CS 360: Machine Learning

Prof. Sara Mathieson

Fall 2020

HAVERFORD COLLEGE

Admin

- Office hours **today 11-12pm** (stay on the link)
  - Will talk first to people I didn't get to in lab
- **Lab 8** due Friday Nov 20
  - Let me know if you would like individual deadlines
- After Thanksgiving – **two options for capstone**
  - Midterm 2 (midterm material ends Nov 20)
  - Final project (posted soon)
  - I will update grade percentages soon

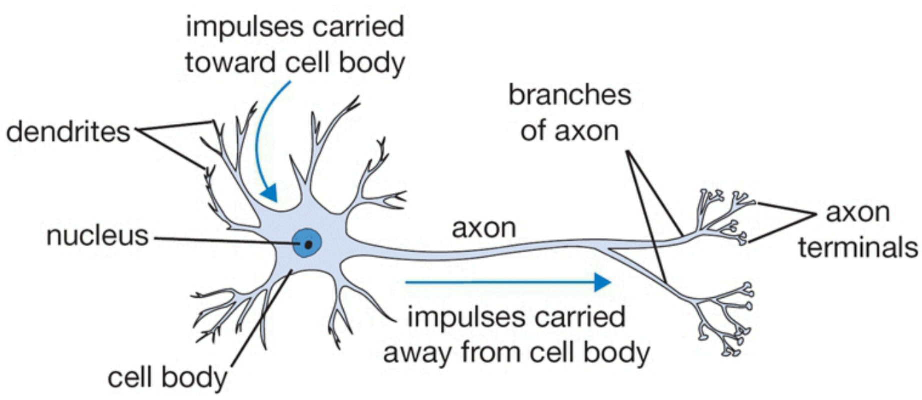
Outline for November 13

- Introduction to neural networks
- Fully connected (FC) neural networks
- Convolutional neural networks (CNNs)
- Next week: more details on training NNs

Outline for November 13

- Introduction to neural networks
- Fully connected (FC) neural networks
- Convolutional neural networks (CNNs)
- Next week: more details on training NNs

Biological Inspiration



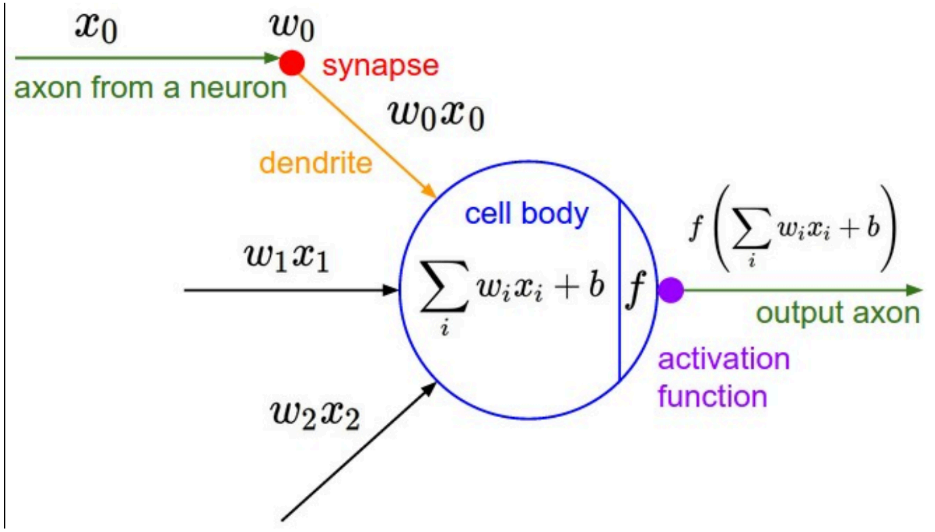


Figure: Stanford CS231n http://cs231n.github.io/neural-networks-1/

Goal: learn from complicated inputs

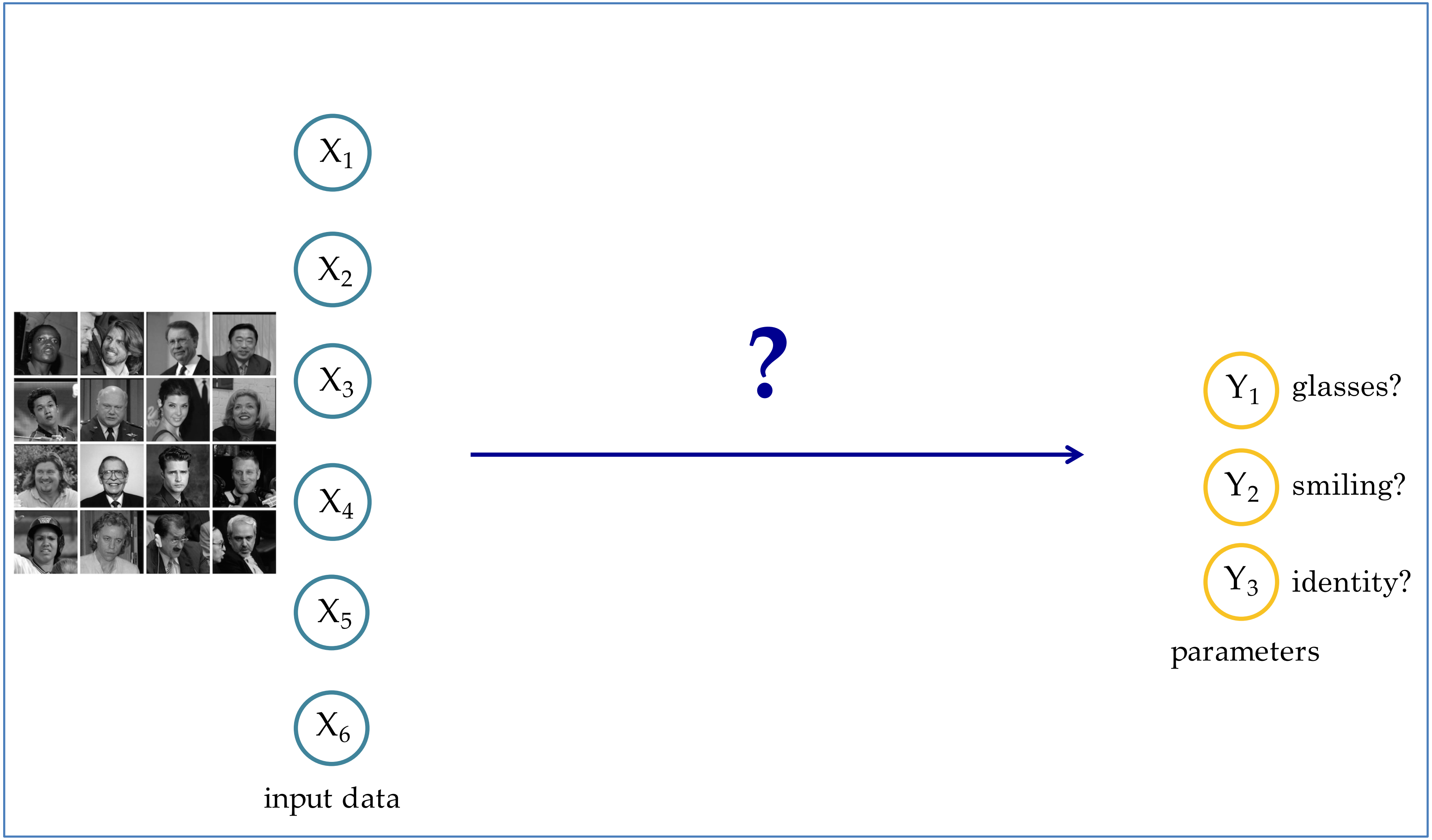


Image: Labeled Faces in the Wild (UMass)

Idea: transform data into lower dimension

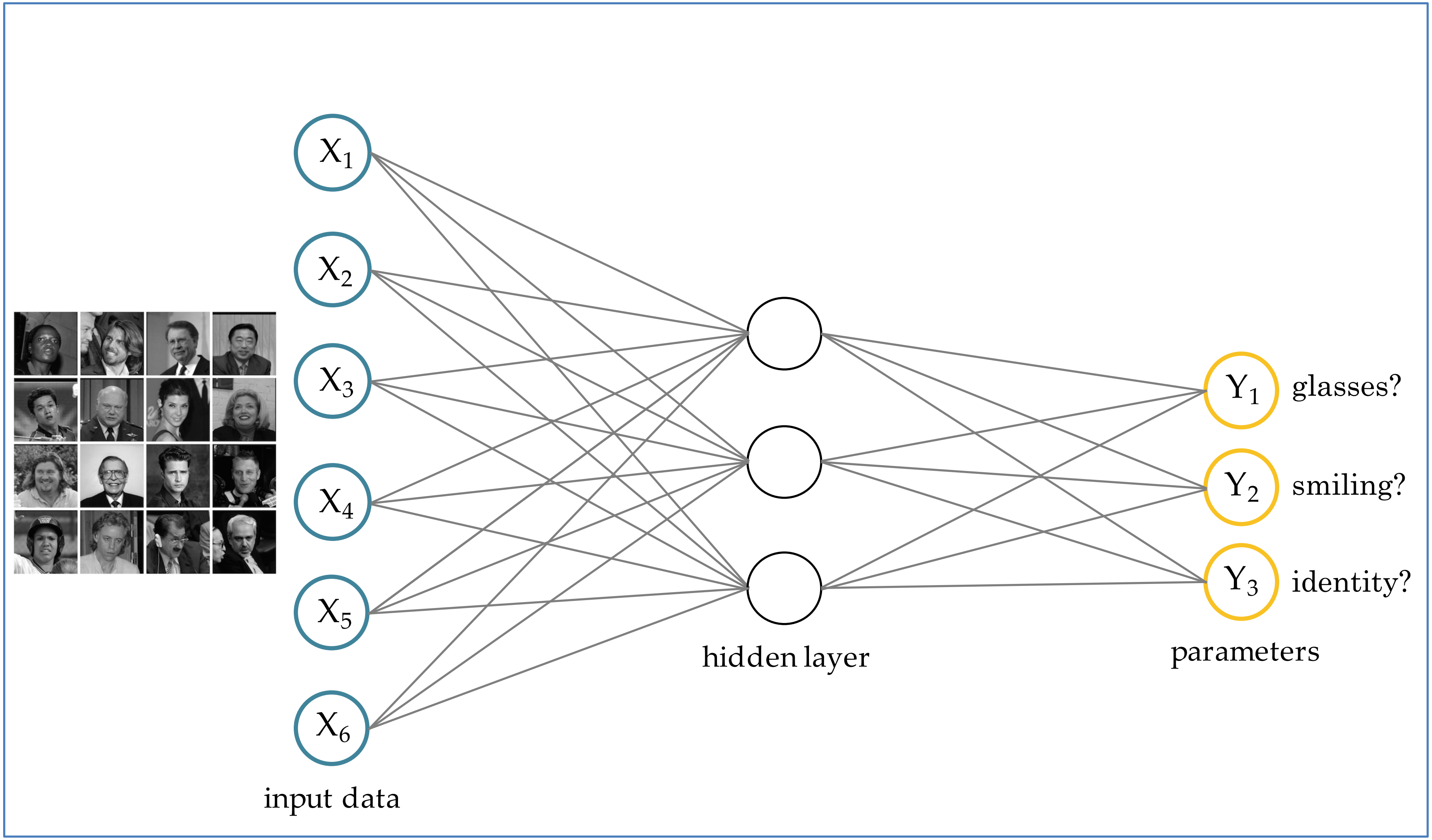


Image: Labeled Faces in the Wild (UMass)

Multi-layer networks = “deep learning”

