Artificial Neural Networks

CS 534: Machine Learning

Slides adapted from Jinho Choi, Stuart Russell, Fei-Fei Li, Andrej Karpathy, Justin Johnson, John Buillinaria, and Kyunghyun Cho

Class Logistics

- Homework #3 due March 21st
- Project proposal feedback on Canvas
- Project madness at beginning of class on March 21st
  - 1 slide, **90 seconds** presentation per group — submission on Canvas by 11:59 pm March 20th
  - Overview of project

Class Logistics: Project Presentation

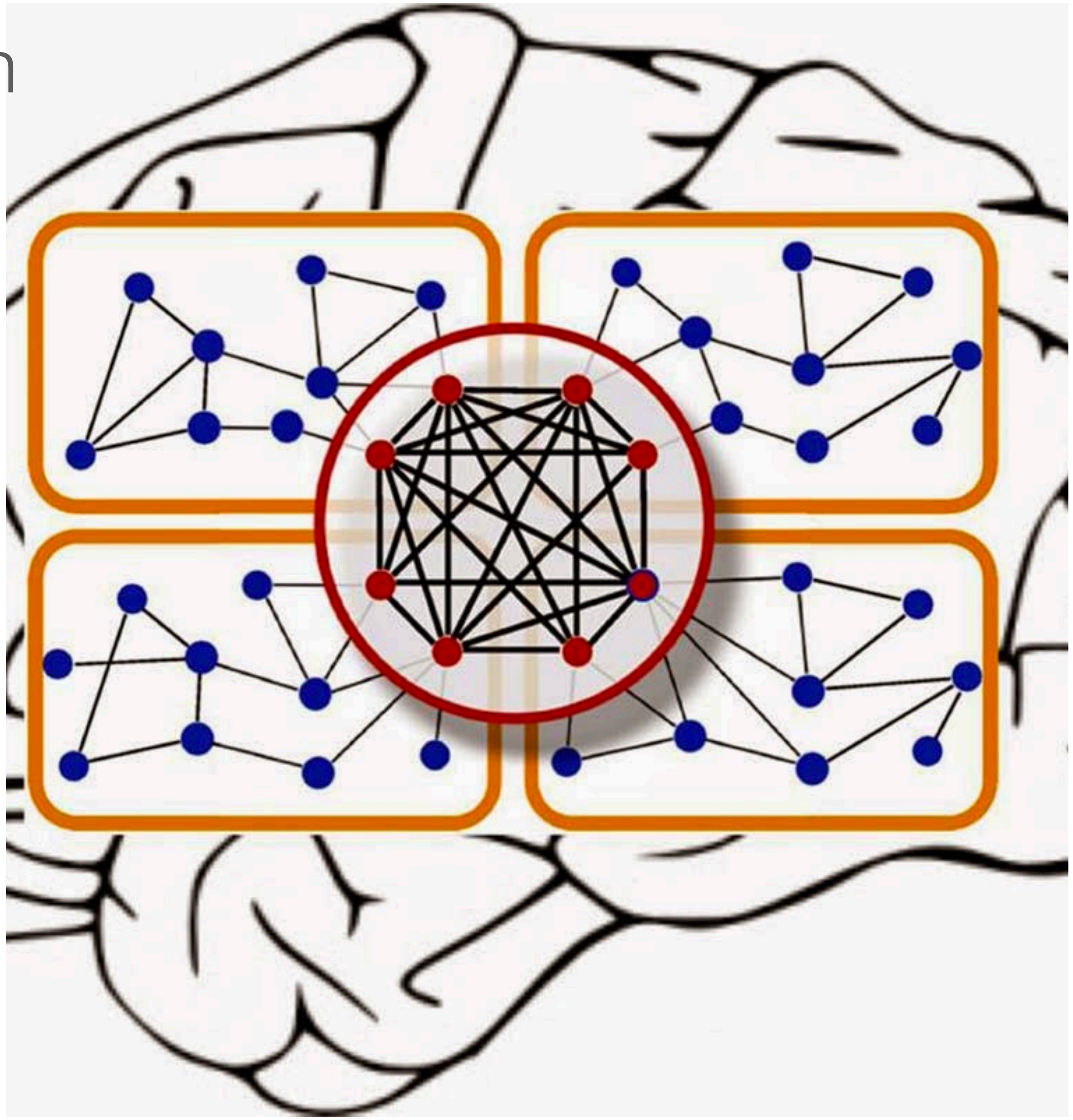
- 8 group projects —> 4 groups per class
- 18 minutes per group (includes Q&A)
  - Allocate 2-3 minutes for question and answer
  - Avoid downtime by using single computer for presentations — must be sent (email) to me by 9 AM on the morning of class

Class Logistics: Presentation Order

- 4/18
  1. Reza, Zelalem
  2. Qiyang, Zining, Jiayu
  3. Yidong, Qiyang
  4. Jing, Yi
- 4/20
  1. Steve, Katherine
  2. Funing, Yunyi, Xiaokun
  3. Damian
  4. Olivia, Tomer

Motivation: Human Brain

- Contains 10^{11} neurons, each with up to 10^5 connections
- Each neuron is fairly slow with switching time of 1 ms
- Computers at least 10^6 times faster in raw switching speed
- Brain is fast, reliable, and fault-tolerant



Motivation: Neuron

- Electrically excitable cell that processes and transmits information
- Information comes in on the dendrites (input)
- If neuron excited/activated, send a spike of electrical activity to axon (output)

Artificial Neural Networks

- Based on assumption that a computational architecture similar to brain would duplicate its abilities
- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process
- Many different kinds of architectures

Review: Linear Regression (MLR)

- Hypothesis of the form

$$f(\mathbf{x}) = \beta_0 + \sum_{i=1}^{p} x_i \beta_i$$

- Learn weights to minimize least squares problem

$$\min_{\beta} (\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^\top (\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) \implies \hat{\boldsymbol{\beta}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$$

