CS 260: Foundations of Data Science

Prof. Sara Mathieson

Fall 2023

HAVERFORD COLLEGE

Admin

- Exam will be handed back on Tuesday

- Last candidate talk at **4:15pm TODAY**
  - Tea at 4pm
  - Student lunch Friday 12:30-1:30pm

Outline for November 30

- Gaussian Mixture Models (GMMs)
- Kernel Density Estimation (KDE)
- Missing data
- Begin: neural networks

Outline for November 30

- Gaussian Mixture Models (GMMs)
- Kernel Density Estimation (KDE)
- Missing data
- Begin: neural networks

Gaussian Mixture Models | $\boxed{\text{E-step}}$ "soft" assignment

- π_k = prob of class k
  $= \frac{1}{K}$ for all k
- $\vec{\mu}_k$ = mean of cluster k
  = random points
- σ_k^2 = variance of cluster k
  = sample variance of pts closest to each mean.

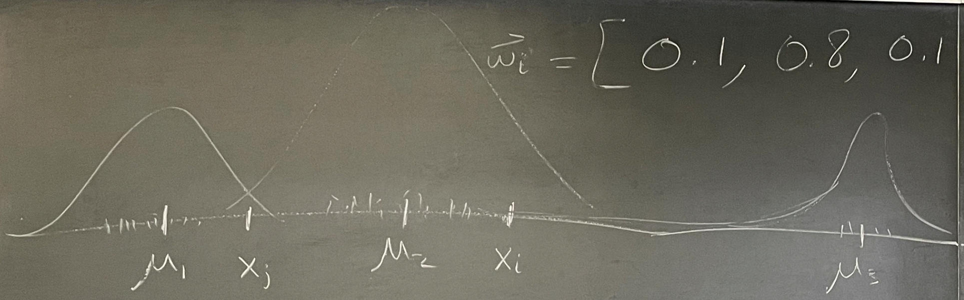
Initialization

w_{ik} = prob that $\vec{x}_i$ came from cluster k

$$= p(k \mid \vec{x}_i) = \frac{p(k)\, p(\vec{x}_i \mid k)}{p(\vec{x}_i)}$$

$$= \frac{\pi_k N(\vec{x}_i ; \vec{\mu}_k, \sigma_k^2)}{\sum_{j=1}^{K} \pi_j N(\vec{x}_i ; \vec{\mu}_j, \sigma_j^2)}$$

$\vec{w}_i = [0.1, 0.8, 0.1]$



$\vec{w}_j = [0.49, 0.49, 0.02]$

$$W = \begin{array}{c} \begin{array}{ccc} \mu_1 & \mu_2 & \mu_3 \end{array} \\ \begin{bmatrix} 0.1 & 0.8 & 0.1 \\ 0.49 & 0.49 & 0.02 \\ \vdots & \vdots & \vdots \end{bmatrix}_{n \times K} \end{array} \begin{array}{l} \to 1 \\ \to 1 \\ \\ \end{array}$$

$\boxed{\text{M-step}}$ param update

$$M_k = \sum_{i=1}^{n} w_{ik} \quad \Big\} \text{ \# pts assigned to cluster } k$$

① $\pi_k = \dfrac{M_k}{n}$

② $\vec{\mu}_k = \dfrac{1}{M_k} \sum_{i=1}^{n} w_{ik} \vec{x}_i$

K-means

$$\cancel{\frac{1}{|C_k|} \sum_{\vec{x}_i \in C_k} \vec{x}_i}$$

③ σ_k^2 = weighted sample variance.

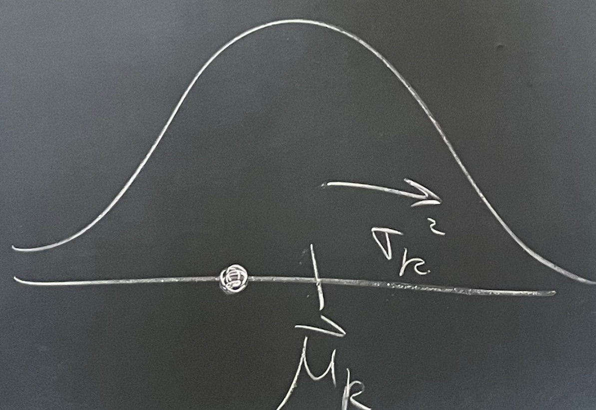
Generative process

① choose a cluster k

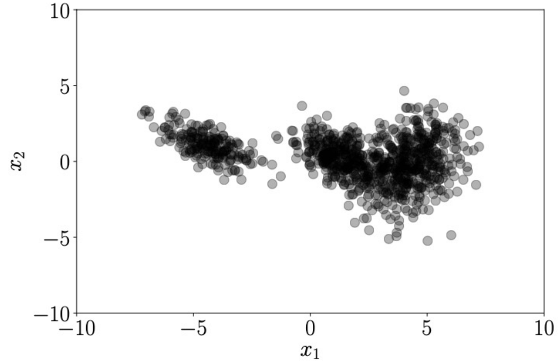
using $[\pi_1, \pi_2 \cdots \pi_K]$

② use $\vec{\mu}_k$ & σ_k^2 to sample

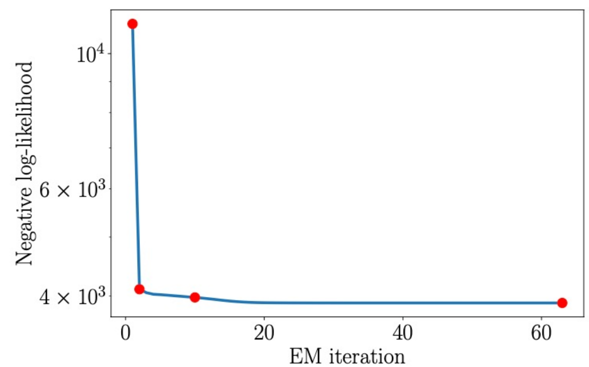
from $N(\vec{\mu}_k, \sigma_k^2)$



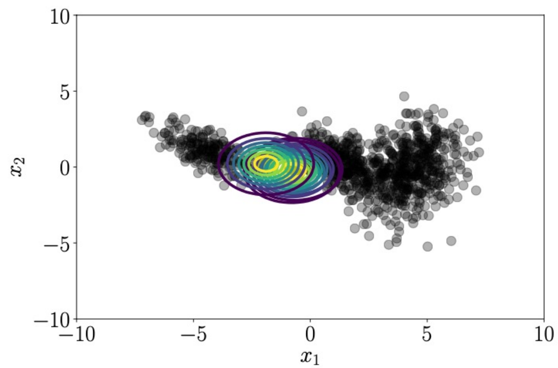
$$\frac{x_i}{\sum_{j=1}^{n} x_j}$$



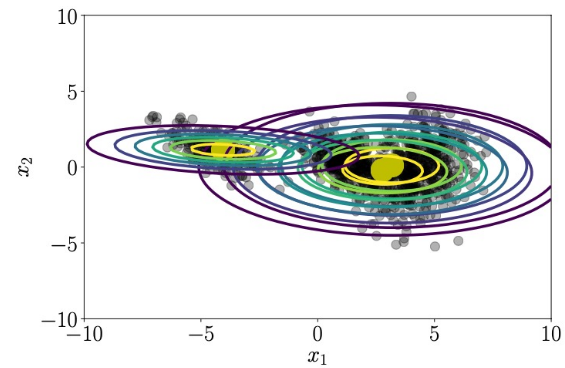
(a) Dataset.



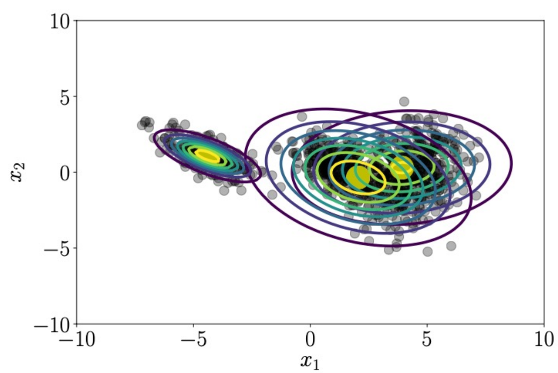
(b) Negative log-likelihood.



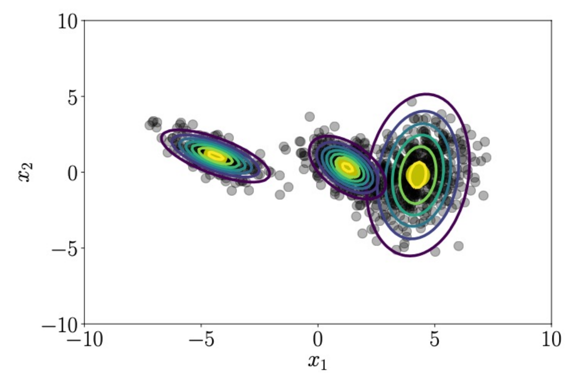
(c) EM initialization.



(d) EM after one iteration.



(e) EM after 10 iterations.



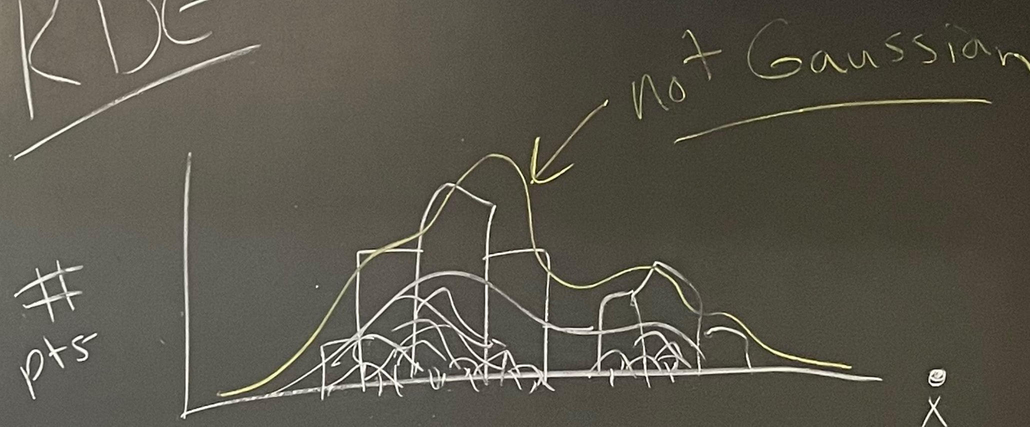
(f) EM after 62 iterations.

