

CS 260: Foundations of Data Science

Prof. Sara Mathieson

Fall 2023

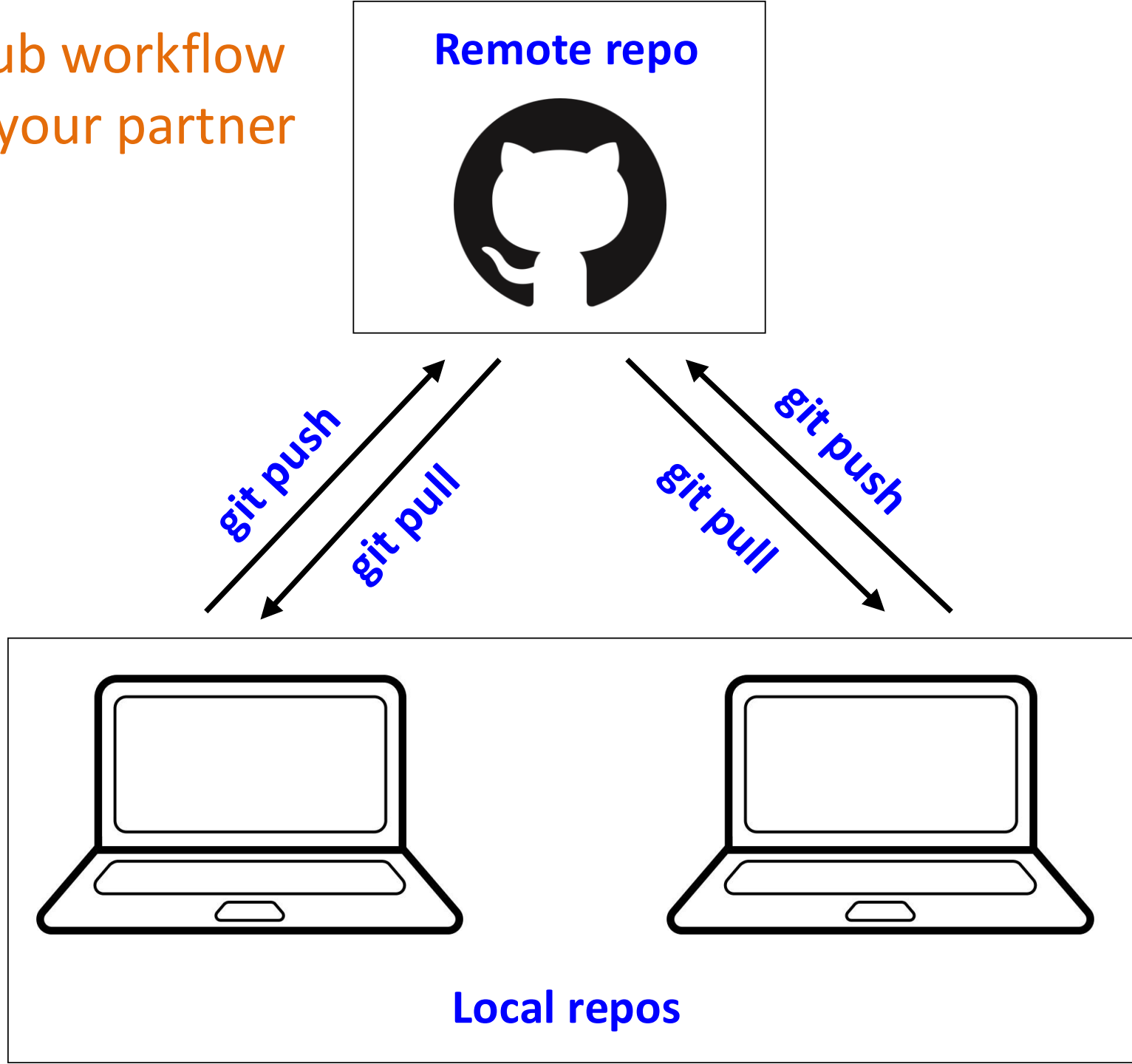


HAVERFORD
COLLEGE

Admin

- Final project presentations next week!
 - Schedule on Piazza

Github workflow with your partner



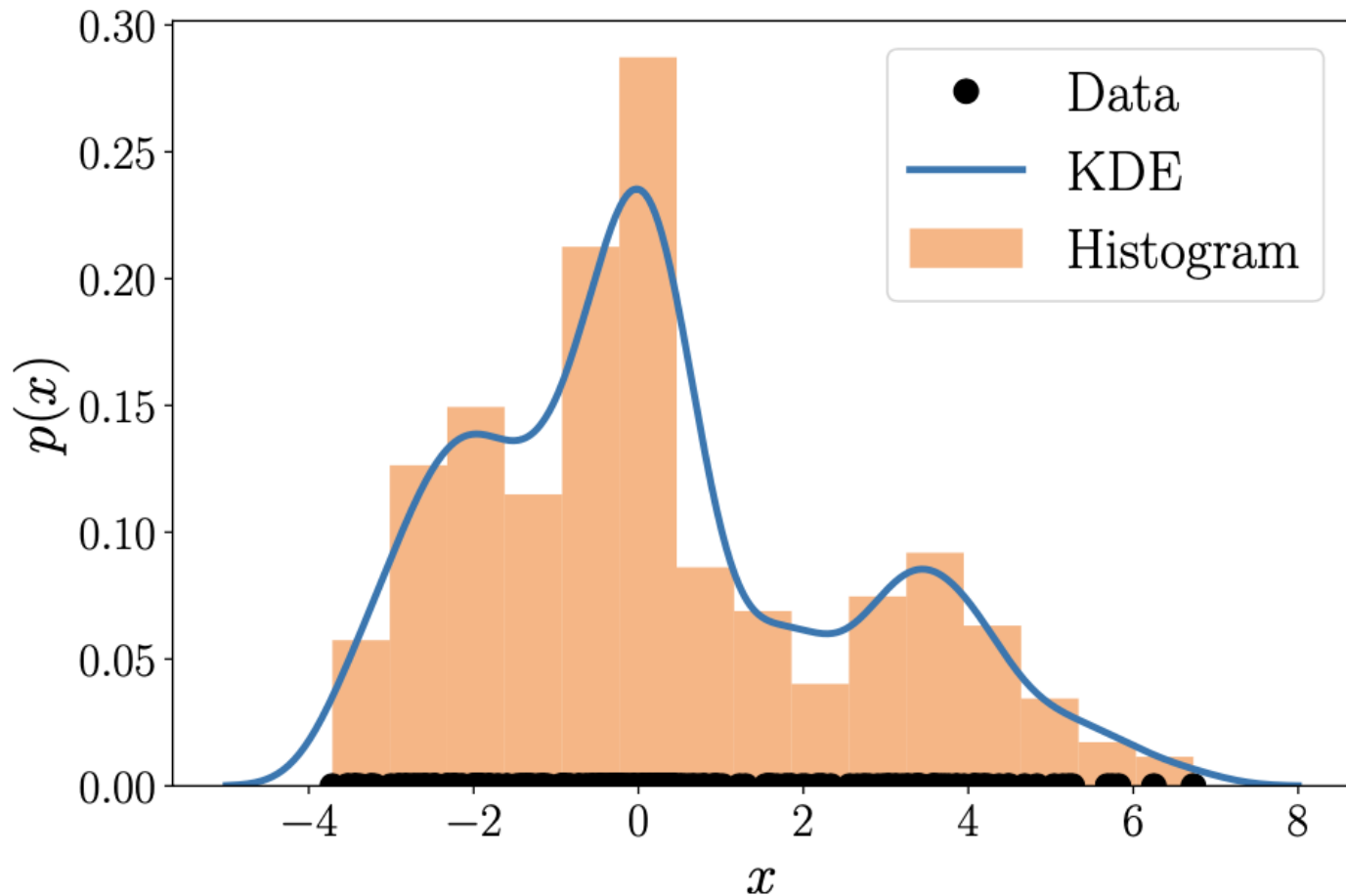
Outline

- Kernel Density Estimation (KDE)
- Missing data
- Neural networks
- Go over Midterm 2

Outline

- Kernel Density Estimation (KDE)
- Missing data
- Neural networks
- Go over Midterm 2

KDE (Kernel Density Estimation)



In code: use seaborn library

Figure 11.9 from MML textbook

Outline

- Kernel Density Estimation (KDE)
- **Missing data**
- Neural networks
- Go over Midterm 2

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

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)

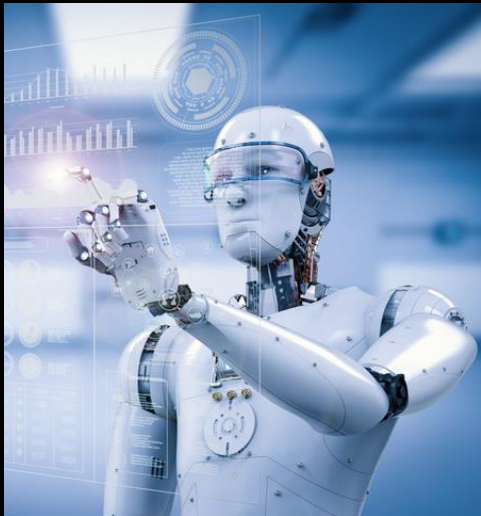
Techniques for handling missing data

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

Outline

- Kernel Density Estimation (KDE)
- Missing data
- **Neural networks**
- Go over Midterm 2

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 into the Lagrangian (Equation 8), and simplify, we get

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

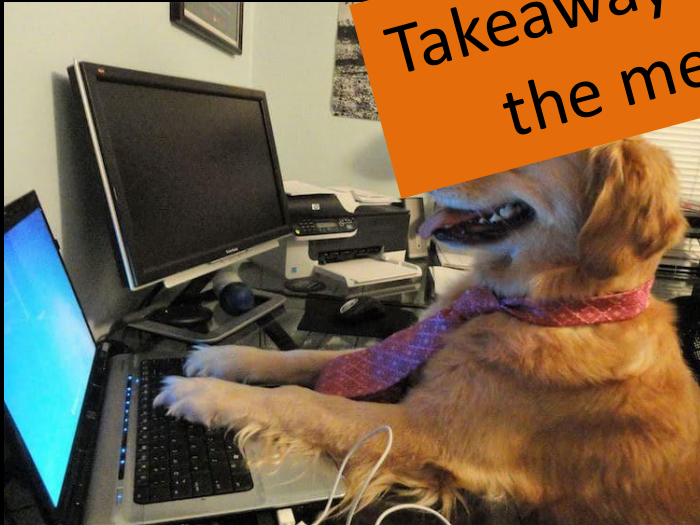
But from Equation (10), the last term must be zero, so we obtain

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

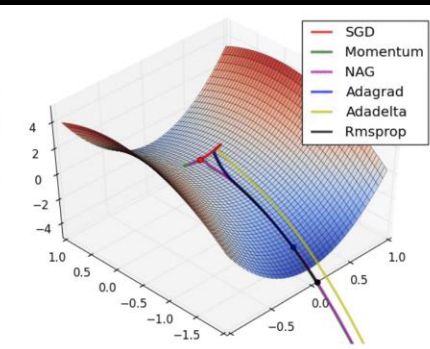
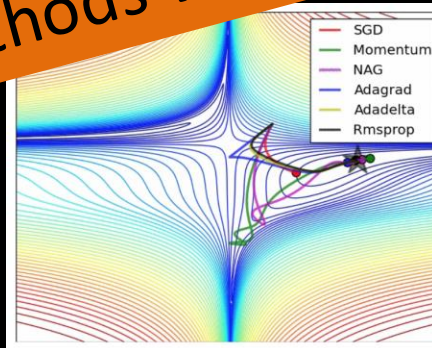


What other computer scientists think I do

Takeaway: we should understand the methods we are using!