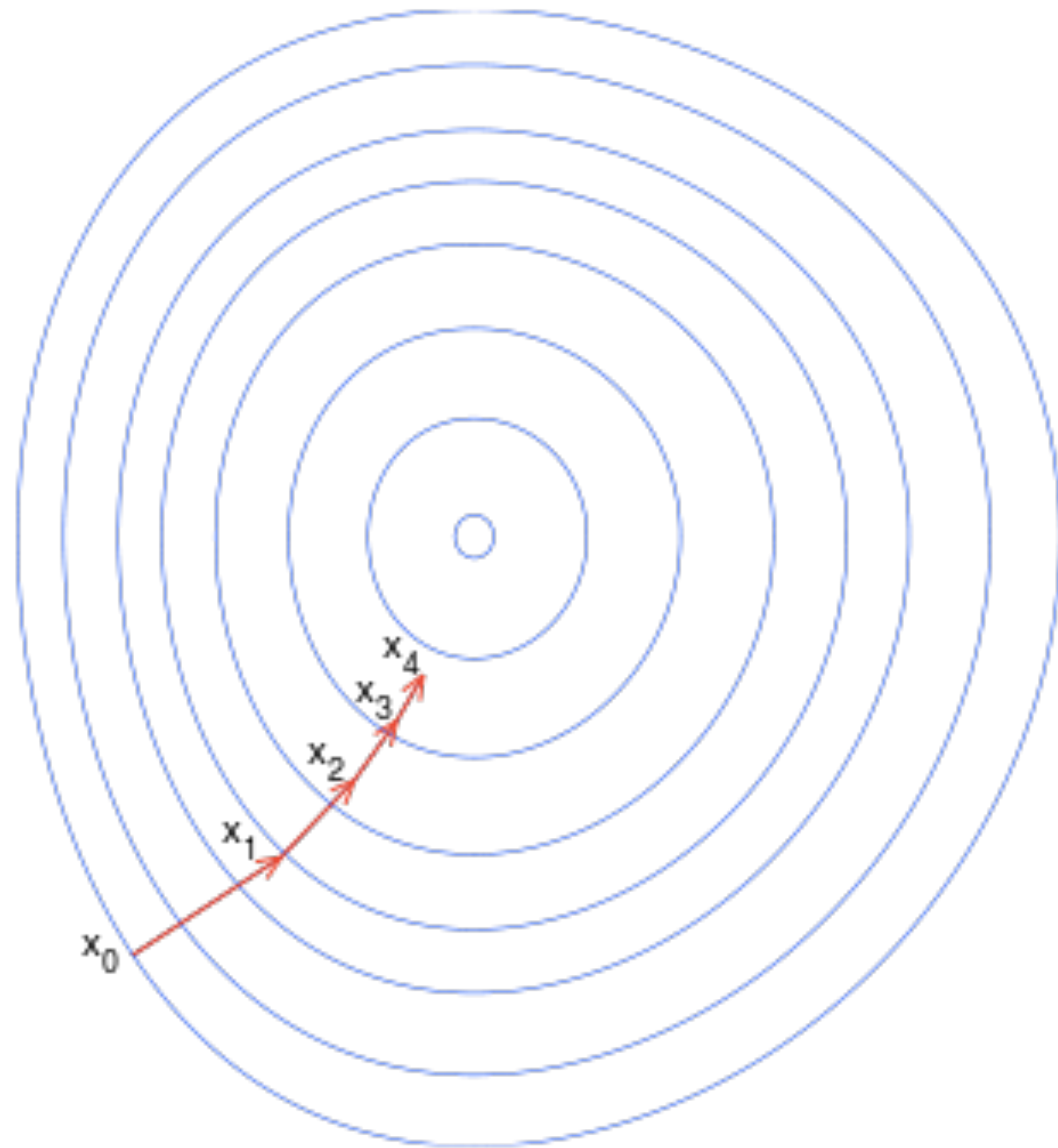
- Alternative to matrix inversion: gradient descent

Review: Gradient Descent (GD)

- Simple and popular algorithm
- Idea: Take a step proportional to the negative of the gradient

$$\theta_i := \theta_i - \eta \frac{\partial L}{\partial \theta_i}$$

- Eventually will find the optimal (minimum) point



Example: GD for MLR

- Optimization problem:

$$\min_{\boldsymbol{\beta}} ||\mathbf{y} - \boldsymbol{\beta}\mathbf{X}||_2^2$$

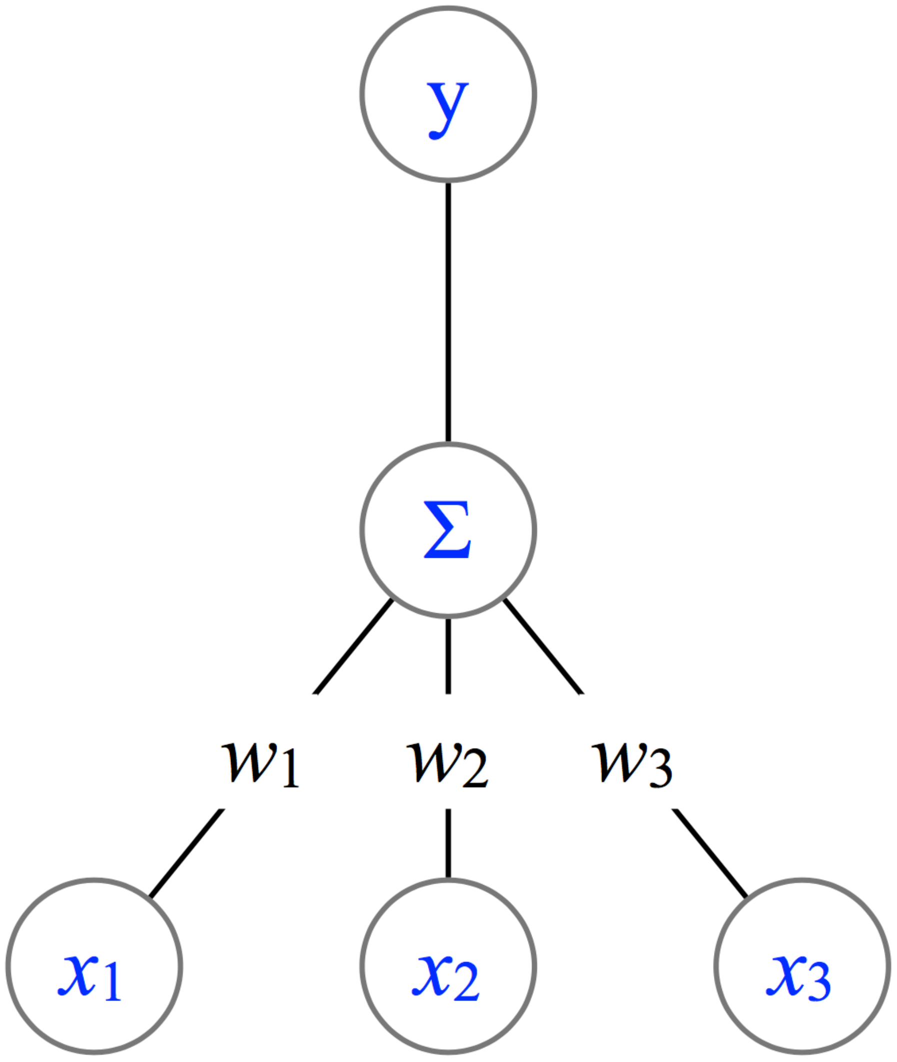
- Gradient update:

$$\boldsymbol{\beta}^+ = \boldsymbol{\beta} + \frac{\eta}{N} \sum_i (y_i - \mathbf{x}_i \boldsymbol{\beta}) \mathbf{x}_i$$