Figure 11.7 from MML textbook

Outline for November 30

- Gaussian Mixture Models (GMMs)
- Kernel Density Estimation (KDE)
- Missing data
- Begin: neural networks

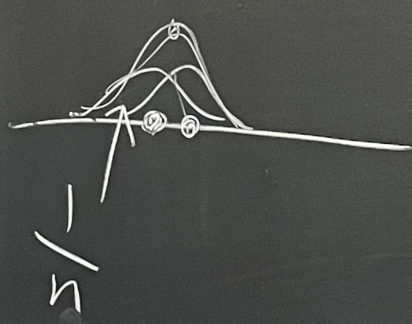
KDE



$$p(x) = \frac{1}{nh}\sum_{i=1}^{n} k\left(\frac{x - x_i}{h}\right)$$

kernel

width



KDE (Kernel Density Estimation)

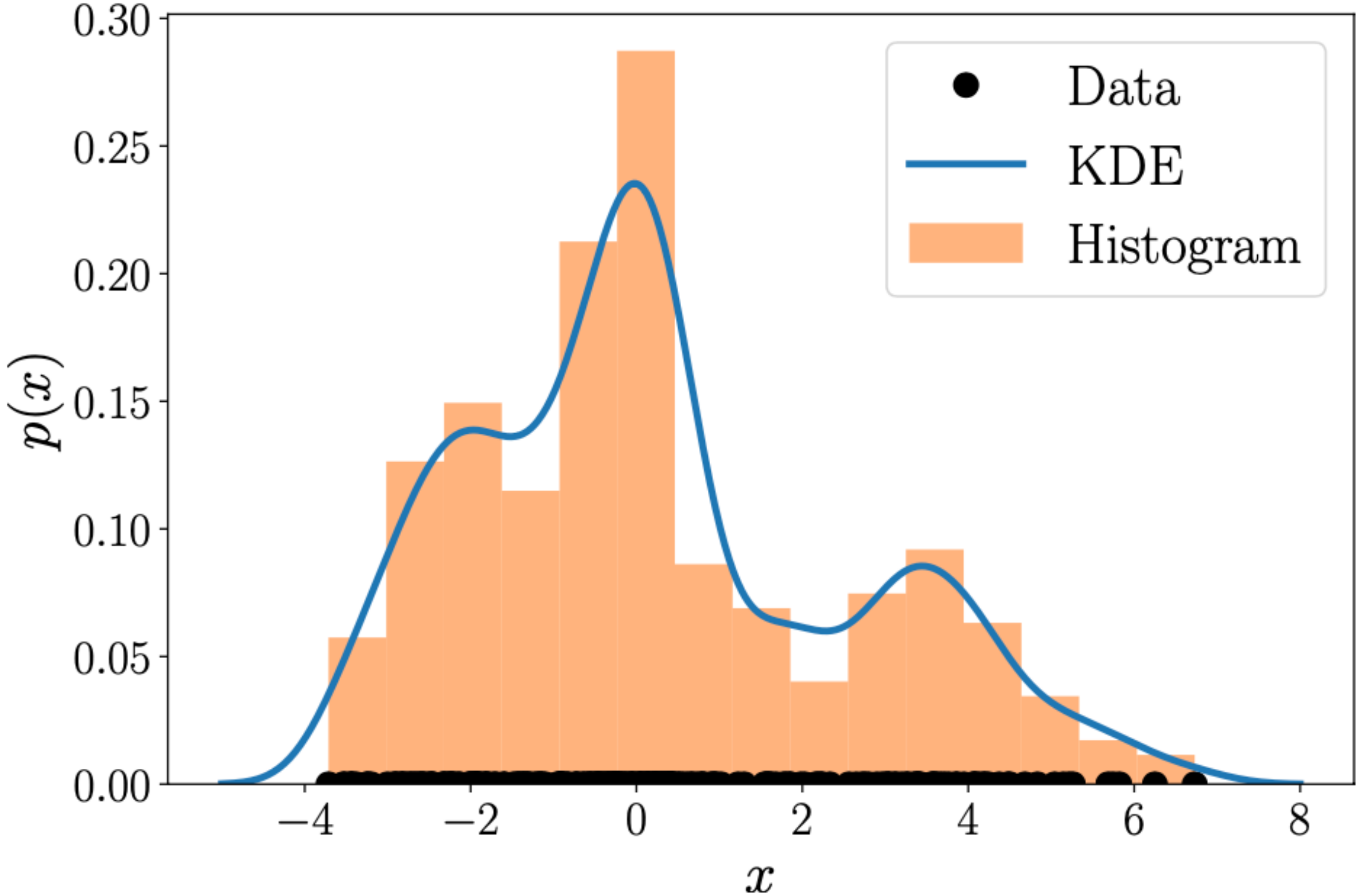


Figure 11.9 from MML textbook

Outline for November 30

- Gaussian Mixture Models (GMMs)
- Kernel Density Estimation (KDE)
- Missing data
- Begin: neural networks

Types of missing data

- MCAR: Missing Completely At Random. Not related to:
  - Specific values
  - Observed responses
- MAR: Missing At Random. Not related to:
  - Specific values
- MNAR: Missing Not At Random

Reference: "The prevention and handling of the missing data" Kang (2013)

Techniques for handling missing data

- Try to prevent the problem in the first place
  - Careful study design, follow-up with participants, etc
- Omit rows with missing data (reduces *n*)
- Omit only when value is needed
  - i.e. Naïve Bayes, per-feature estimates
- Mean substitution (per feature)

Reference: "The prevention and handling of the missing data" Kang (2013)

Techniques for handling missing data

- Imputation
  - Use similar examples to guess the missing values
  - Can be done locally or globally
- Last observation carried forward
  - Useful for time-series data

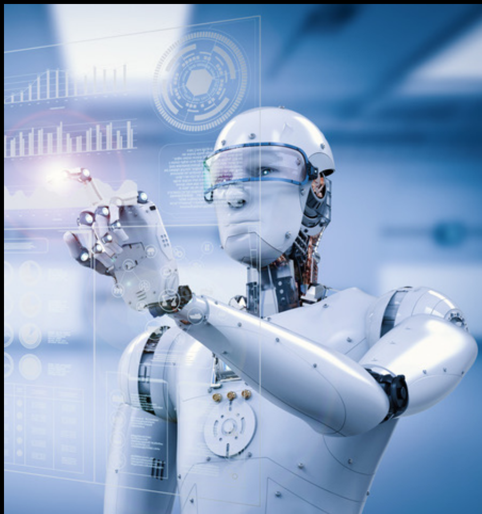
Reference: “The prevention and handling of the missing data” Kang (2013)

Outline for November 30

- Gaussian Mixture Models (GMMs)
- Kernel Density Estimation (KDE)
- Missing data
- Begin: neural networks

Adapted from: "Know Your Meme"

MACHINE LEARNING



What society thinks I do

$$\nabla_w \mathcal{L}(w,b,\alpha) = w - \sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)} = 0$$

This implies that

$$w = \sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)}.$$

As for the derivative with respect to b, we obtain

$$\frac{\partial}{\partial b}\mathcal{L}(w,b,\alpha) = \sum_{i=1}^{m} \alpha_i y^{(i)} = 0.$$

If we take the definition of w in Equation (9) and plug that back int[o]
Lagrangian (Equation 8), and simplify, we get

$$\mathcal{L}(w,b,\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2}\sum_{i,j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j (x^{(i)})^T x^{(j)} - b \sum_{i=1}^{m} \alpha_i y^{(i)}.$$

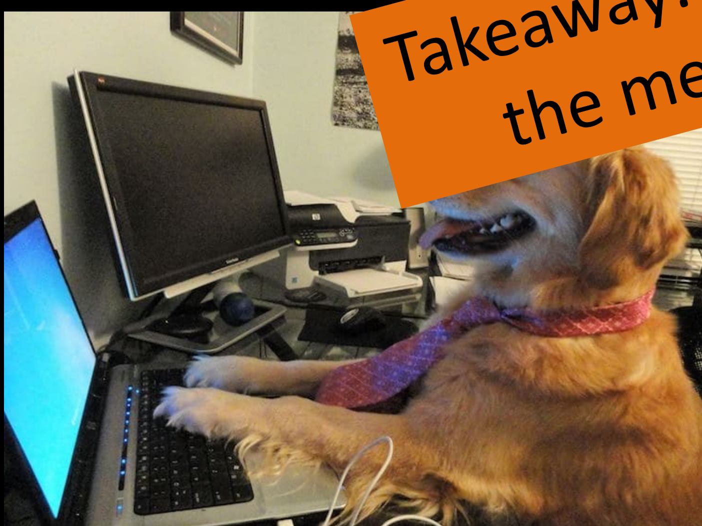
But from Equation (10), the last term must be zero, so we obtain

$$\mathcal{L}(w,b,\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2}\sum_{i,j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j \ldots$$

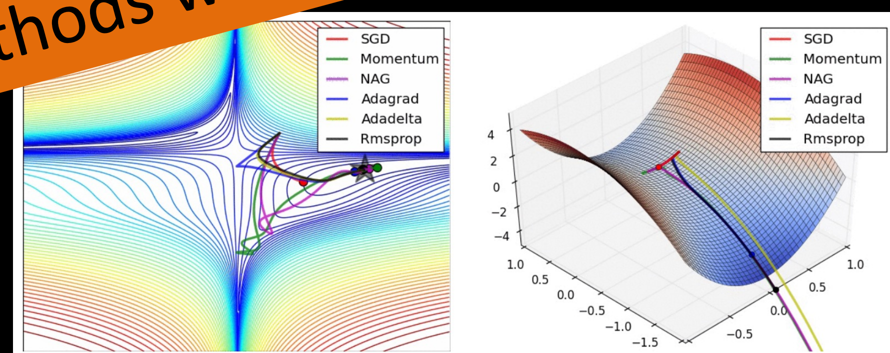
Wha[...]

[...]t other computer scientists think I do

Takeaway: we should understand the methods we are using!



What mathematicians think I do



What I think I do

```
>>> from sklearn import svm
>>> import tensorflow as tf
```

What I really do

Biological Inspiration

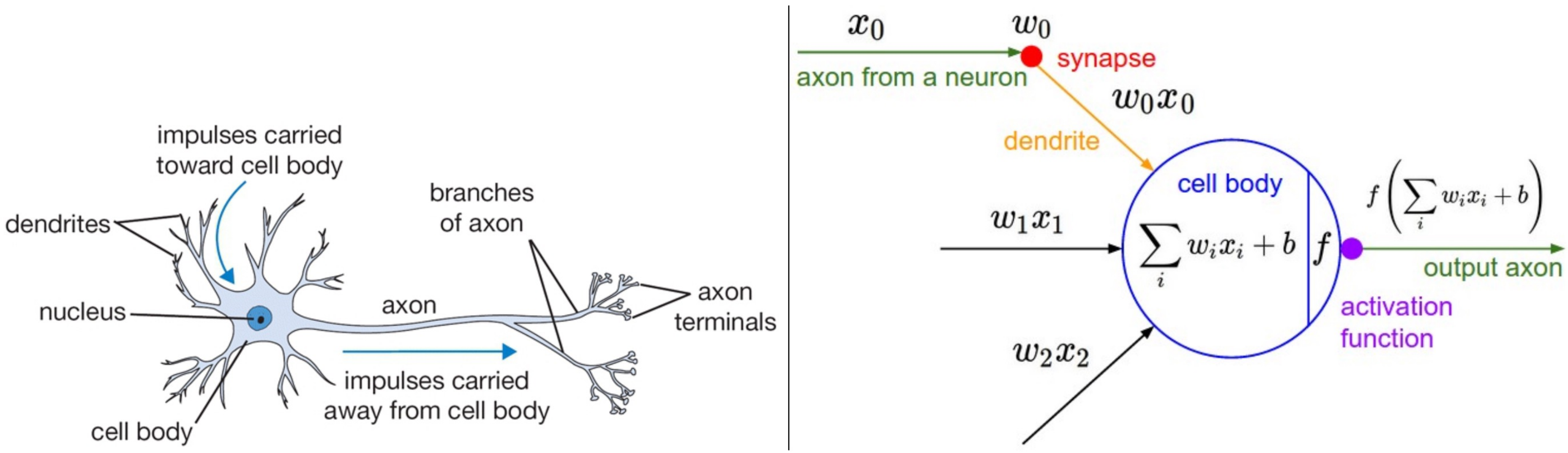


Figure: Stanford CS231n http://cs231n.github.io/neural-networks-1/

Goal: learn from complicated inputs

X_1

X_2

X_3

X_4

X_5

X_6

input data

?

