE hessian!**

# Stochastic gradient descent

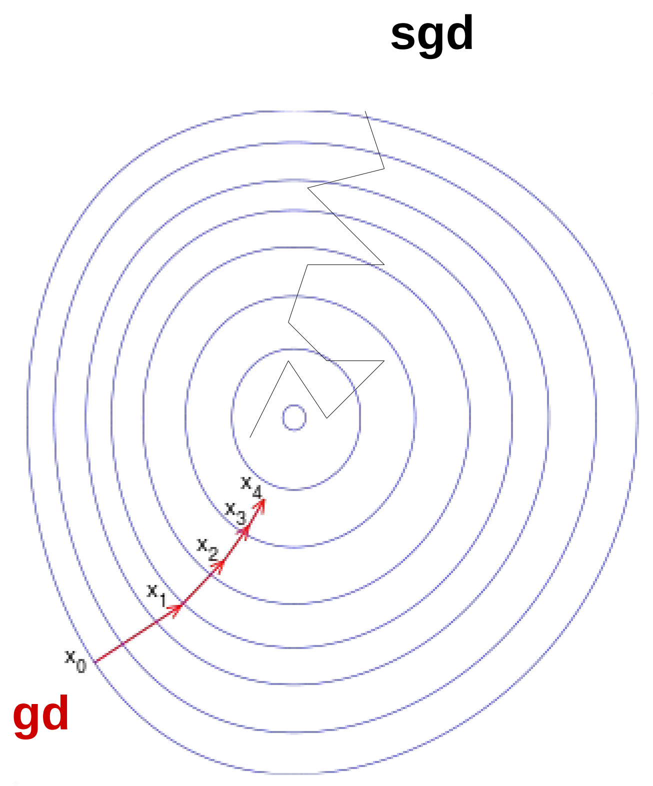
Loss function is mean over all data samples.

Approximate with 1 or few random samples.

Update:

$$w_{i+1} \leftarrow w_i - \alpha E \frac{\partial L}{\partial w}$$

- E – expectation
- Learning rate should decrease



# SGD with momentum

Idea: move towards "overall gradient direction",
Not just current gradient.

$$w_0 \leftarrow 0 ; v_0 \leftarrow 0$$

$$\nu_{i+1} \leftarrow \alpha \frac{\partial L}{\partial w} + \mu \nu_i$$

$$w_{i+1} \leftarrow w_i - \nu_{i+1}$$

Helps for noisy gradient / canyon problem

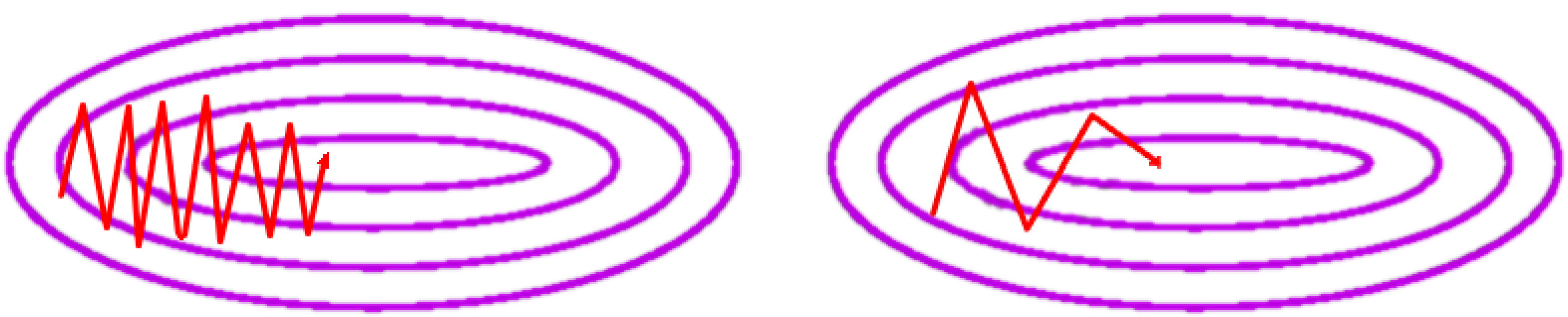
# SGD with momentum

Idea: move towards "overall gradient direction",
Not just current gradient.

$$w_0 \leftarrow 0 ; v_0 \leftarrow 0$$

$$\nu_{i+1} \leftarrow \alpha \frac{\partial L}{\partial w} + \mu \nu_i$$

$$w_{i+1} \leftarrow w_i - \nu_{i+1}$$



# AdaGrad

Idea: decrease learning rate individually for each parameter in proportion to sum of it's gradients so far.

$$G_t = \sum_{\tau=1}^{t} \left[ \frac{\partial L}{\partial w} \right]^2$$

"Total update path length"
(for each parameter)

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \frac{\partial L}{\partial w}$$