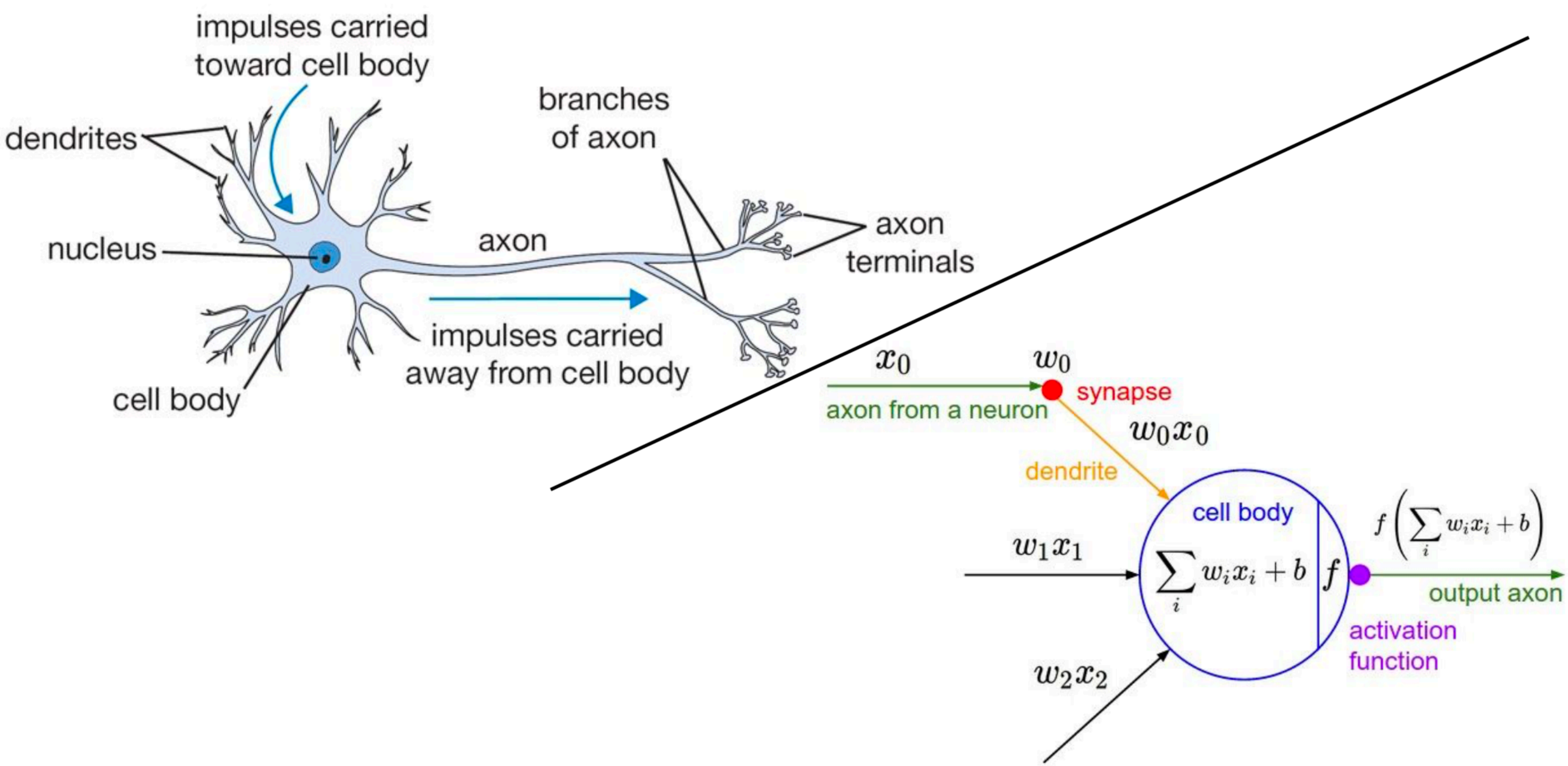
Perceptron [Rosenblatt, 1957]

- Uses hyperplane classifier to map input to binary output
- Compute linear combination of the inputs and threshold it

$$f_{\mathbf{w}}(\mathbf{x}) = \mathrm{sign}(\mathbf{x} \cdot \mathbf{w}) = \begin{cases} +1 & \text{if } \mathbf{x} \cdot \mathbf{w} > 0 \\ -1 & \text{otherwise} \end{cases}$$



Neuron —> Perceptron



Perceptron Algorithm

- Loss function uses functional margin

$$\ell(y, f_{\mathbf{w}}(\mathbf{x})) = \sum_n \mathbf{w}^\top \mathbf{x}_i y_i$$

- Solve via gradient descent
- But what if we want it to be online (does not need to consider the entire data set at the same time)?