Y_1 glasses?

Y_2 smiling?

Y_3 identity?

parameters

Image: Labeled Faces in the Wild (UMass)

Idea: transform data into lower dimension

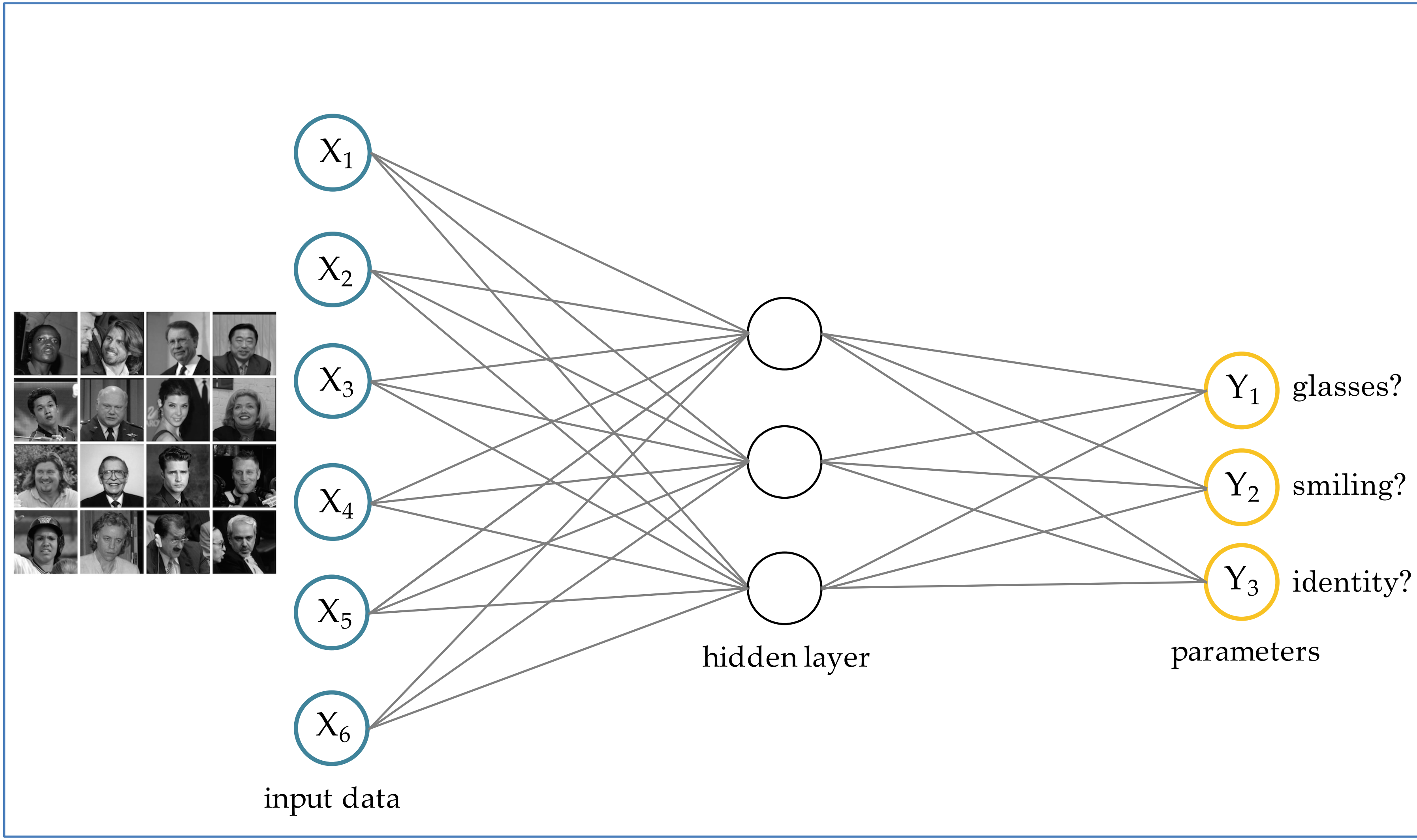


Image: Labeled Faces in the Wild (UMass)

Multi-layer networks = “deep learning”

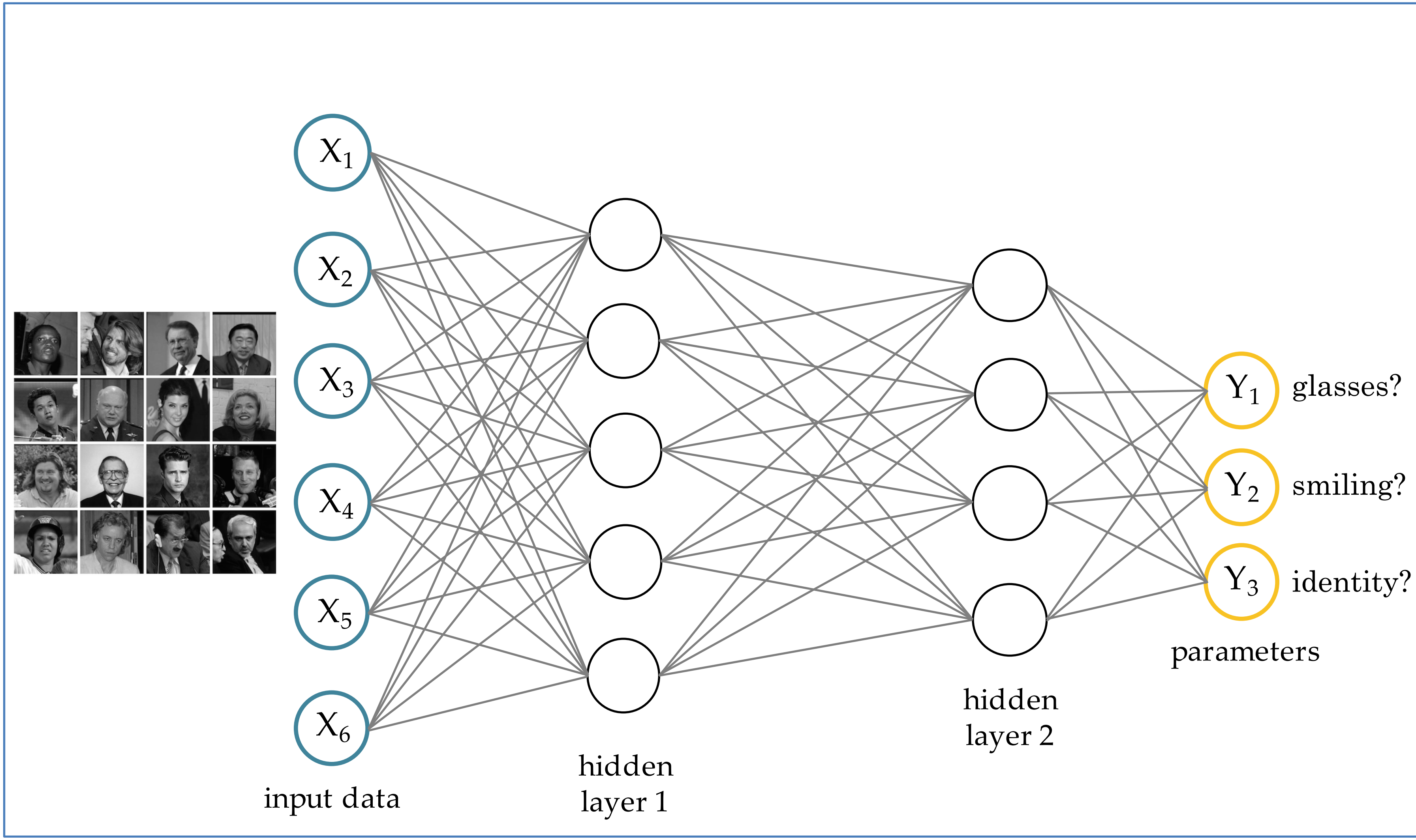
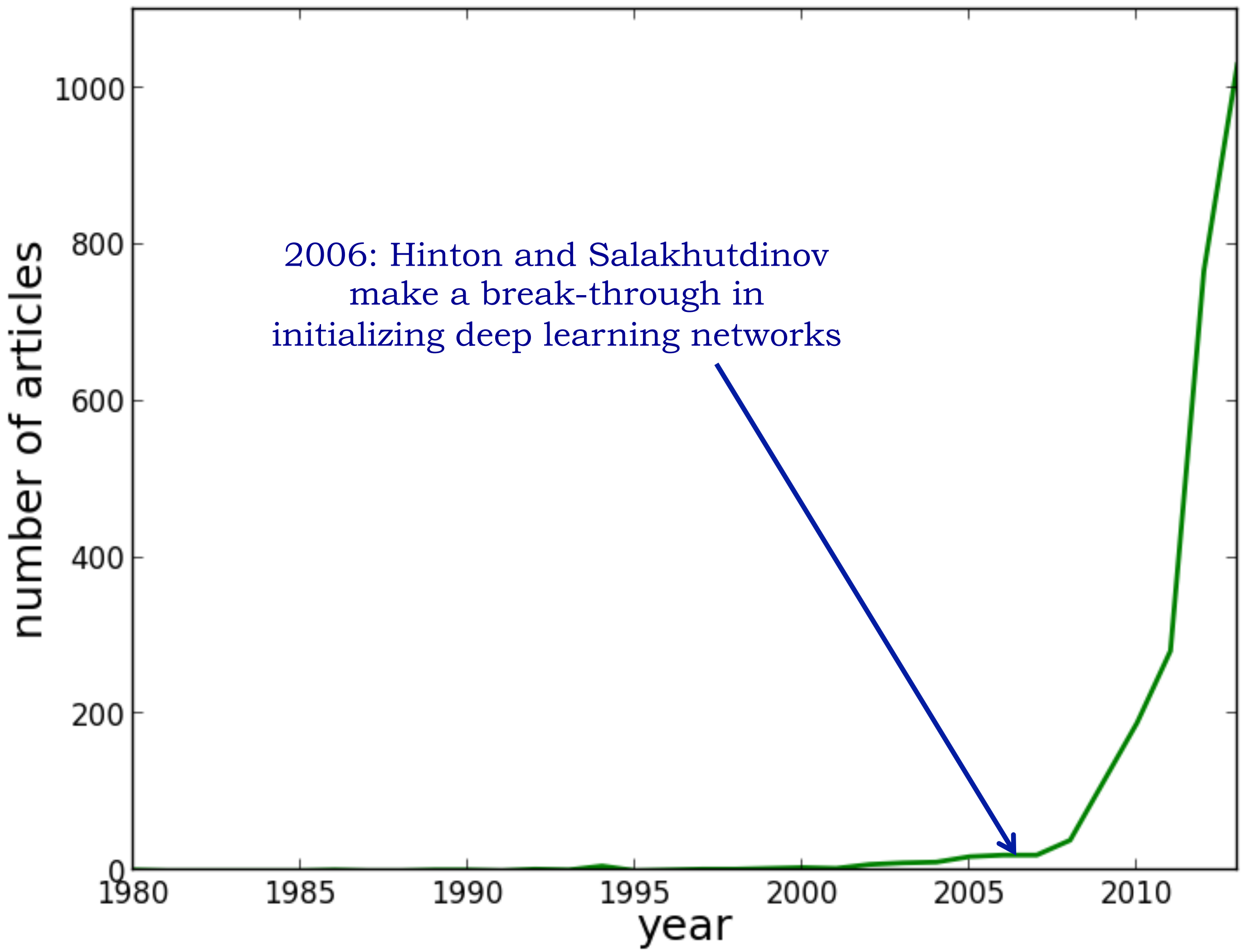