What mathematicians think I do

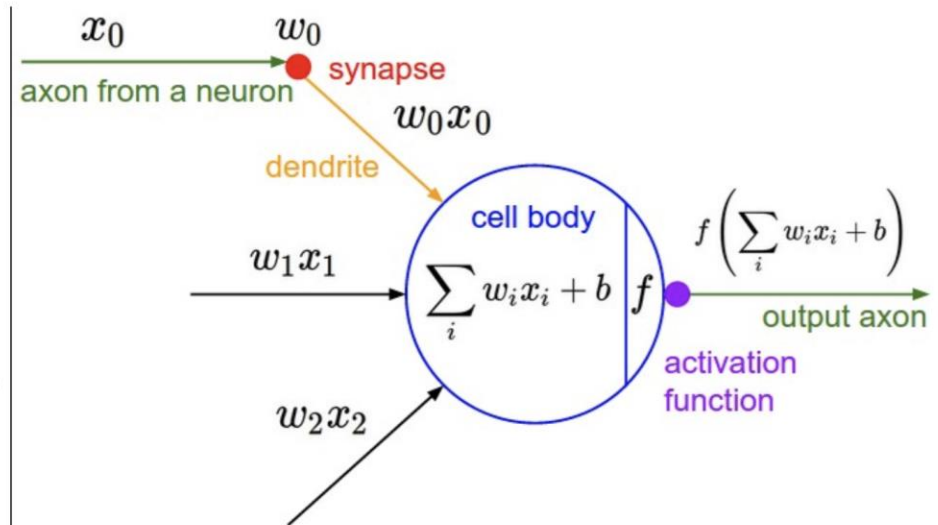
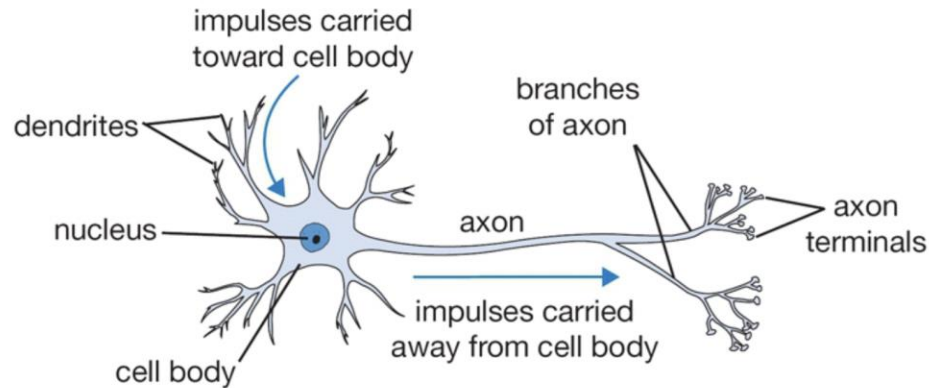


What I think I do

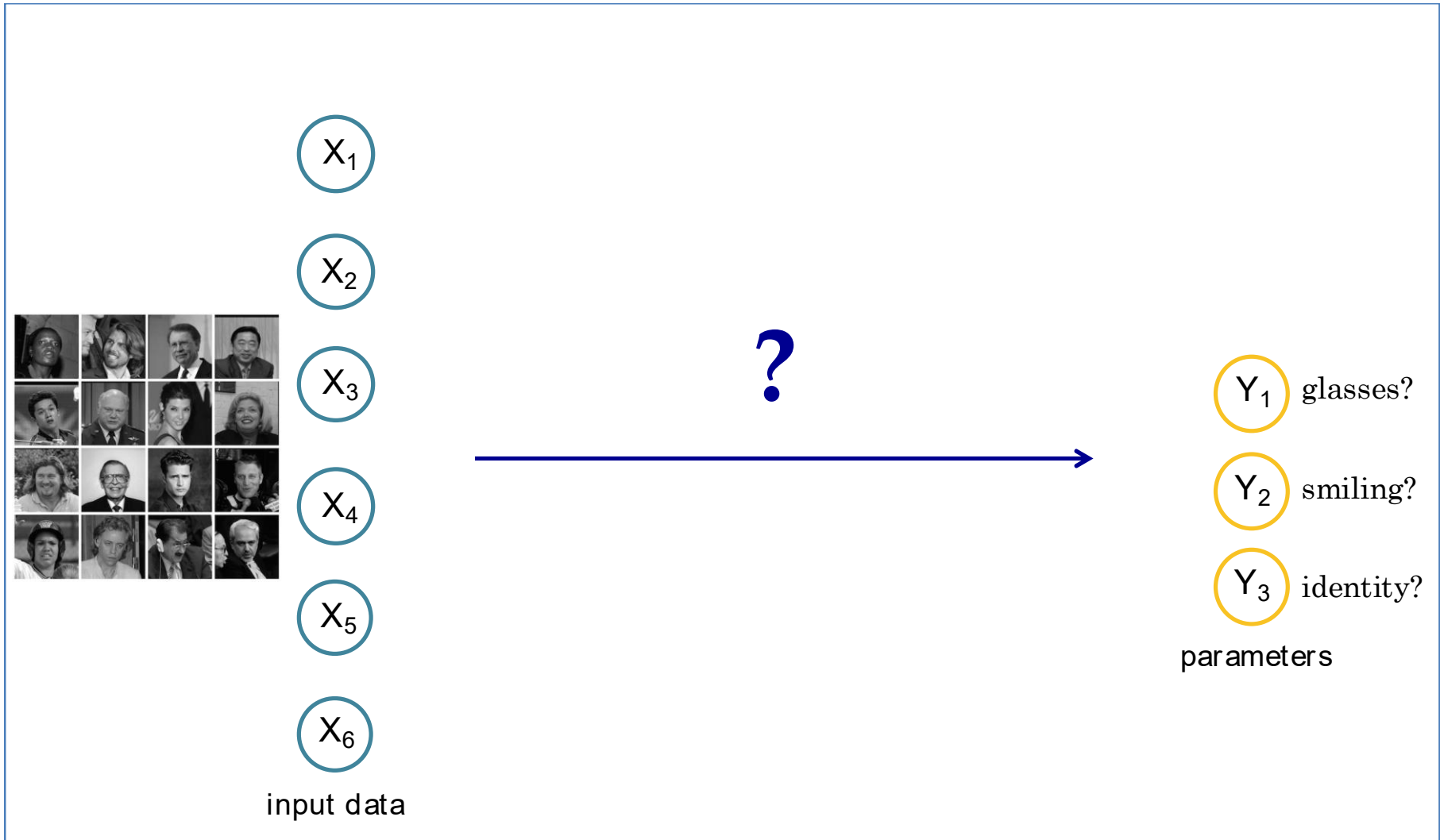
```
>>> from sklearn import svm
>>> import tensorflow as tf
```