Gradient Descent: Reformulated

- Recall empirical risk:

$$R_{\text{EMP}}[f(\mathbf{x})] = \frac{1}{N} \sum_n \ell(f(\mathbf{x_n}), y_n)$$

- Think of GD in terms of ERM:

$$\theta^+ = \theta - \gamma \frac{1}{N} \sum_n \nabla_\theta \ell(f(\mathbf{x}_n), y_n)$$

γ: learning rate or gain

$\frac{1}{N} \sum_n \nabla_\theta \ell(f(\mathbf{x}_n), y_n) = \nabla R_{\text{EMP}}[f(\mathbf{x})]$

- "True" gradient descent is a batch algorithm

Motivation: Stochastic Optimization

- Online / streaming data —> can't wait for all
- Non-stationary data (moving target) —> model should not be static
- Sufficient samples means information is redundant amongst samples —> more frequent, noisy updates

Stochastic Optimization

- Idea: Estimate function and gradient from a small, current subsample of your data
    - Function: $f(x) \to \tilde{f}(x)$
    - Gradient: $\nabla f(x) \to \tilde{\nabla} f(x)$
- With enough iterations and data, you will converge in expectation to the true minimum

Stochastic Optimization

- Pro: Better for large datasets and often faster convergence
- Con: Hard to reach high accuracy
- Con: Best classical methods can't handle stochastic approximation
- Con: Theoretical definitions for convergence not as well-defined

Stochastic Gradient Descent (SGD)

- Randomized gradient estimate to minimize the function using a single randomly picked example

$$E[\tilde{\nabla} f] = \nabla f$$

- The resulting update is of the form:

$$\theta^+ = \theta - \gamma \nabla_\theta \ell(f(\mathbf{x}_i), y_i)$$

- Although random noise is introduced, it behaves like gradient descent in its expectation

SGD Algorithm

Initialize parameter θ and learning rate η

while *not converged* **do**

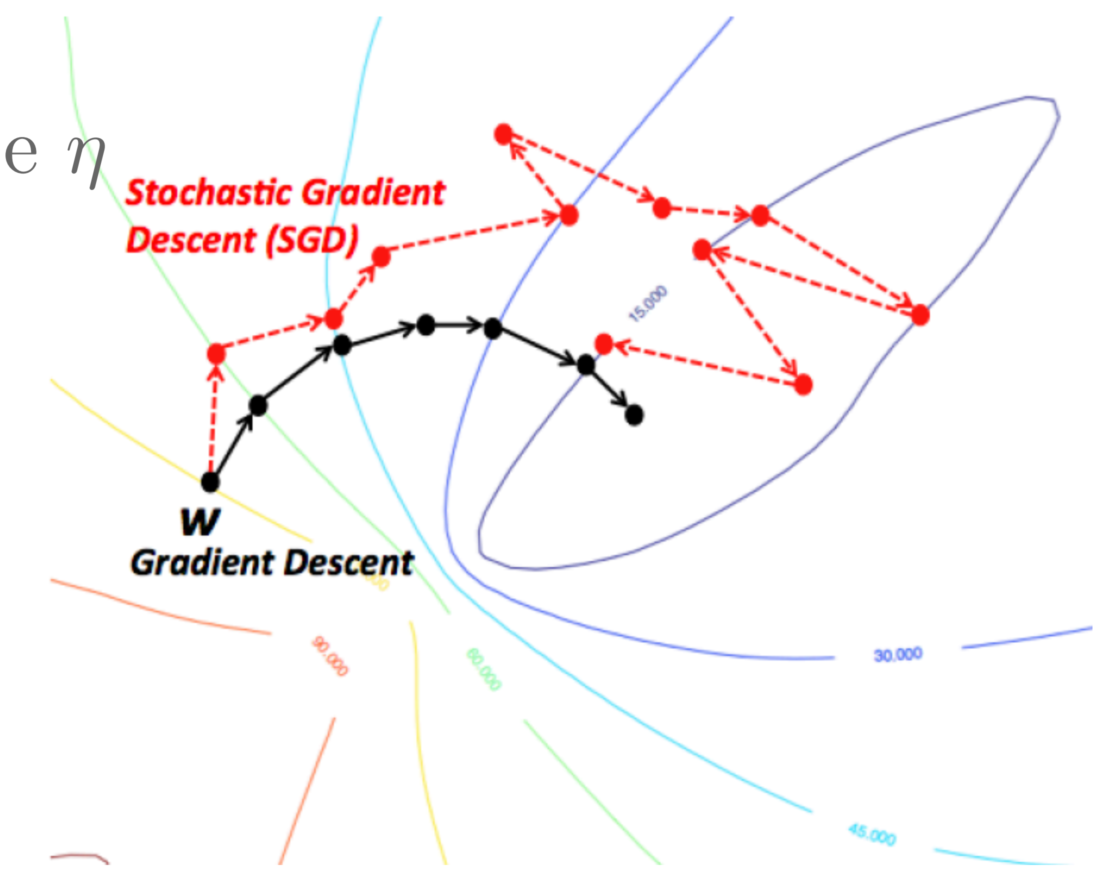
  Randomly shuffle training data

  for $i = 1, \cdot, N$ **do**

    $\theta^{+} = \theta - \gamma \nabla_{\theta} \ell(f(\mathbf{x}_i), y_i)$

  end

end



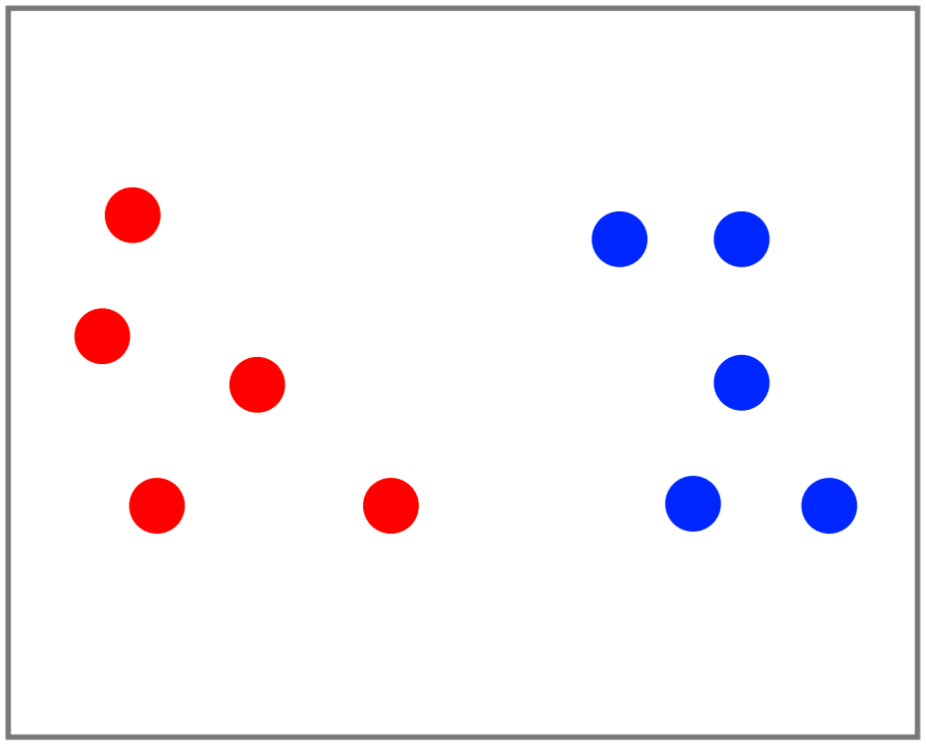
https://wikidocs.net/3413

Perceptron: Learning

- Perceptron uses SGD to learn the parameters
- Without loss of generality, can set learning parameter to be 1
- For each point:
  - If successfully classified, do nothing
  - Incorrectly classified, update weight vector