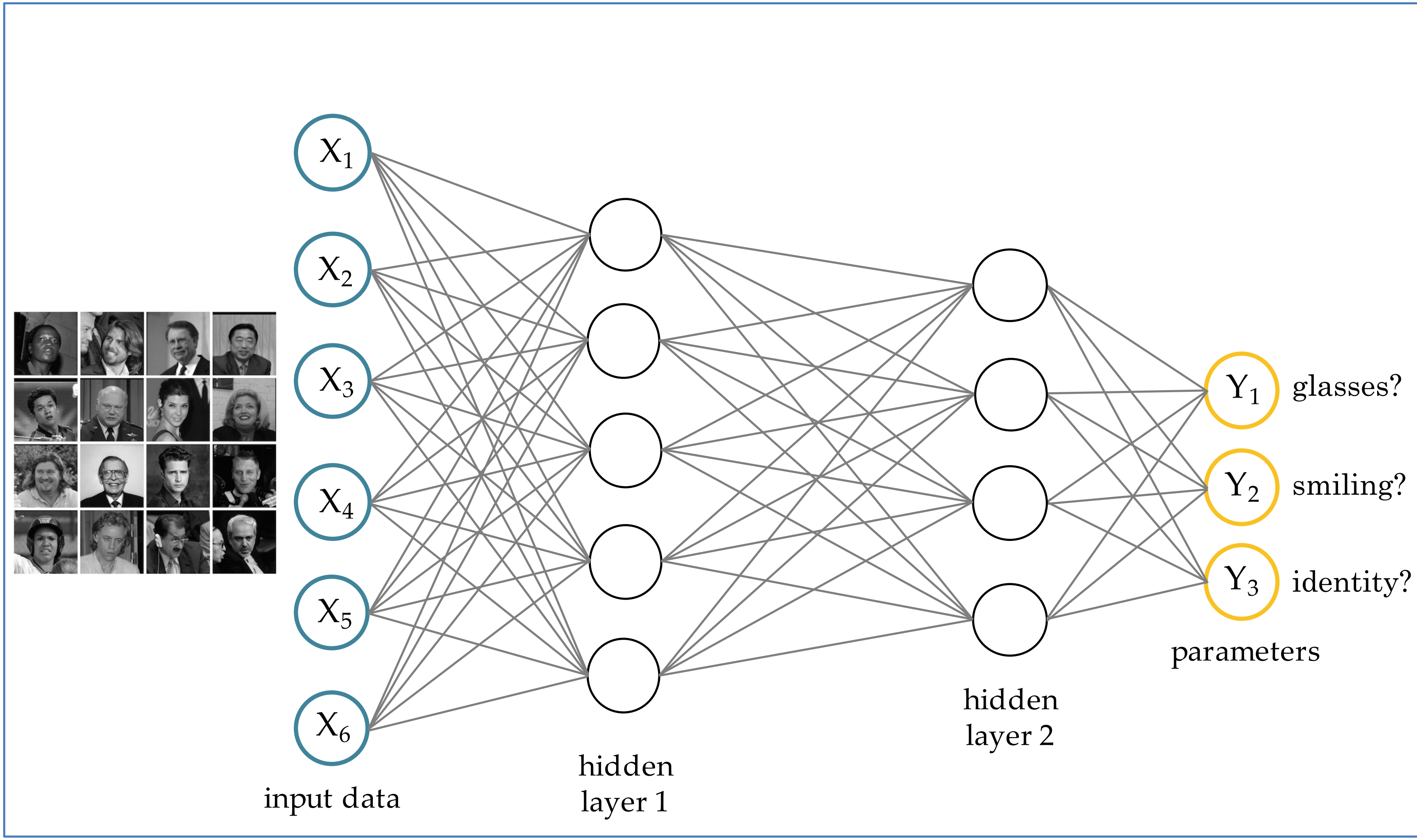
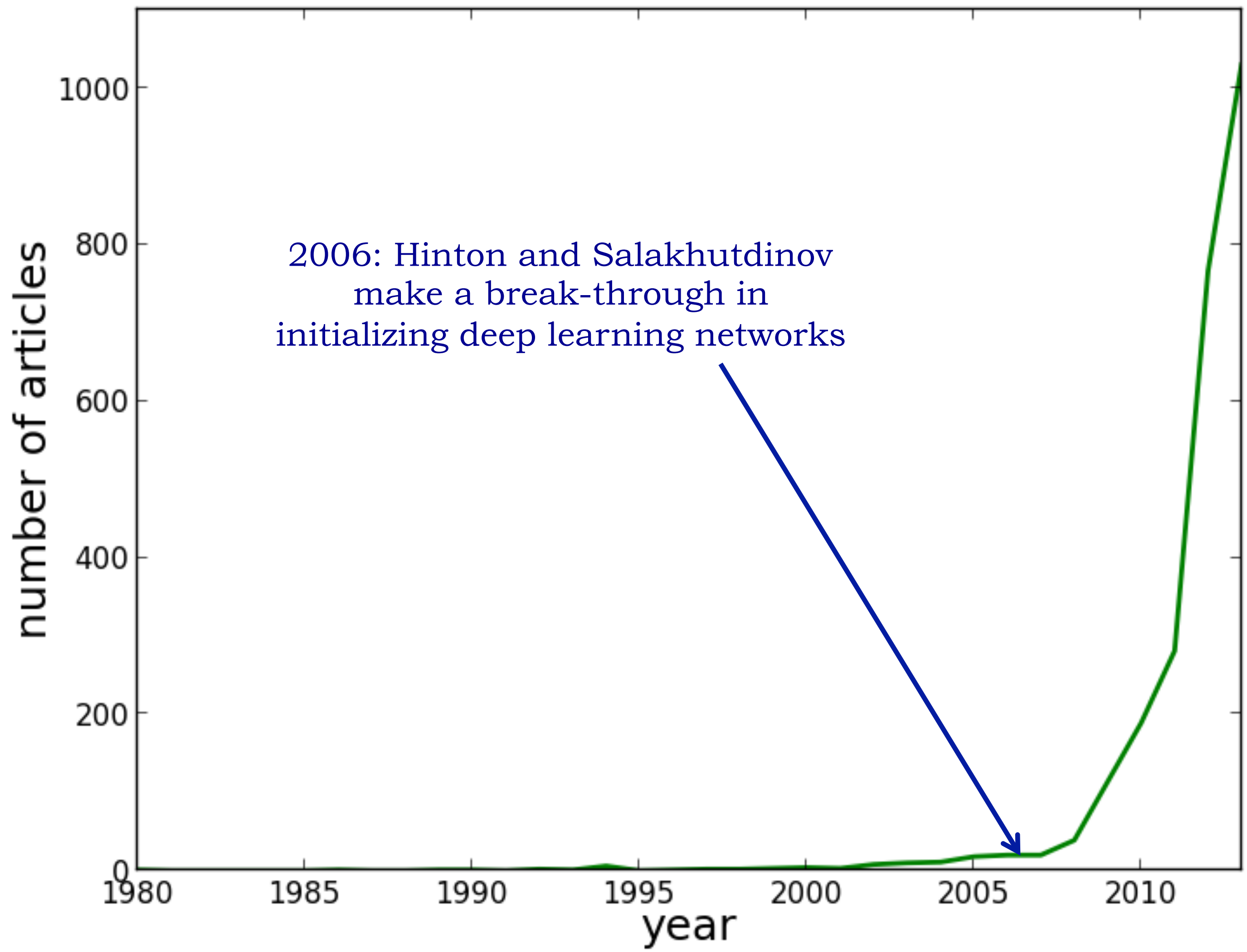


Image: Labeled Faces in the Wild (UMass)

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

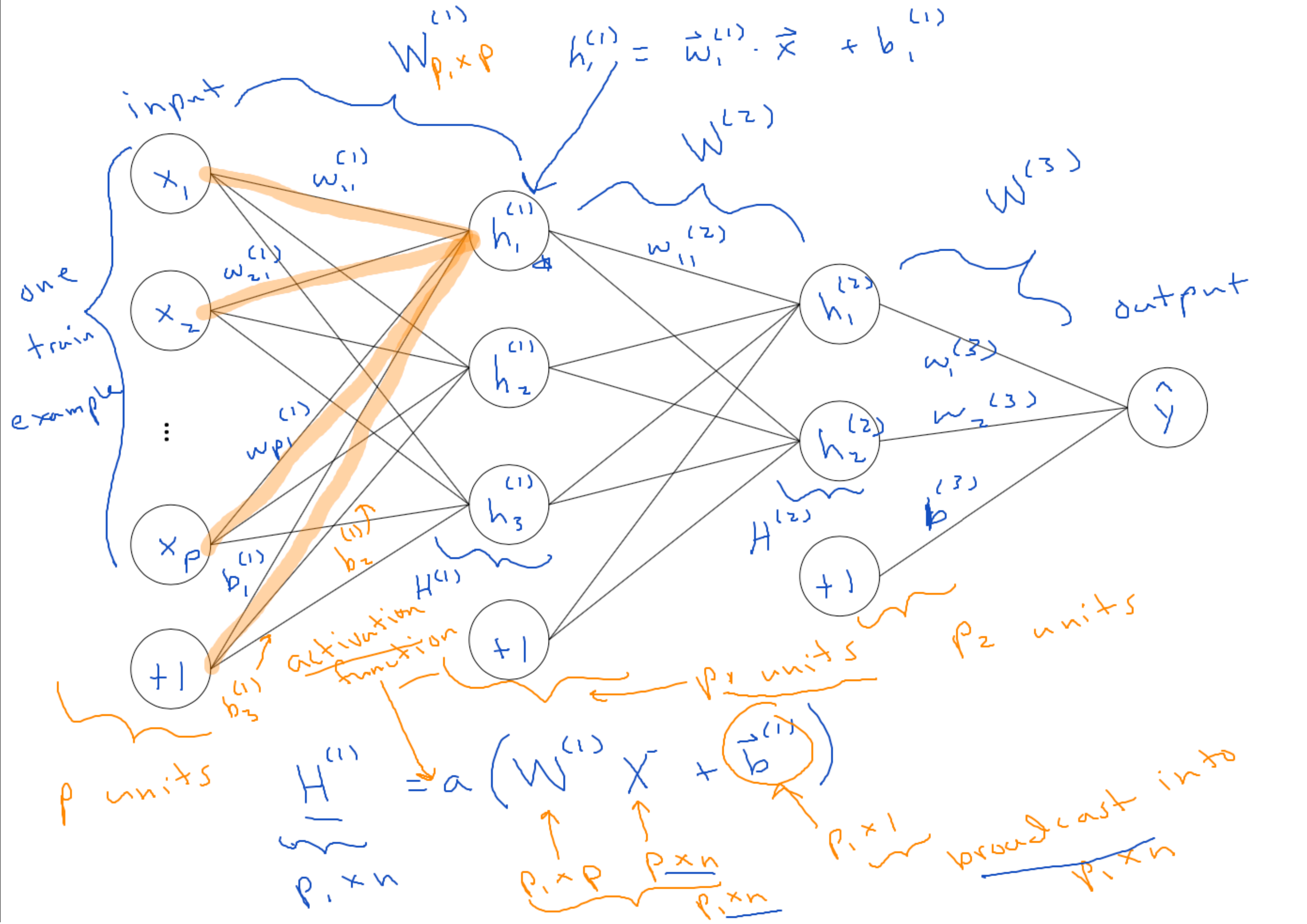
- We will train our network by asking it to minimize the loss between its output and the true output

- We will use SGD-like approaches to minimize loss

Outline for November 13

- Introduction to neural networks
- Fully connected (FC) neural networks
- Convolutional neural networks (CNNs)
- Next week: more details on training NNs