What I really do

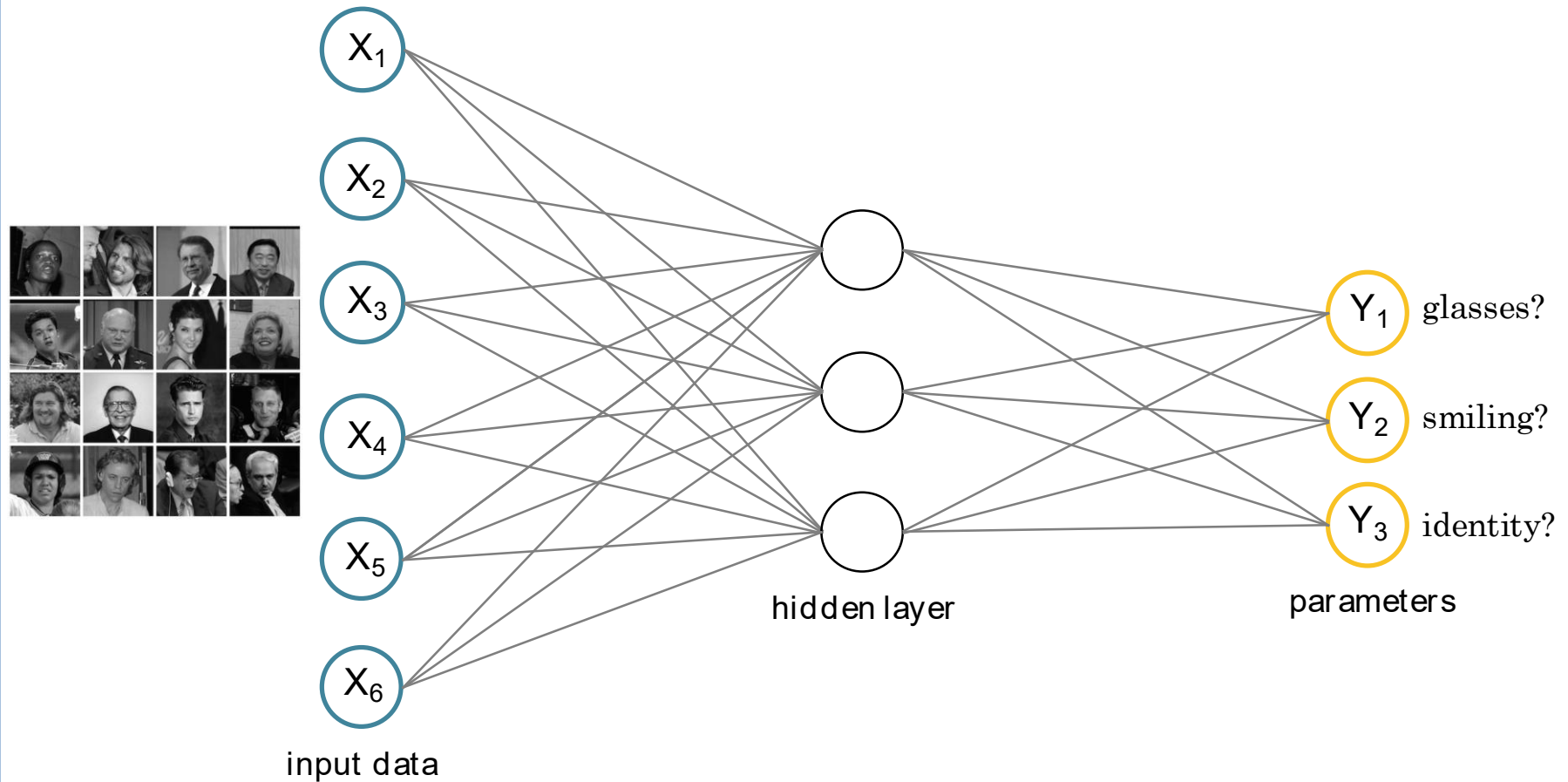
Biological Inspiration



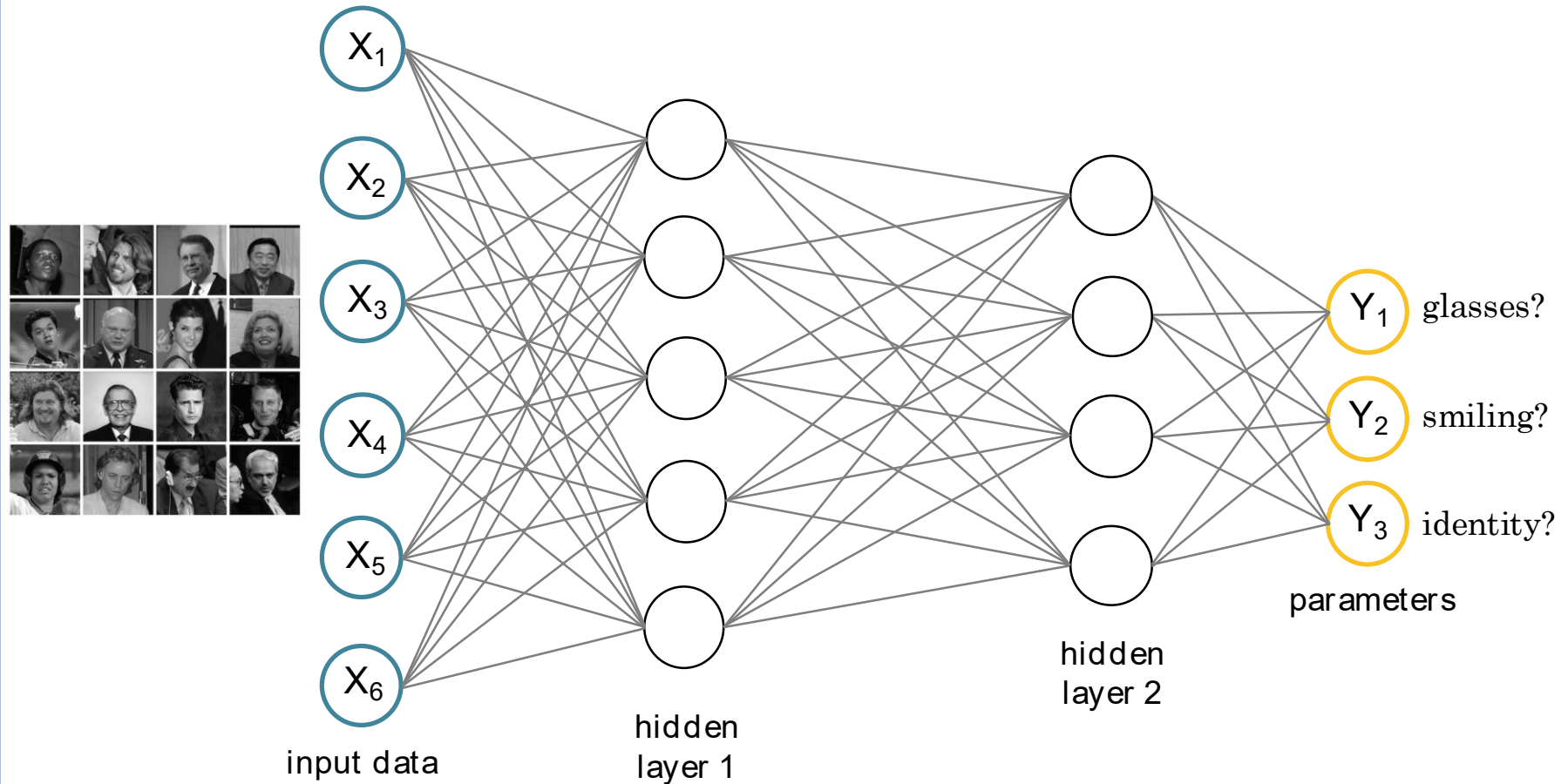
Goal: learn from complicated inputs



Idea: transform data into lower dimension



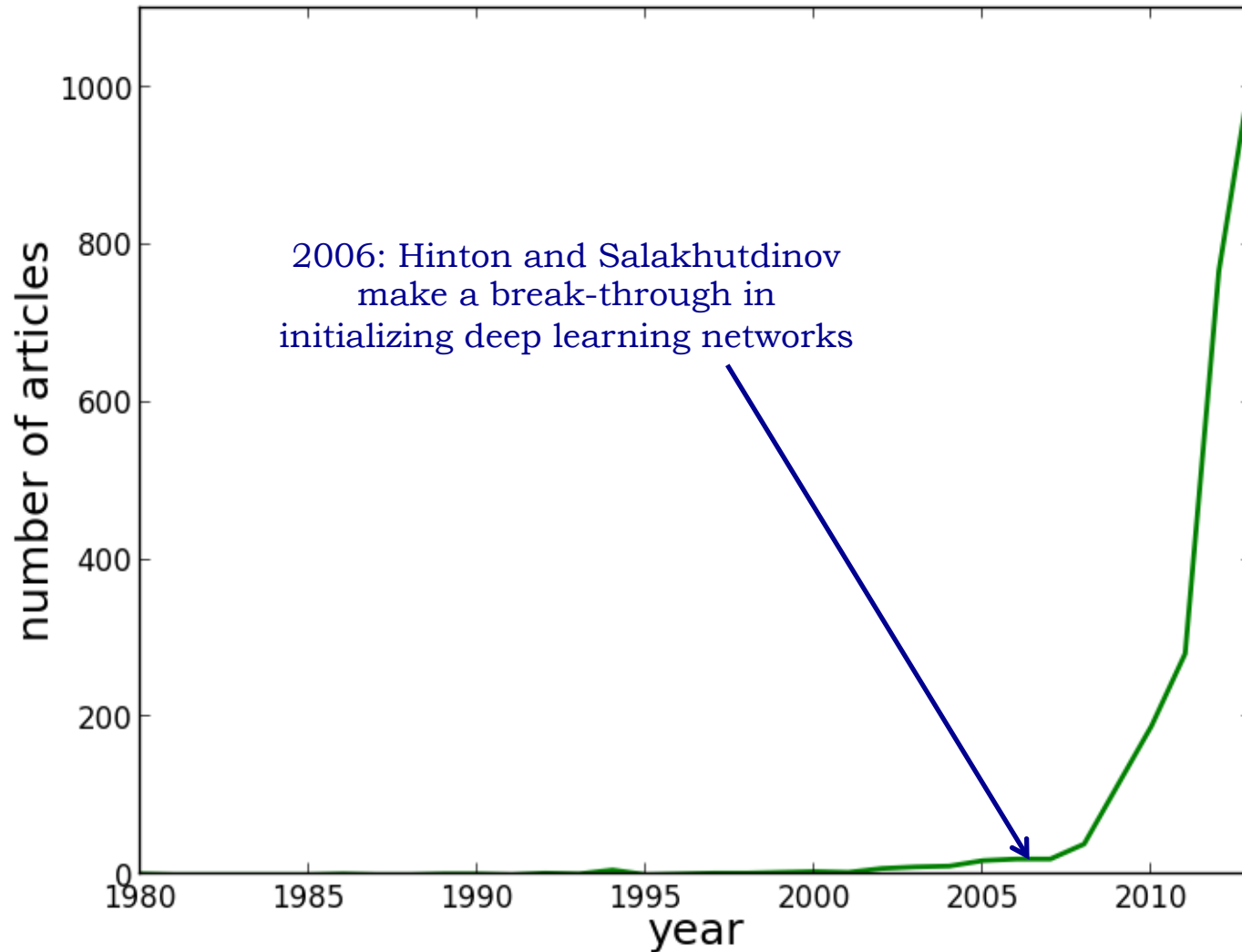
Multi-layer networks = “deep learning”



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

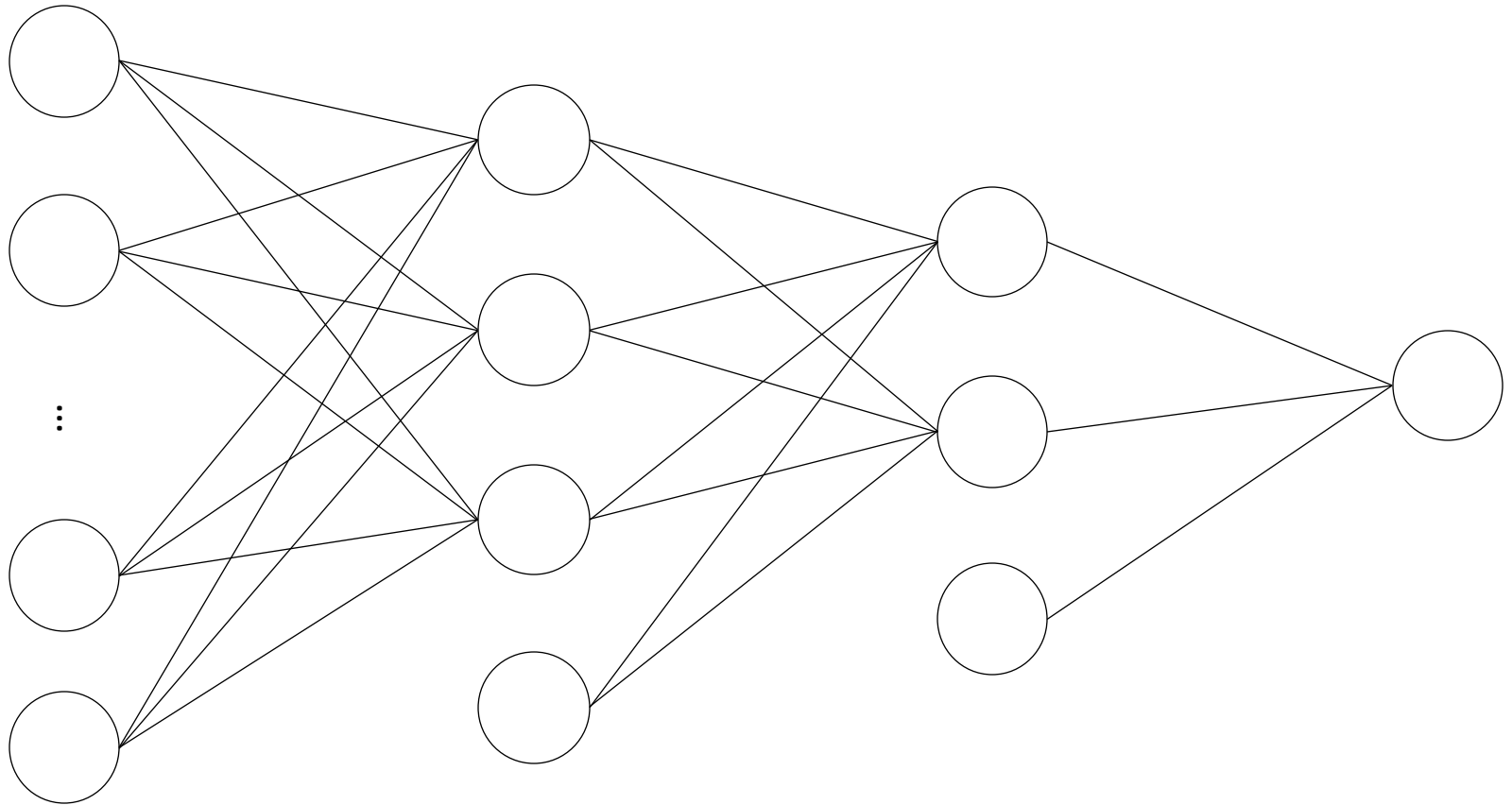
Big picture for today

- 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

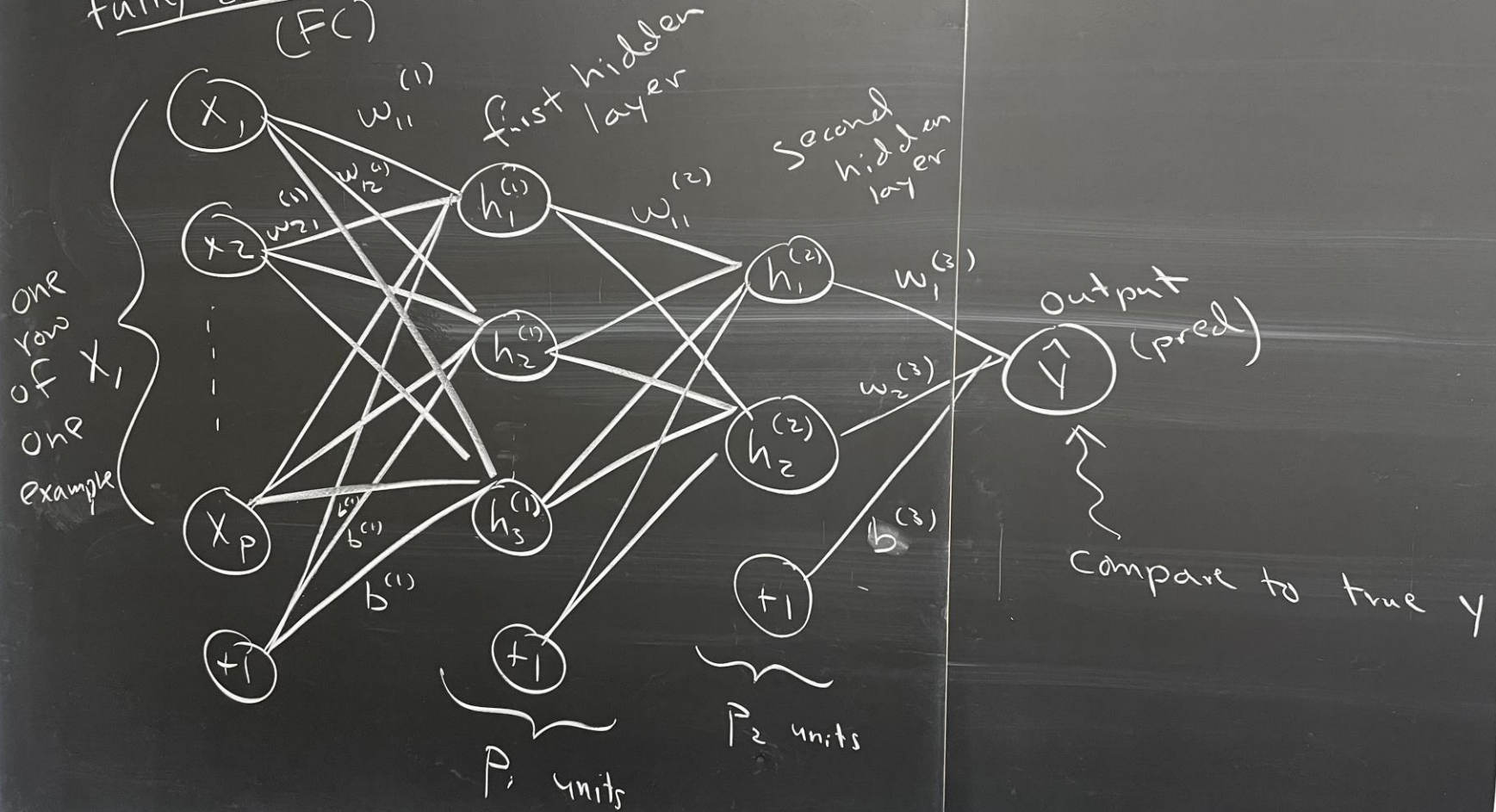
Big picture for today

- 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



fully connected architecture
(FC)