Image: Labeled Faces in the Wild (UMass)

History of Neural Networks

- Perceptron can be interpreted as a simple neural network
- Misconceptions about the weaknesses of perceptrons contributed to declining funding for NN research
- Difficulty of training multi-layer NNs contributed to second setback
- Mid 2000's: breakthroughs in NN training contribute to rise of "deep learning"

Number of papers that mention “deep learning” over time



Big picture for today

- Neural networks can approximate any function!

Big picture for today

- Neural networks can approximate any function!
- For our purposes in ML, we want to use them to approximate a function from our inputs to our outputs

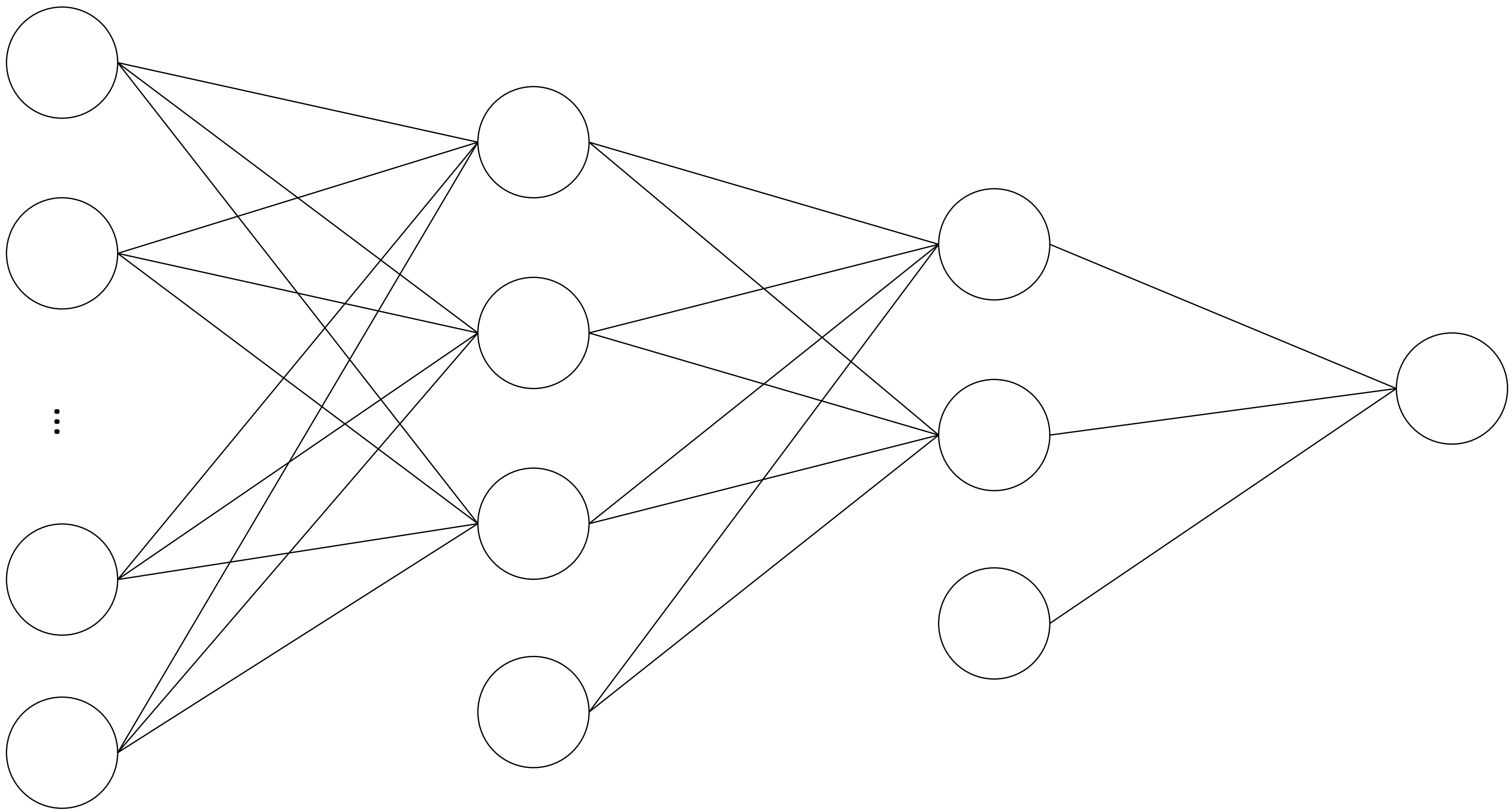
Big picture for today

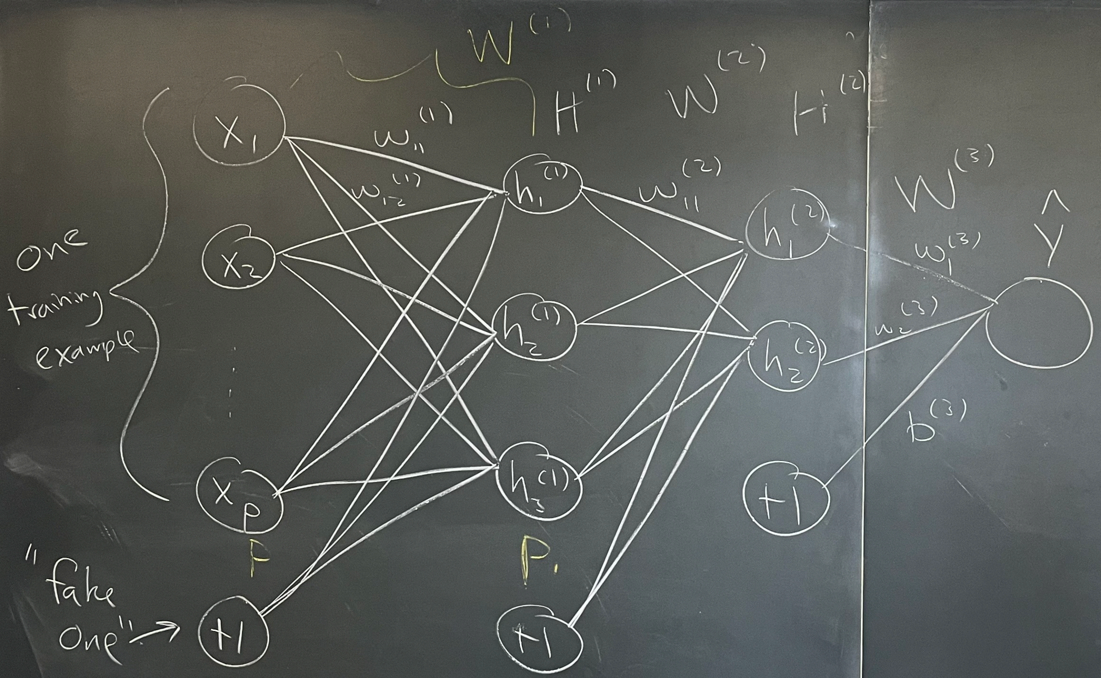
- Neural networks can approximate any function!
- For our purposes in ML, we want to use them to approximate a function from our inputs to our outputs
- We will train our network by asking it to minimize the loss between its output and the true output

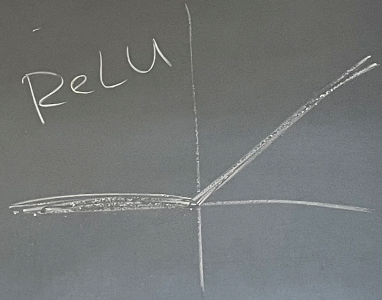
Big picture for today

- Neural networks can approximate any function!
- For our purposes in ML, we want to use them to approximate a function from our inputs to our outputs
- We will train our network by asking it to minimize the loss between its output and the true output
- We will use SGD-like approaches to minimize loss

Fully Connected Neural Network Architecture







$$\begin{bmatrix} -1 & 2 & 5 \\ 0 & -2 & 1 \\ -3 & -4 & 8 \end{bmatrix}$$

a

$$\begin{bmatrix} 0 & 2 & 5 \\ 0 & 0 & 1 \\ 0 & 0 & 8 \end{bmatrix}$$

$$H^{(1)} = a\Big(\underbrace{\underset{p_1 \times p}{W^{(1)}}\ \underset{p \times n}{X}}_{p_1 \times n} + \underbrace{\vec{b}^{(1)}}_{p_1 \times 1}\Big)$$

activation function

$$\underset{p \times n}{\begin{bmatrix} | & | & \\ x_1 & x_2 & \cdots \\ | & | & \end{bmatrix}}$$

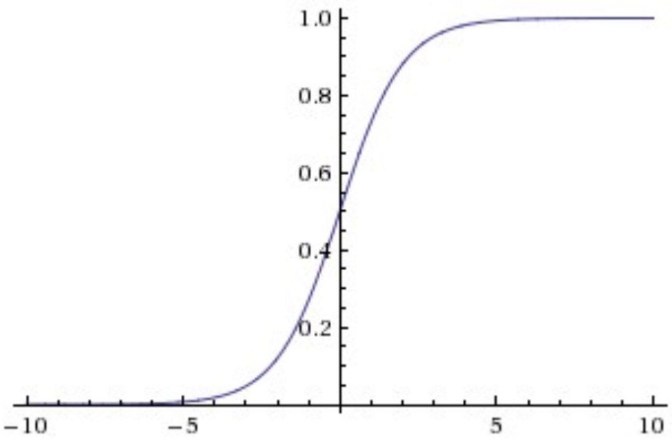
$$H^{(2)} = a\left(W^{(2)} H^{(1)} + \vec{b}^{(2)}\right)$$

$$\hat{Y} = a\left(W^{(3)} H^{(2)} + b^{(3)}\right)$$

Option 1: sigmoid function

- Input: all real numbers, output: [0, 1]