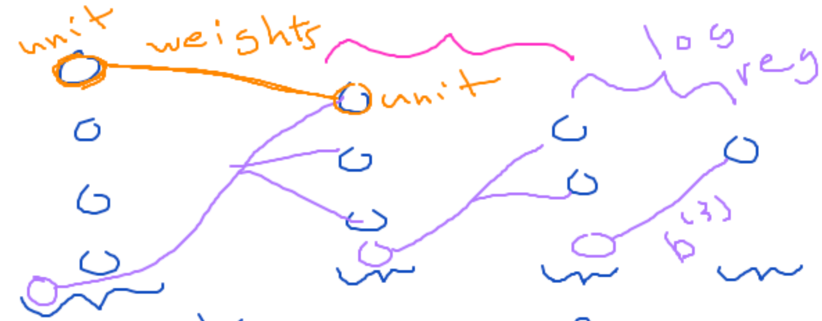
Fully Connected Neural Network Architecture



Fully Connected Neural Network Architecture

input

$$X = \begin{bmatrix} | & | & & | \\ \vec{x}_1 & \vec{x}_2 & \cdots & \vec{x}_n \\ | & | & & | \end{bmatrix}_{p \times n}$$



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

$$\underset{p_2 \times n}{H^{(2)}} = a\left(\underset{p_2 \times p_1}{W^{(2)}}\ \underset{p_1 \times n}{H^{(1)}} + \underset{p_2 \times 1}{\vec{b}^{(2)}}\right)$$

$$\underset{1 \times n}{\hat{\vec{y}}} = a\left(\underset{1 \times p_2}{W^{(3)}}\ \underset{p_2 \times n}{H^{(2)}} + \underset{1 \times 1}{b^{(3)}}\right)$$

scalar

prediction

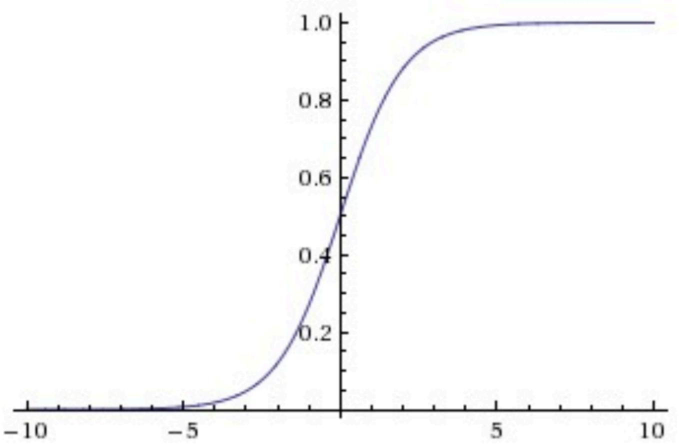
activation function

element-wise

Option 1: sigmoid function

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

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



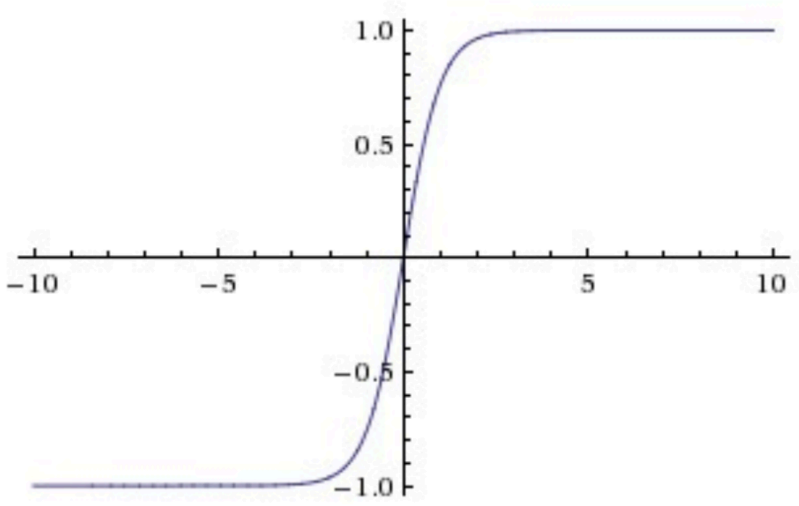
- Derivative is convenient

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

Option 2: hyperbolic tangent

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

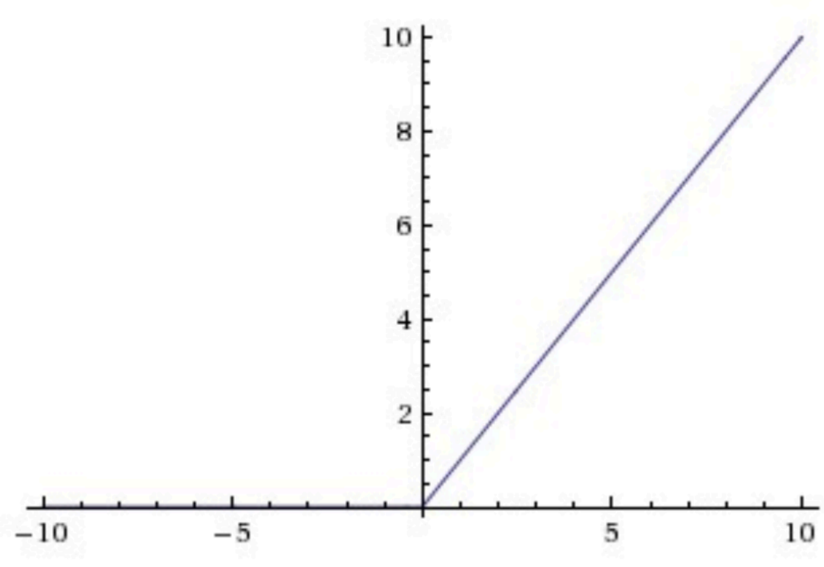
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



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

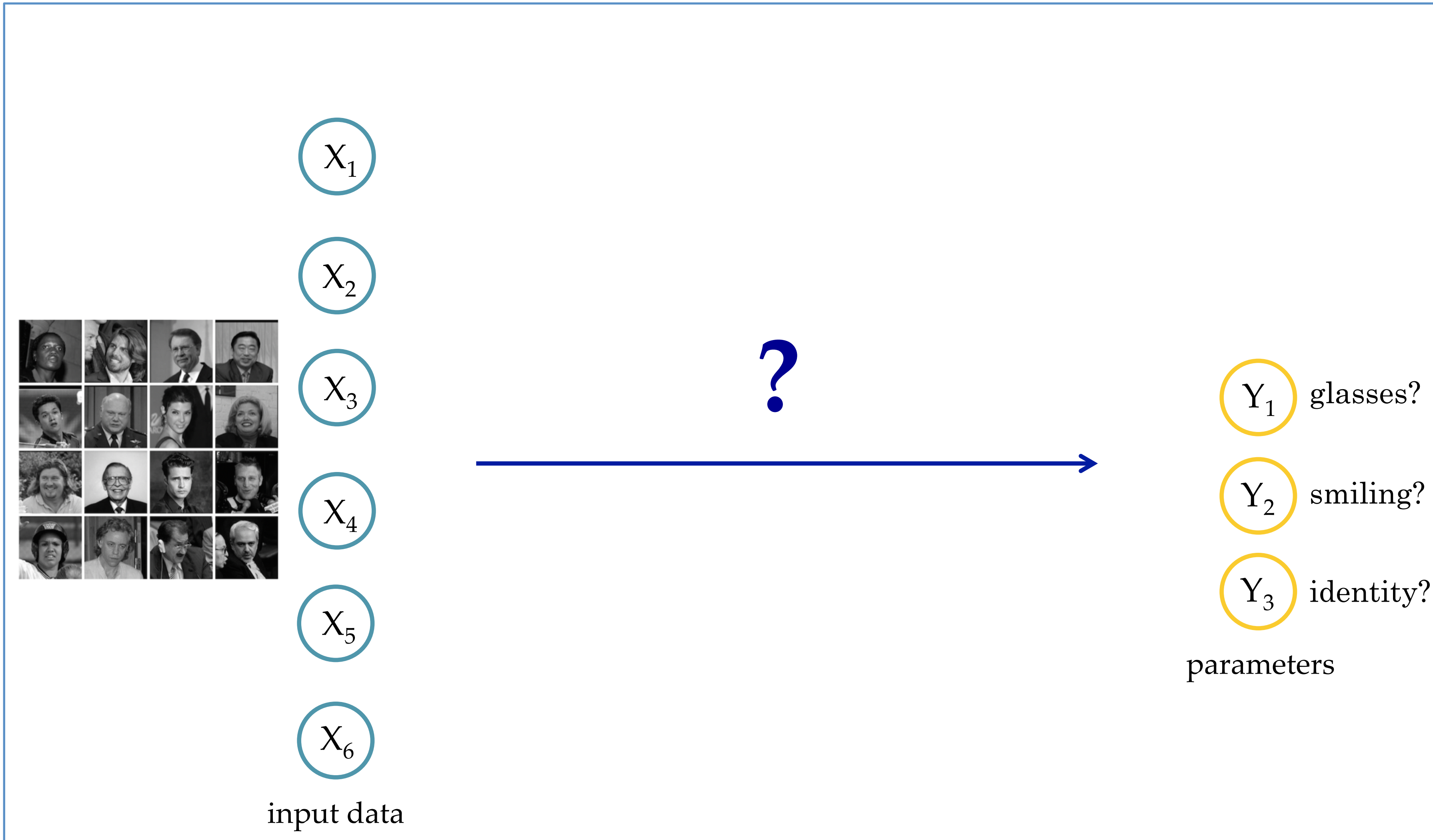
- (+) Works well in practice (accelerates convergence)
- (+) Function value very easy to compute! (no exponentials)
- (-) Units can "die" (no signal) if input becomes too negative throughout gradient descent