$$H^{(1)} = a(W^{(1)}X + \vec{b}^{(1)})$$

non-linear
activation
function

$P_1 \times P$

$P \times n$

$P_1 \times 1$

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

$P_2 \times P_1$

$P_1 \times n$

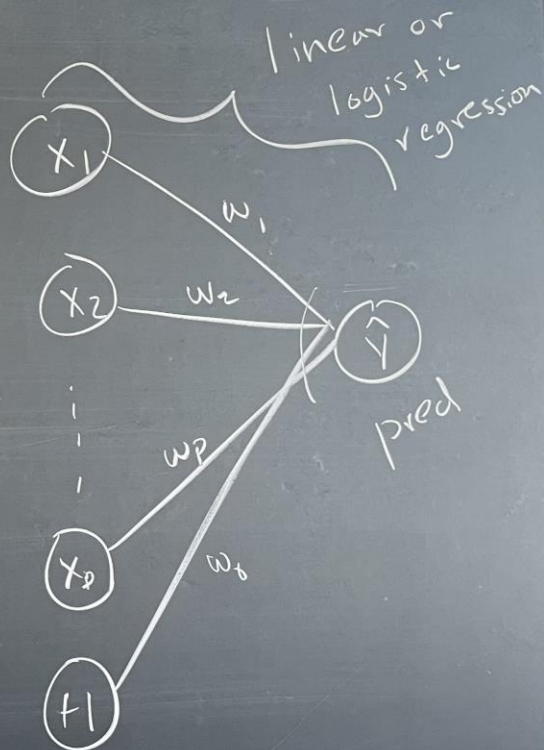
$P_2 \times 1$

$$\hat{y} = a(W^{(3)}H^{(2)} + \vec{b}^{(3)})$$

$1 \times n$

need to
transpose X

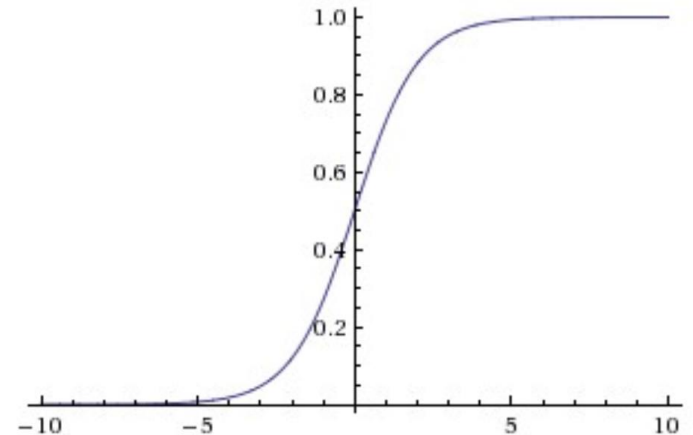
non-linear
function from
 $X \Rightarrow \hat{y}$



Option 1: sigmoid function

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

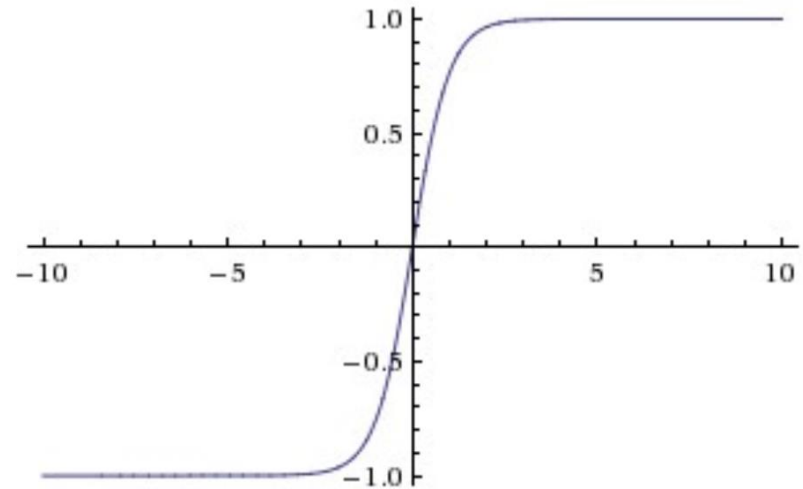
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



Option 2: hyperbolic tangent

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

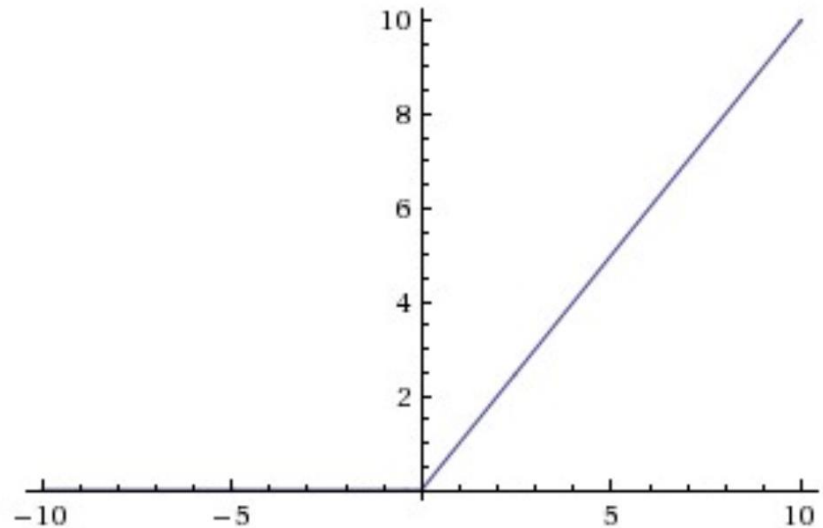
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



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!

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

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!