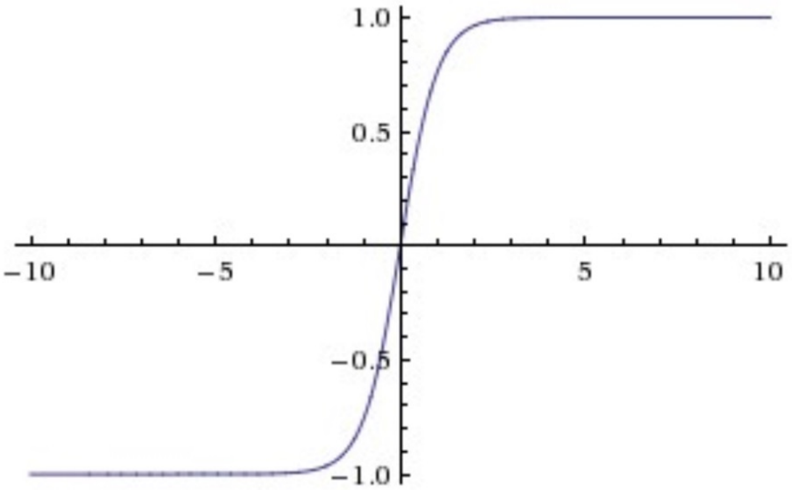
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



Option 2: hyperbolic tangent

- Input: all real numbers, output: [-1, 1]

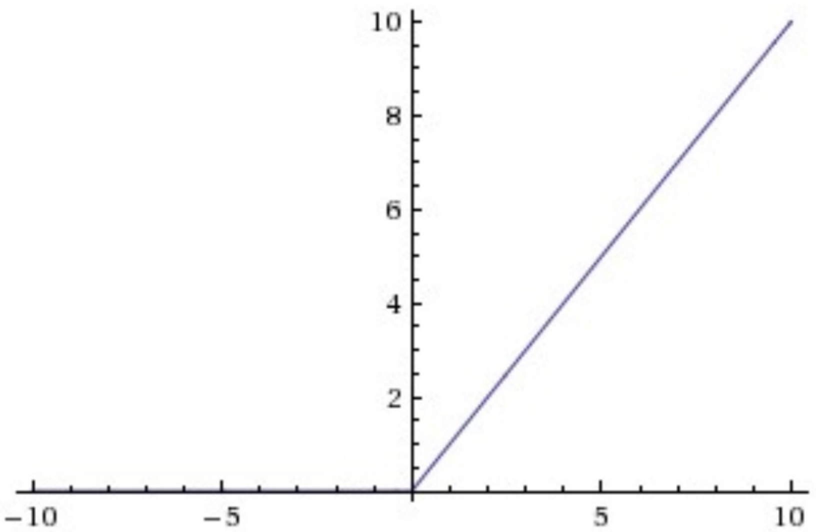
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



Option 3: Rectified Linear Unit (ReLU)

- Return *x* if *x* is positive (i.e. threshold at 0)

$$f(x) = \max(0, x)$$



Pros and Cons of Activation Functions

1) Sigmoid

- (-) When input becomes very positive or very negative, gradient approaches 0 (saturates and stops gradient descent)
- (-) Not zero-centered, so gradient on weights can end up all positive or all negative (zig-zag in gradient descent)
- (+) Derivative is easy to compute given function value!

More info:
http://cs231n.github.io/neural-networks-1/

Pros and Cons of Activation Functions

1) Sigmoid

- (-) When input becomes very positive or very negative, gradient approaches 0 (saturates and stops gradient descent)
- (-) Not zero-centered, so gradient on weights can end up all positive or all negative (zig-zag in gradient descent)
- (+) Derivative is easy to compute given function value!

2) Tanh

- (-) Still has a tendency to prematurely kill the gradient
- (+) Zero-centered so we get a range of gradients
- (+) Rescaling of sigmoid function so derivative is also not too difficult

More info:
http://cs231n.github.io/neural-networks-1/

Pros and Cons of Activation Functions

1) Sigmoid

- (-) When input becomes very positive or very negative, gradient approaches 0 (saturates and stops gradient descent)
- (-) Not zero-centered, so gradient on weights can end up all positive or all negative (zig-zag in gradient descent)
- (+) Derivative is easy to compute given function value!

2) Tanh

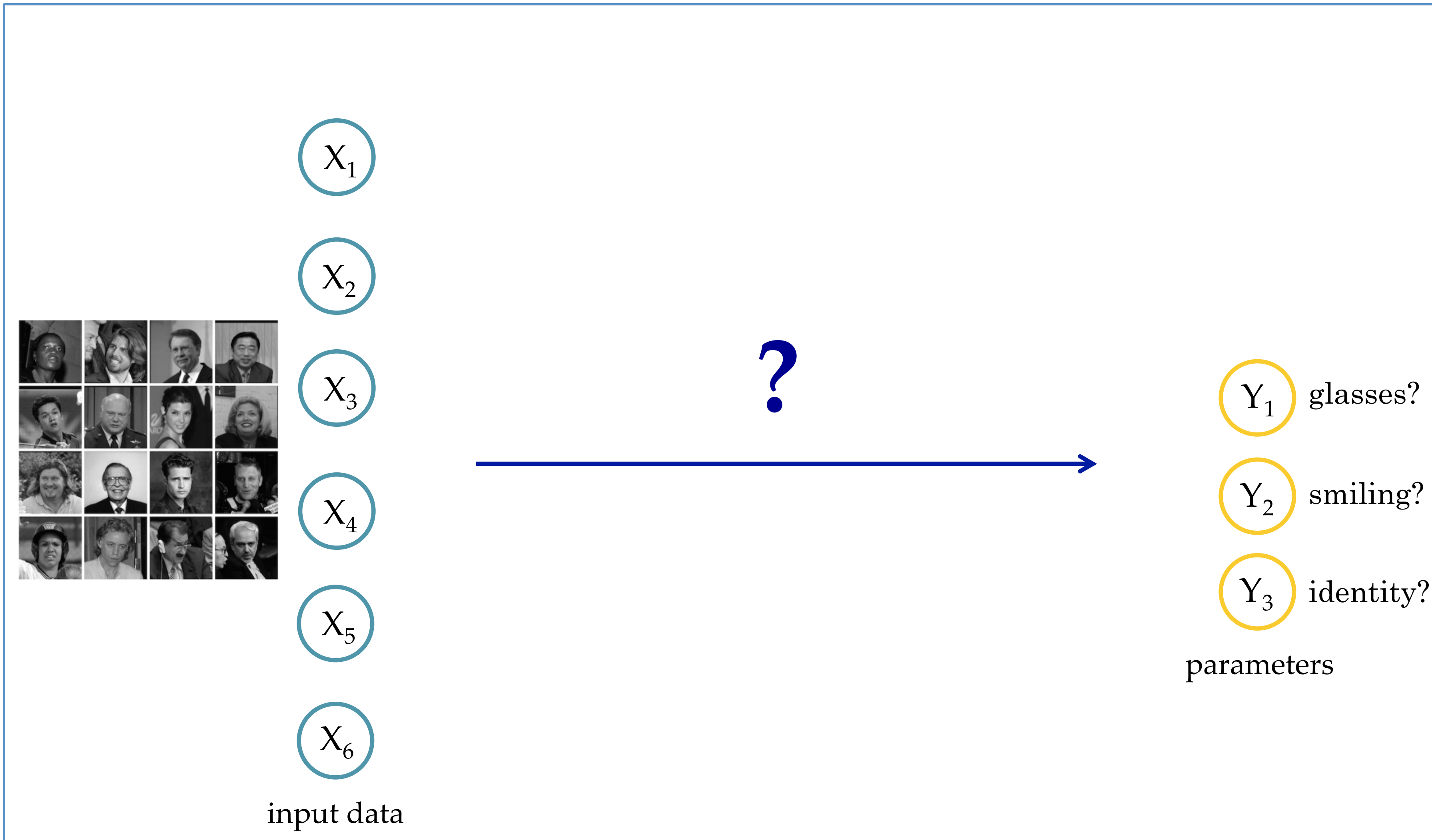
- (-) Still has a tendency to prematurely kill the gradient
- (+) Zero-centered so we get a range of gradients
- (+) Rescaling of sigmoid function so derivative is also not too difficult

3) ReLU

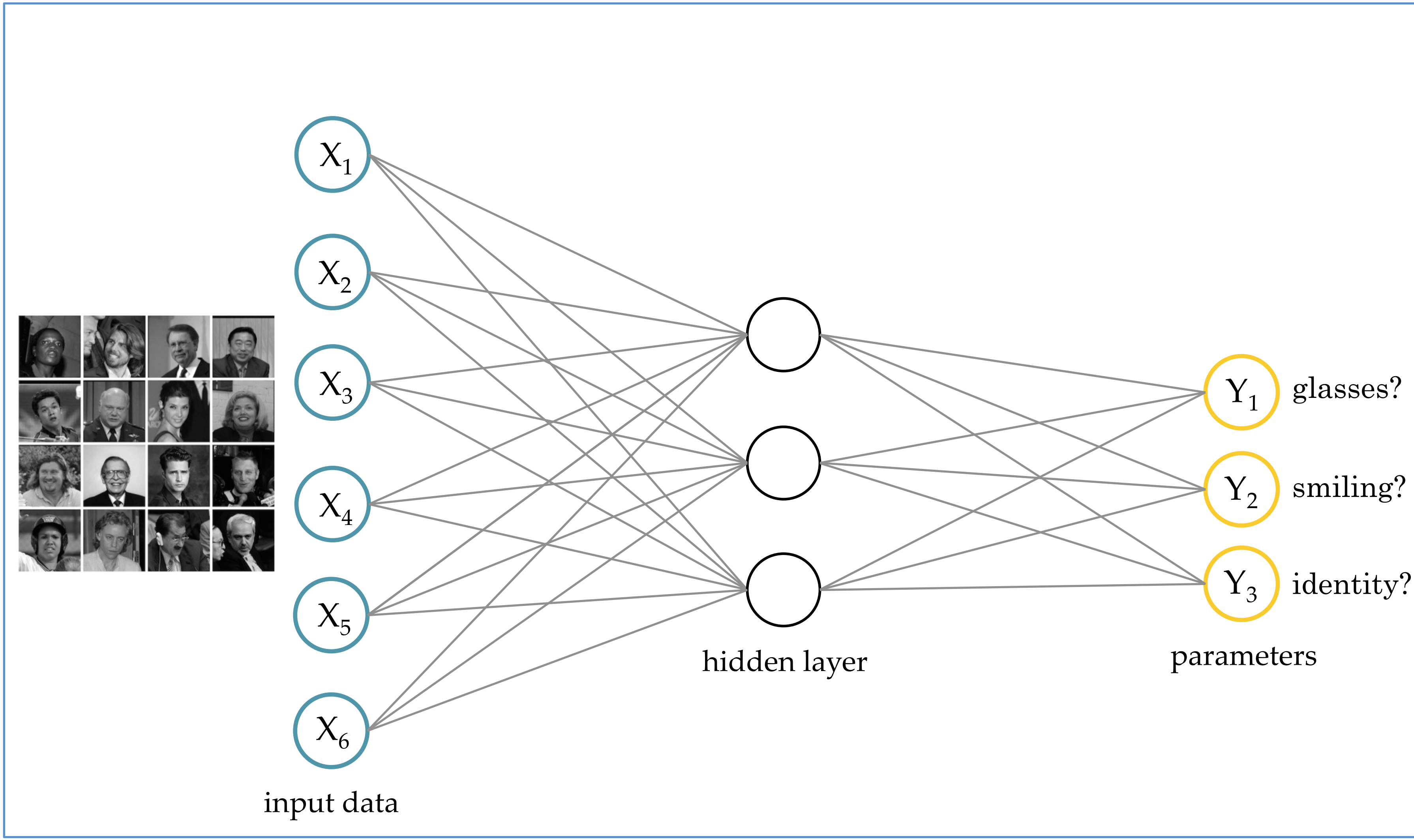
- (+) Works well in practice (accelerates convergence)
- (+) Function value very easy to compute! (no exponentials)
- (-) Units can have no signal if input becomes too negative throughout gradient descent

