CS 260: Foundations of Data Science

Prof. Thao Nguyen

Fall 2024

HAVERFORD
COLLEGE

Materials by Sara Mathieson

Admin

- **Lab 8** grades & feedback posted on Moodle

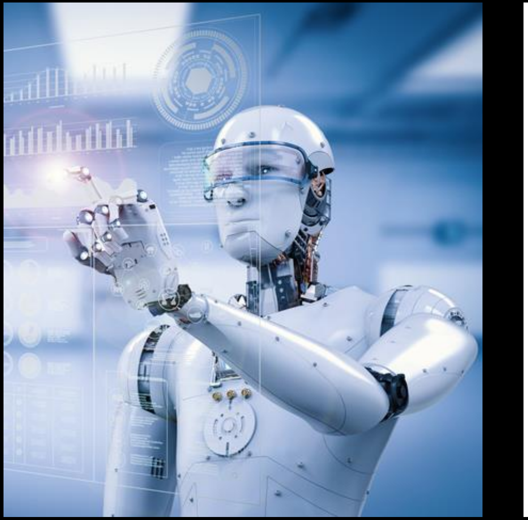
- **End-of-semester survey (link on Piazza)**

Outline for today

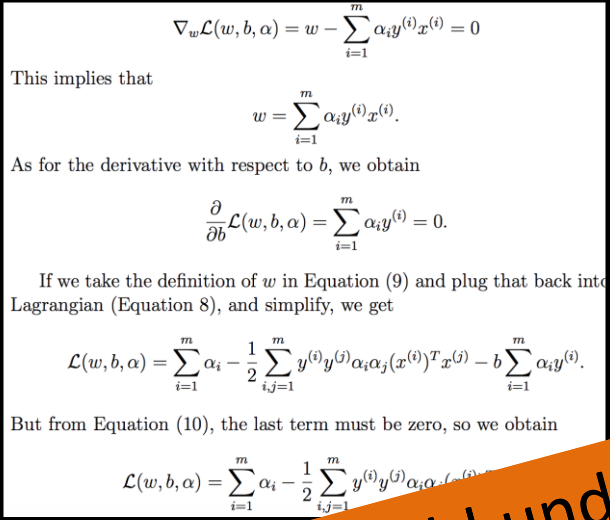
- Neural networks

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 Lagrangian (Equation 8), and simplify, we get

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

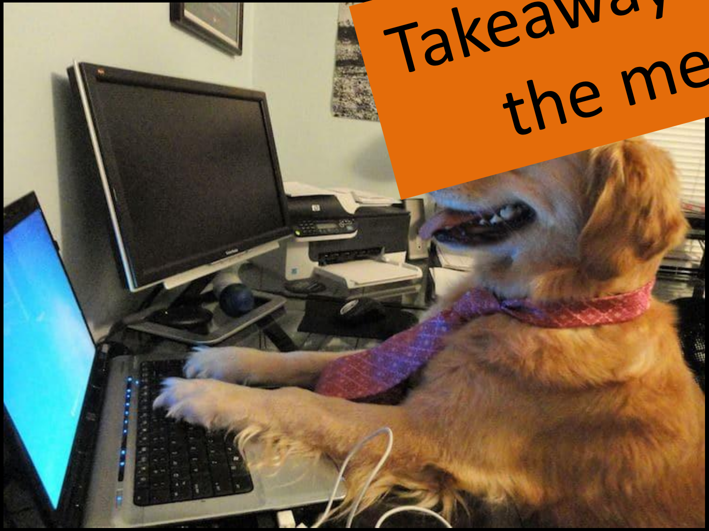
But from Equation (10), the last term must be zero, so we obtain

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

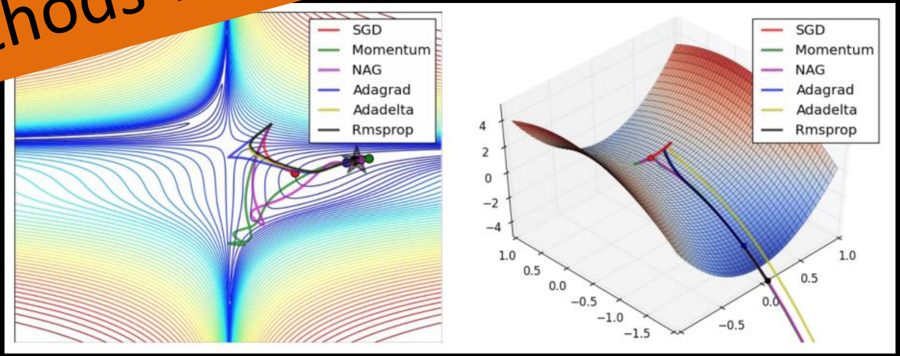
Wh

t other computer scientists think I do

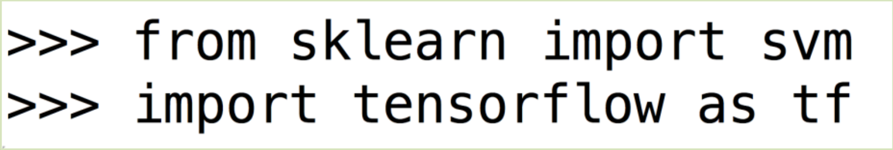
Takeaway: we should understand the methods we are using!



What mathematicians think I do



What I think I do



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

What I really do

Biological Inspiration for Neural Networks

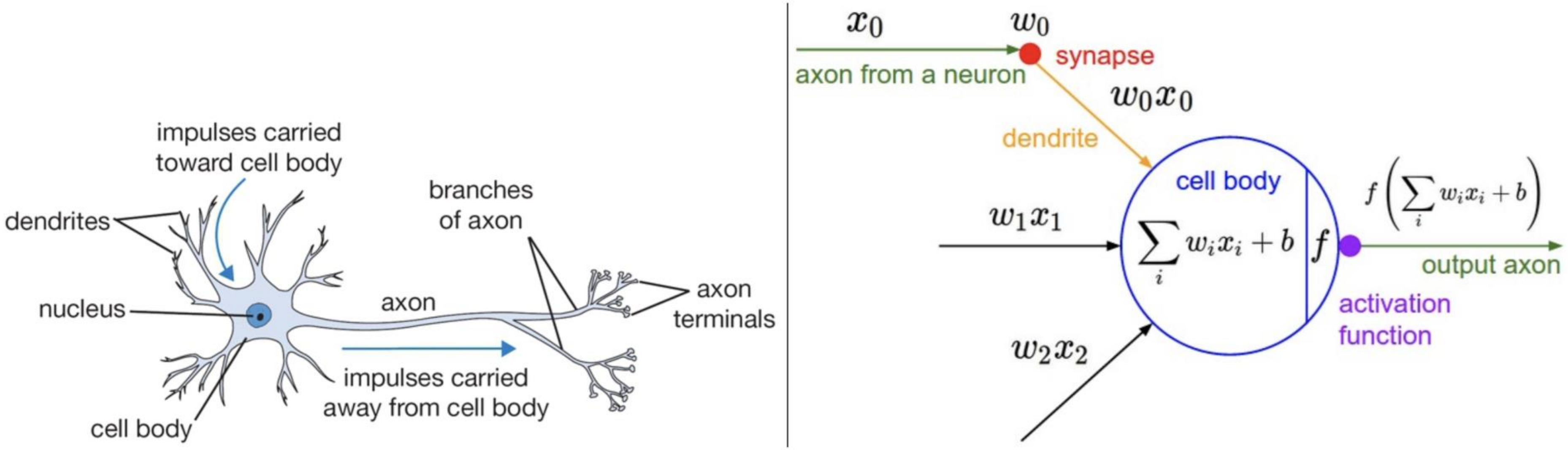
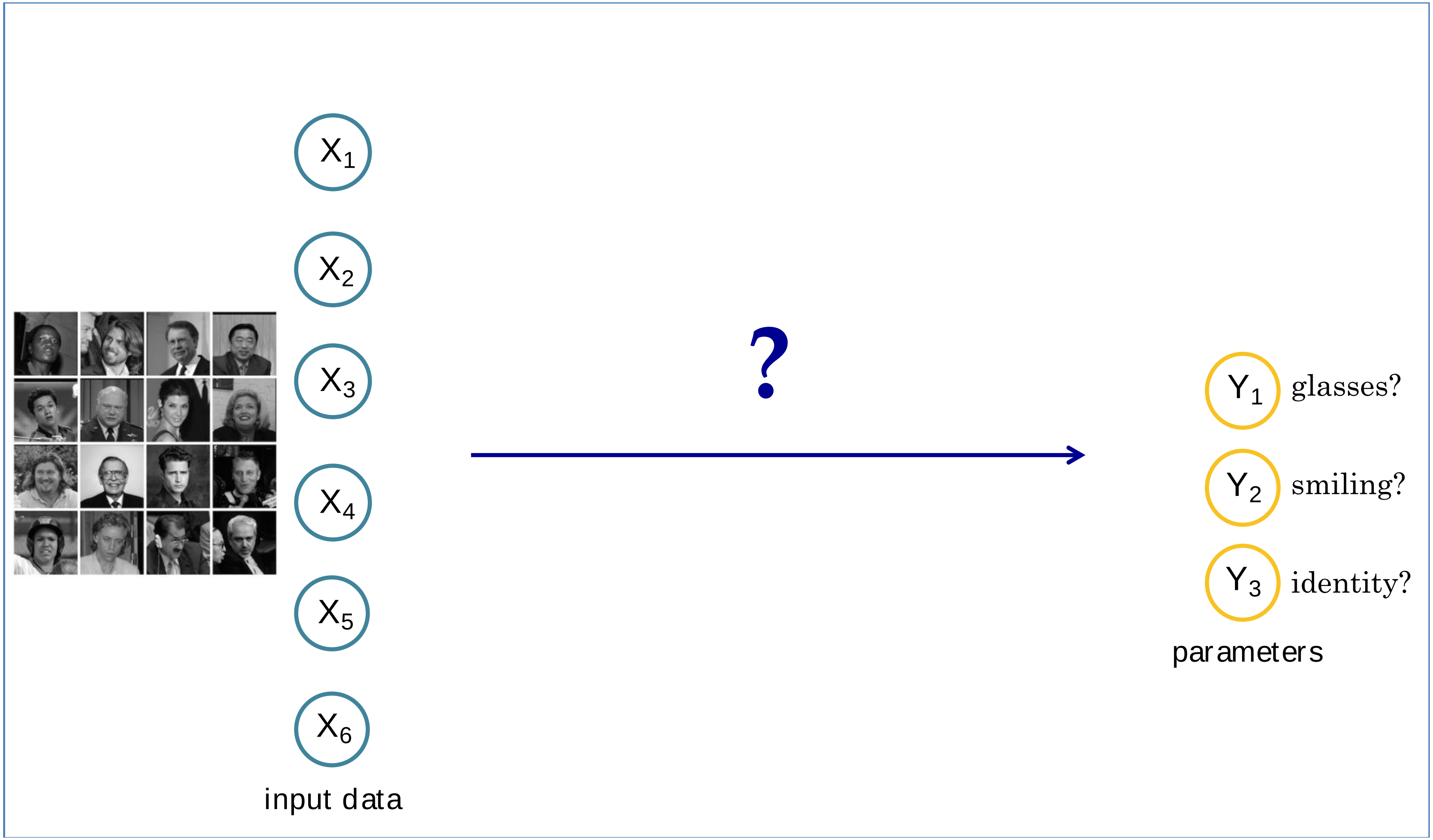