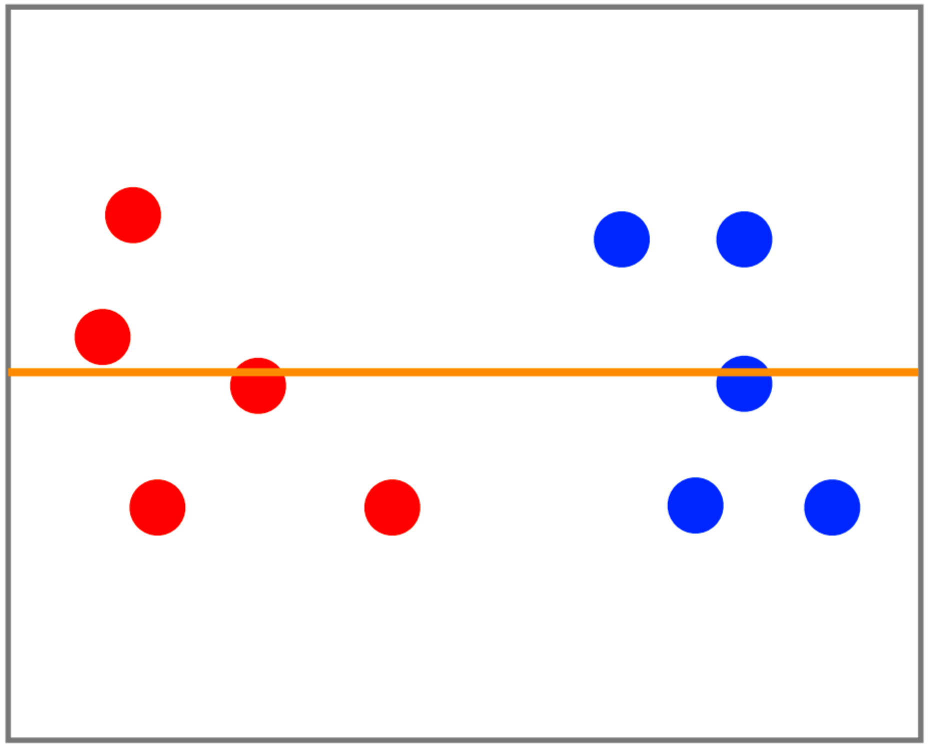
$$\mathbf{w}^{+} = \mathbf{w} + \mathbf{x}_i y_i$$