More info:
http://cs231n.github.io/neural-networks-1/

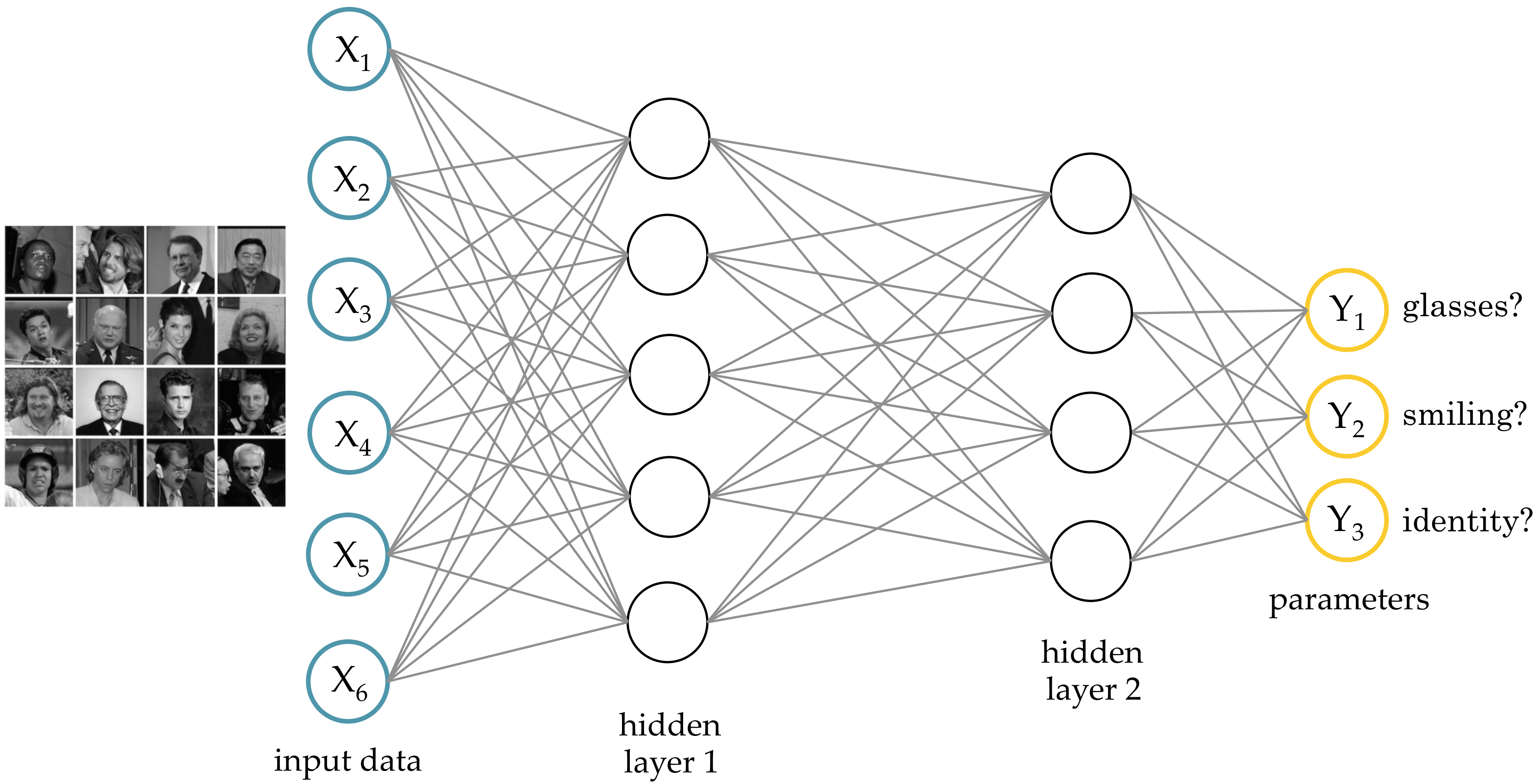
Goal: find a function between input and output



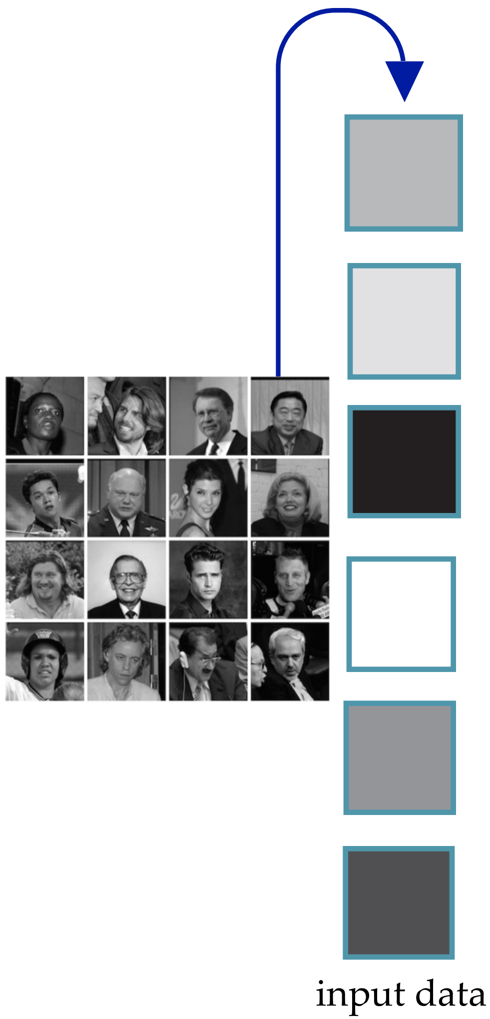
First idea: one hidden layer



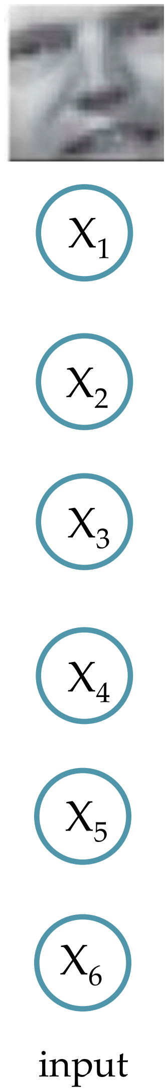
Second idea: more hidden layers ("deep" learning)



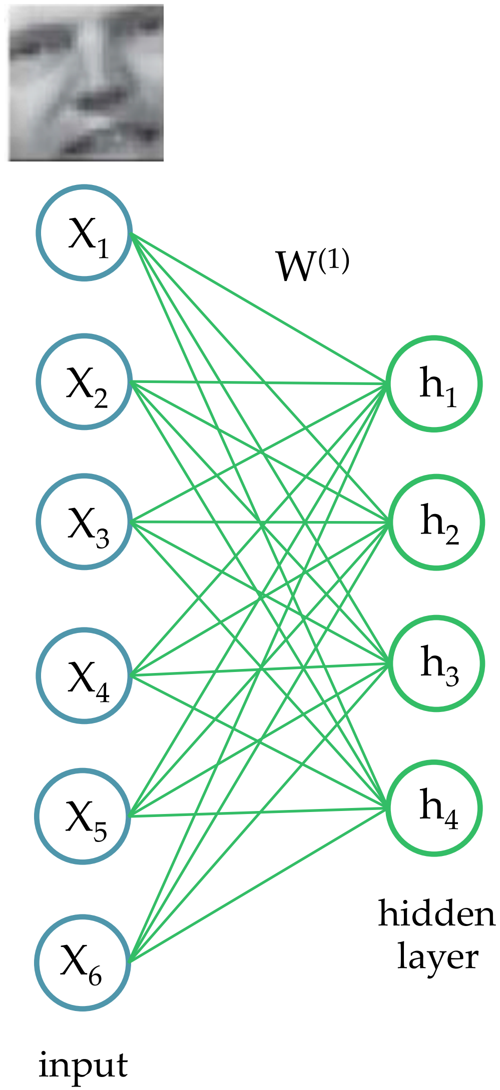
Flatten pixels of image into a single vector