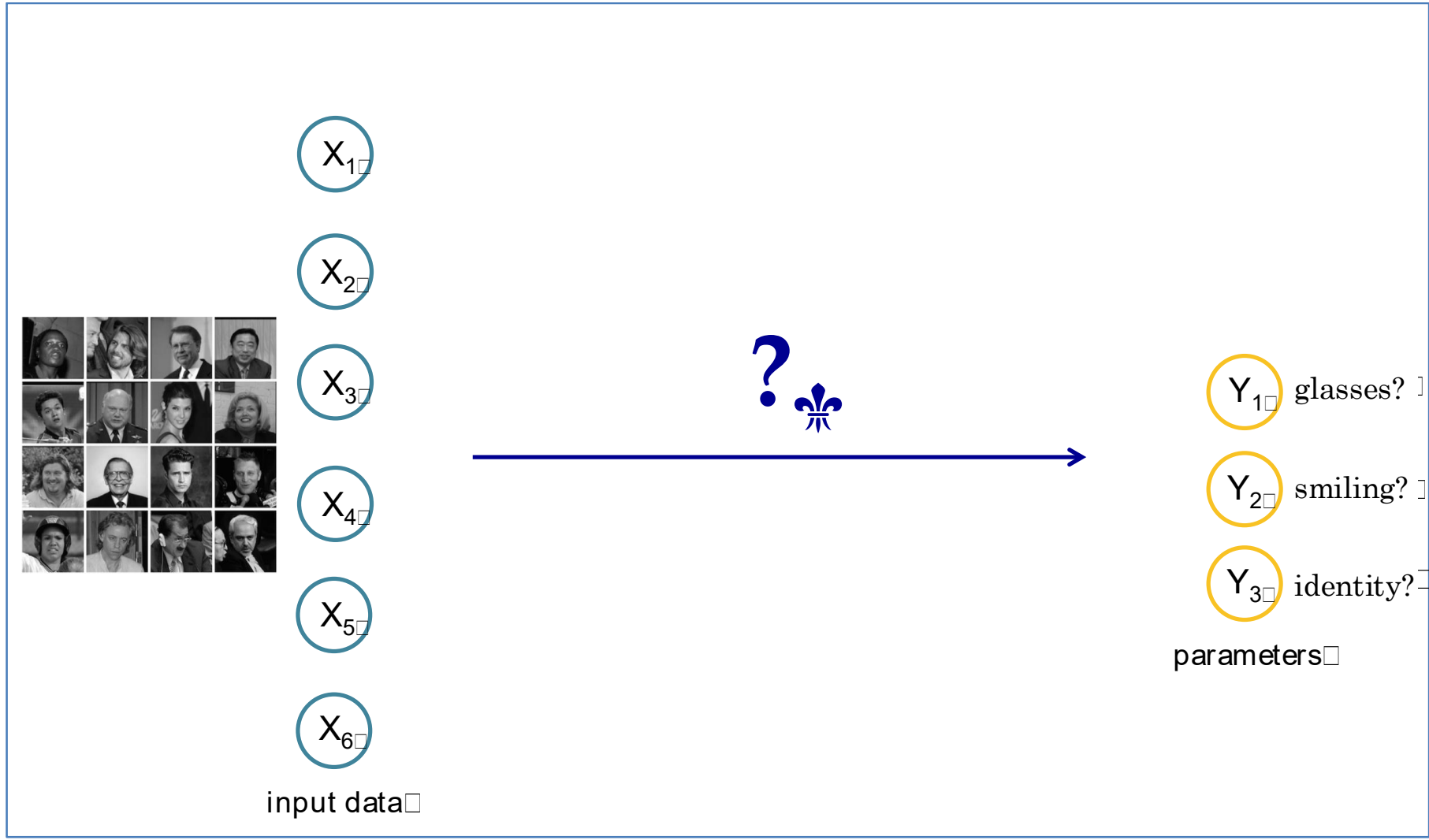
2) Tanh

- (-) 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

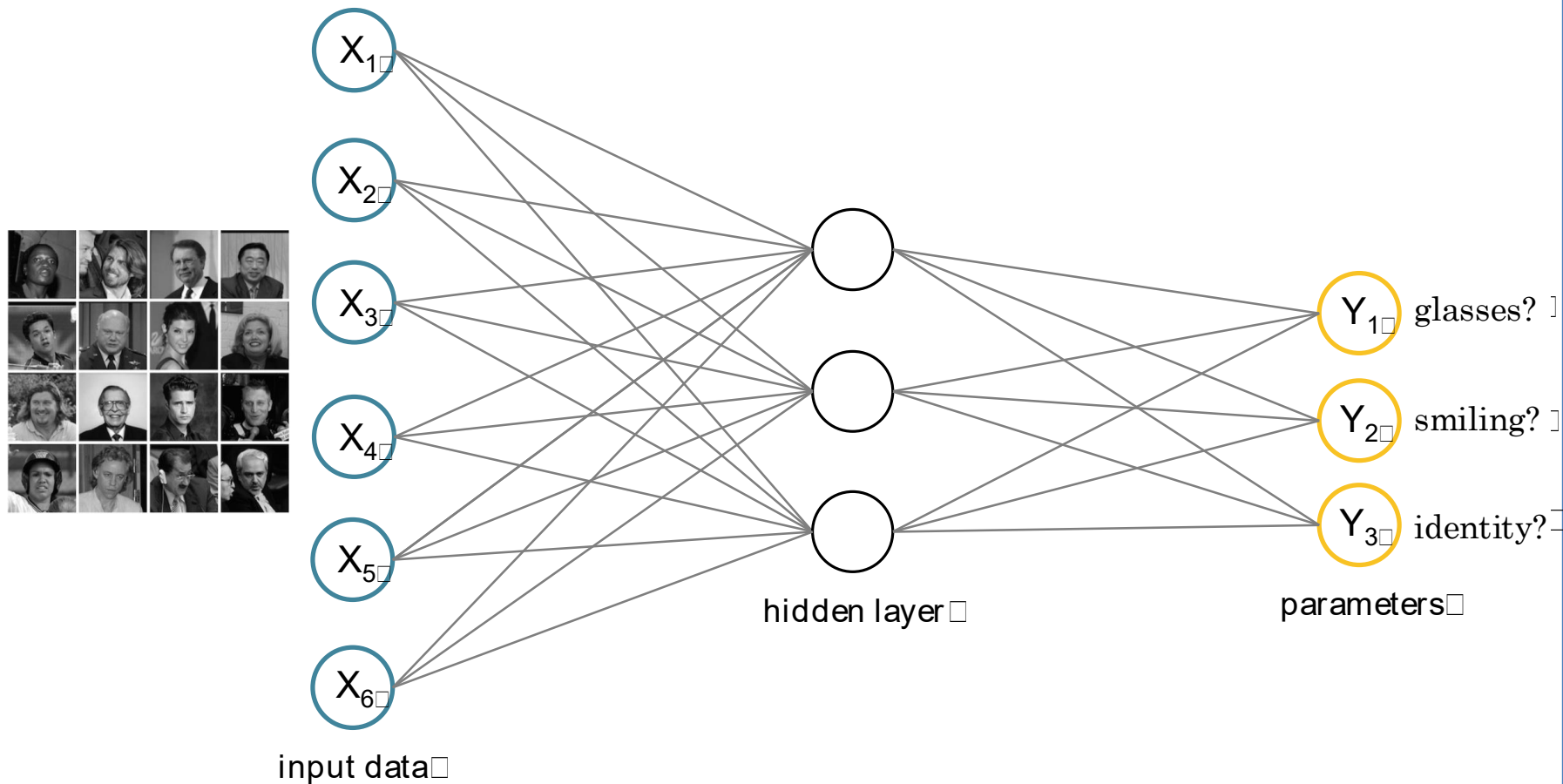
3) ReLU

- (+) 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

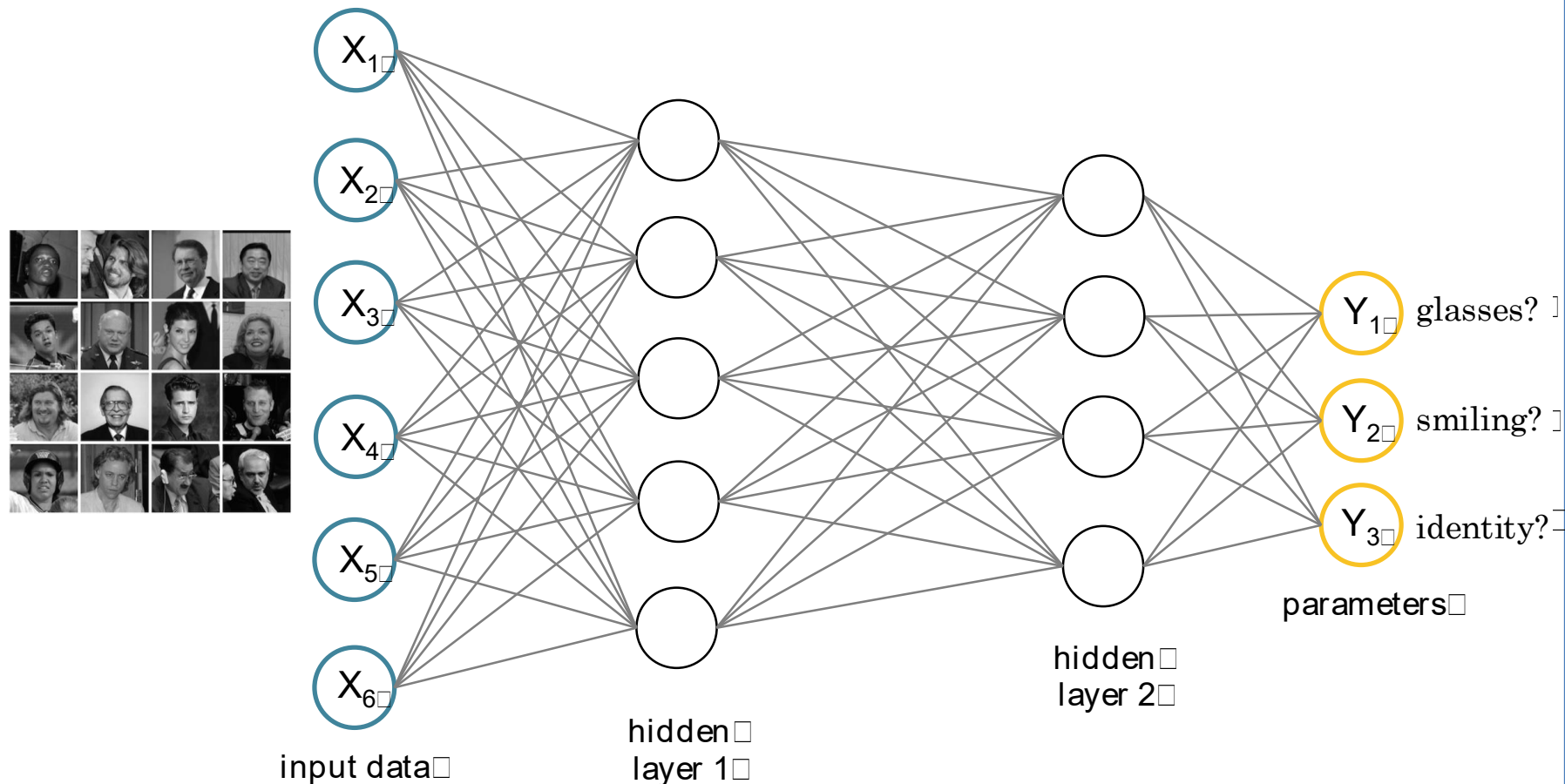
Goal: find a function between input and output



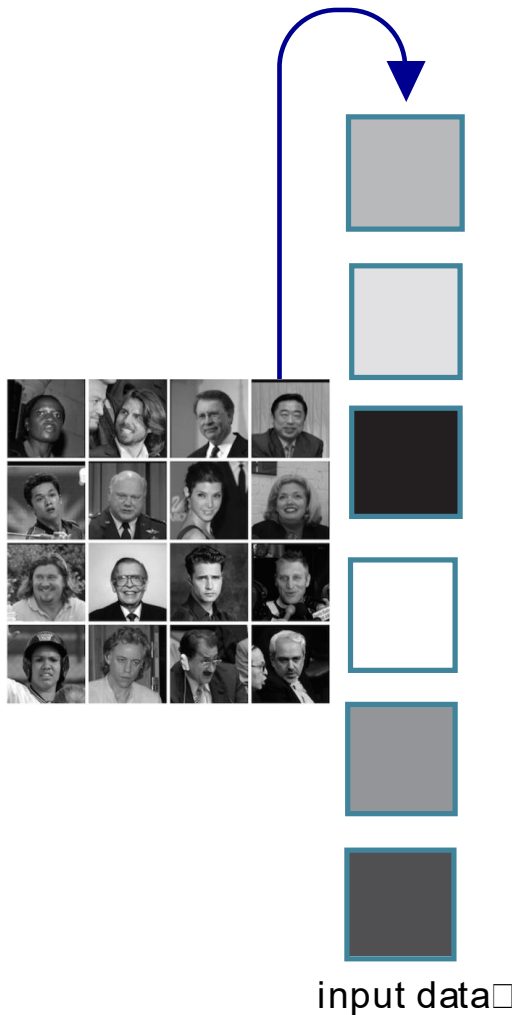
First idea: one hidden layer



Second idea: more hidden layers (“deep” learning)