Figure: Stanford CS231n http://cs231n.github.io/neural-networks-1/

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

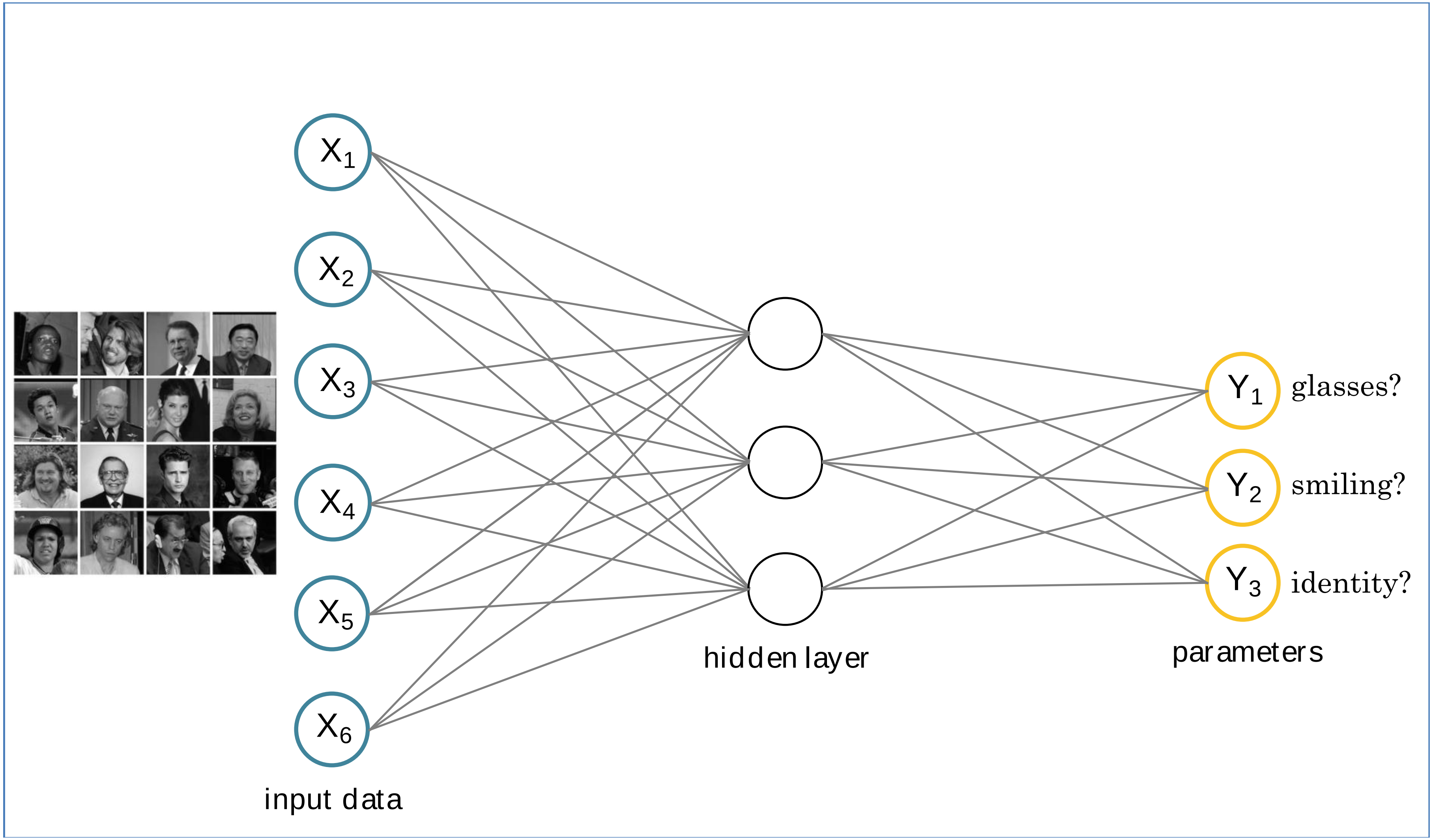


Image: Labeled Faces in the Wild (UMass)

Multi-layer networks = “deep learning”

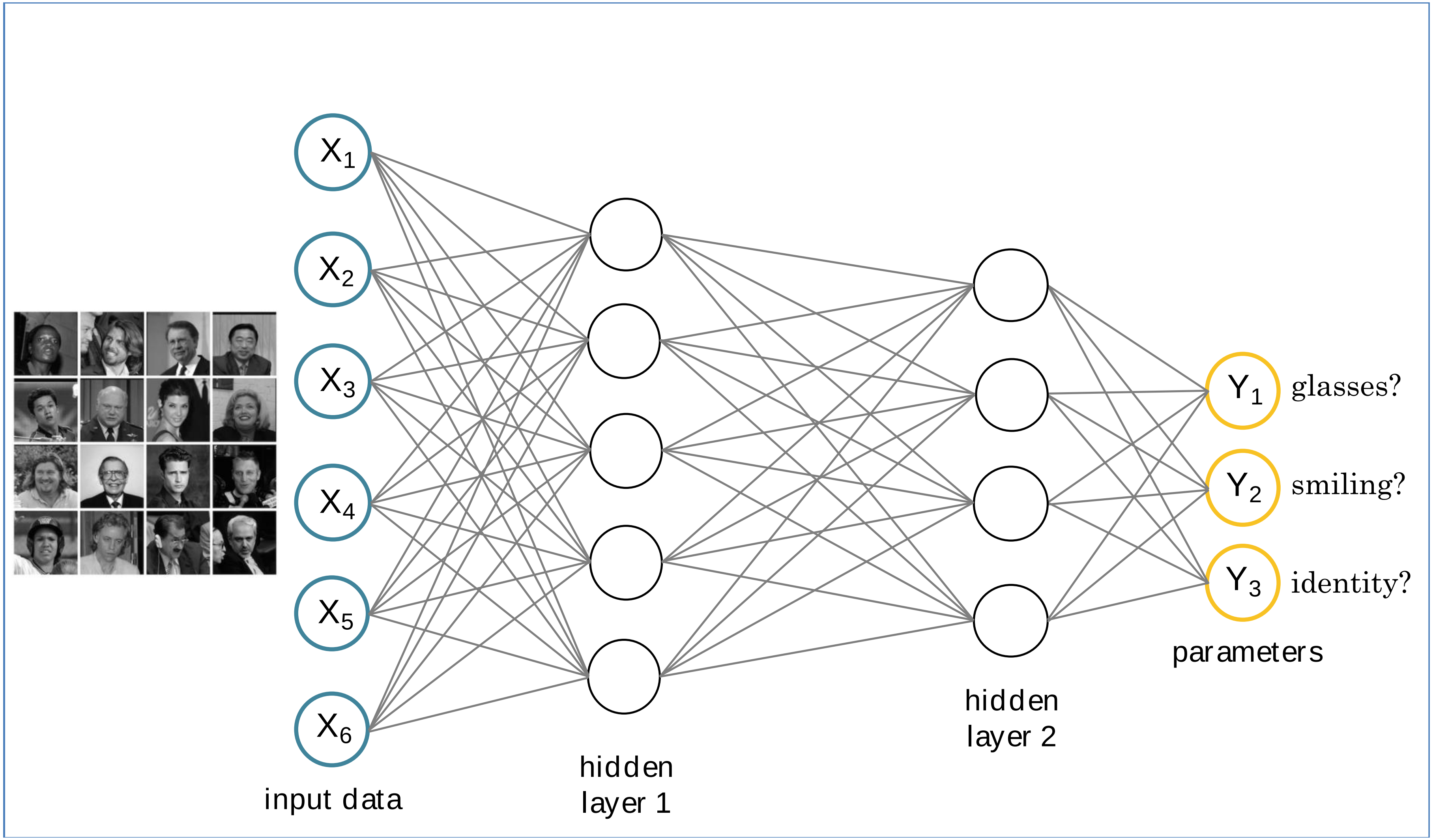
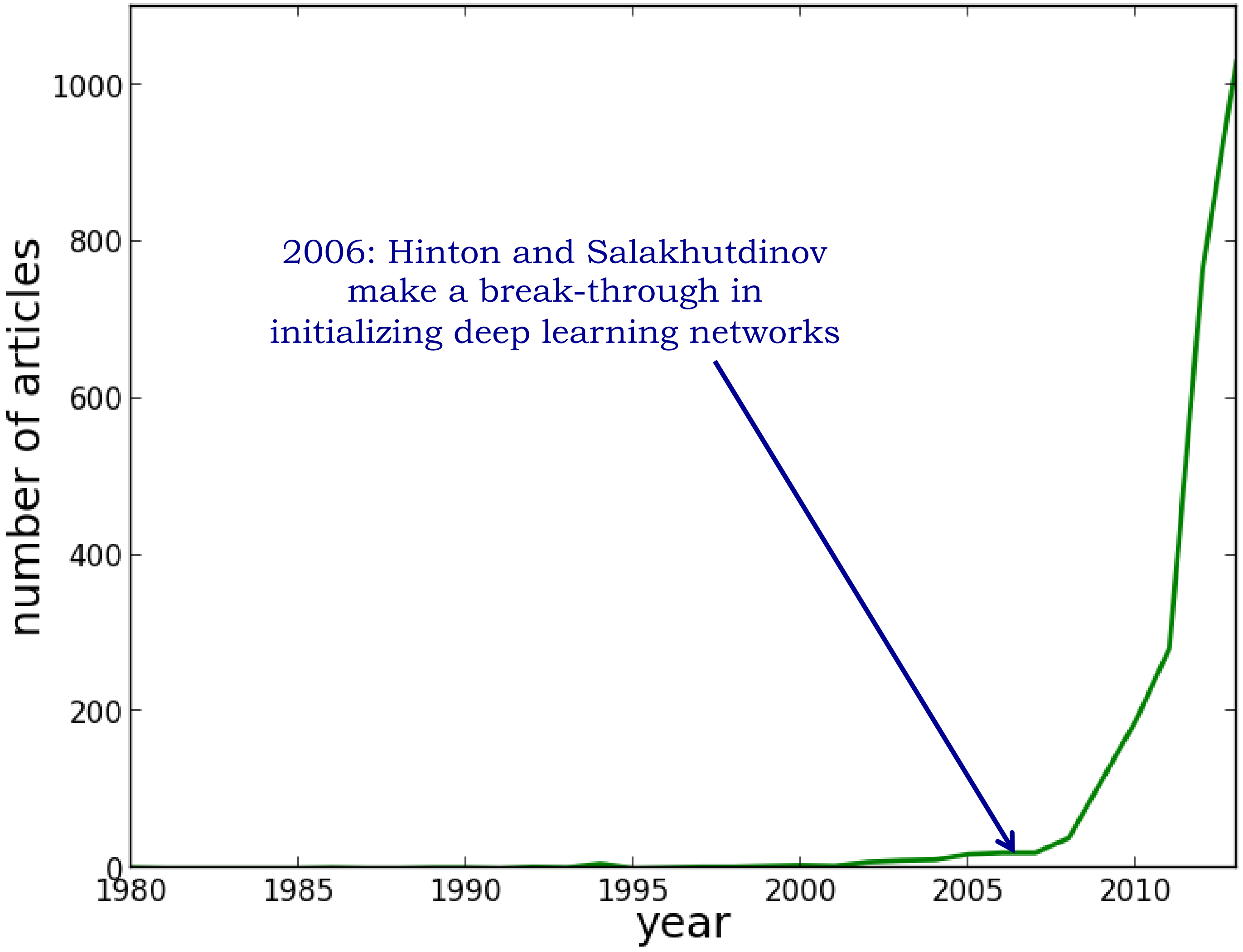


Image: Labeled Faces in the Wild (UMass)

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



2006: Hinton and Salakhutdinov make a break-through in initializing deep learning networks