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

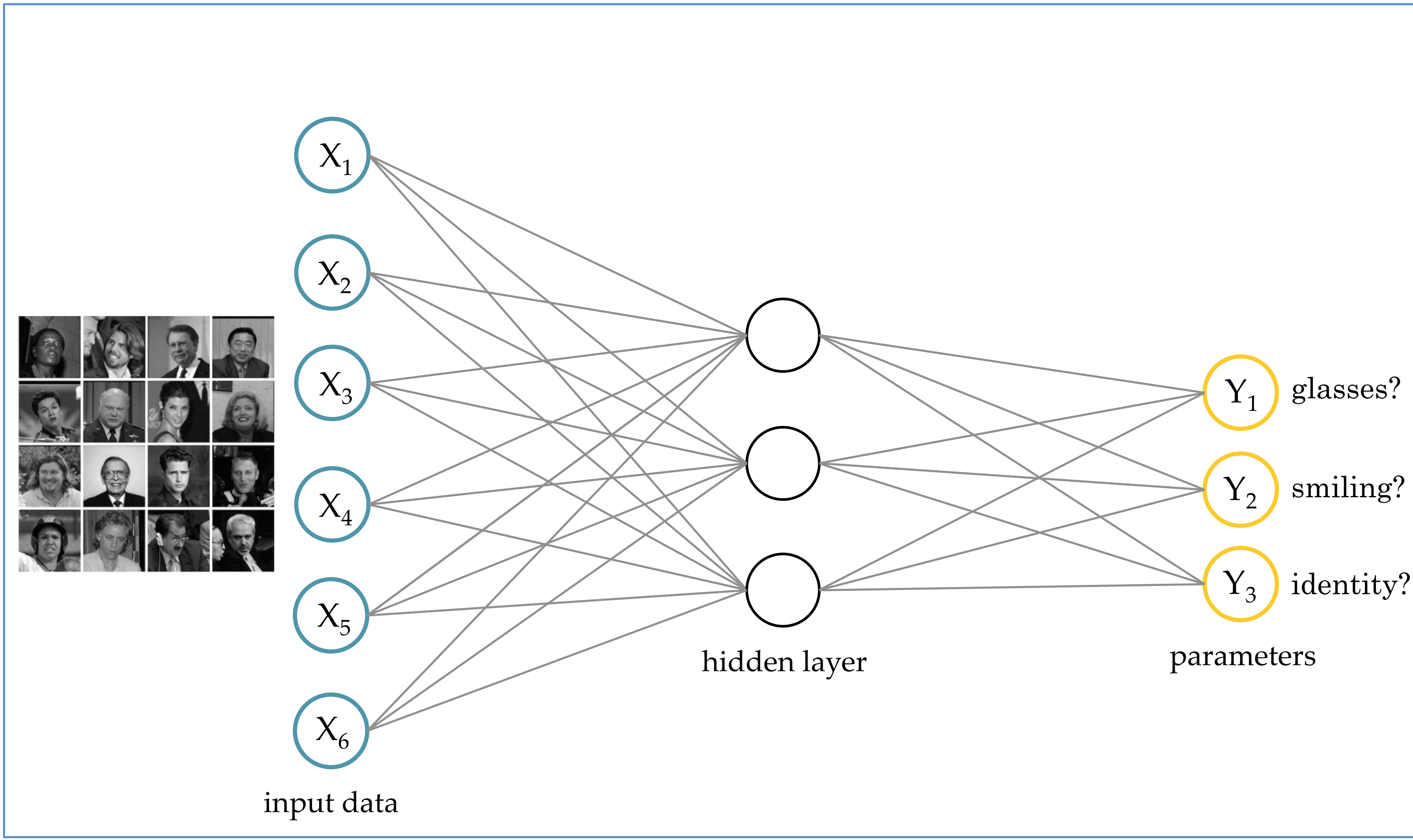
Cross Entropy Loss

NEXT TIME!

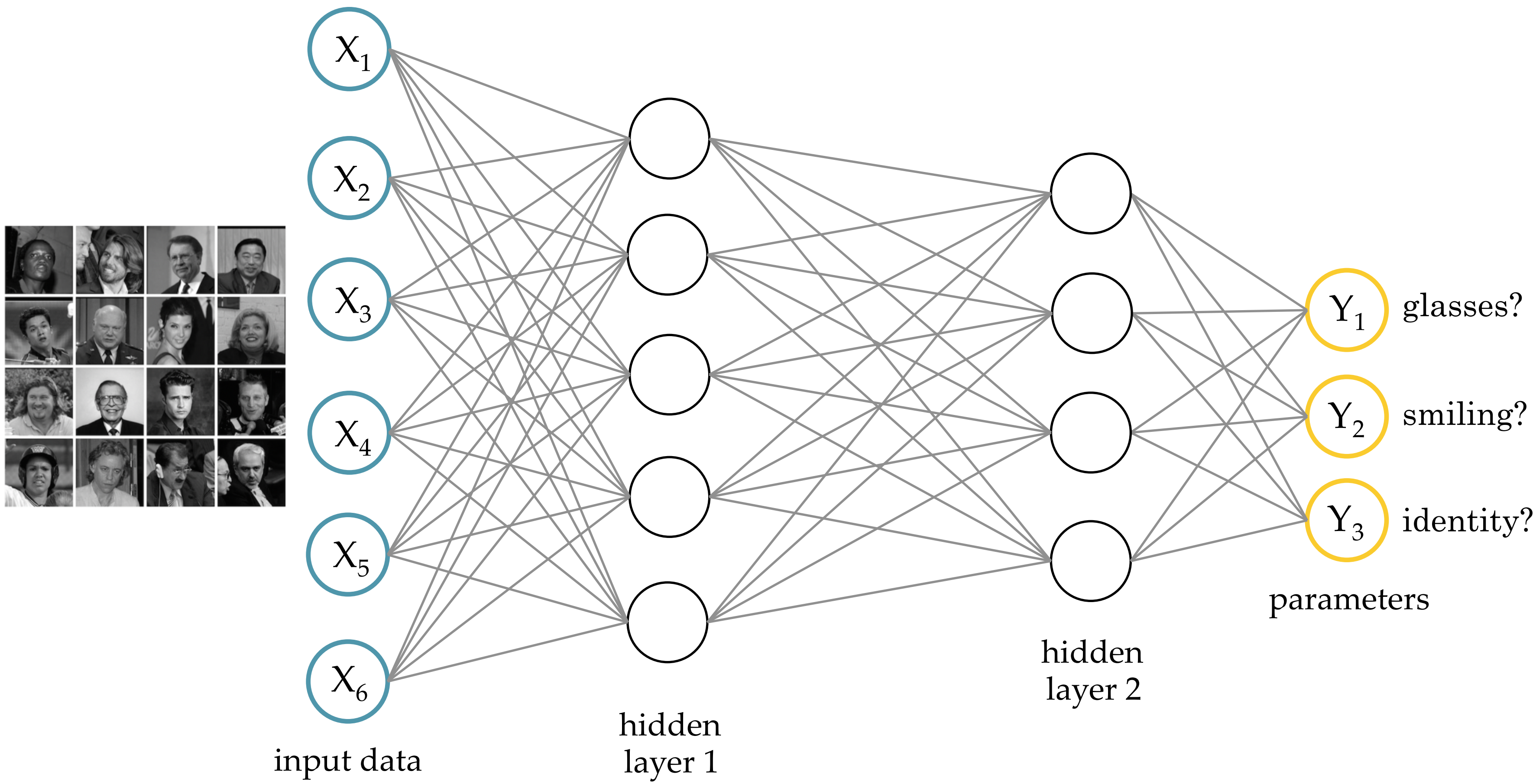
Goal: find a function between input and output



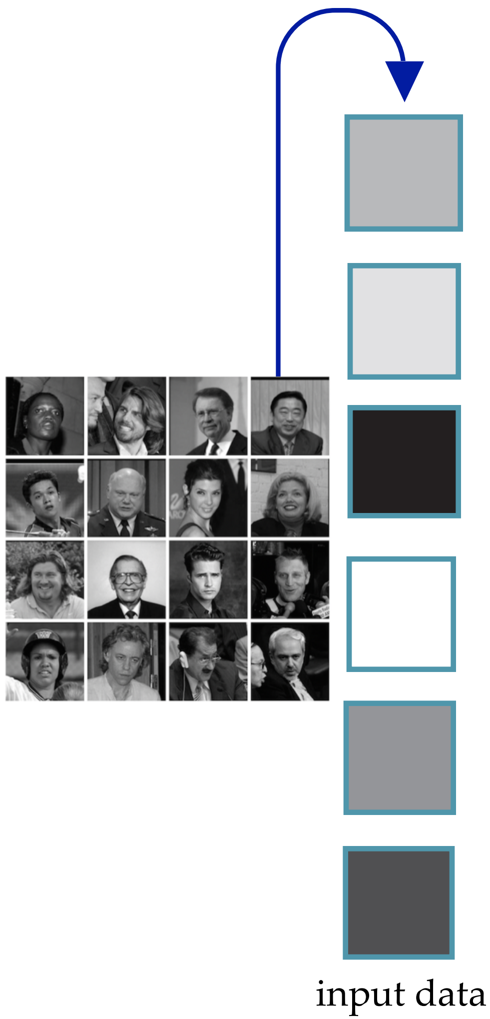
First idea: one hidden layer



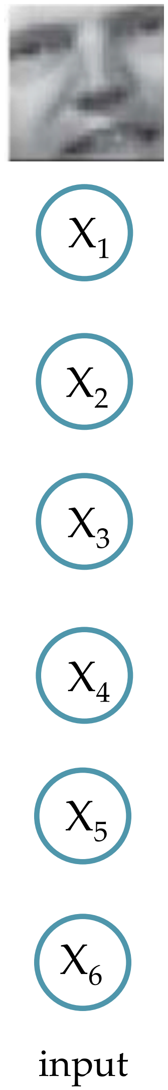
Second idea: more hidden layers (“deep” learning)



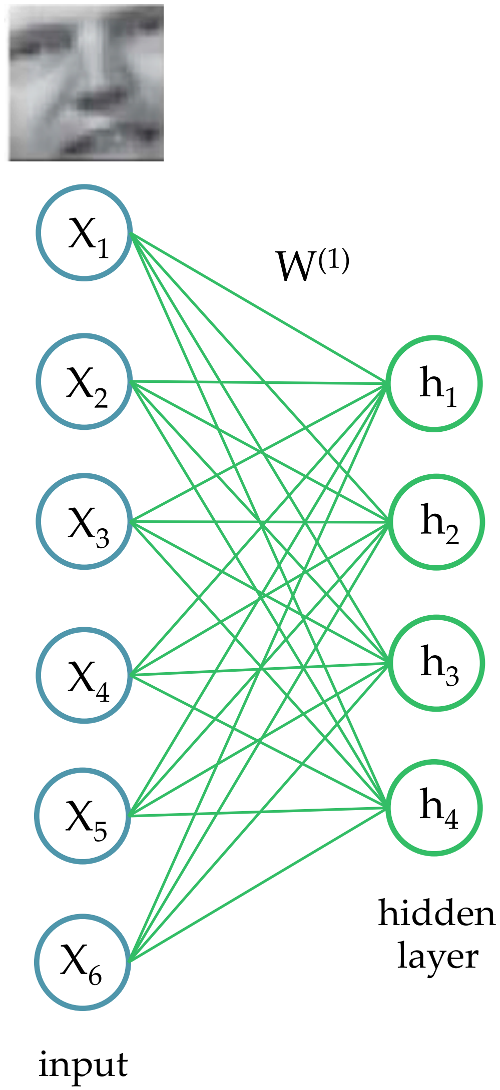
Flatten pixels of image into a single vector