Flatten pixels of image into a single vector



Detour to autoencoders



$X_{1\#}$

$X_{2\#}$

$X_{3\#}$

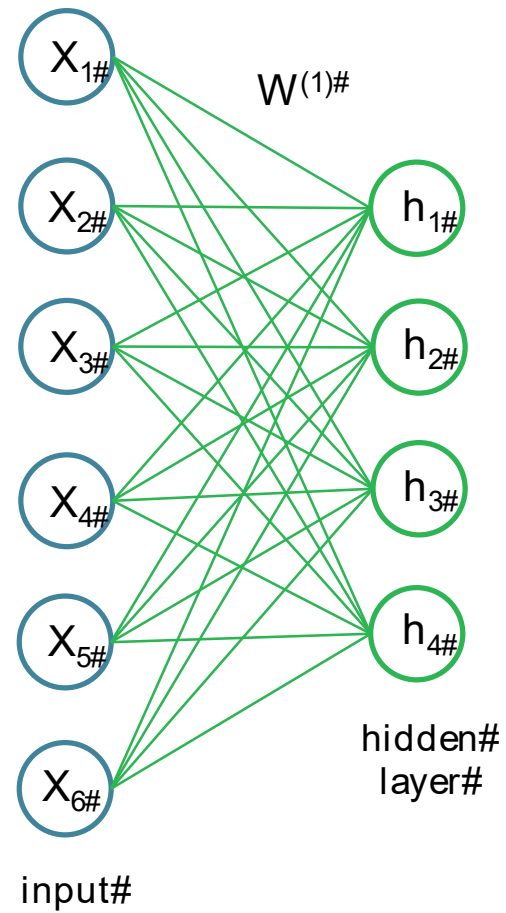
$X_{4\#}$

$X_{5\#}$

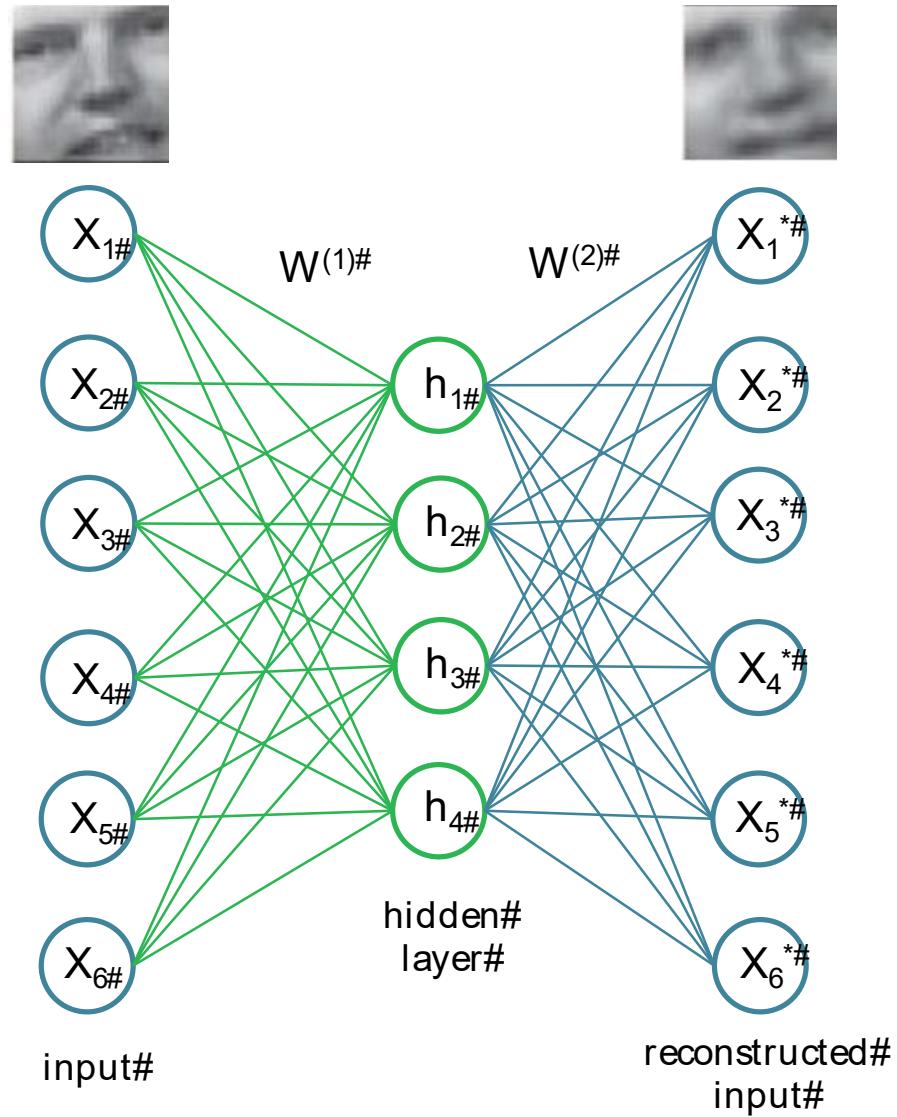
$X_{6\#}$

input#

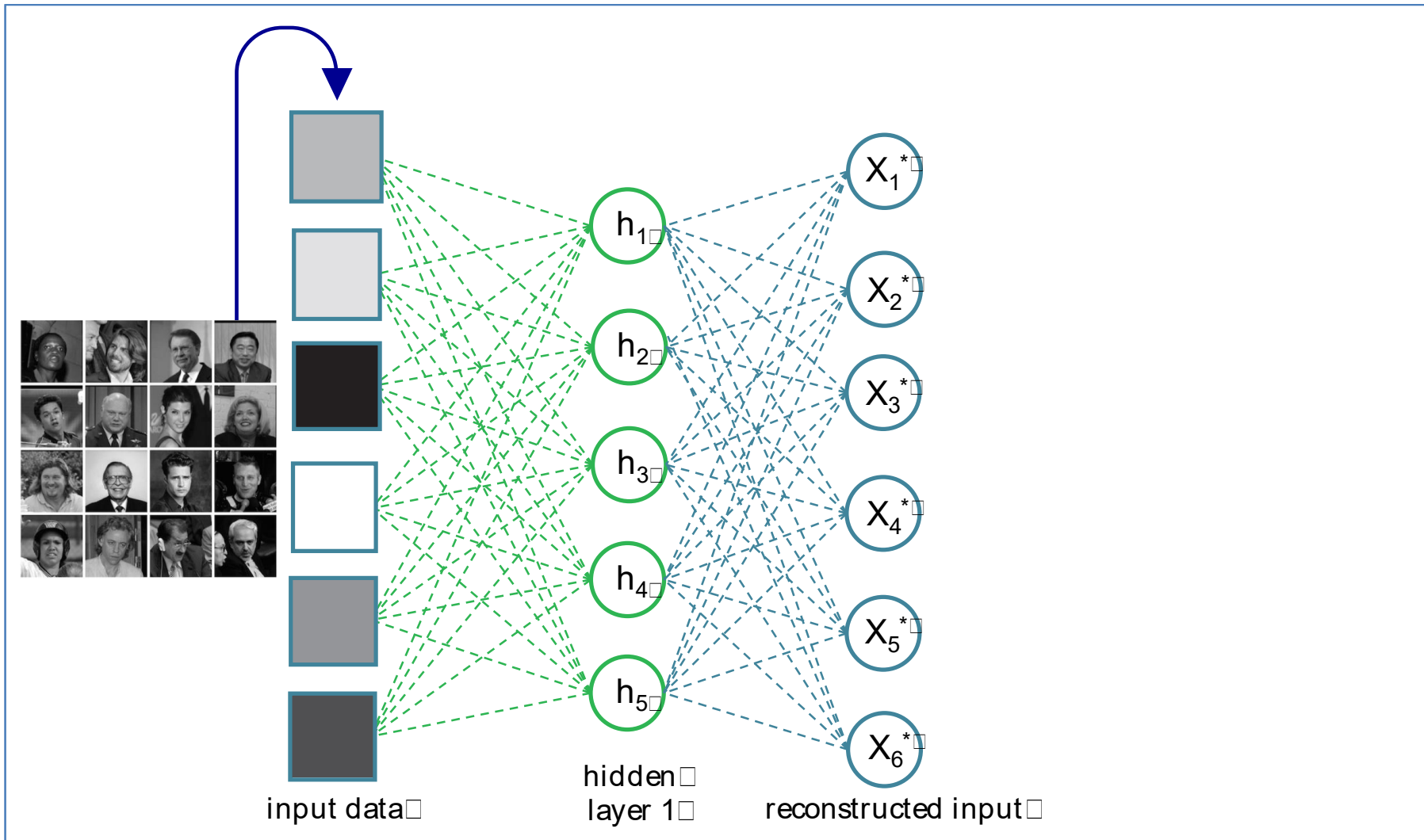
Detour to autoencoders



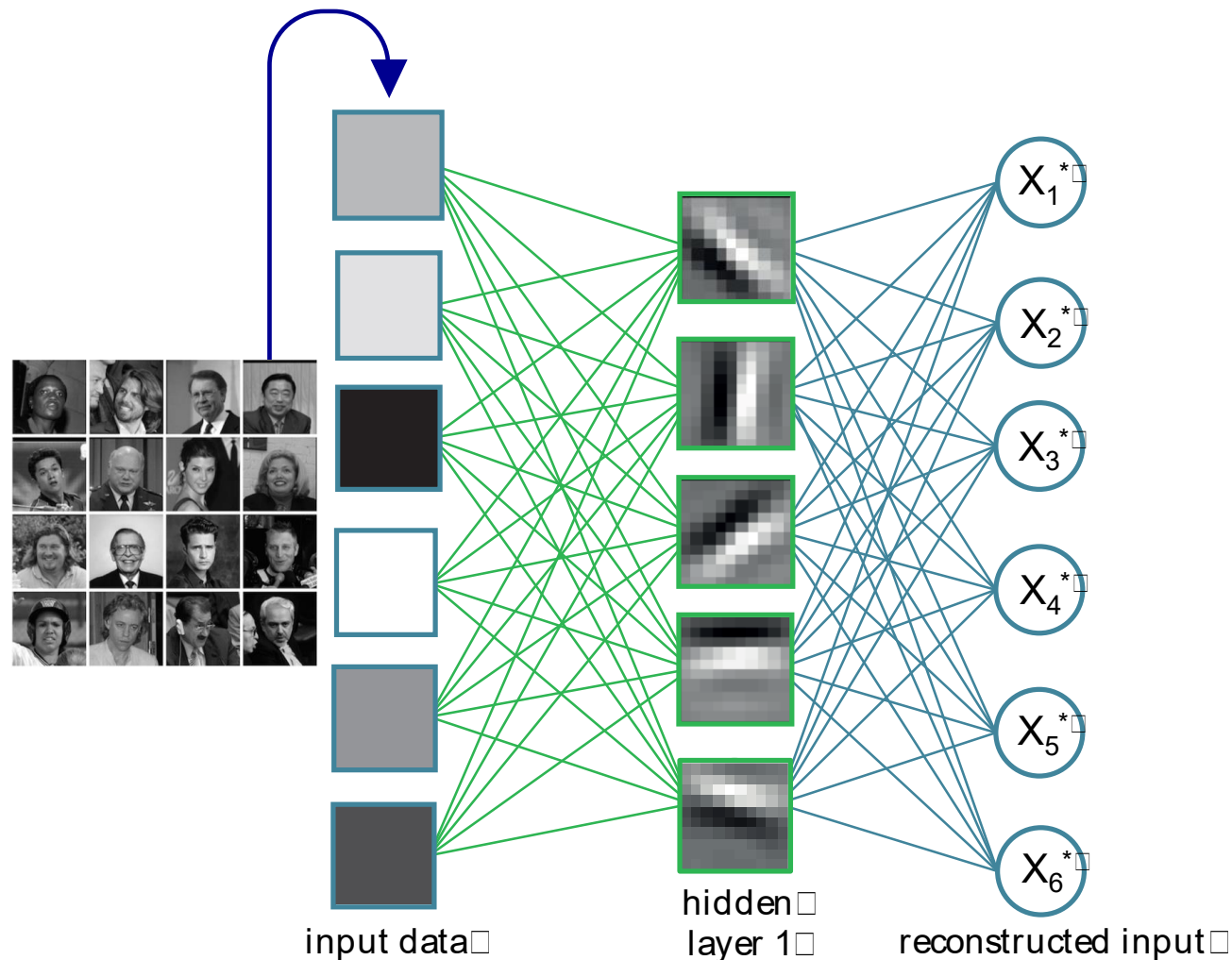
Detour to autoencoders



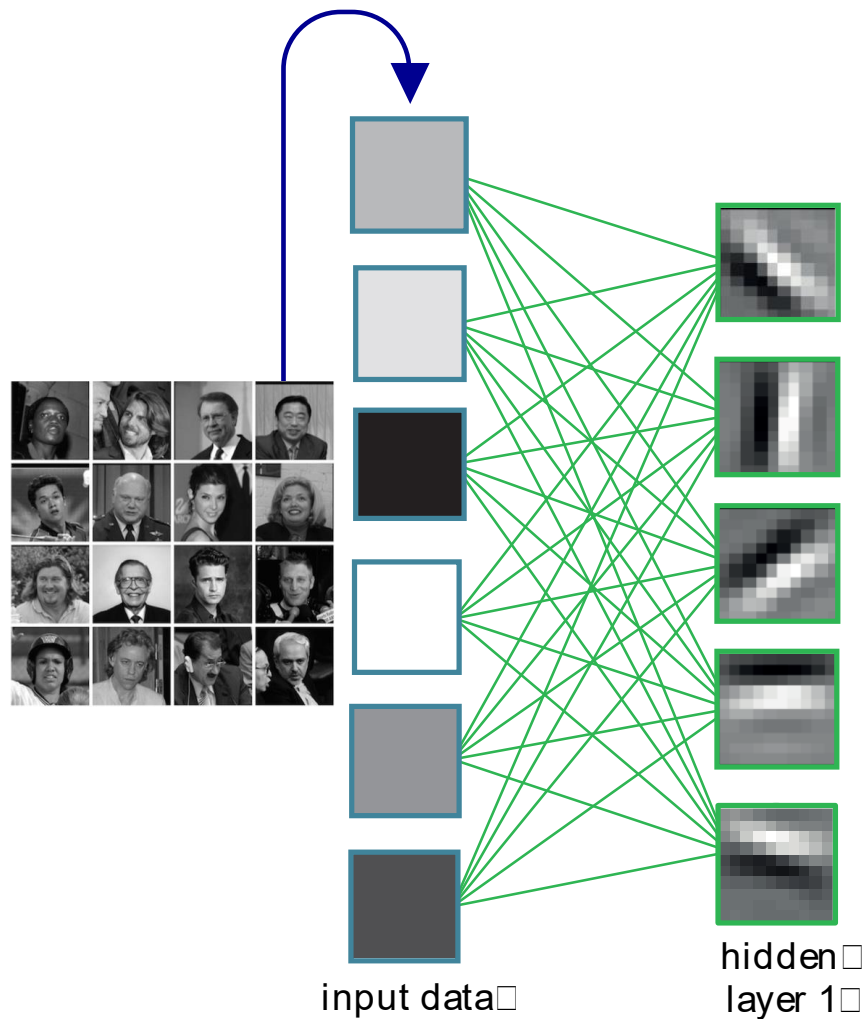
Use unsupervised pre-training to find a function from the input to itself