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

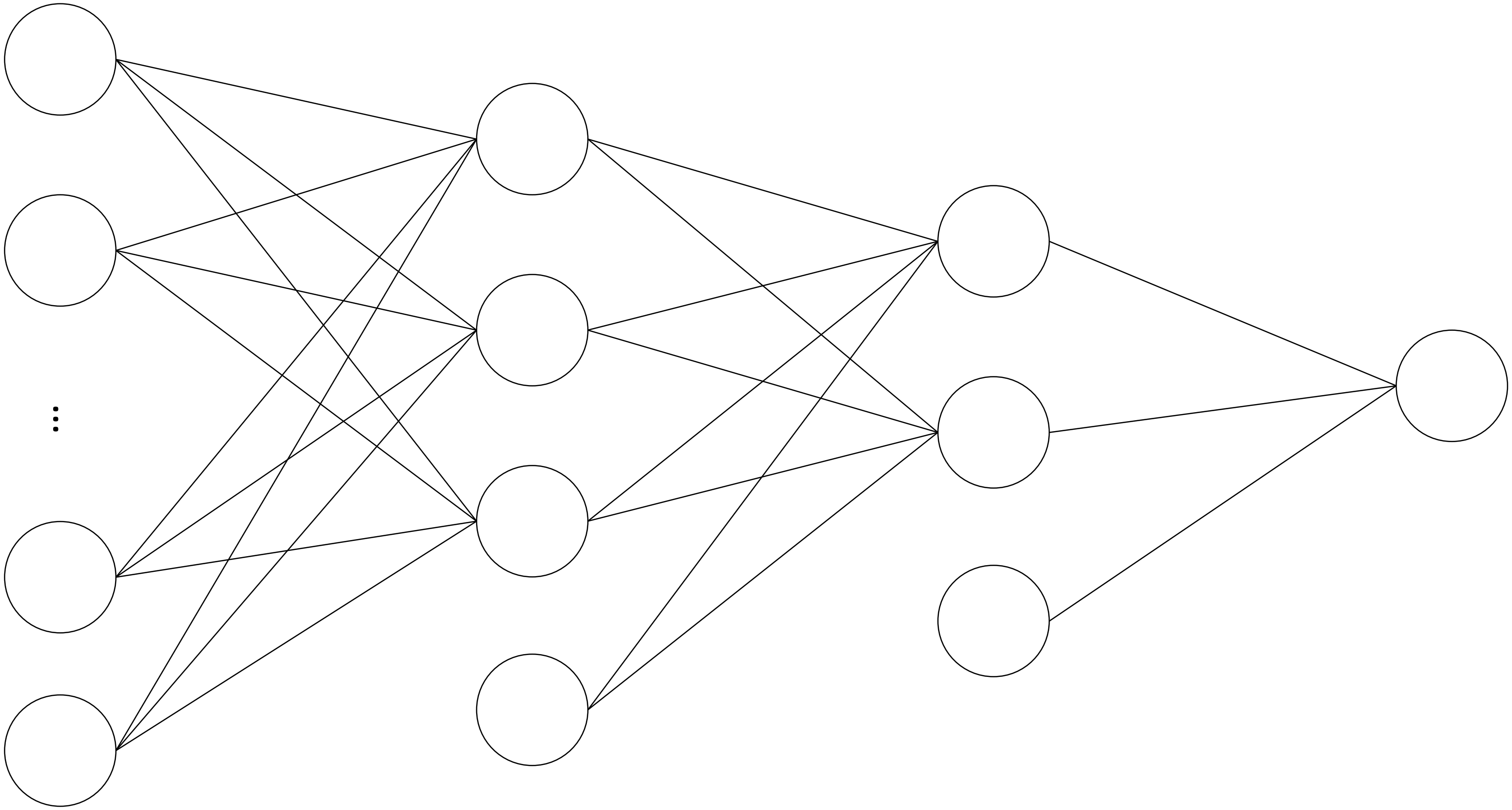
- Neural networks can approximate any function!

- For our purposes in ML, we want to use them to approximate a function from our inputs to our outputs

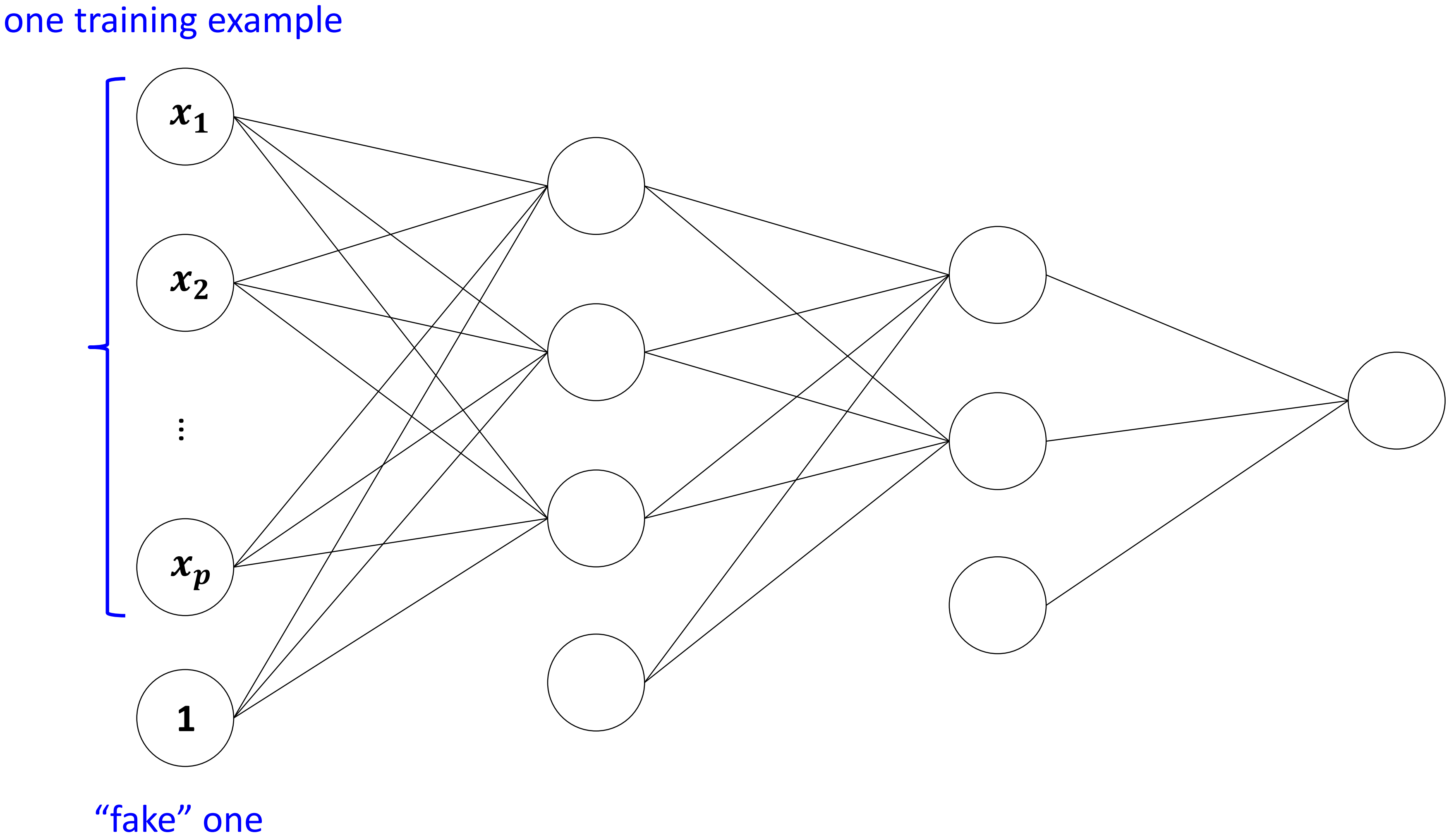
- We will train our network by asking it to minimize the loss between its output and the true output