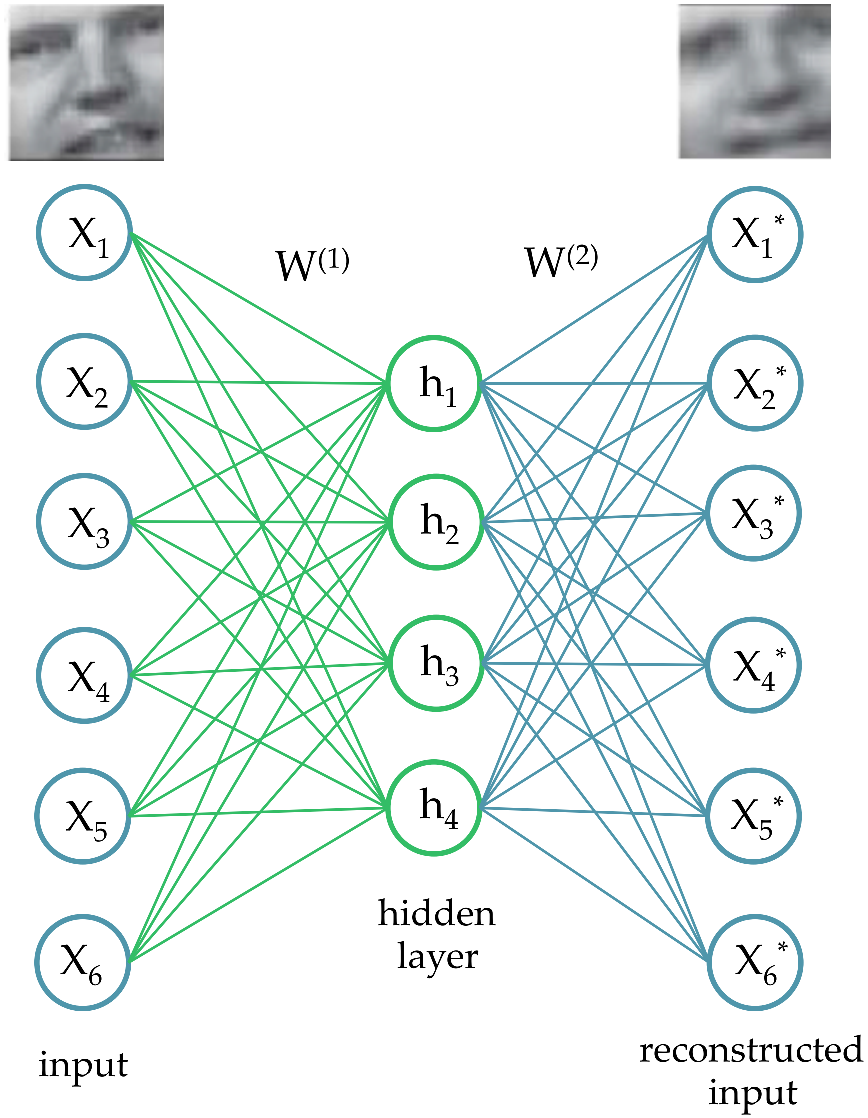
Detour to autoencoders



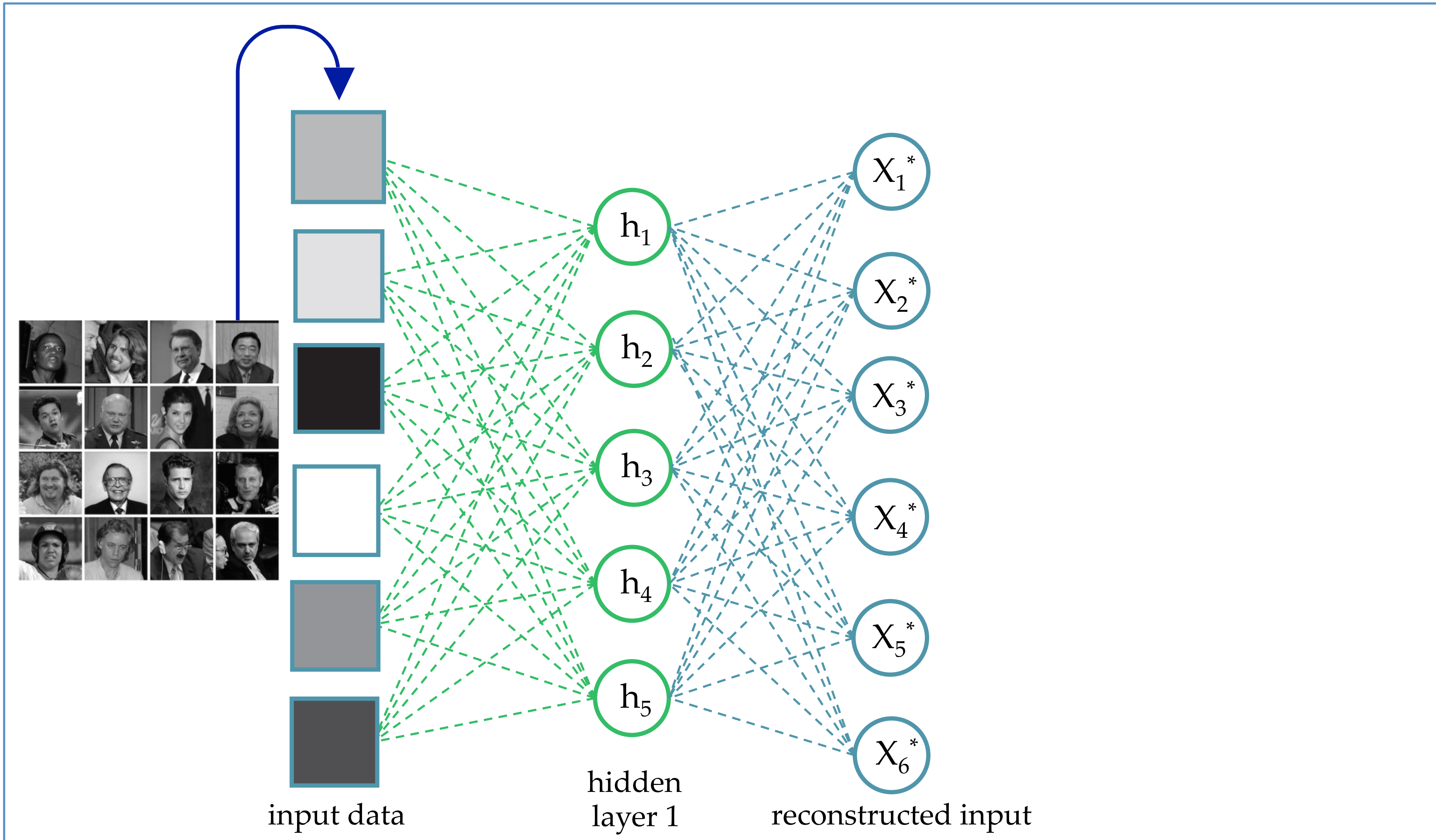
Detour to autoencoders



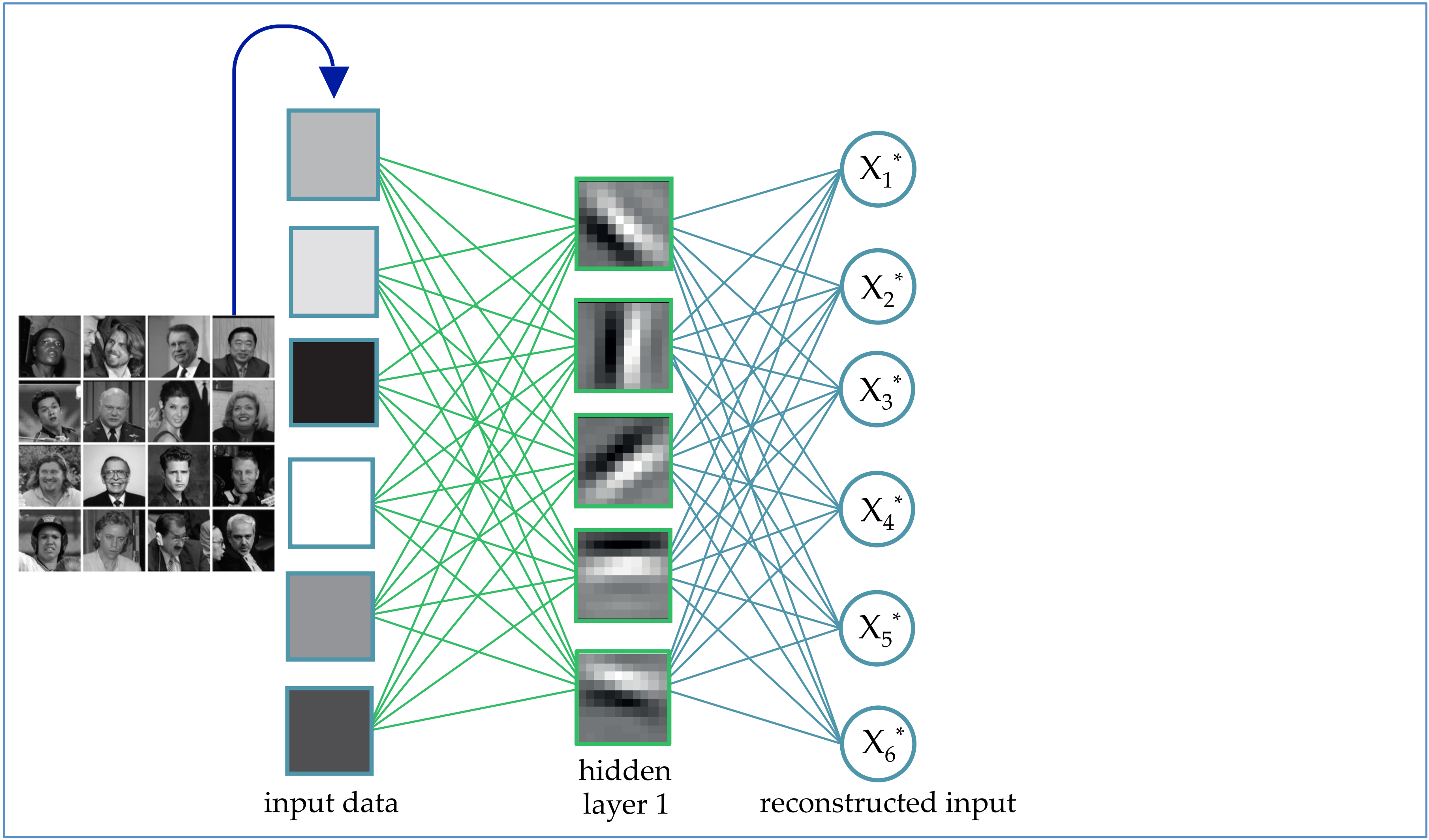
Detour to autoencoders



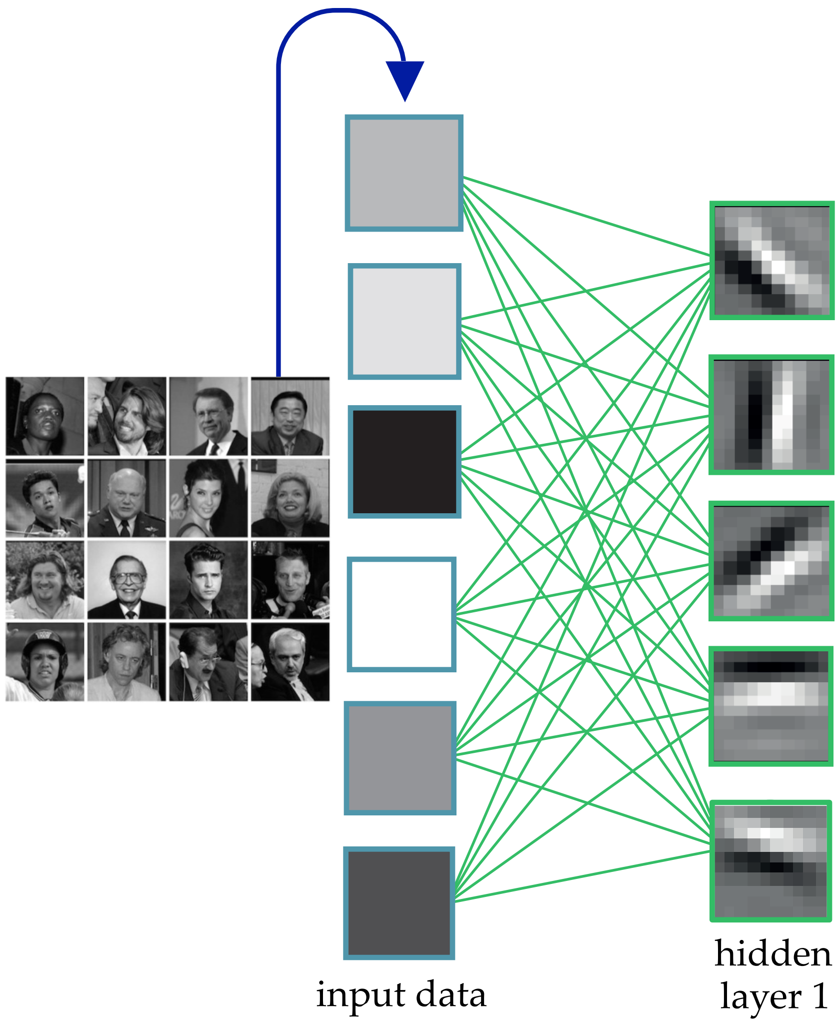
Use <u>unsupervised pre-training</u> to find a function from the input to itself