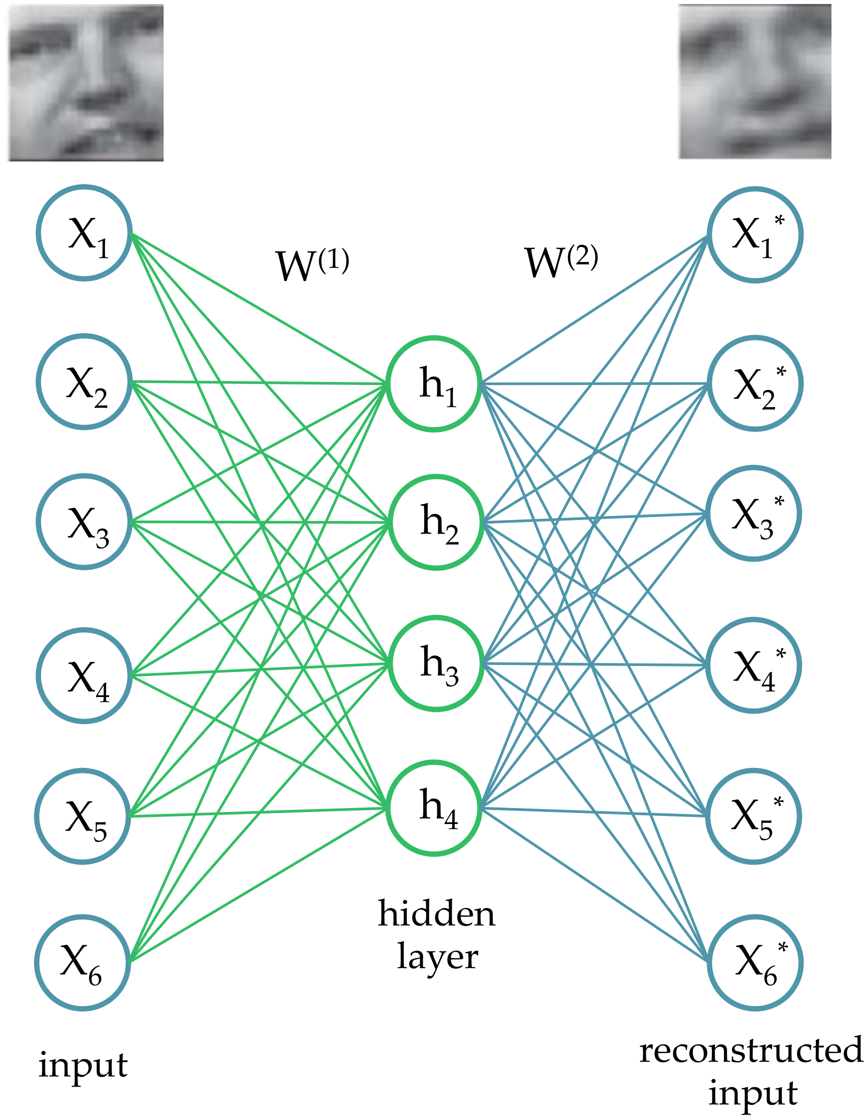
Detour to autoencoders



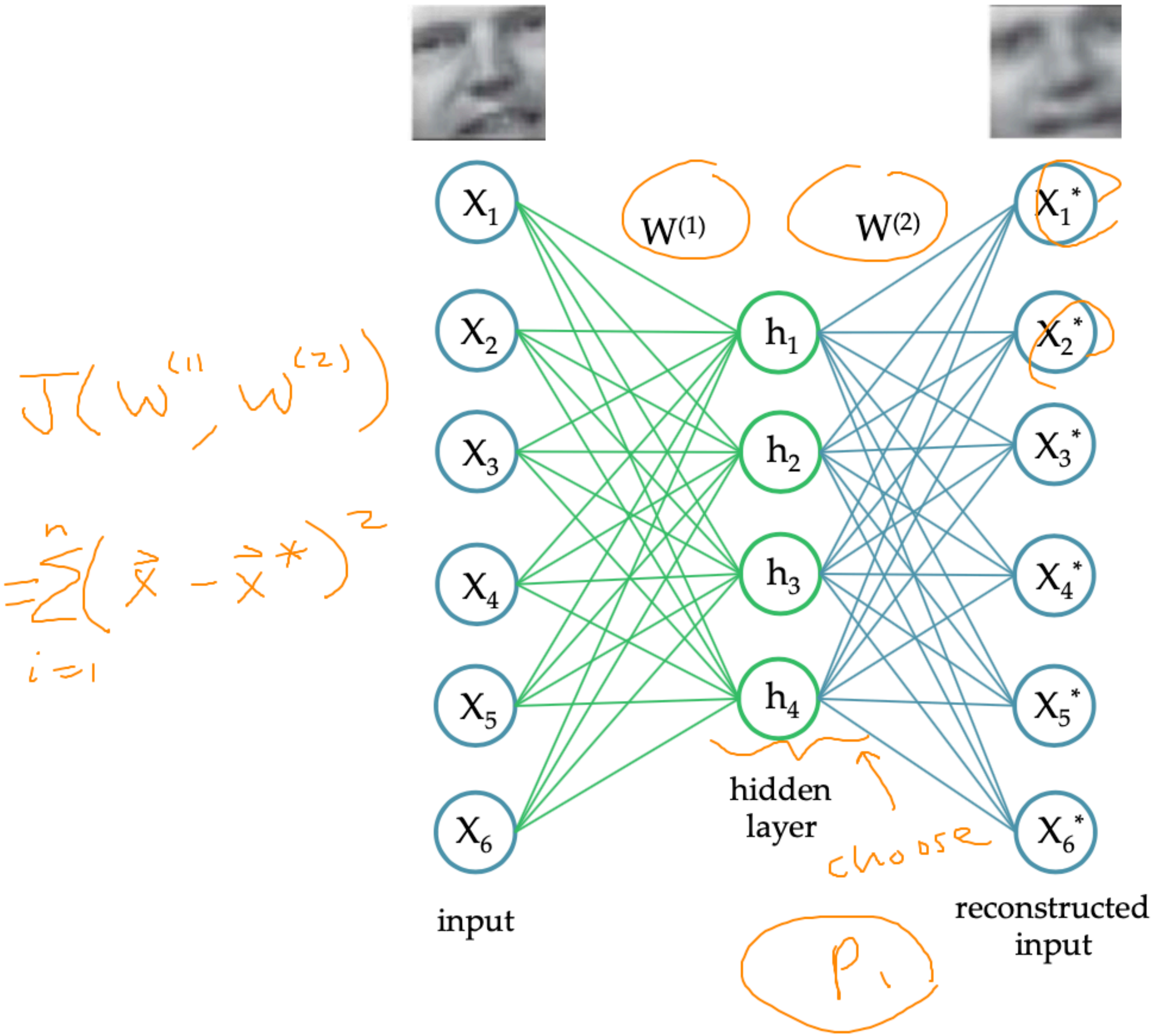
Detour to autoencoders



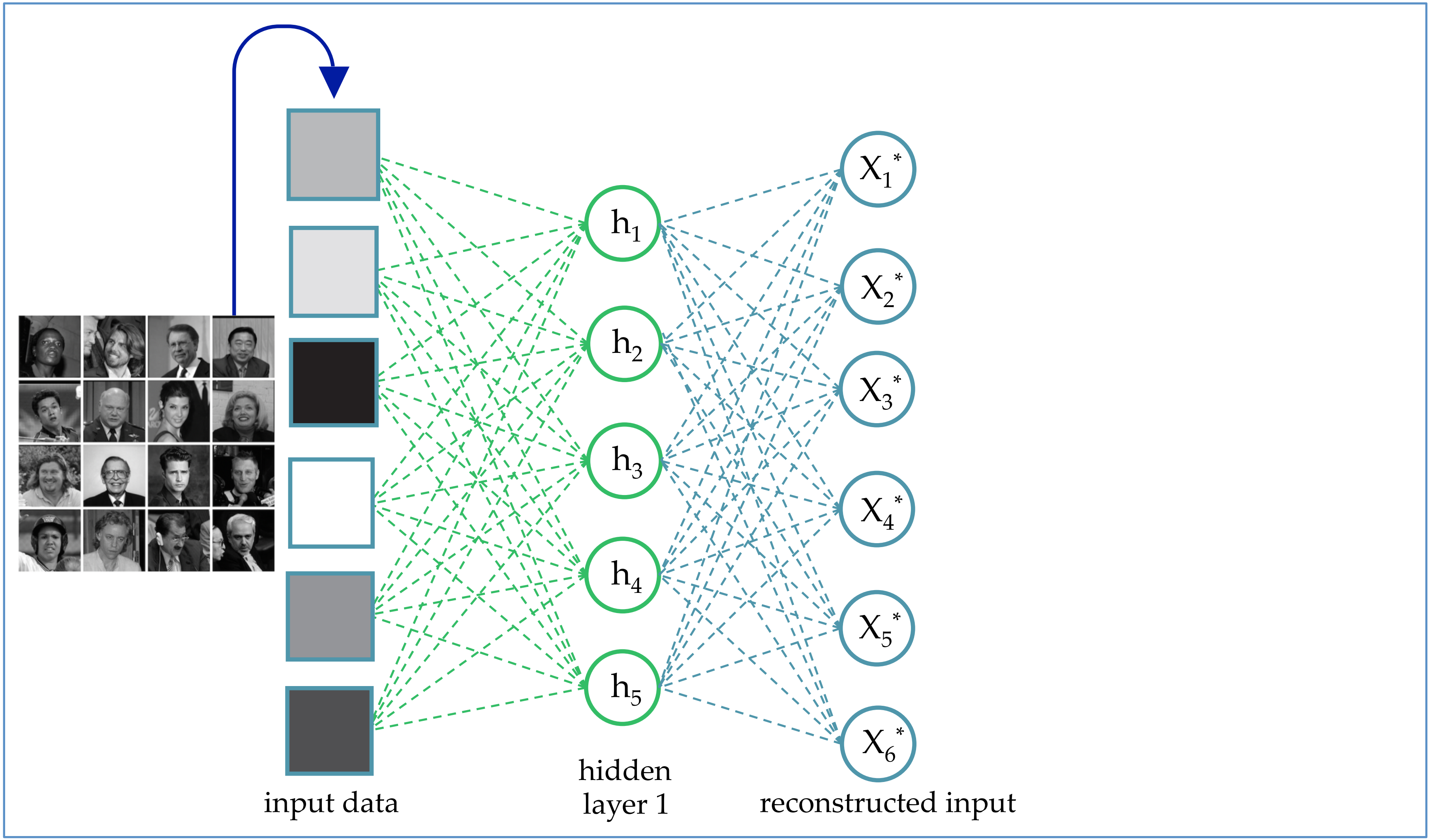
Detour to autoencoders



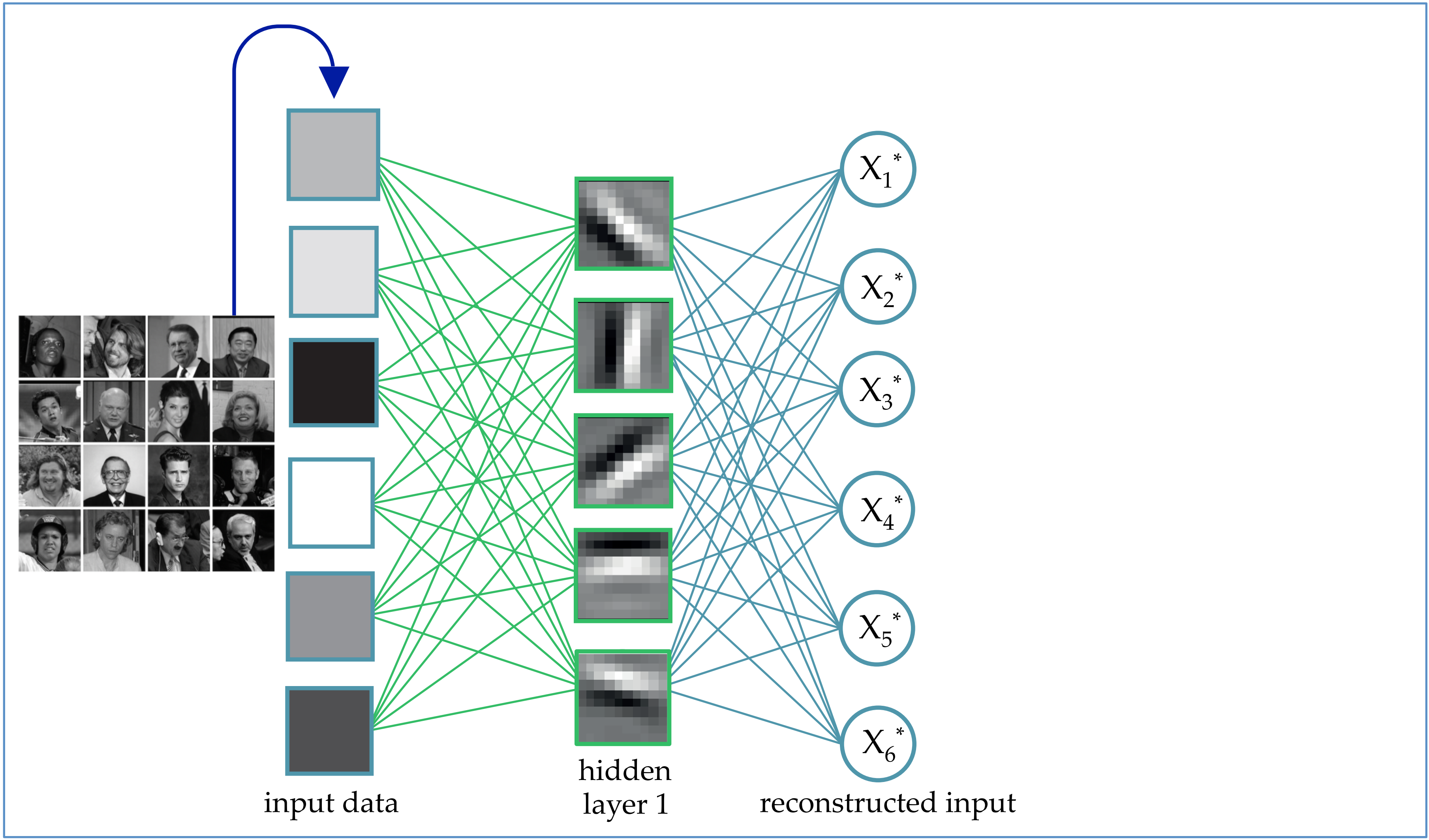
Detour to autoencoders