- We will use SGD-like approaches to minimize loss

Fully Connected Neural Network Architecture

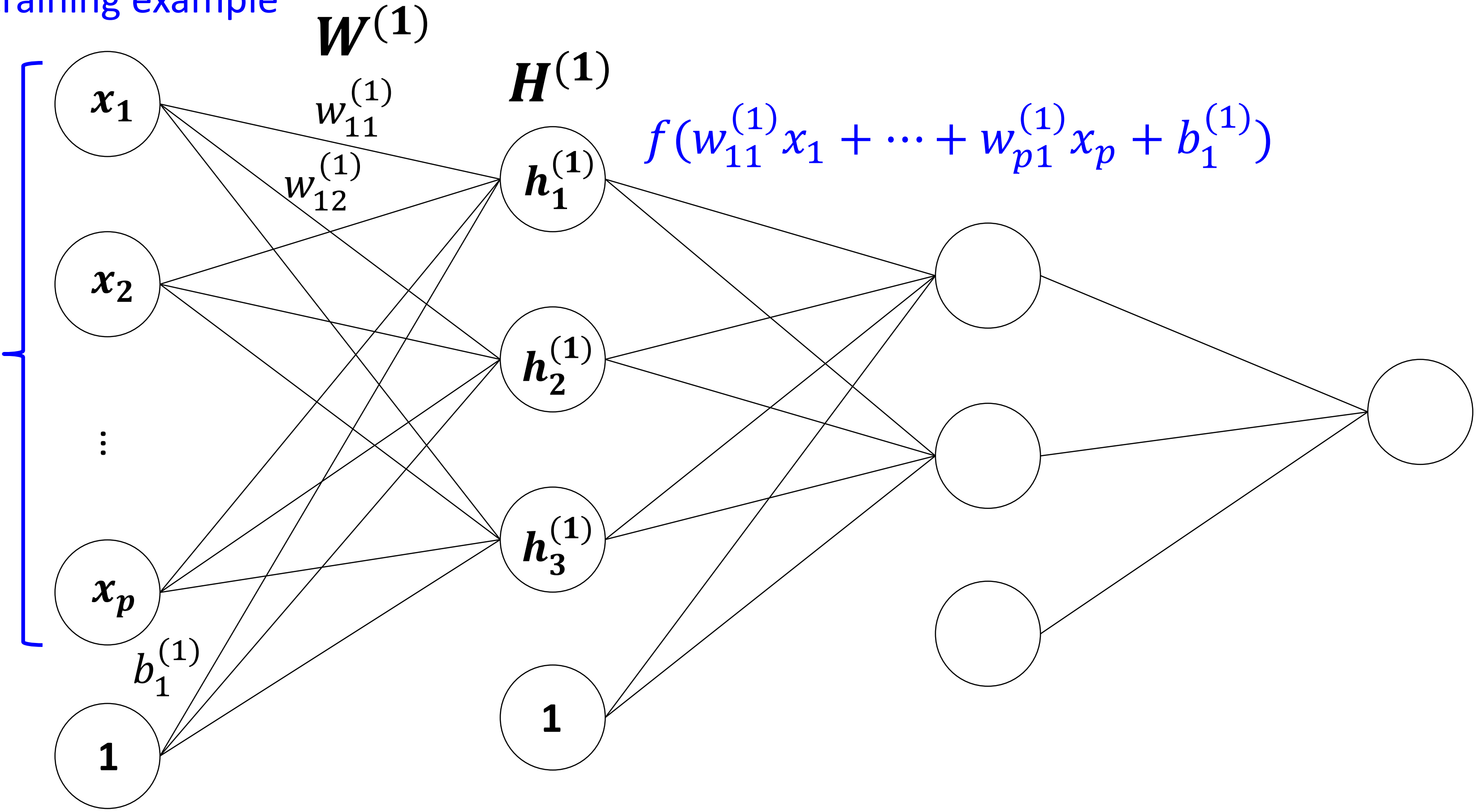


Fully Connected Neural Network Architecture



Fully Connected Neural Network Architecture

one training example

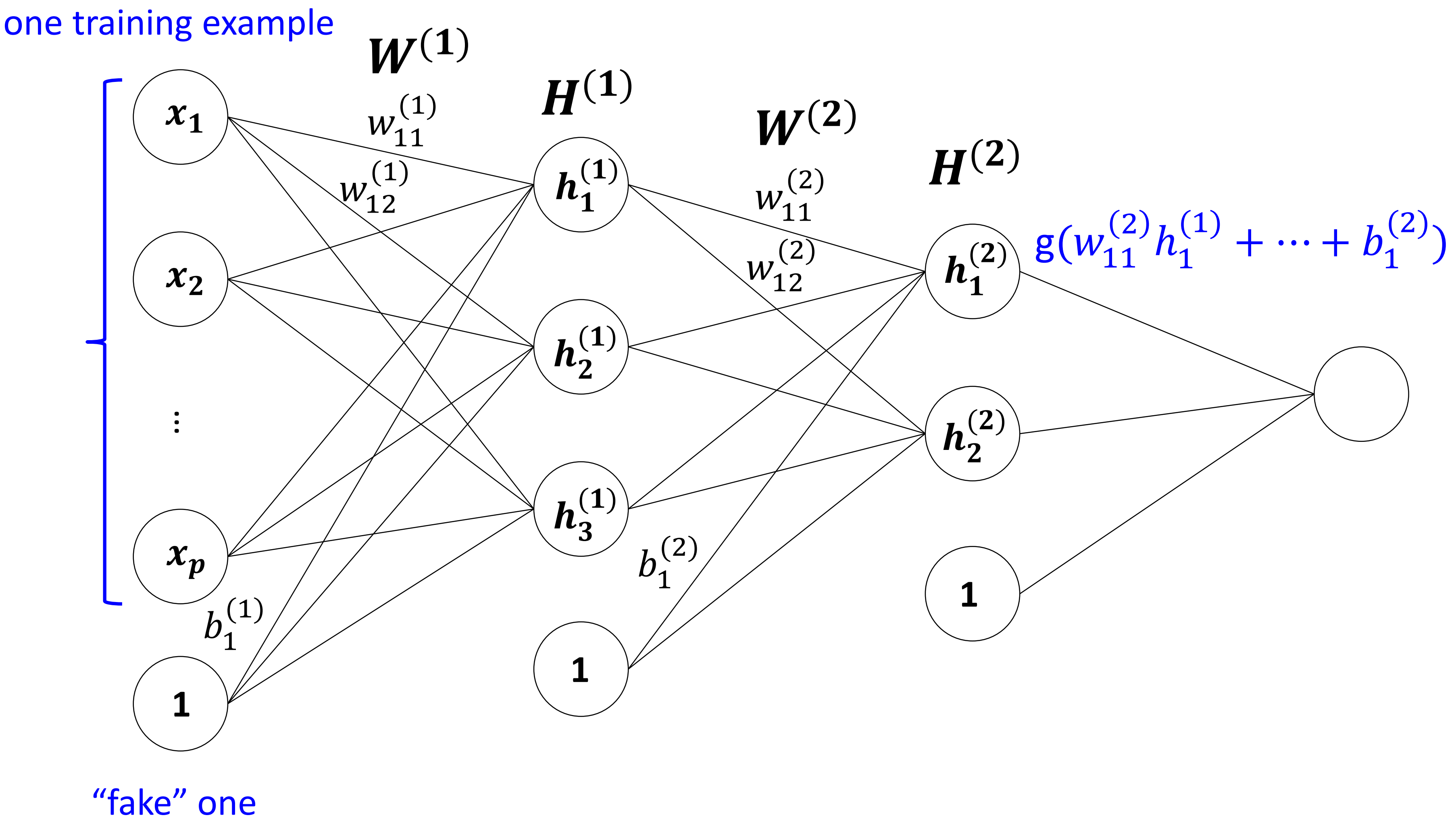


$\boldsymbol{W}^{(\mathbf{1})}$
$\boldsymbol{H}^{(\mathbf{1})}$
$w_{11}^{(1)}$
$w_{12}^{(1)}$
$\boldsymbol{x}_{\mathbf{1}}$
$\boldsymbol{x}_{\mathbf{2}}$
⋮
$\boldsymbol{x}_{\boldsymbol{p}}$
$b_1^{(1)}$
1
$\boldsymbol{h}_{\mathbf{1}}^{(\mathbf{1})}$
$\boldsymbol{h}_{\mathbf{2}}^{(\mathbf{1})}$
$\boldsymbol{h}_{\mathbf{3}}^{(\mathbf{1})}$
1

$f(w_{11}^{(1)}x_1 + \cdots + w_{p1}^{(1)}x_p + b_1^{(1)})$

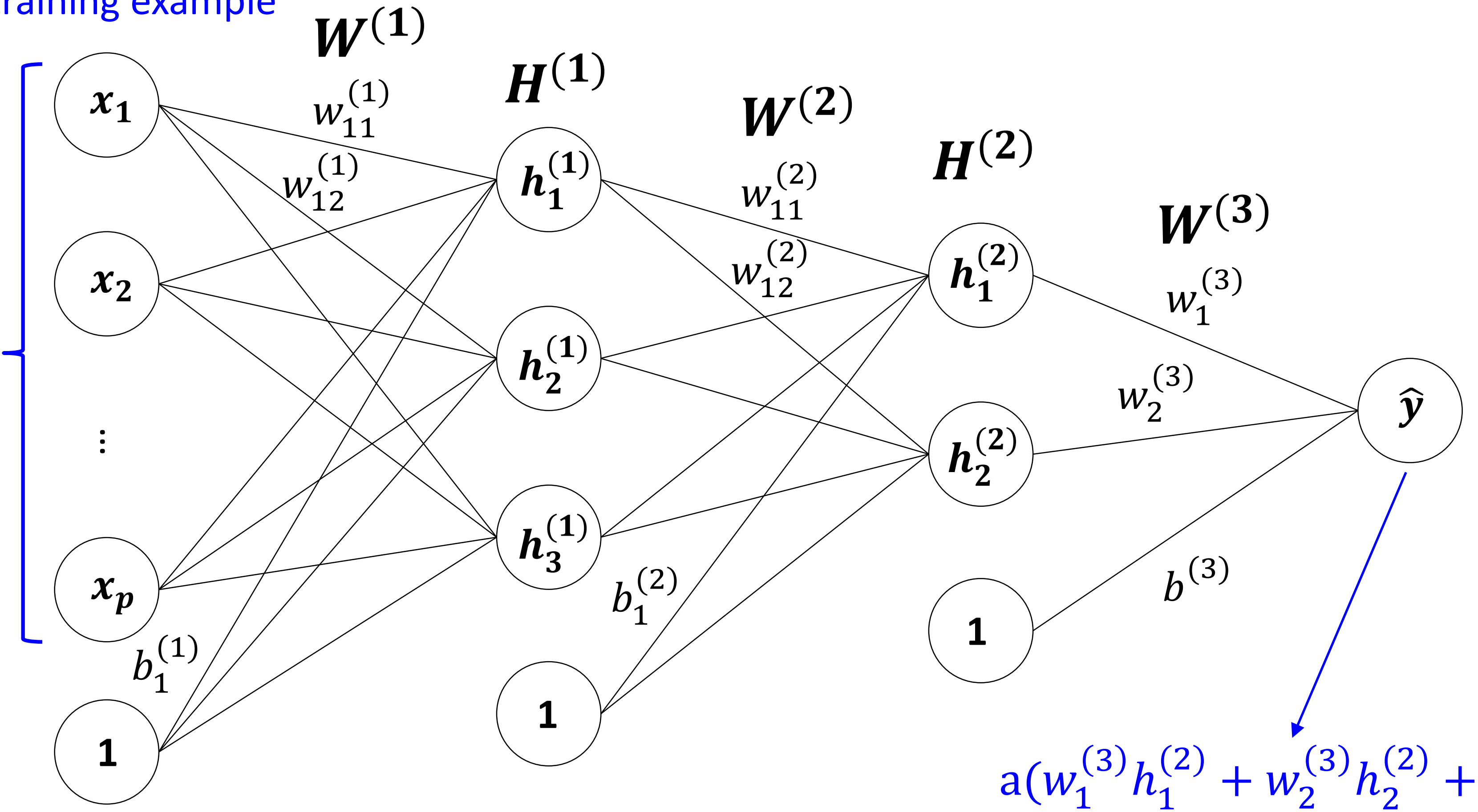
“fake” one

Fully Connected Neural Network Architecture



Fully Connected Neural Network Architecture

one training example



$\mathrm{a}(w_1^{(3)}h_1^{(2)} + w_2^{(3)}h_2^{(2)} + b^{(3)})$

“fake” one

Layer Output

- $$H^{(1)} = \underset{\text{activation function}}{a}\left(\underbrace{\underset{p_1 \times p}{W^{(1)}}\,\underset{p \times n}{X}}_{p_1 \times n} + \underset{p_1 \times 1}{\vec{b}^{(1)}}\right)$$

p_1 = # of nodes in layer 1

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

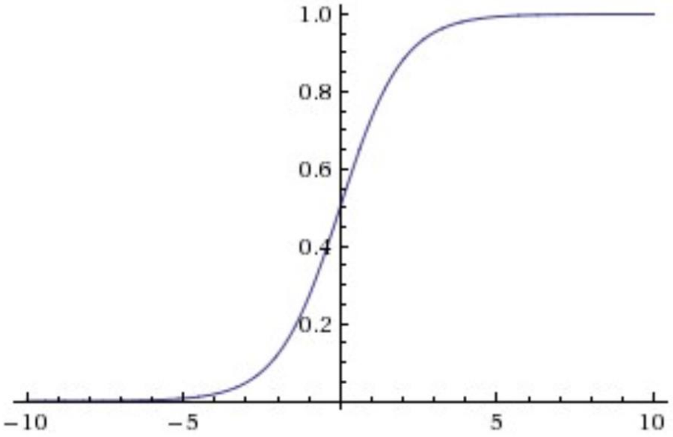
- $$\hat{y} = a\left(W^{(3)}H^{(2)} + b^{(3)}\right)$$

Activation Functions

Option 1: sigmoid function

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

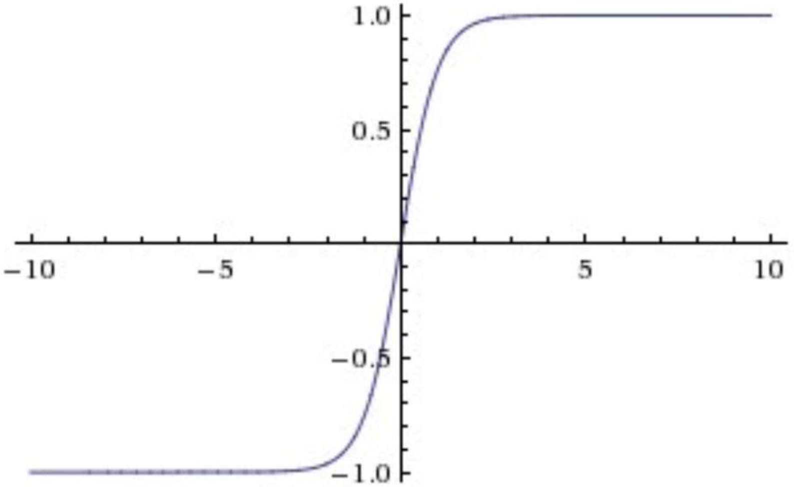
$$\boxed{\sigma(x) = \frac{1}{1 + e^{-x}}}$$



Option 2: hyperbolic tangent

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

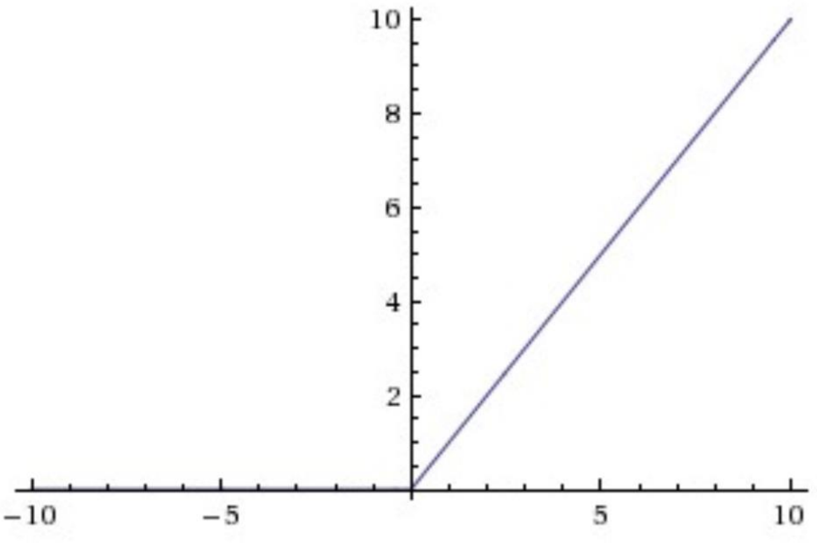
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