Perceptron: Learning Example

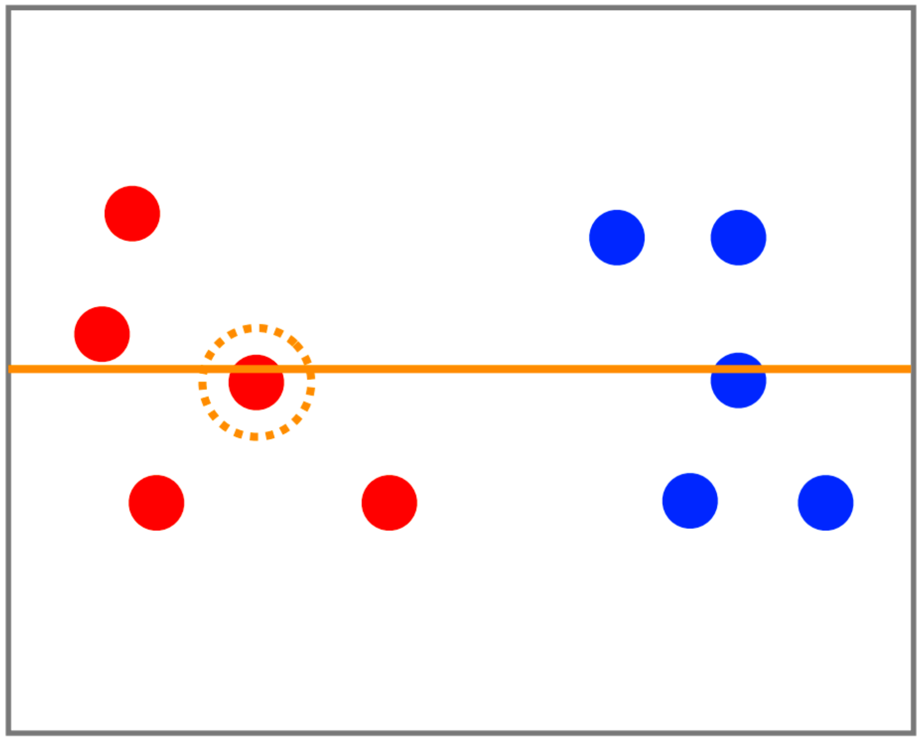


Training data

Initialize parameters

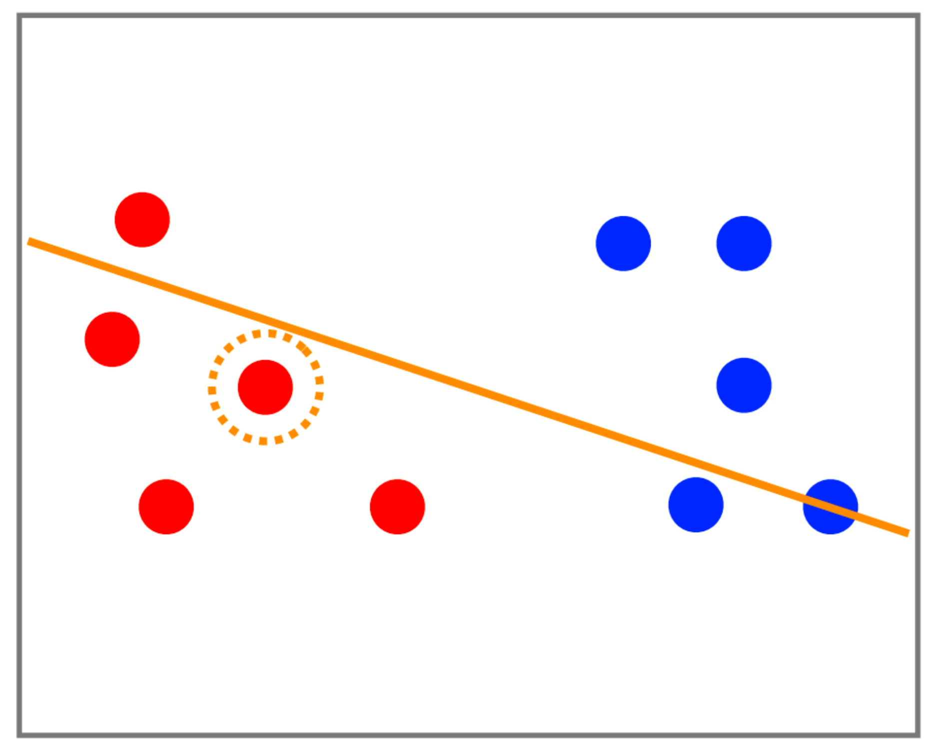


Perceptron: Learning Example

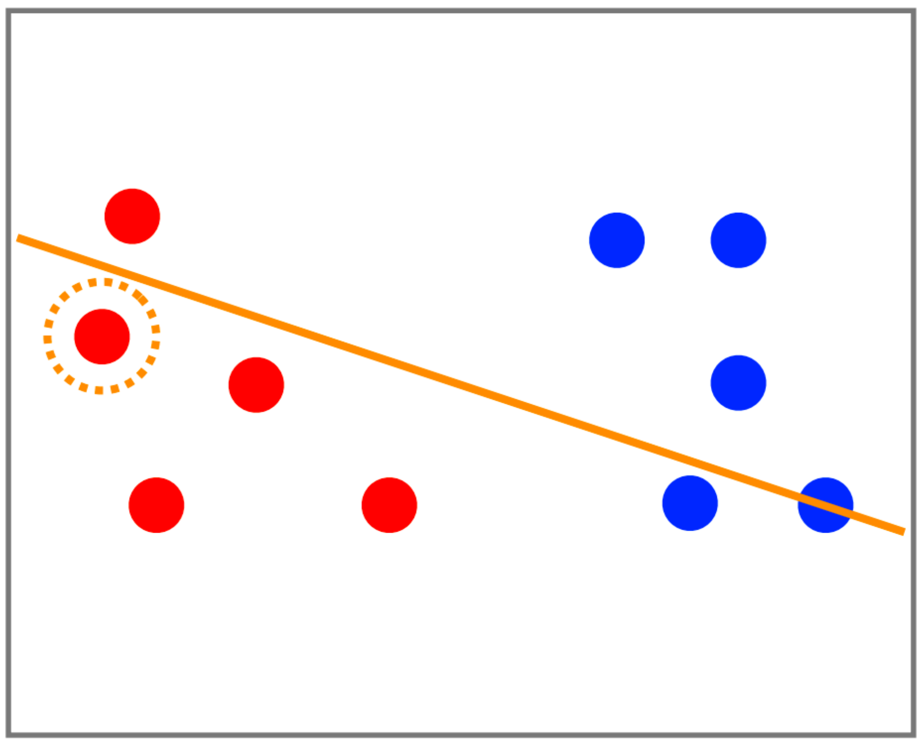


Randomly select point
— incorrectly classified

Update parameters

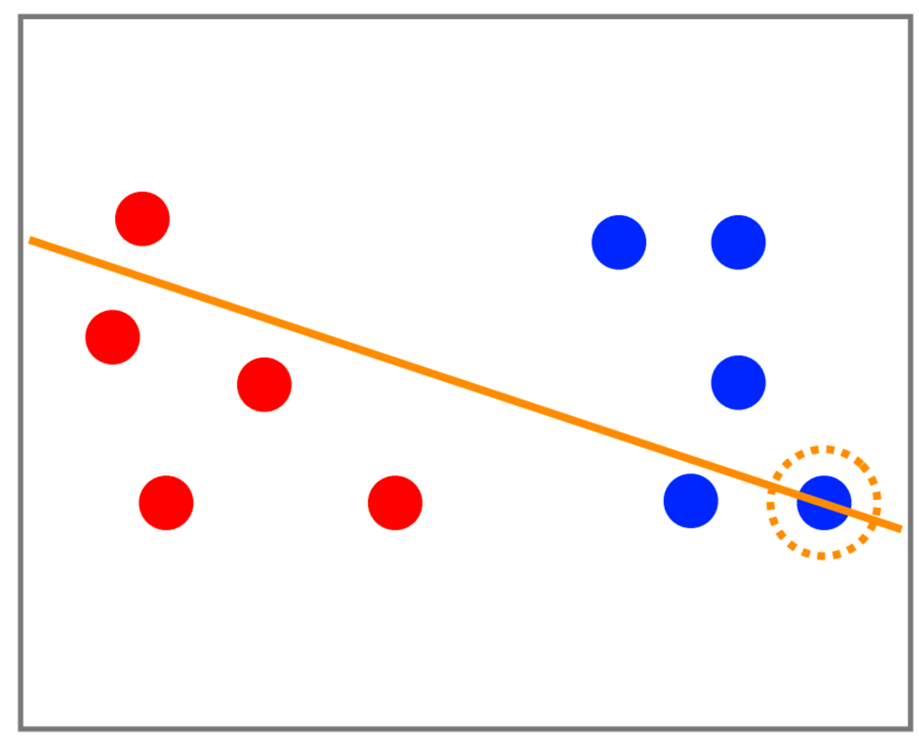


Perceptron: Learning Example

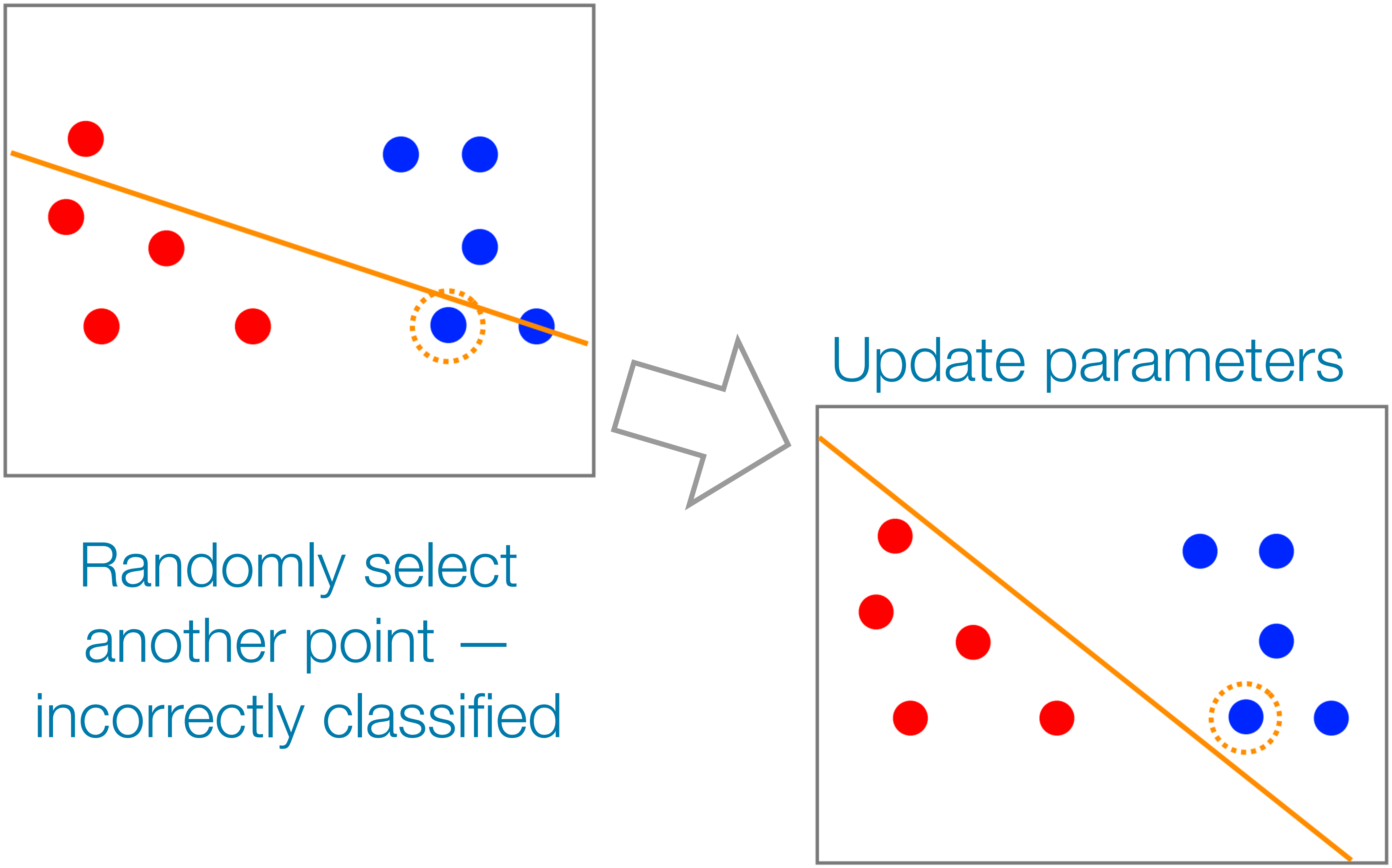


Randomly select another point — correct classified do nothing

Randomly select another point — correct classified do nothing



Perceptron: Learning Example



Randomly select another point — incorrectly classified

Update parameters

Perceptron Convergence Theorem