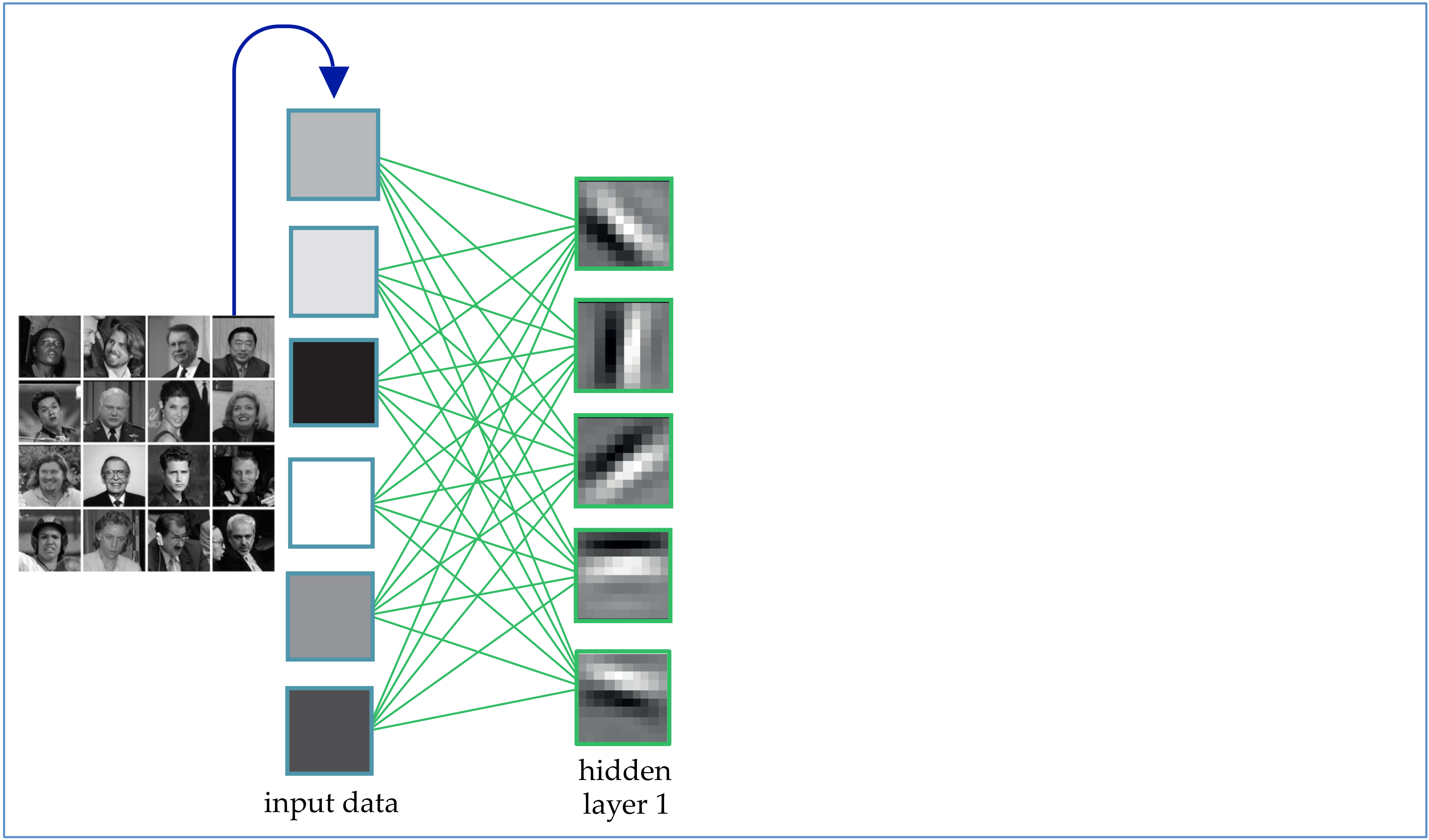
Use unsupervised pre-training to find a function from the input to itself



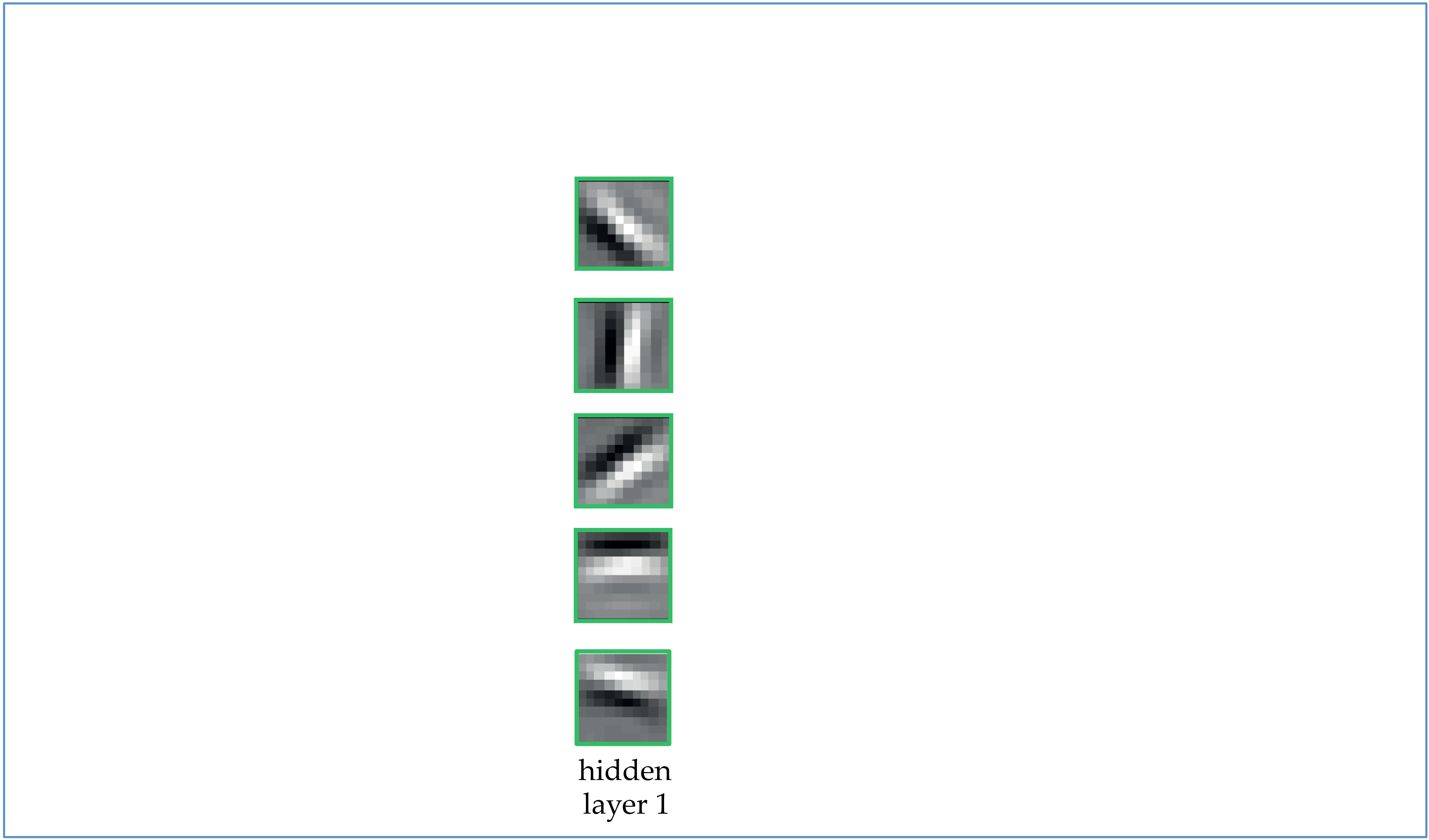
Hidden units can be interpreted as edges



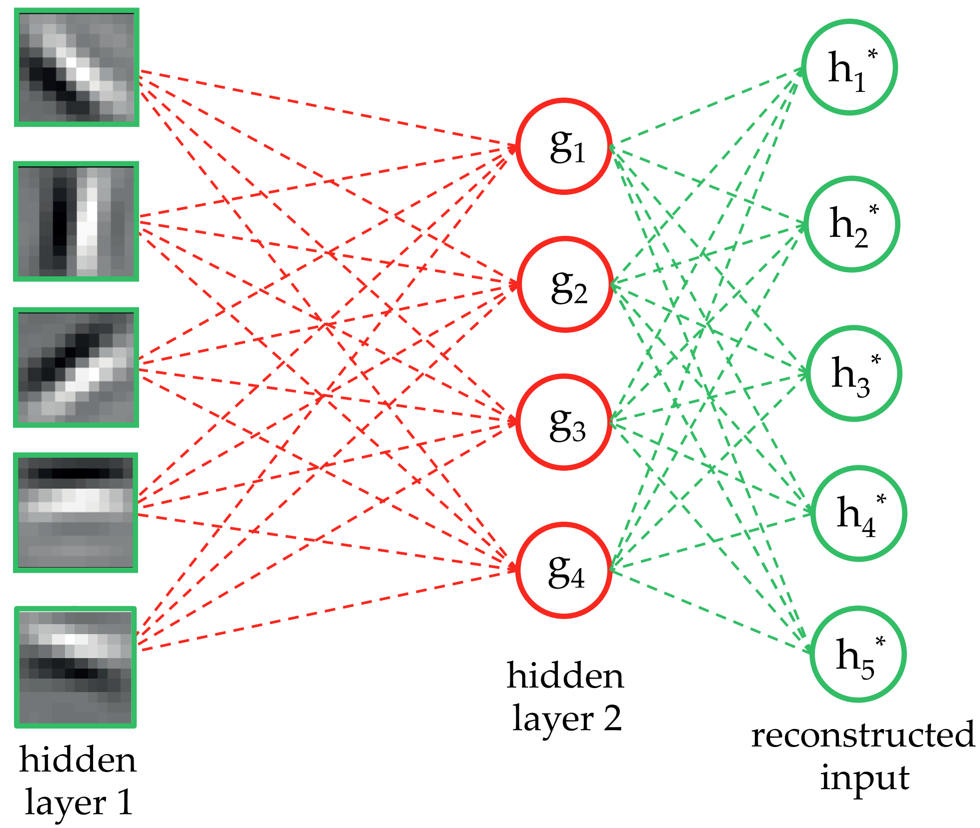
Now: throw away reconstruction and input