Hidden units can be interpreted as edges



Now: throw away reconstruction and input

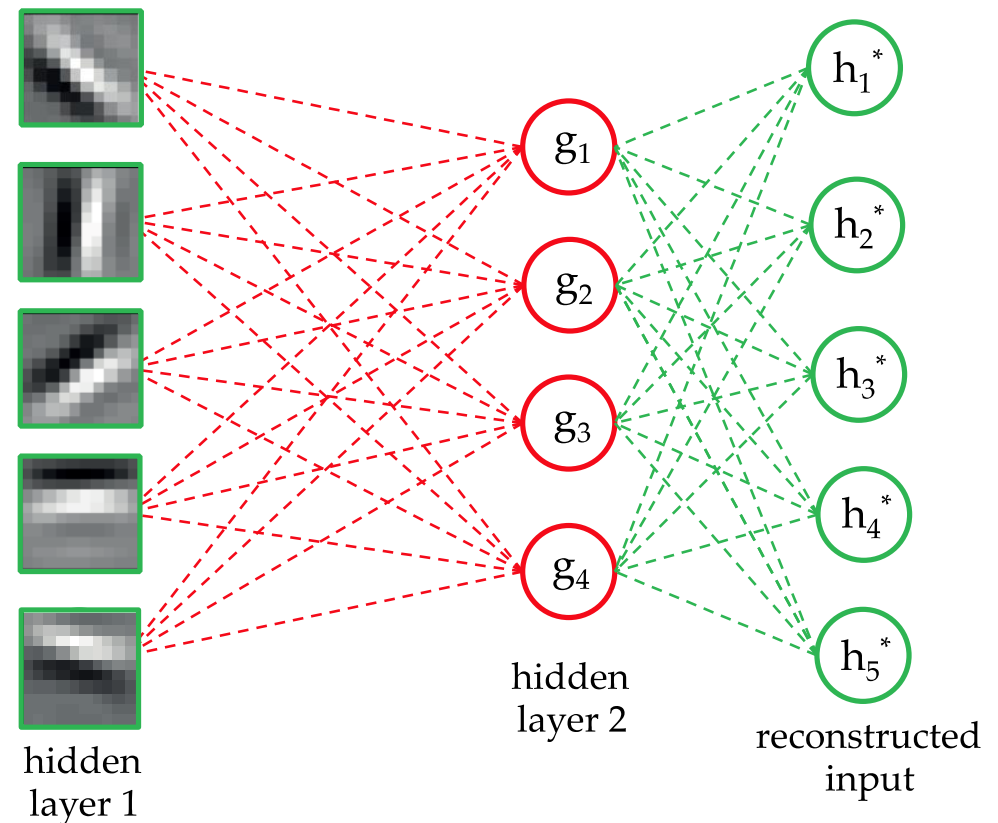


Now: throw away reconstruction and input

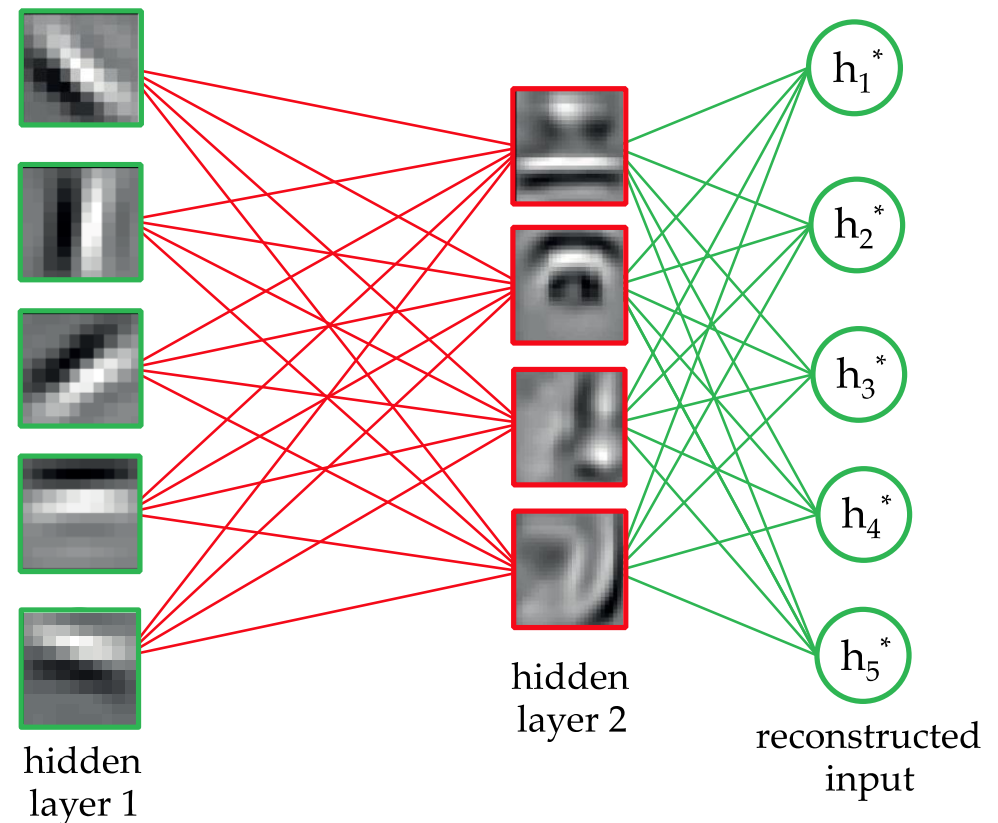


hidden
layer 1

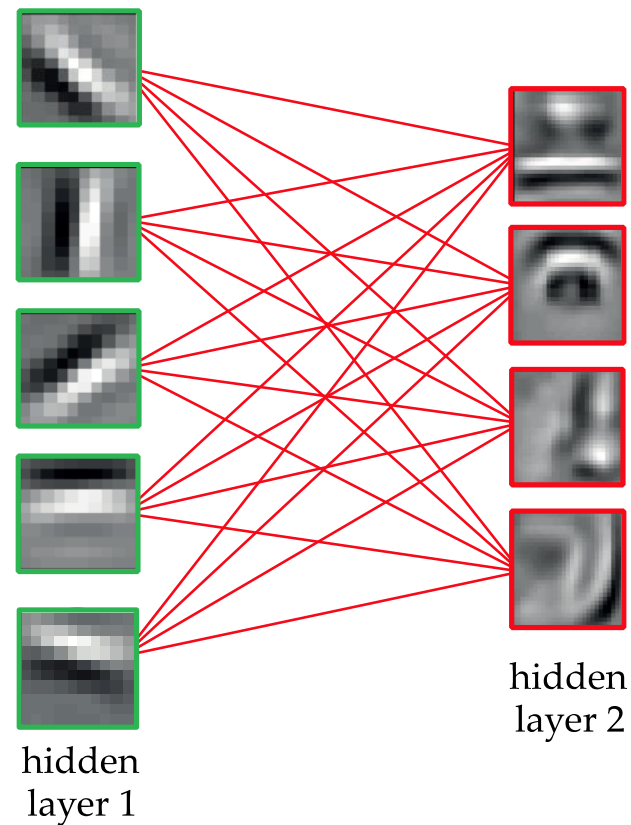
Then repeat the entire process for each layer



Then repeat the entire process for each layer



Then repeat the entire process for each layer

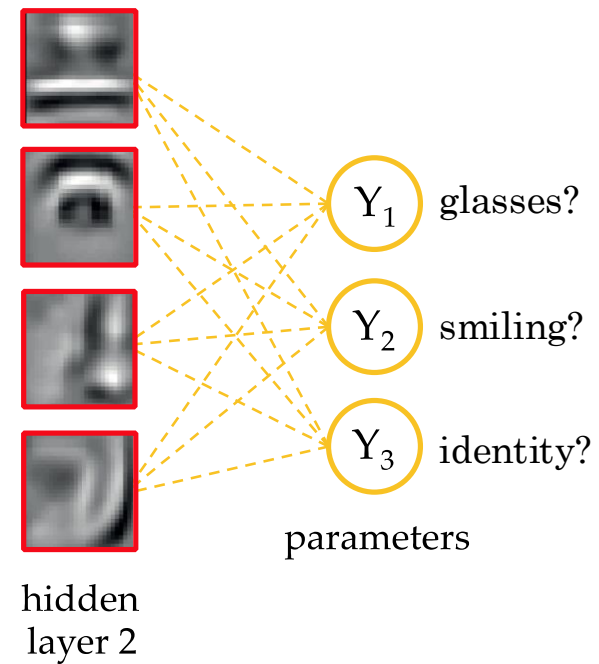


Then repeat the entire process for each layer

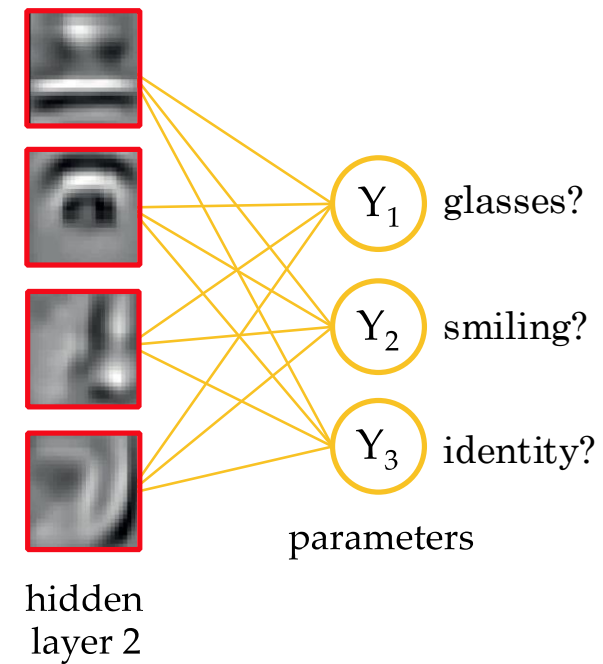


hidden
layer 2

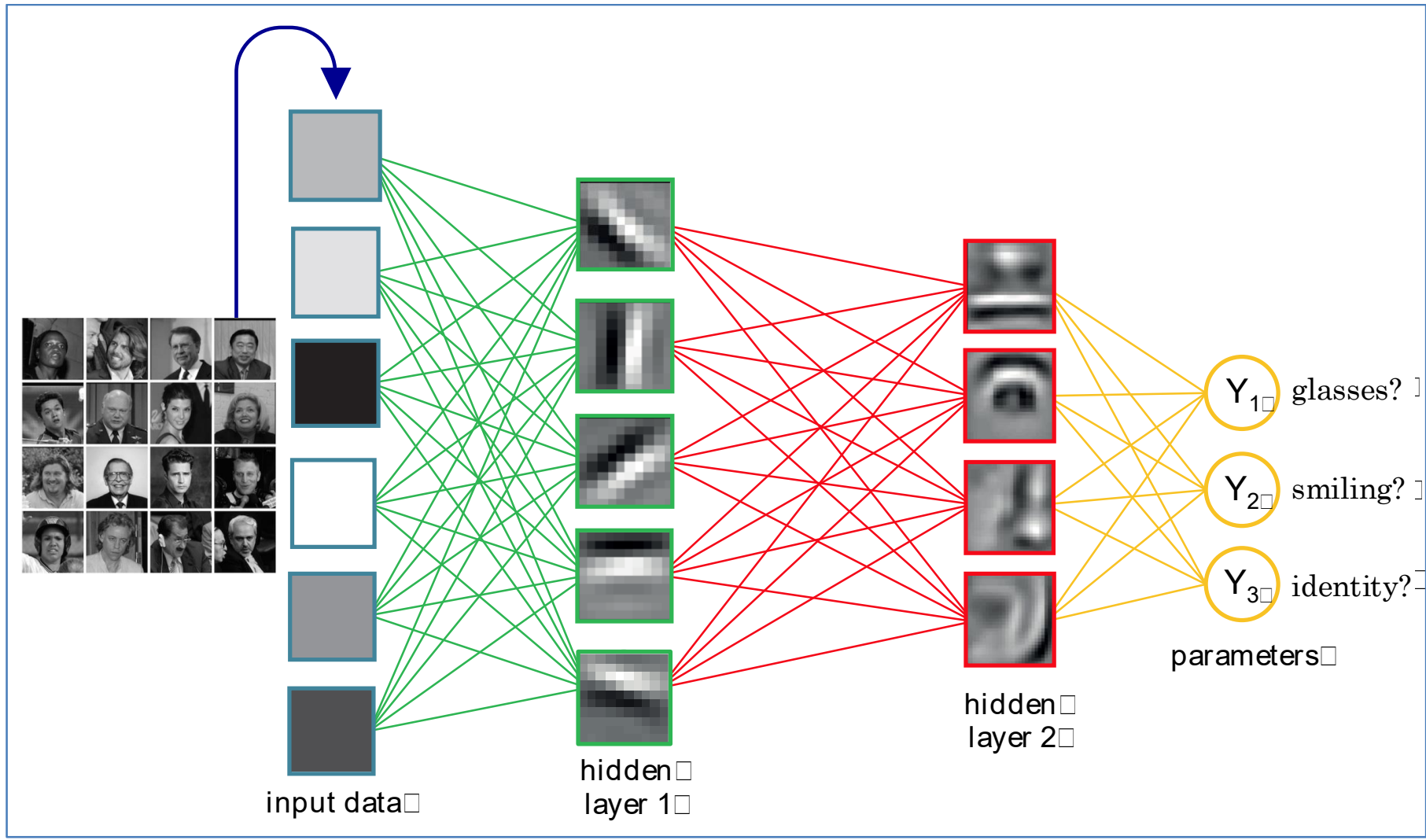
In the last layer, use the outputs (supervised)



In the last layer, use the outputs (supervised)

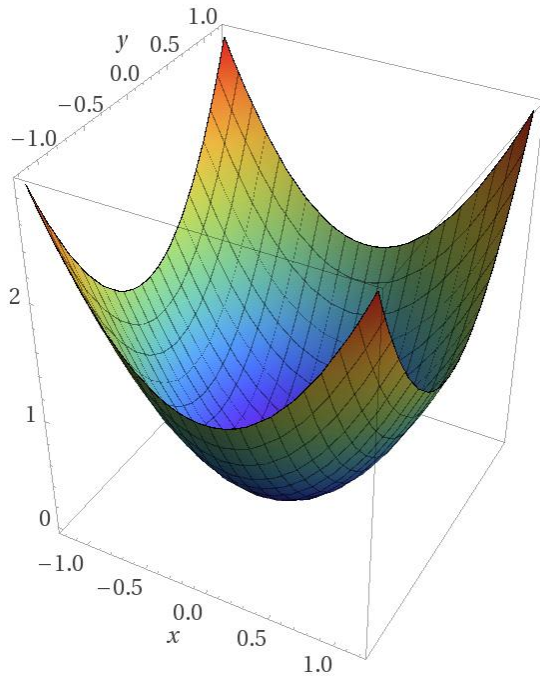


Finally, “fine-tune” the entire network!