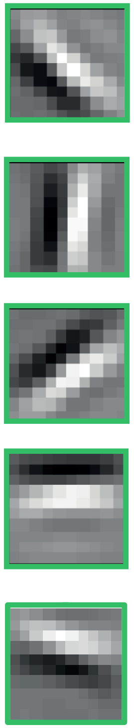
Hidden units can be interpreted as edges



Now: throw away reconstruction and input

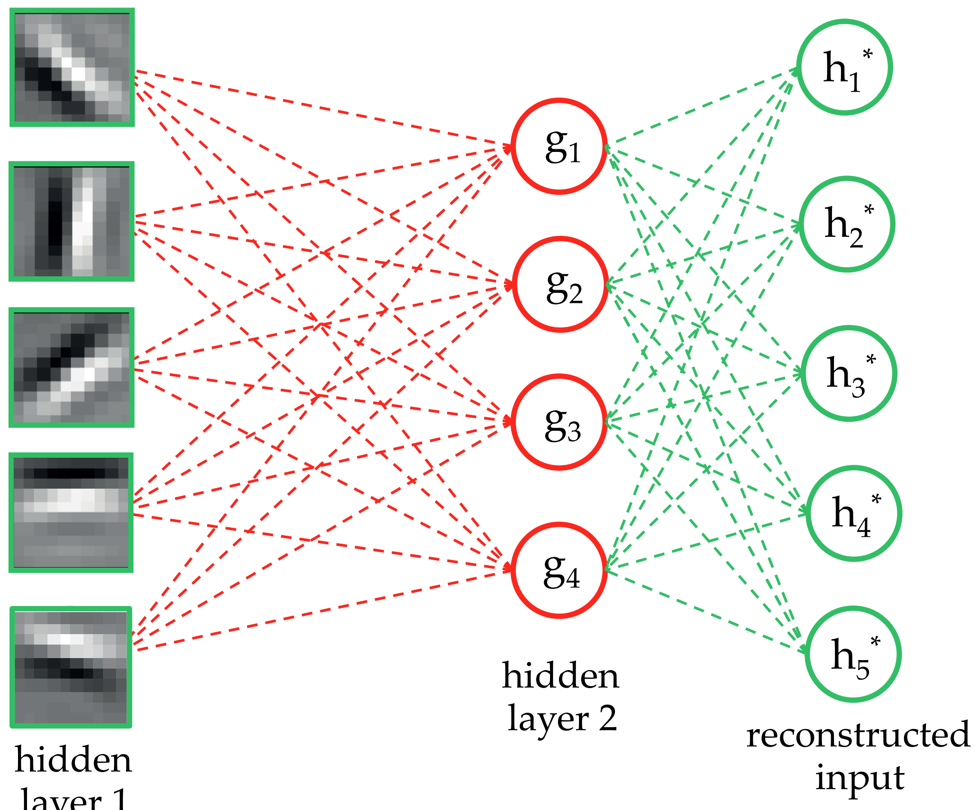


Now: throw away reconstruction and input

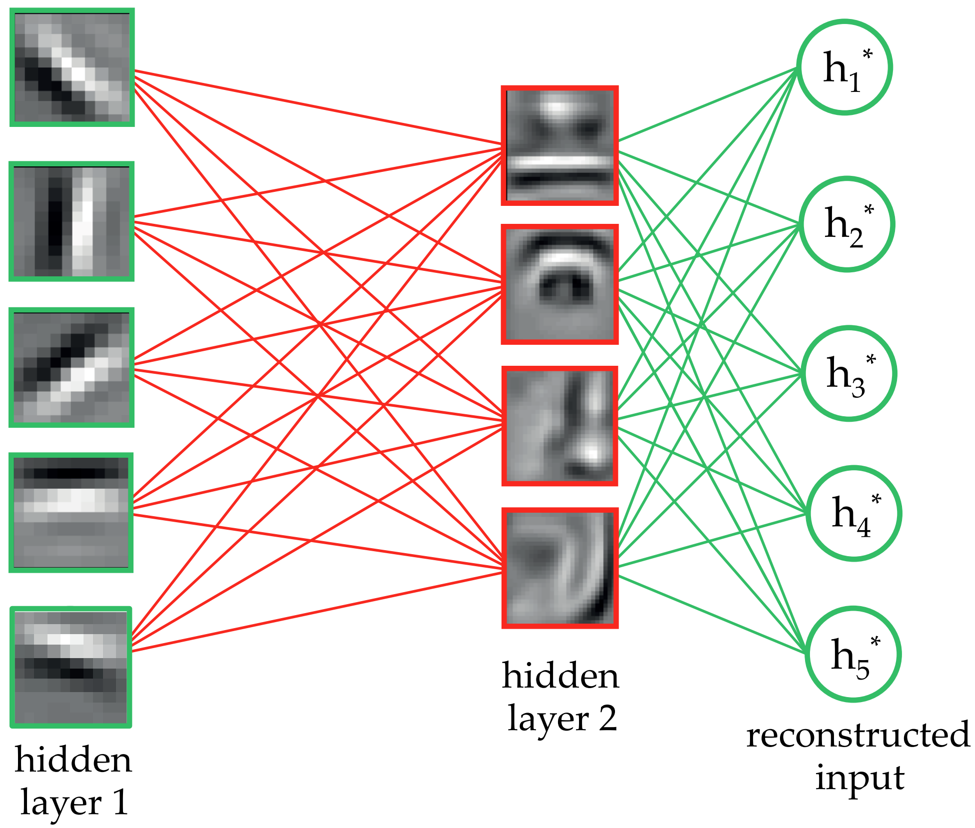


hidden
layer 1

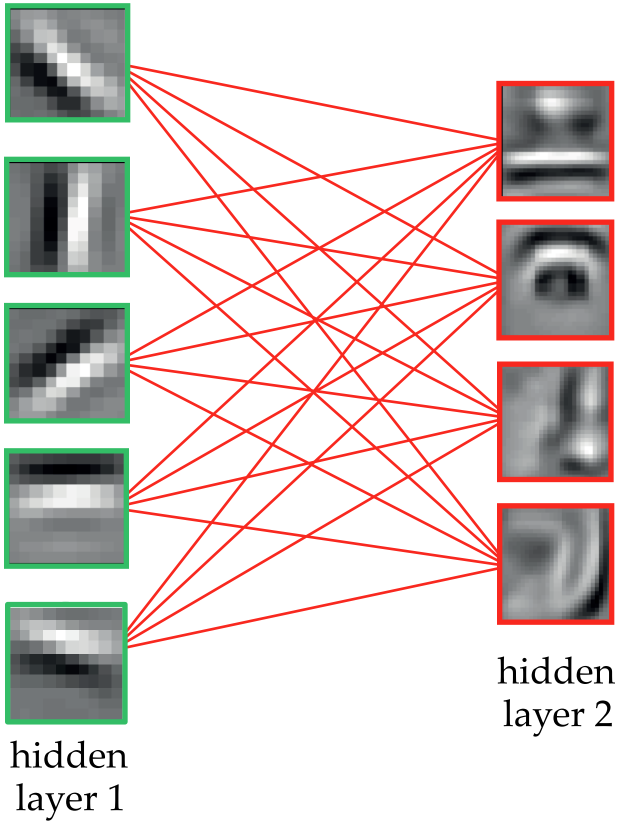
Then repeat the entire process for each layer



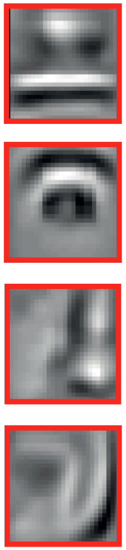
Then repeat the entire process for each layer



Then repeat the entire process for each layer

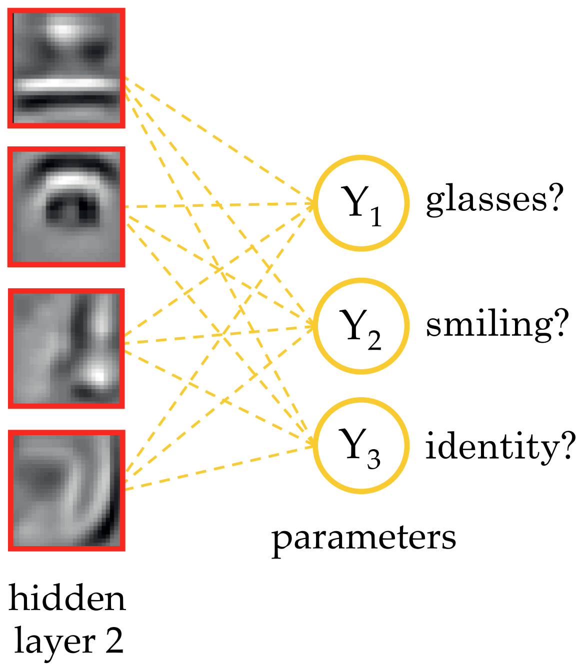


Then repeat the entire process for each layer

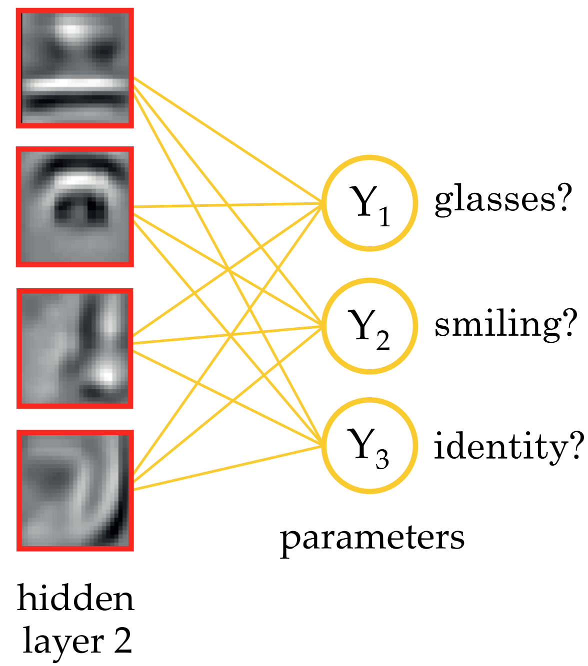


hidden
layer 2

In the last layer, use the outputs (supervised)



In the last layer, use the outputs (supervised)



Finally, “fine-tune” the entire network!