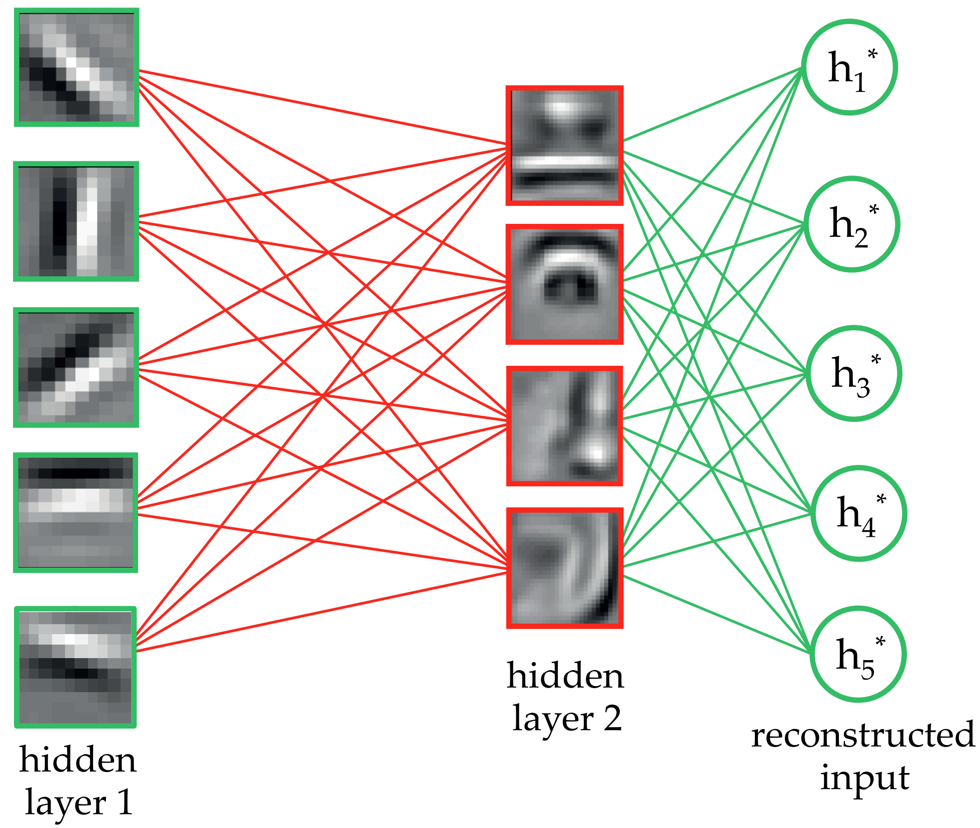
Now: throw away reconstruction and input



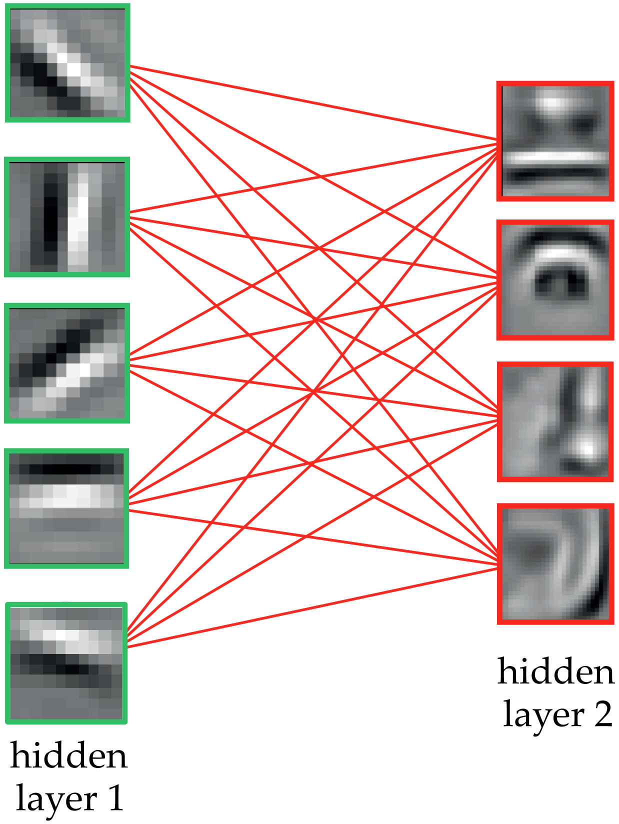
Then repeat the entire process for each layer



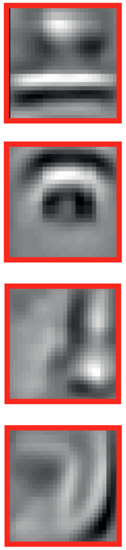
Then repeat the entire process for each layer



Then repeat the entire process for each layer

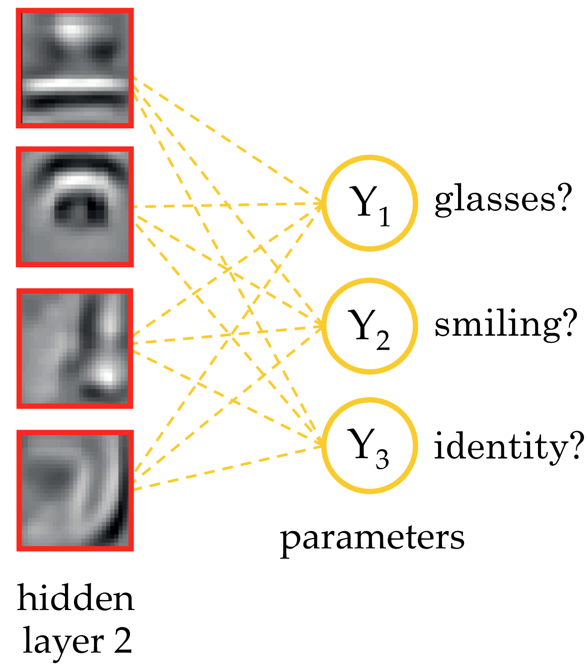


Then repeat the entire process for each layer

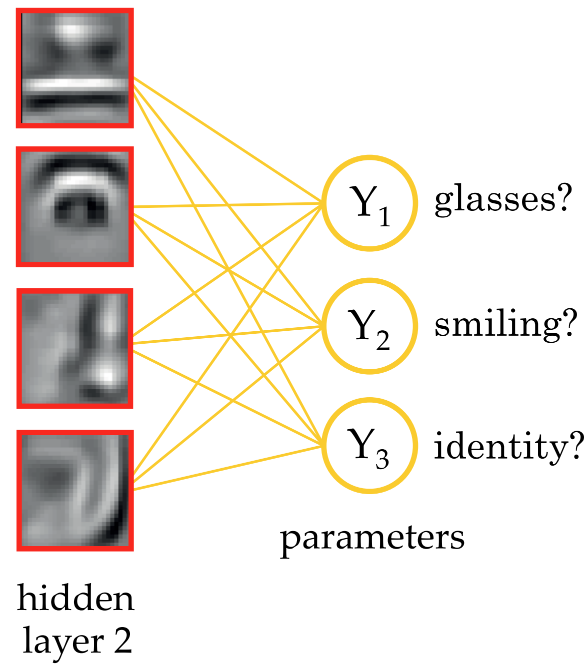


hidden
layer 2

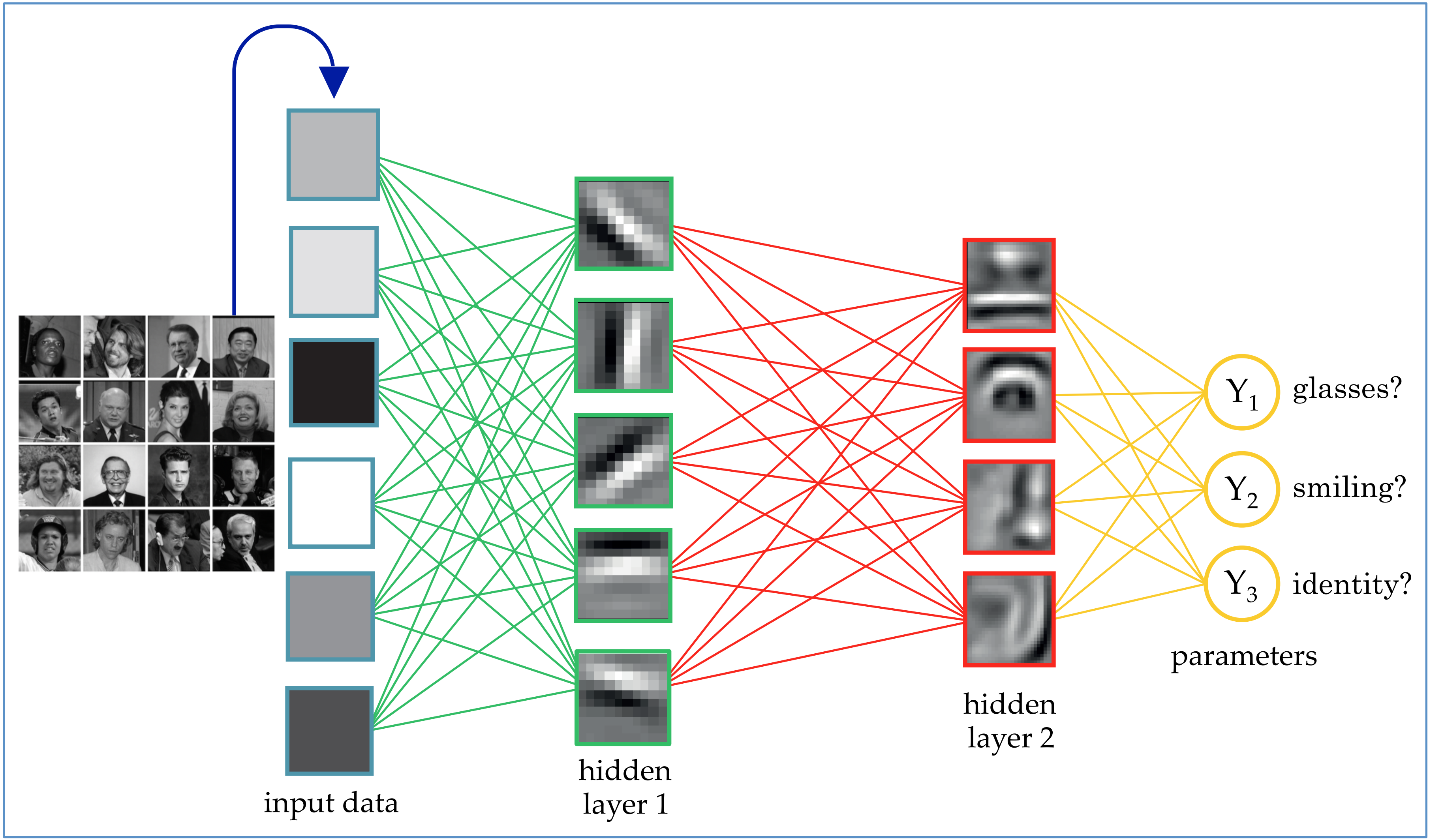
In the last layer, use the outputs (supervised)



In the last layer, use the outputs (supervised)

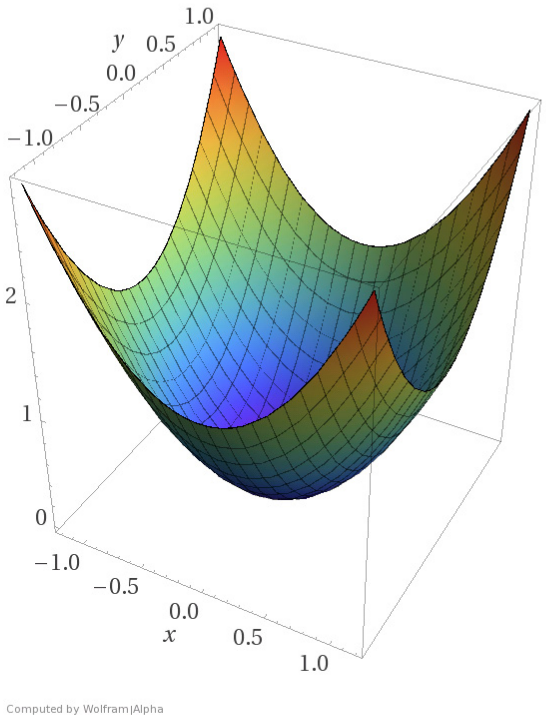


Finally, "fine-tune" the entire network!