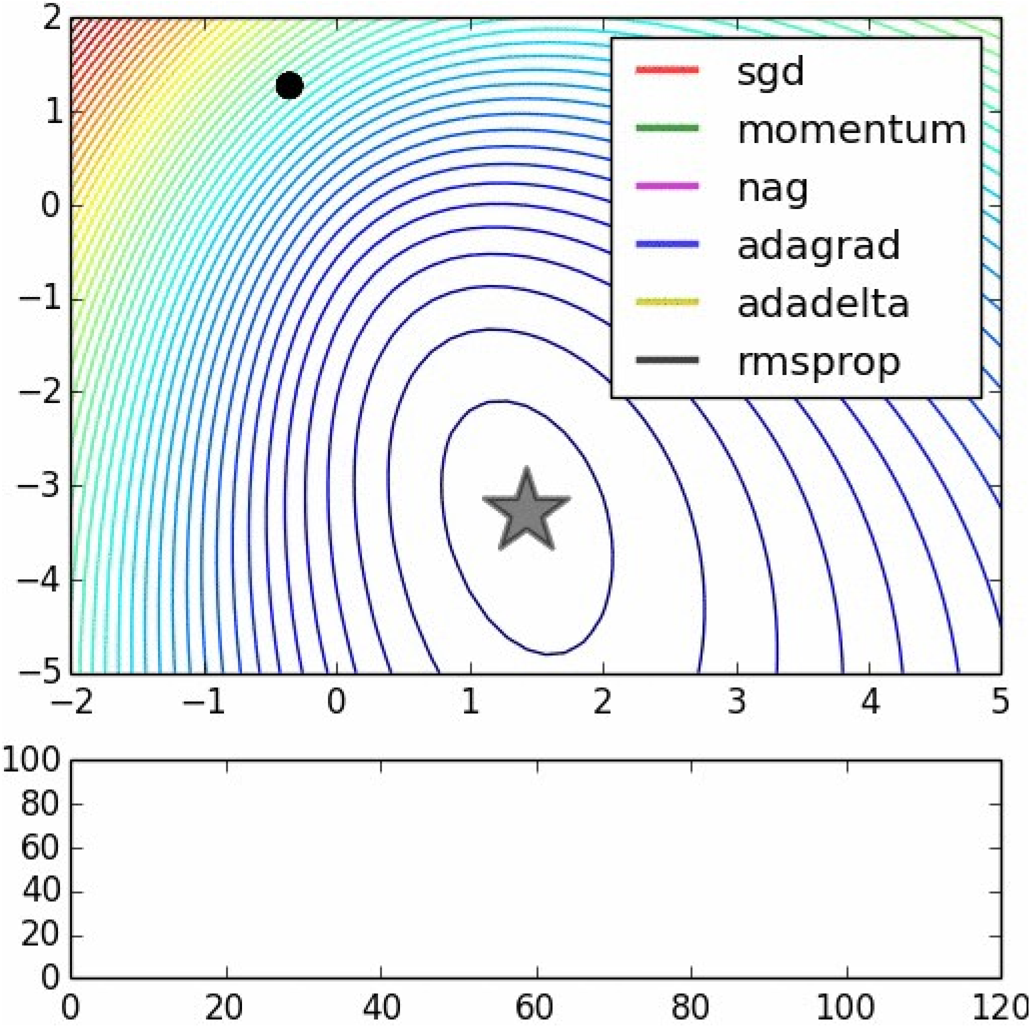
# RMSProp

Idea: make sure all gradient steps have approximately same magnitude (by keeping moving average of magnitude)

$$ms_{t+1} = \gamma \cdot ms_t + (1-\gamma) \| \frac{\partial L}{\partial w} \|^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{ms + \epsilon}} \frac{\partial L}{\partial w}$$

# Alltogether



# Moar stuff

**Without Hessian**

- Adadelta ~ adagrad with window
- Adam ~ rmsprop + momentum
- Nesterov-momentum
- Hessian-free (narrow)
- Conjugate gradients

**Estimate inverse Hessian**

- BFGS
- L-BFGS
- ****-BFGS

# Regularization (weight)

General idea:

$$L_{new} = L + reg$$

performance = how_i_fit_data + how_reasonable_i_am

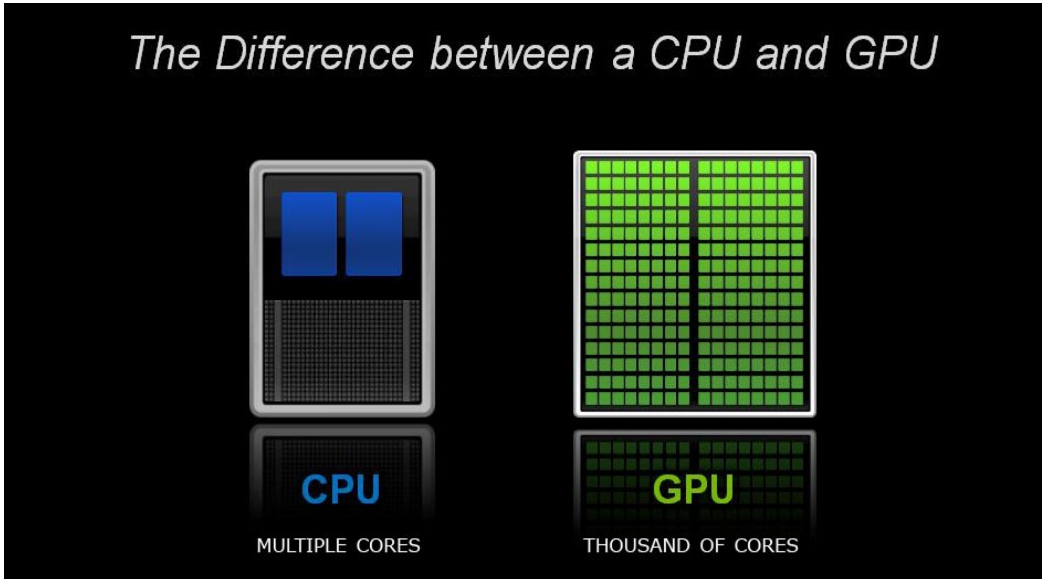
L2 regularizer

$$L_{new} = L + \beta \|\theta\|_2 = L + \beta \sum_i \theta_i^2$$

linear models: theta = {w,b}

- a.k.a. weight decay
- a.k.a. Tikhonov regularizer
- a.k.a. normal prior on params

# Computation

# Nuff

**Let's code some neural networks!**