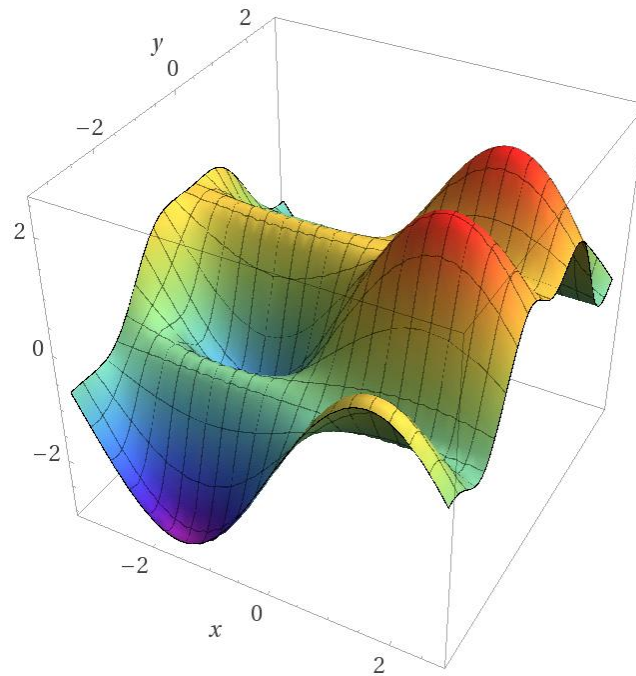
Takeaways

- As the number of parameters grows, a non-convex function often has more and more local minima
- Starting at a “good” point is crucial!



Computed by Wolfram|Alpha

Convex



Computed by Wolfram|Alpha

Non-convex

Takeaways

- Unsupervised pre-training uses latent structure in the data as a starting point for weight initialization
- After this process, the network is “fine-tuned”
- In practice this has been found to increase accuracy on specific tasks (which could be specified after feature learning)

Weight initialization

- We still have to initialize the pre-training
- All 0's initialization is bad! Causes nodes to compute the same outputs, so then the weights go through the same updates during gradient descent
- Need asymmetry! => usually use small random values

Mini-batches

- So far in this class, we have considered *stochastic gradient descent*, where one data point is used to compute the gradient and update the weights
- On the flipside is *batch gradient descent*, where we compute the gradient with respect to all the data, and then update the weights
- A middle ground uses *mini-batches* of examples before updating the weights

Notes about scores and softmax

- The output of the final fully connected layer is a vector of length K (number of classes)

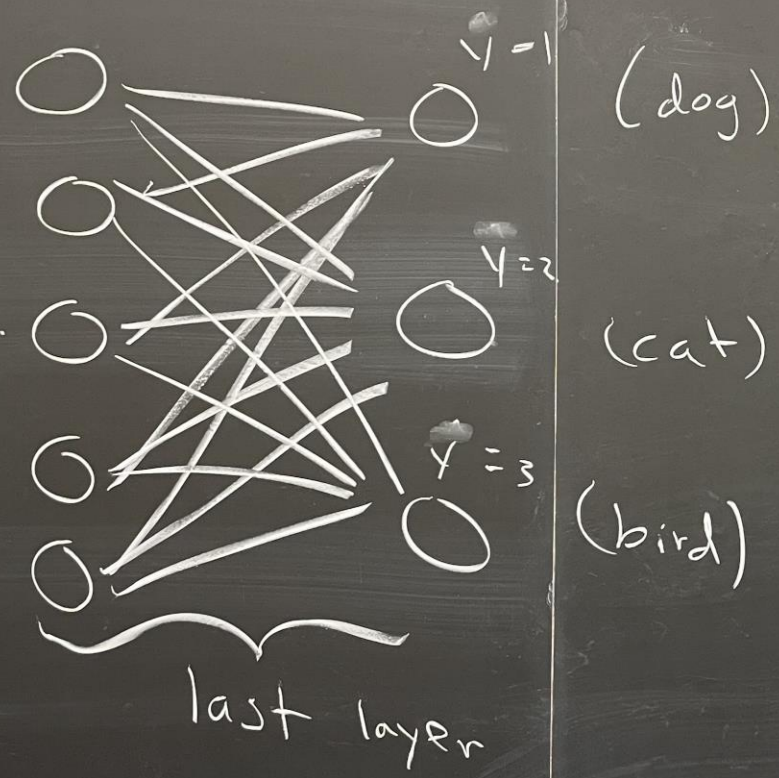
Notes about scores and softmax

- The output of the final fully connected layer is a vector of length K (number of classes)
- The raw scores are transformed into probabilities using the *softmax function*: (let s_k be the score for class k)

$$\hat{y}_k = \frac{e^{s_k}}{\sum_{j=1}^K e^{s_j}}$$

- Then we apply *cross-entropy loss* to these probabilities

Softmax



Output scores

$$\begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} -7.2 \\ 10.9 \\ 0.718 \end{bmatrix}$$

neural network output

probability

take argmax

$$\hat{y} = 2$$

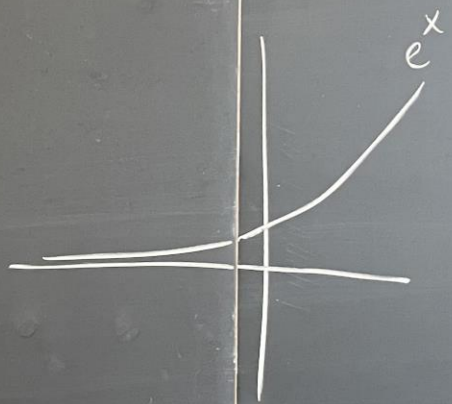
probability?

~~$$\frac{-7.2}{-7.9 + 10.9 + 0.718}$$~~

Can't have negative probability!

SOF tmax

$$\Rightarrow \frac{e^{-7.2}}{e^{-7.2} + e^{10.9} + e^{-0.718}}$$



ork
put

$$\hat{y}_k = \frac{e^{s_k}}{\sum_{j=1}^K e^{s_j}}$$

← probability for class k

Motivation for moving away from FC architectures

- For a $32 \times 32 \times 3$ image (very small!) we have $p=3072$ features in the input layer
- For a $200 \times 200 \times 3$ image, we would have $p=120,000$! *doesn't scale*

Motivation for moving away from FC architectures

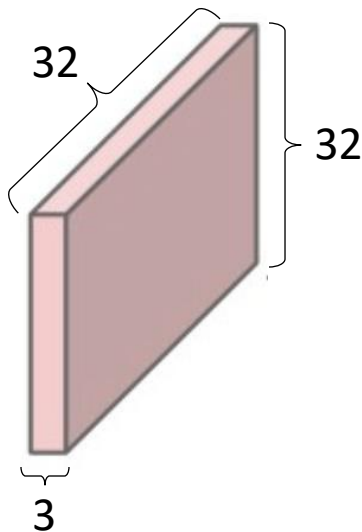
- For a $32 \times 32 \times 3$ image (very small!) we have $p=3072$ features in the input layer
- For a $200 \times 200 \times 3$ image, we would have $p=120,000$! *doesn't scale*
- FC networks do not explicitly account for the structure of an image and the correlations/relationships between nearby pixels

Idea: 3D volumes of neurons

- Do not “flatten” image, keep it as a volume with *width*, *height*, and *depth*

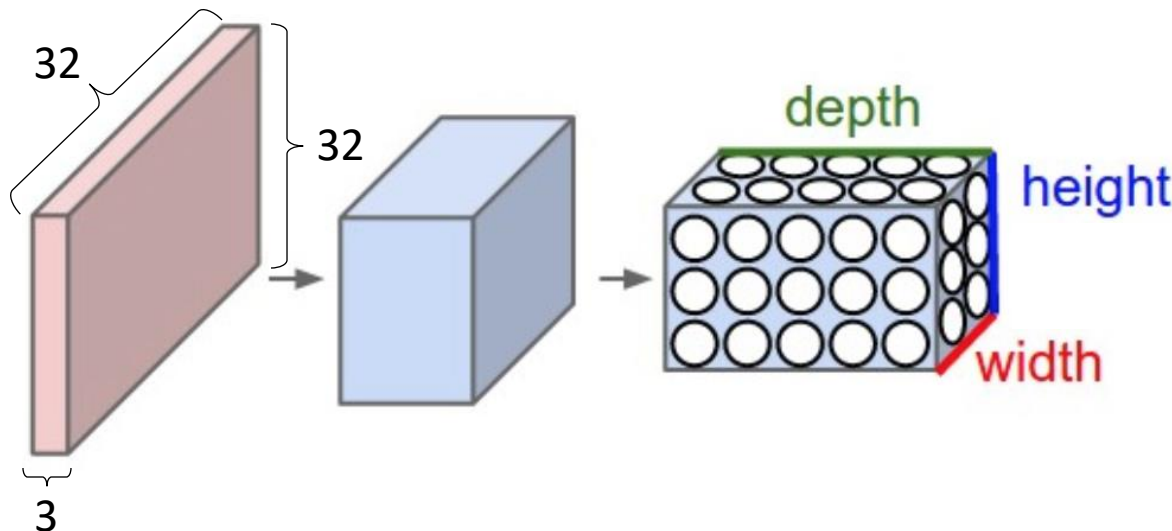
Idea: 3D volumes of neurons

- Do not “flatten” image, keep it as a volume with *width*, *height*, and *depth*
- For **CIFAR-10**, we would have:
 - Width=32, Height=32, Depth=3



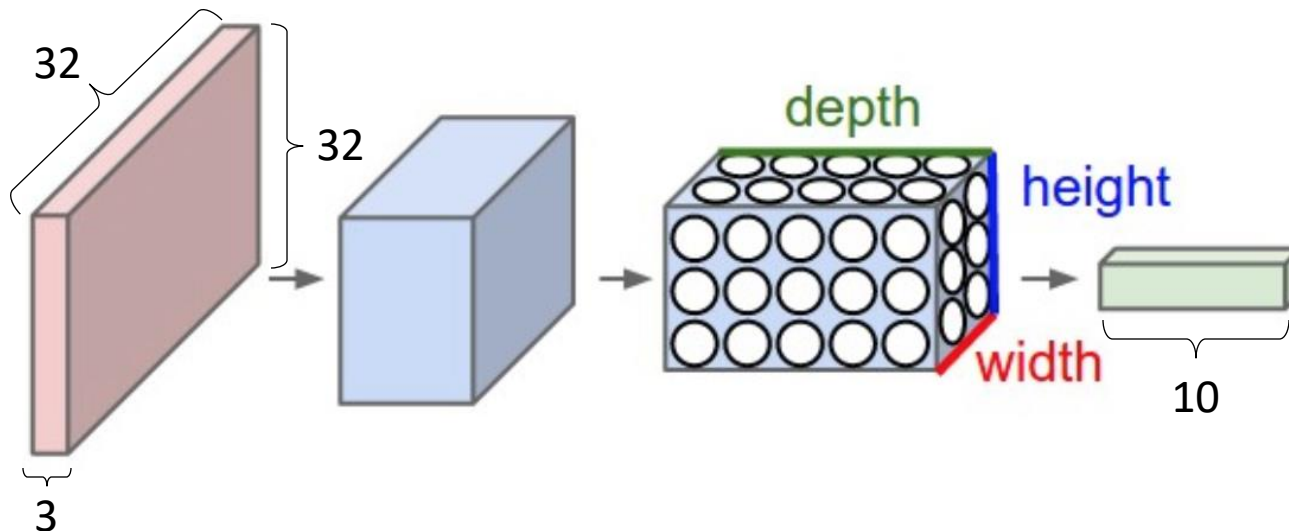
Idea: 3D volumes of neurons

- Do not “flatten” image, keep it as a volume with *width*, *height*, and *depth*
- For **CIFAR-10**, we would have:
 - Width=32, Height=32, Depth=3
- Each layer is also a 3 dimensional volume



Idea: 3D volumes of neurons

- Do not “flatten” image, keep it as a volume with *width*, *height*, and *depth*
- For **CIFAR-10**, we would have:
 - Width=32, Height=32, Depth=3
- Each layer is also a 3 dimensional volume
- The output layer is 1x1xC, where C is the number of classes (10 for CIFAR-10)



Outline

- Kernel Density Estimation (KDE)
- Missing data
- Neural networks
- Go over Midterm 2

Midterm 2 Grades

- 90-100% A
- 80-89% B
- 70-79% C
- 60-69% D
- Below 60%: not passing, please meet with me
- Note: as per the syllabus, you must pass at least one exam to pass the course
- Any questions about the exam: bring to me within one week