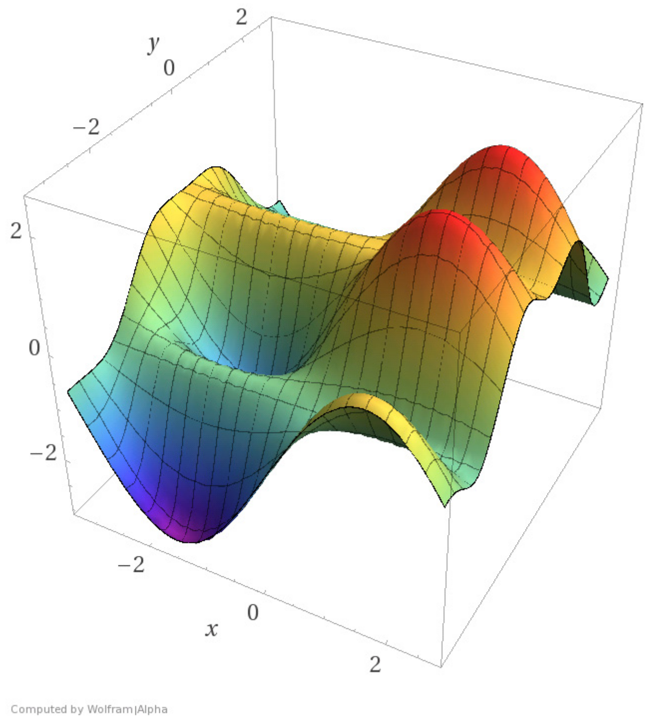


Takeaways

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



Convex



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)

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)

Recent Example: OpenAI's GPT-2

- "Language Models are Unsupervised Multitask Learners" https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf

- Decision not to release full model: https://openai.com/blog/better-language-models/

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. This is the approach we will use in Lab 8.

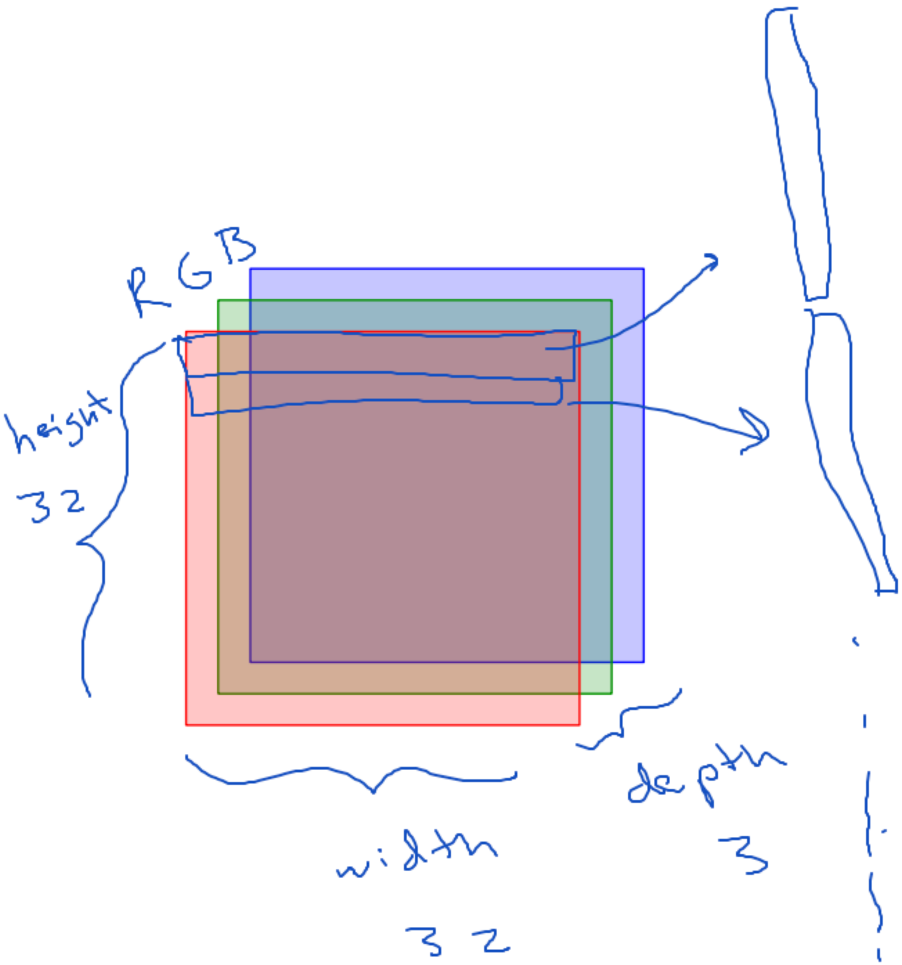
Lab 8 data pre-processing

- It is helpful to have our data be zero-centered, so we will subtract off the mean
- It is also helpful to have the features be on the same scale, so we will divide by the standard deviation
- We will compute the mean and std with respect to the *training data*, then apply the same transformation to all datasets

Lab 8 data pre-processing

- Input is now itself a multi-dimensional array
- For images, often the shape of each image will be (width, height, 3) for RGB channels
- Need to "*flatten*" or "unravel" for fully connected networks

Lab 8 Data



one data point

x.shape = (32, 32, 3)

mini-batch

X.shape = (64, 32, 32, 3)

fully connected

"flatten" unravel

$p = 32 \cdot 32 \cdot 3$

$= 3072$

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

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*)

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

Think about outside of class:
- Why do we use exp?
- Why don't we just take the max score?

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

Outline for November 13

- Introduction to neural networks
- Fully connected (FC) neural networks
- Convolutional neural networks (CNNs)
- Next week: more details on training NNs

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*

- 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

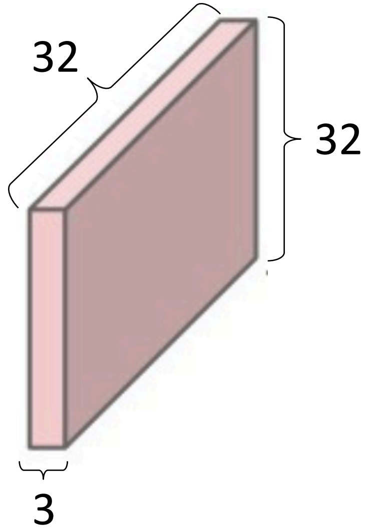


Image: modified from Stanford Course CS231n: http://cs231n.github.io/convolutional-networks/

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

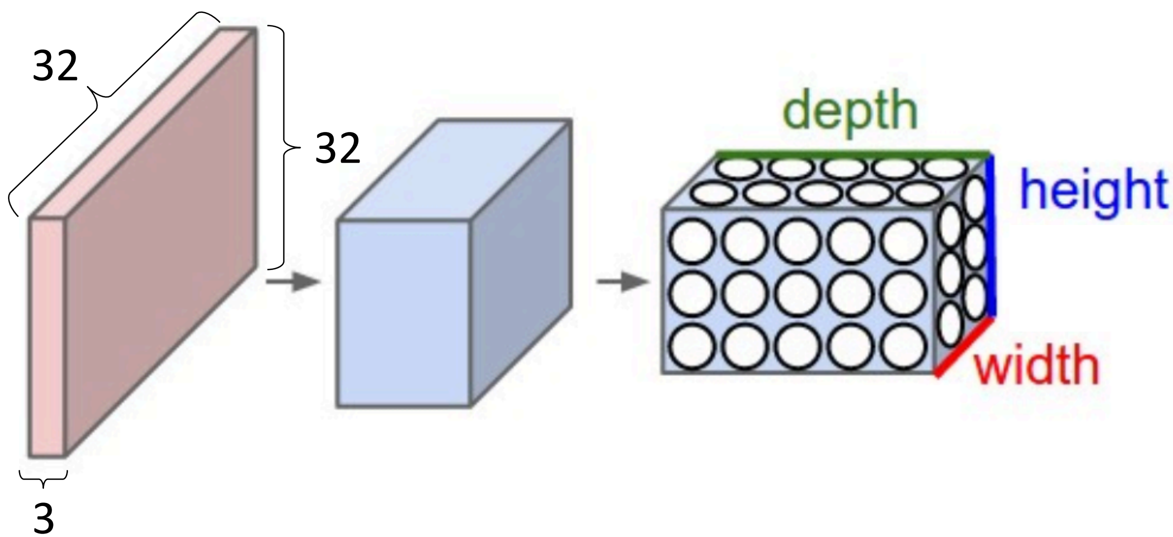


Image: modified from Stanford Course CS231n: http://cs231n.github.io/convolutional-networks/

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)

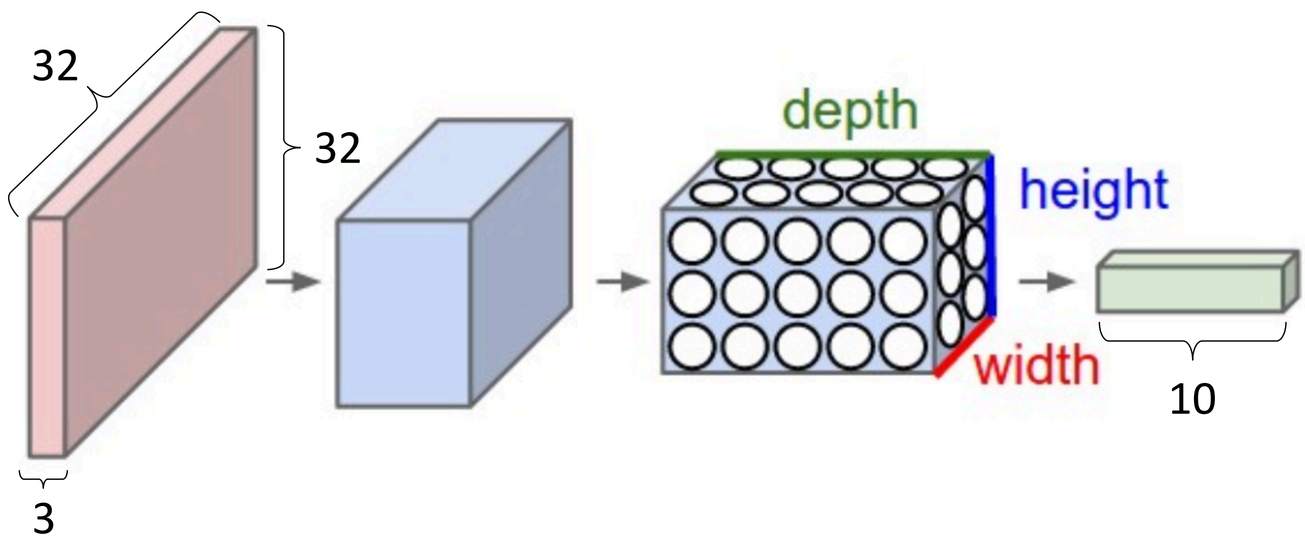


Image: modified from Stanford Course CS231n: http://cs231n.github.io/convolutional-networks/