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

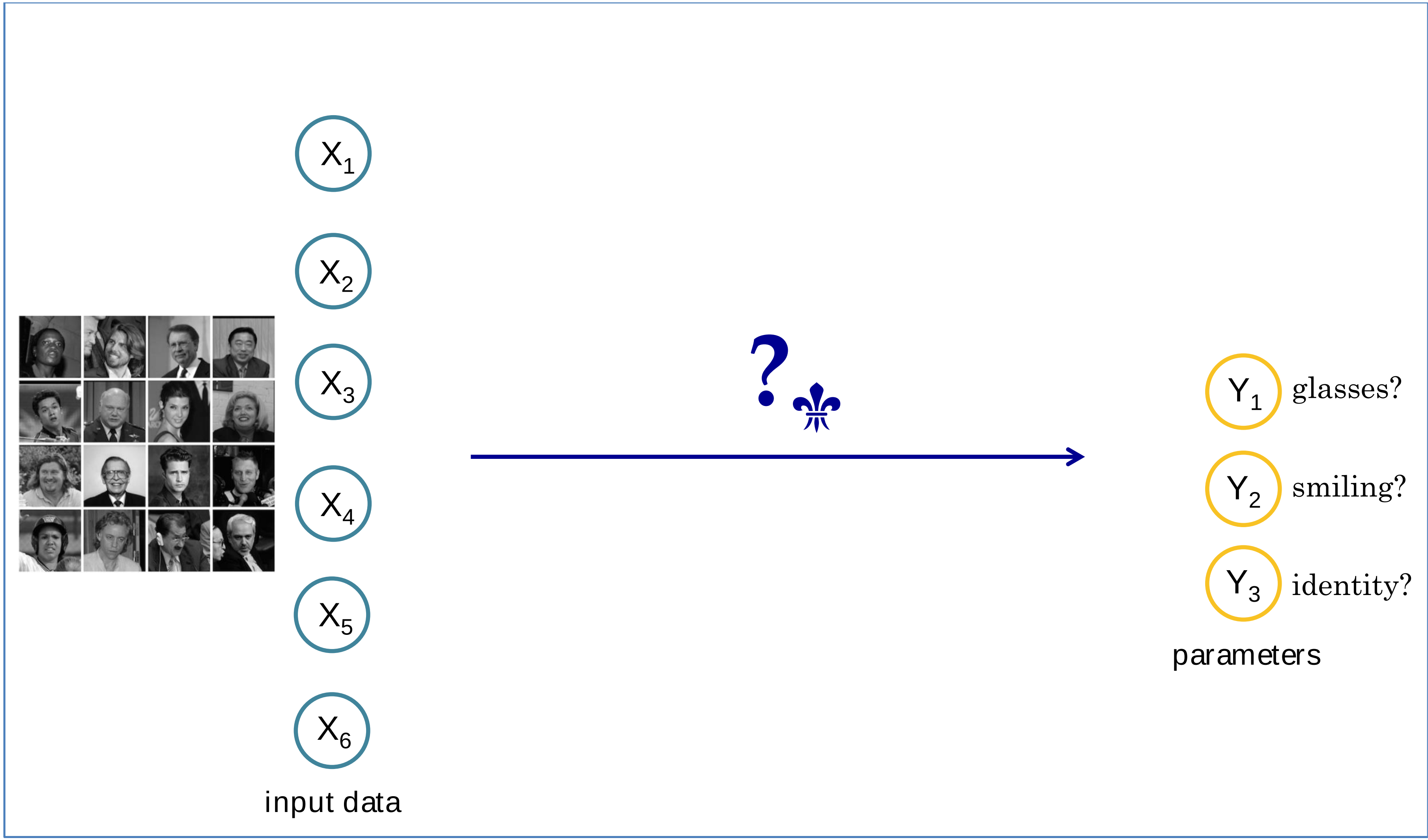
- (-) Still has a tendency to prematurely kill the gradient
- (+) Zero-centered so we get a range of gradients
- (+) Rescaling of sigmoid function so derivative is also not too difficult

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



X_1

X_2

X_3

X_4

X_5

X_6

input data

? ⚜

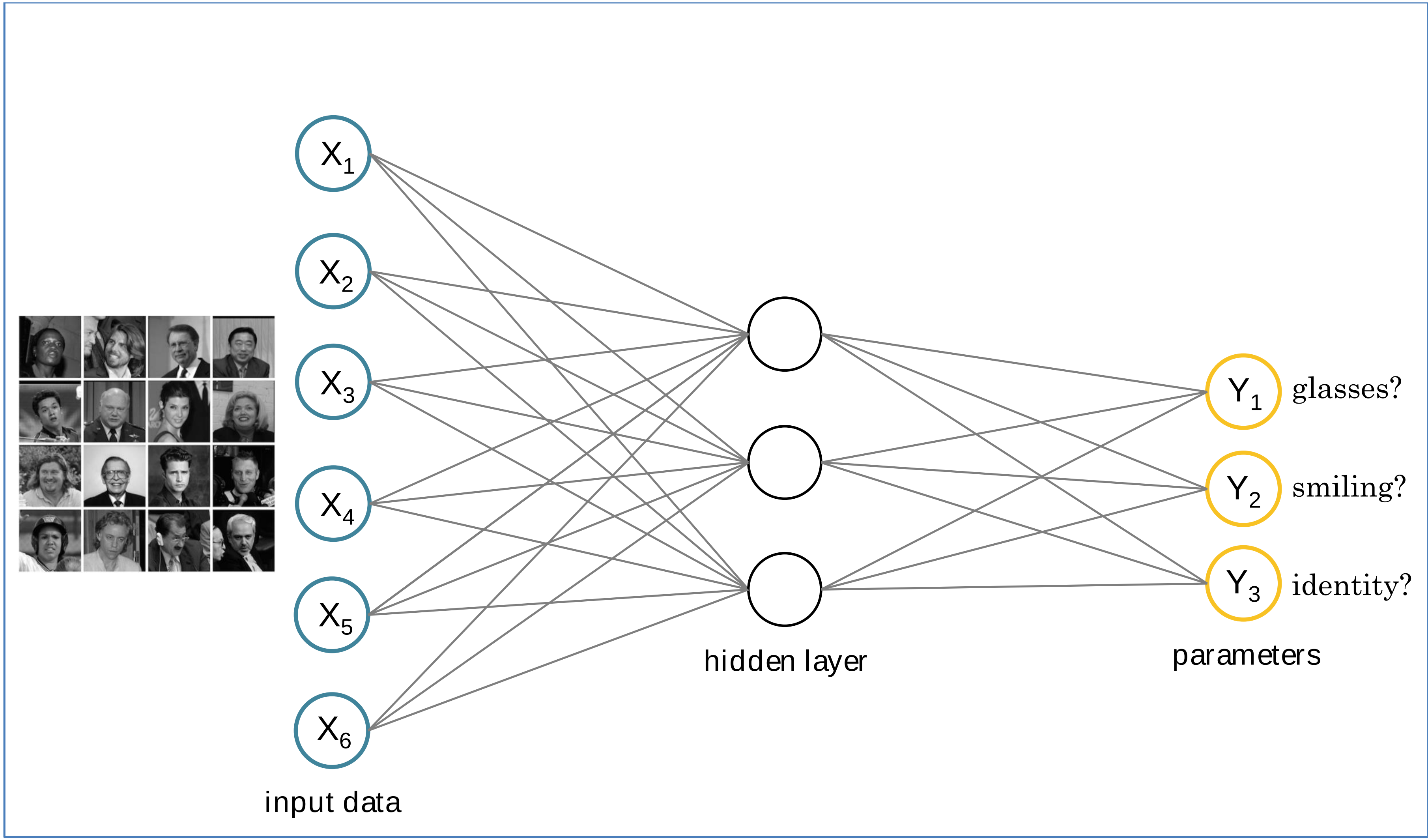
Y_1 glasses?

Y_2 smiling?

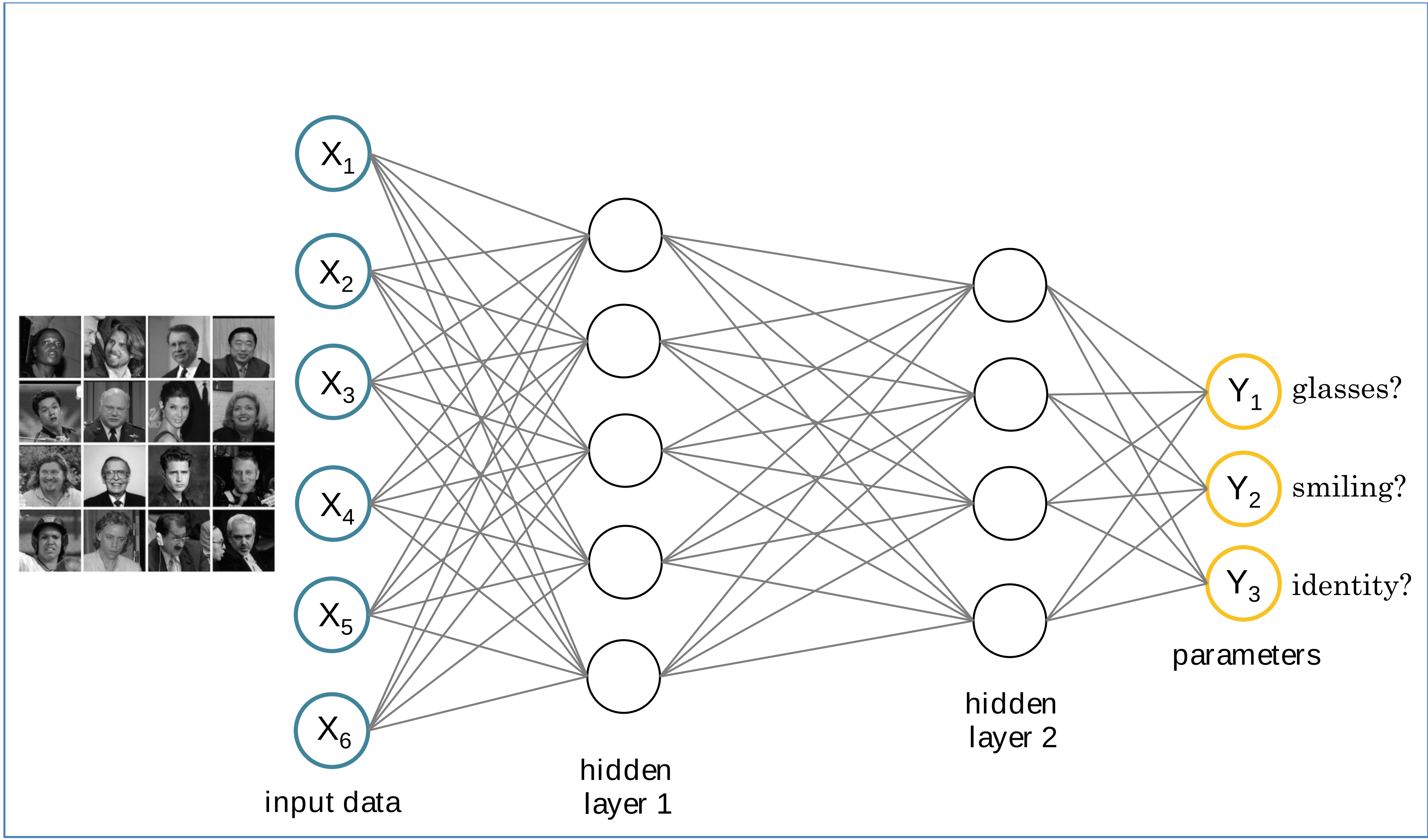
Y_3 identity?

parameters

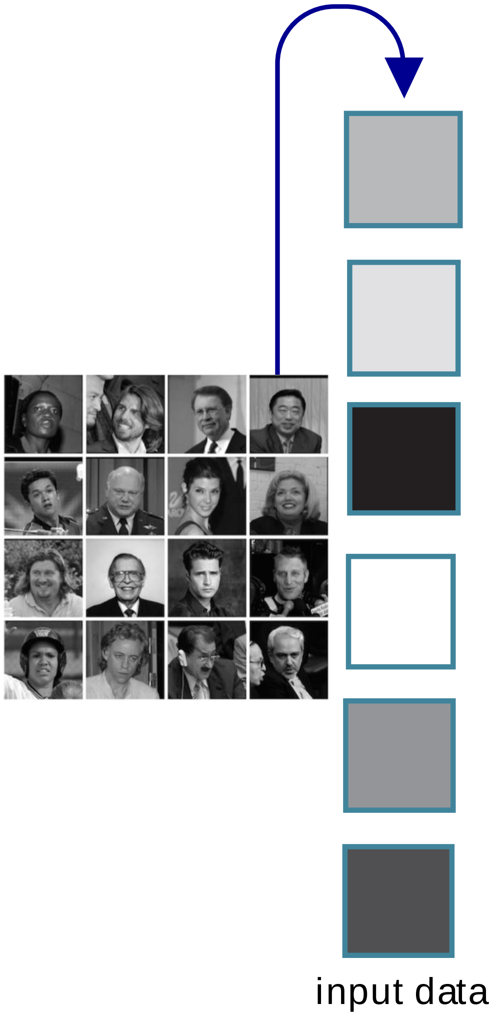
First idea: one hidden layer



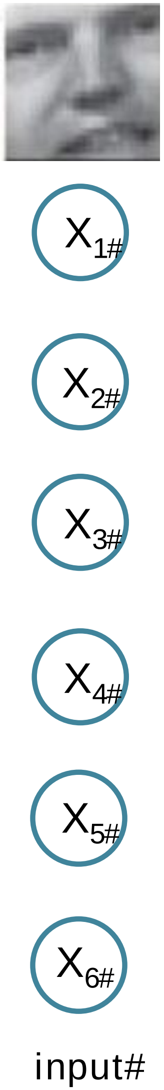
Second idea: more hidden layers (“deep” learning)