- Intuition: perceptron will converge more quickly for easy learning problems compared to large learning problems
  - Classify "easy" and "hard" via the margin
- **Theorem.** *Suppose the perceptron algorithm is run on a linearly separable data set* $\mathbf{D}$ *with margin* $\gamma > 0$*. Assume that* $||x|| \leq 1$ *for all* $x \in \mathbf{D}$*. Then the algorithm will converge after at most* $\frac{1}{\gamma^2}$ *updates.*

Perceptron: Issues

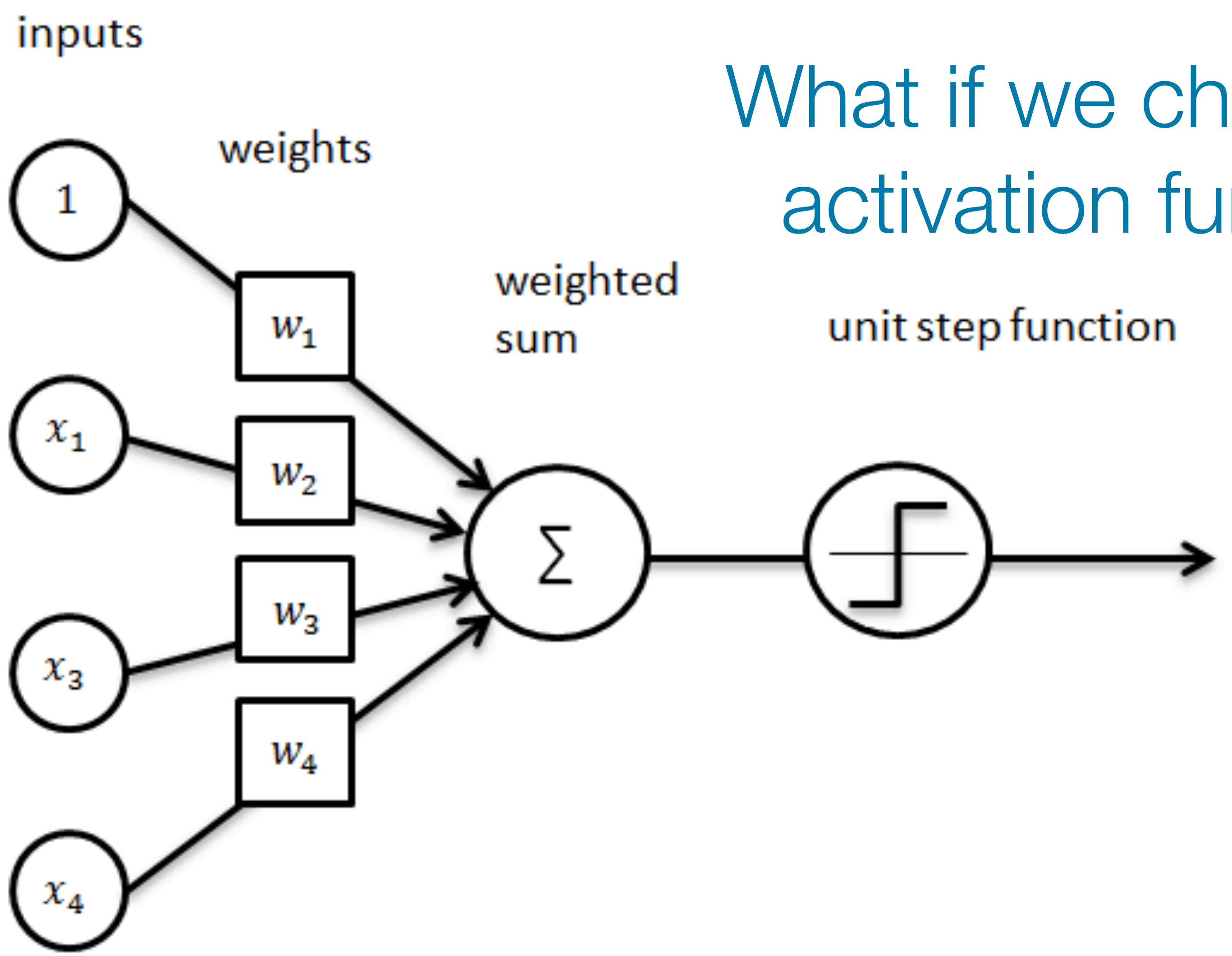
- If data isn't linearly separable, no guarantees of convergence or training accuracy
- Even if training data is linearly separable, perceptron can overfit
- Averaged perceptron (average weight vectors across all iterations) is an algorithmic modification that helps both issues