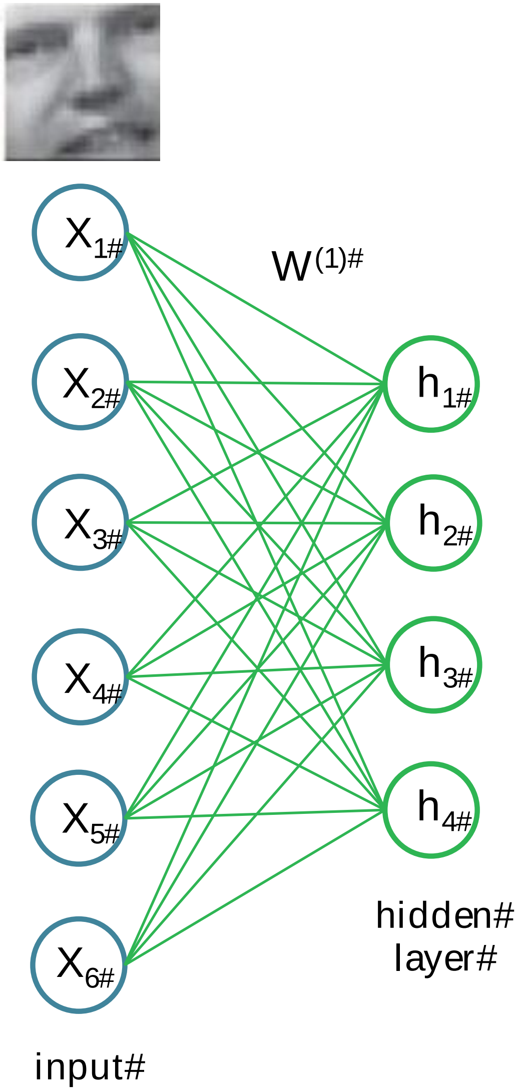
Another idea: Flatten pixels of image into a single vector



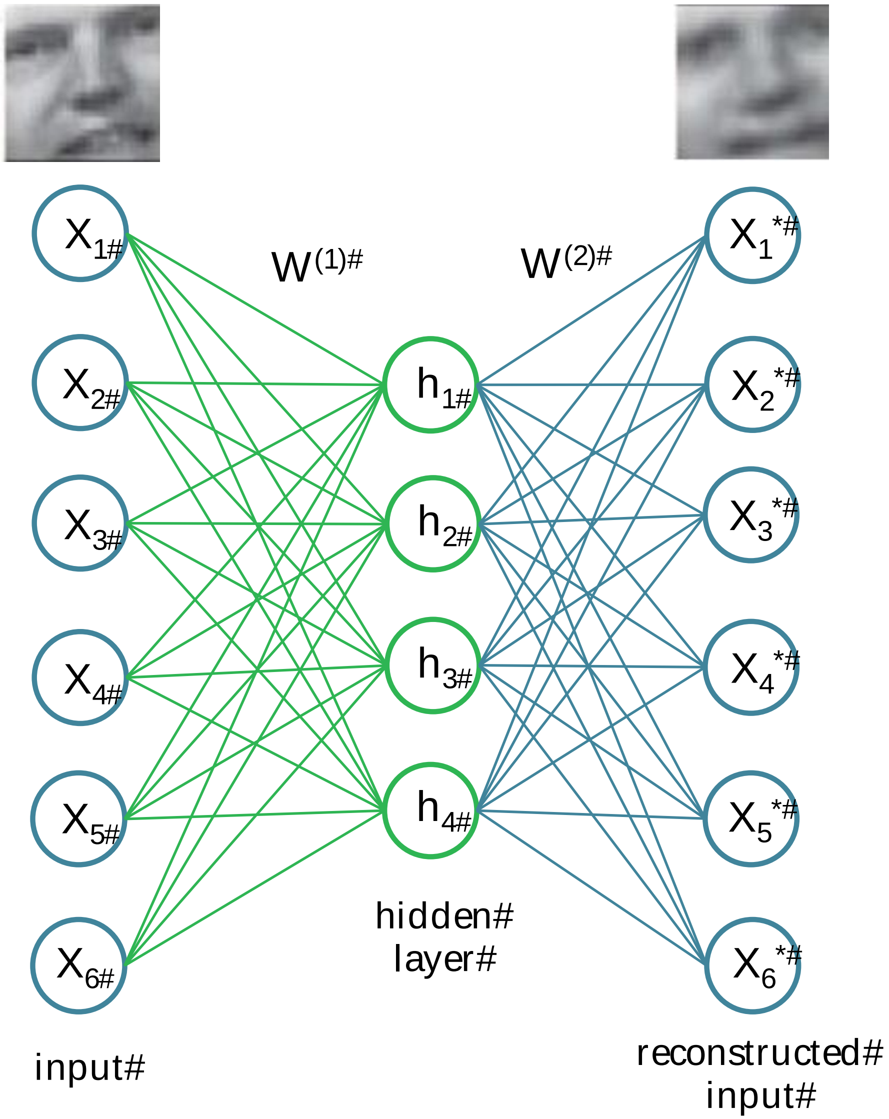
Detour to autoencoders



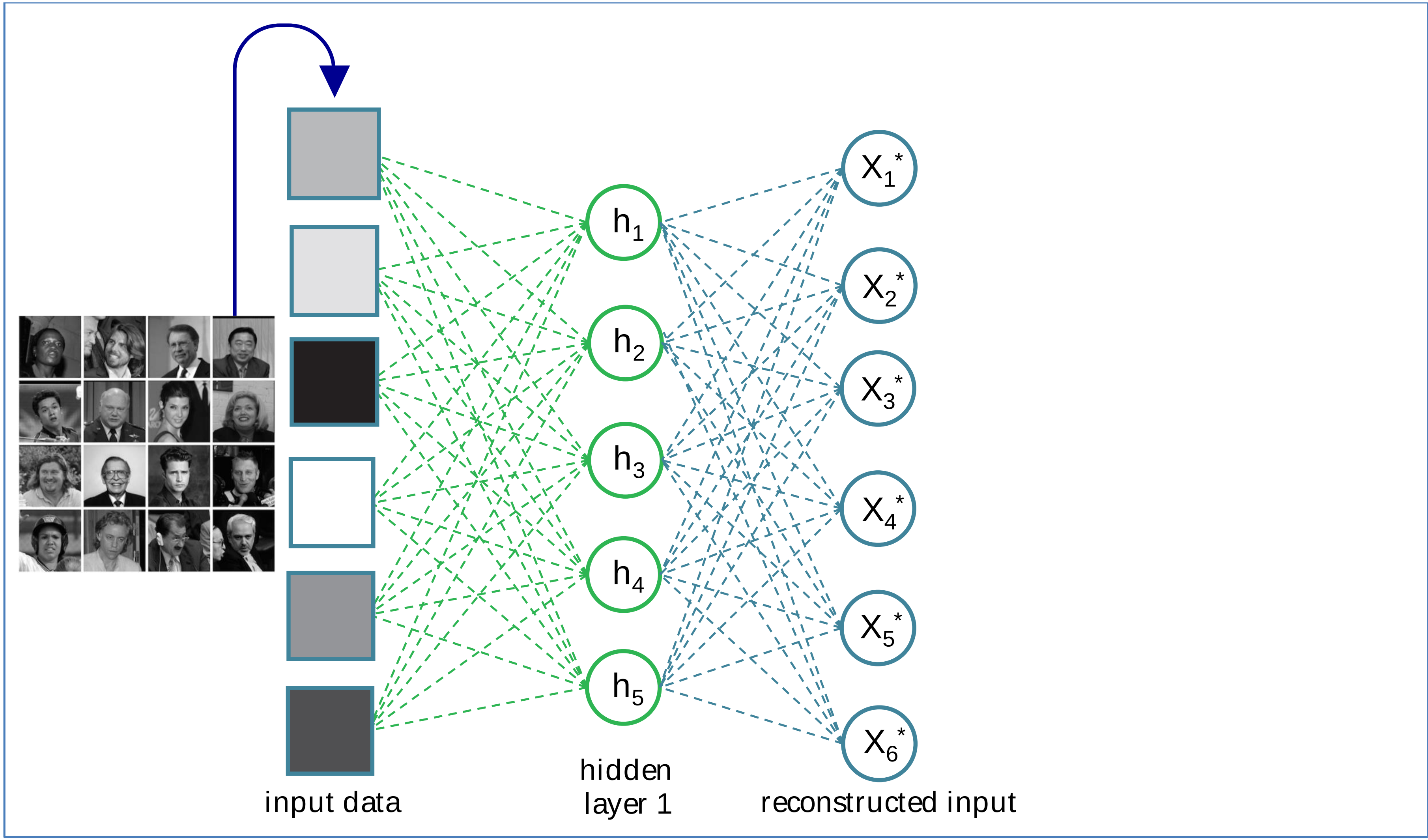
Detour to autoencoders



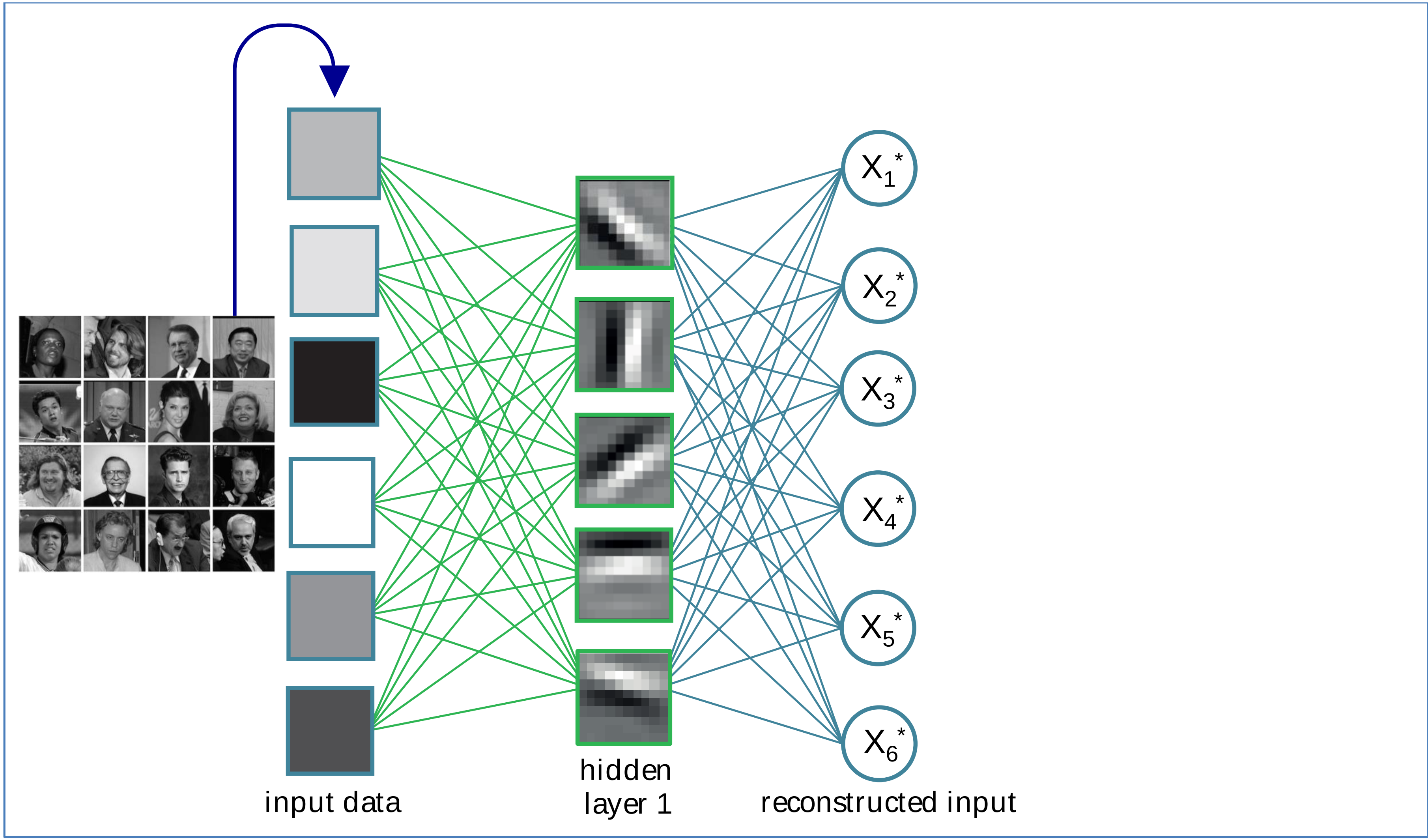
Detour to autoencoders



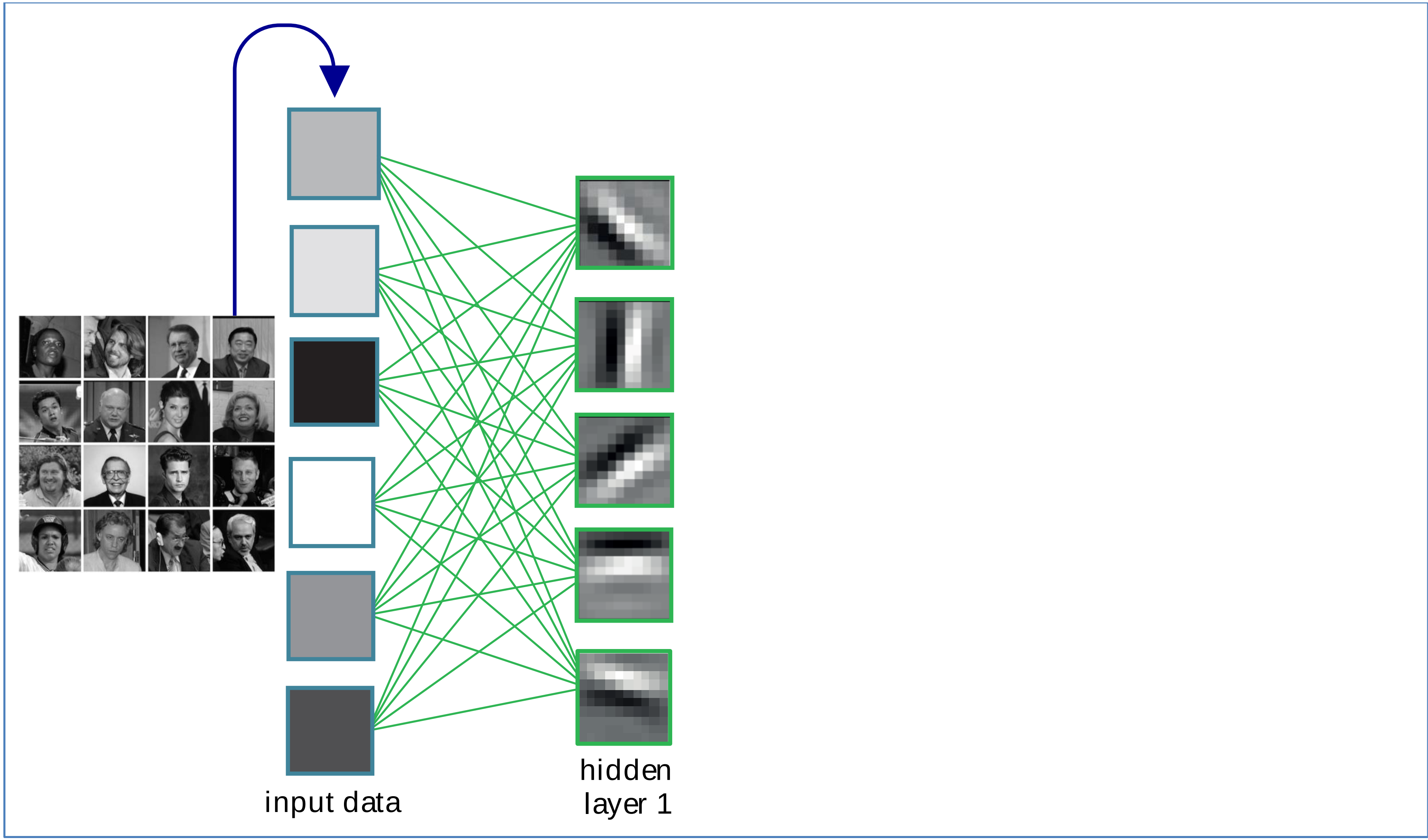
Use unsupervised pre-training to find a function from the input to itself



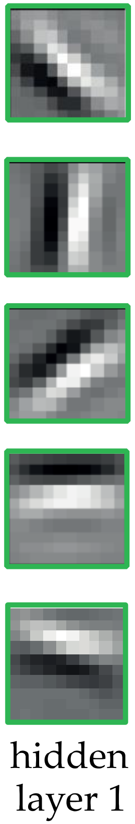
Hidden units can be interpreted as edges



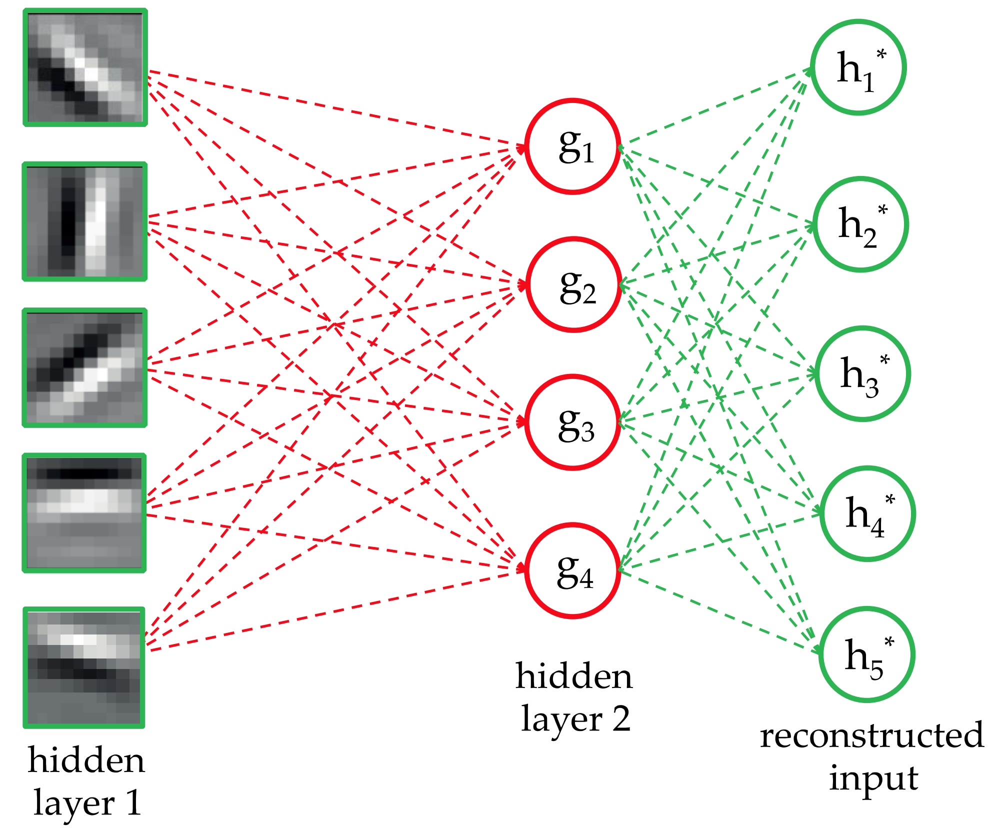
Now: throw away reconstruction and input



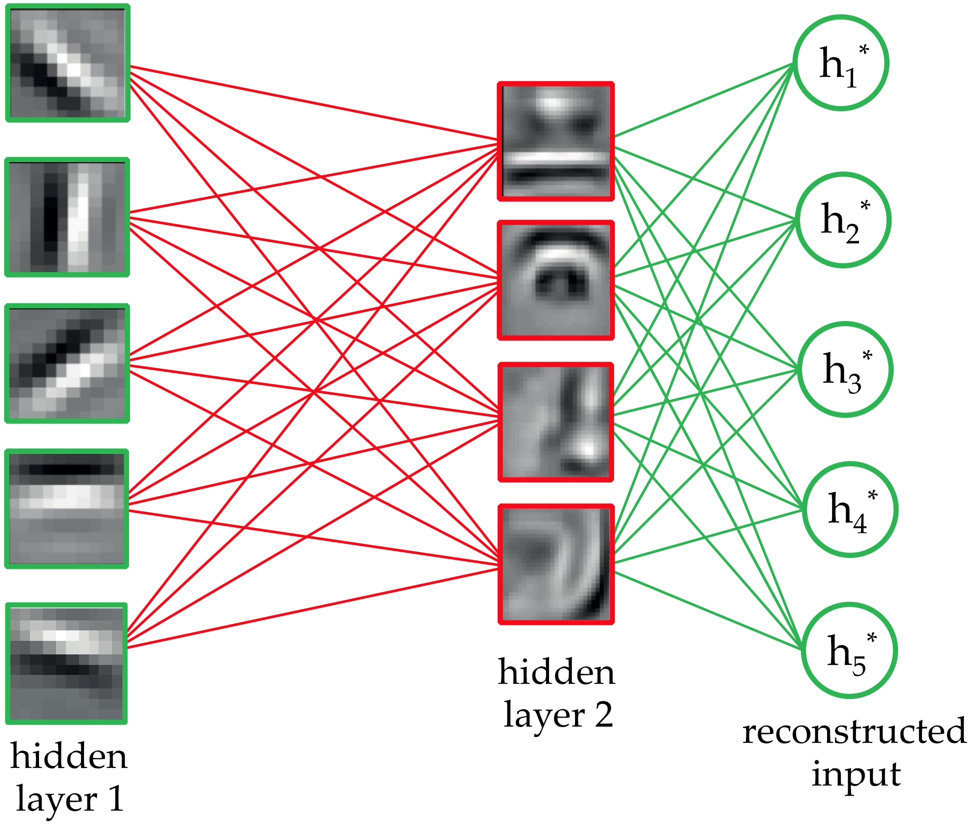
Now: throw away reconstruction and input



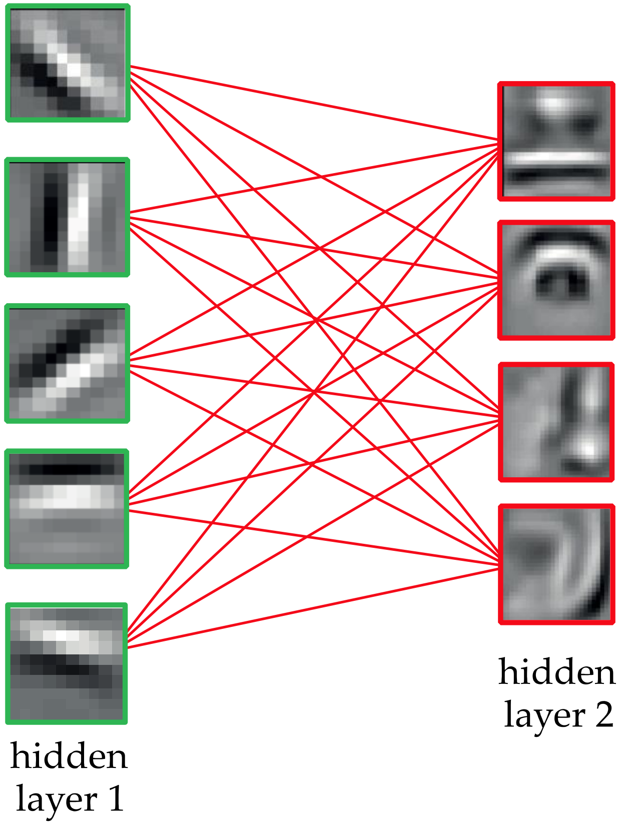
Then repeat the entire process for each layer



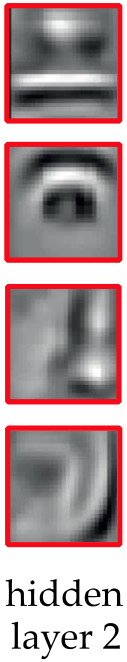
Then repeat the entire process for each layer



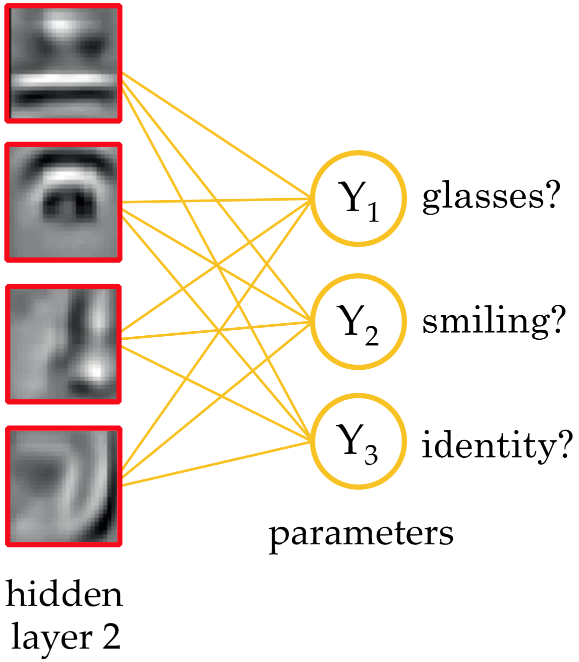
Then repeat the entire process for each layer



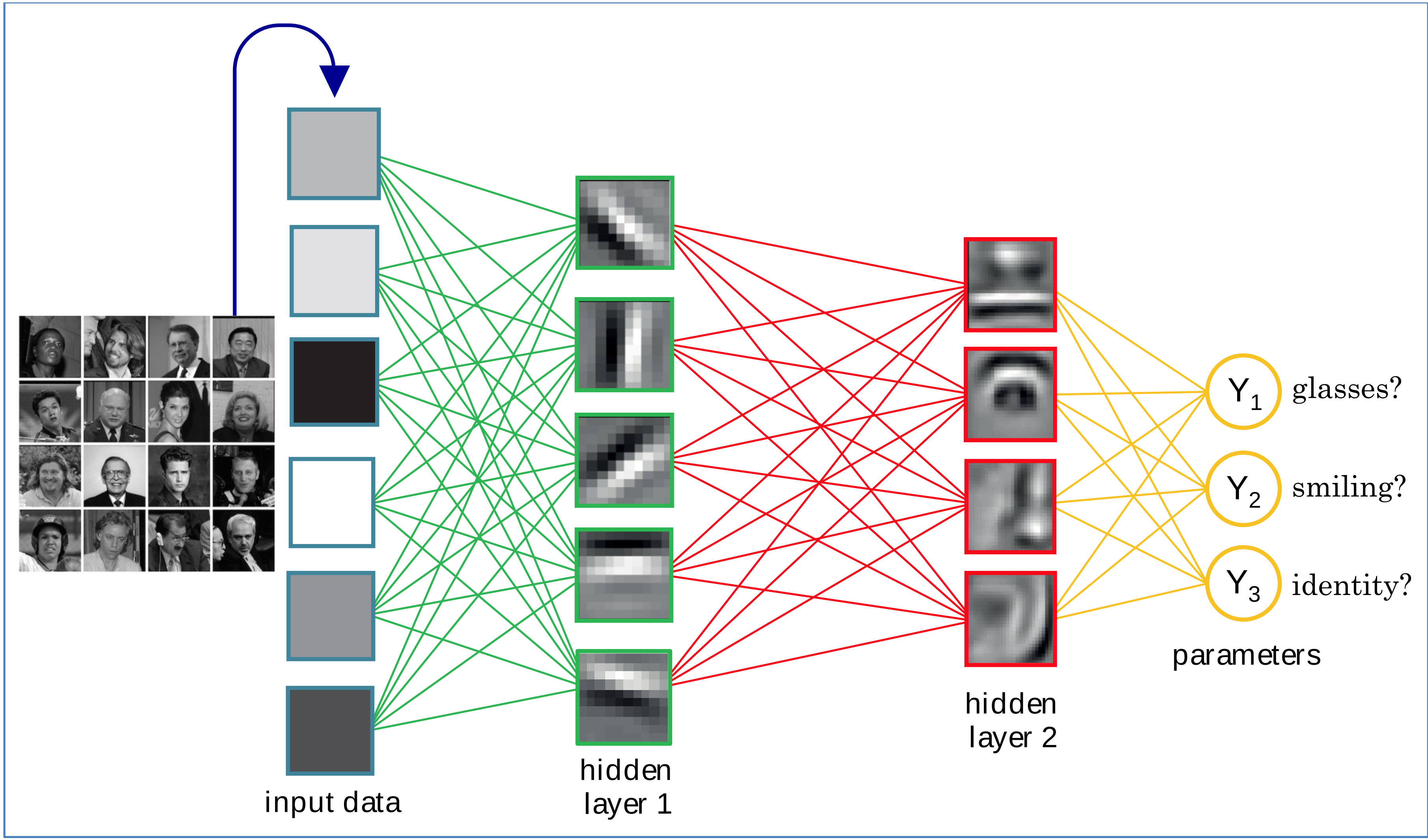
Then repeat the entire process for each layer



In the last layer, use the outputs (supervised)

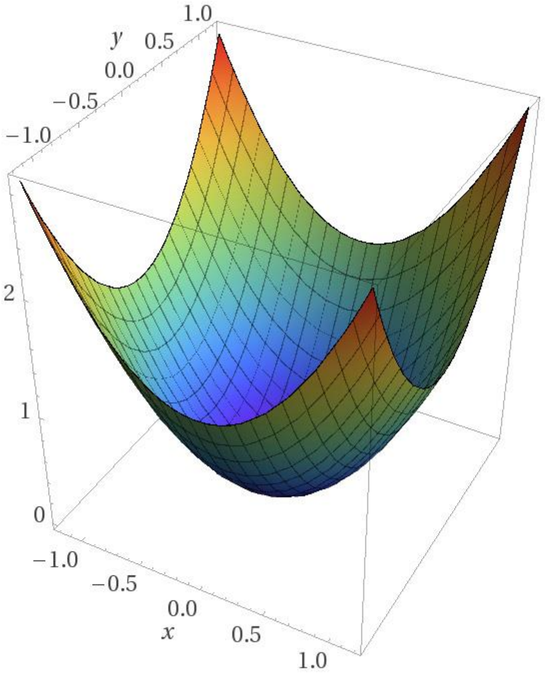


Finally, "fine-tune" the entire network!



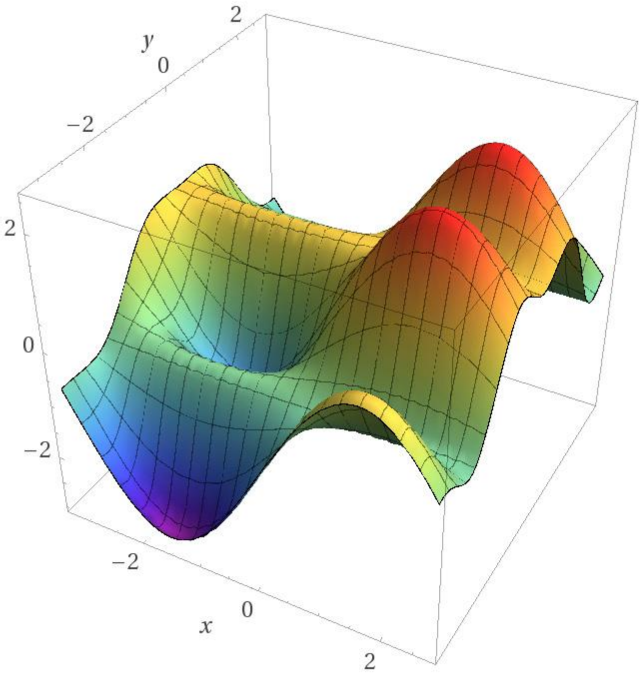
Takeaways

- 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

Image: O’Reilly Media

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

Motivation for moving away from FC architectures

- For a 32x32x3 image (very small!) we have *p*=3072 features in the input layer

- For a 200x200x3 image, we would have *p*=120,000! *doesn’t scale*

Motivation for moving away from FC architectures

- For a 32x32x3 image (very small!) we have *p*=3072 features in the input layer

- For a 200x200x3 image, we would have *p*=120,000! *doesn’t scale*

- Fully connected 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

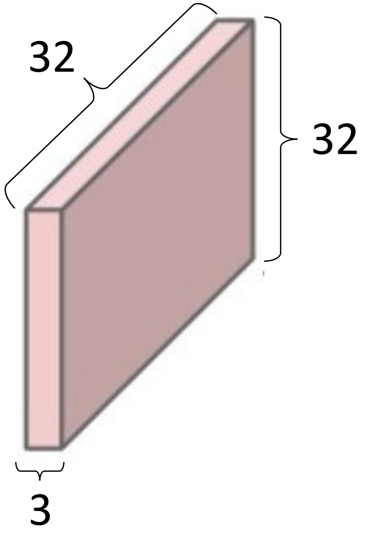


Image: modified from Stanford Course CS231n: http://cs231n.github.io/convolutional-networks/

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

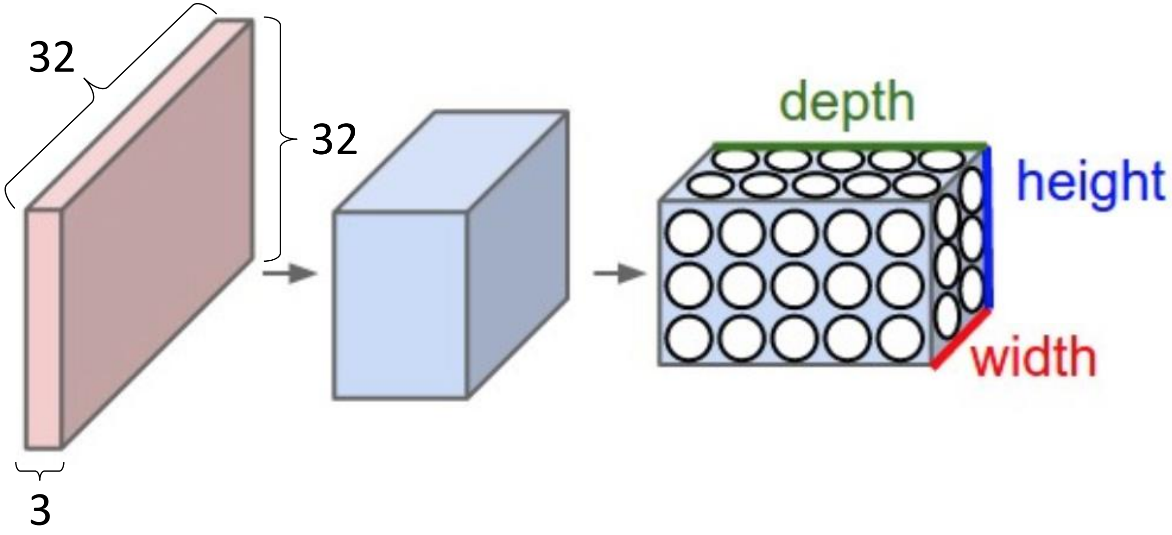


Image: modified from Stanford Course CS231n: http://cs231n.github.io/convolutional-networks/

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)

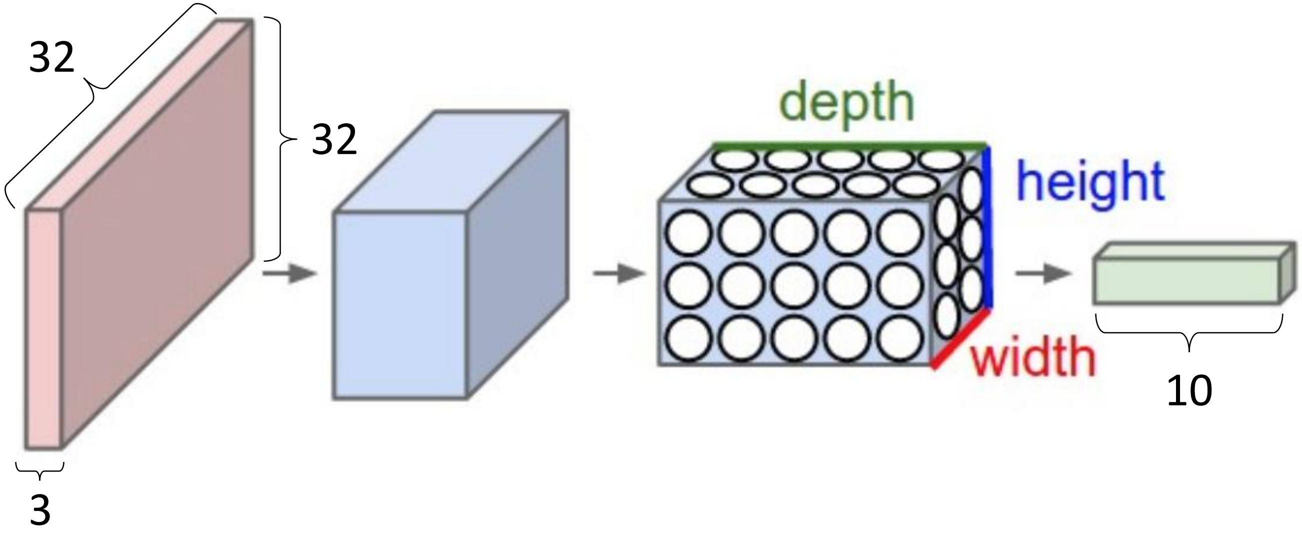


Image: modified from Stanford Course CS231n: http://cs231n.github.io/convolutional-networks/