Motivation: Need for Networks

- Perceptrons have very simple decision surface (linearly separable functions)
- What if we connect several of them together?
  - Error surface is not differentiable — why?
  - Can't apply gradient descent to find a good set of weights

Perceptron: Revisited

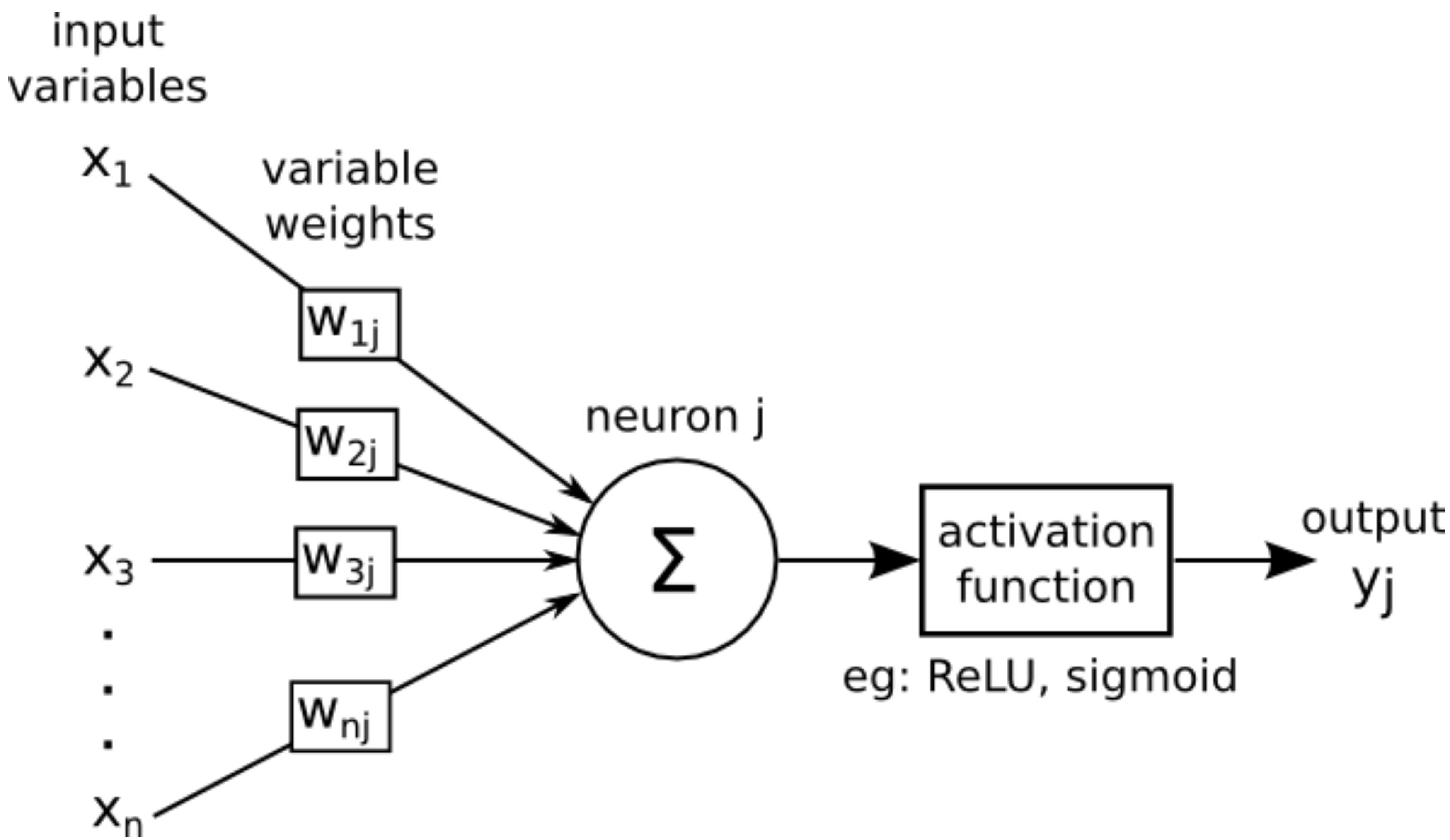
Introduce bias (input = 1)



What if we change the activation function?

http://ataspinar.com/2016/12/22/the-perceptron/

Neuron: Generalized Perceptron



http://dataminingtheworld.blogspot.com/

Neuron: Sigmoid Unit

- Activation function is sigmoid function

$$\sigma(\mathbf{x}) = \frac{1}{1 + \exp(-\mathbf{x})}$$

- Nice property of sigmoid

$$\frac{\partial \sigma(\mathbf{x})}{\partial x} = \sigma(\mathbf{x})(1 - \sigma(\mathbf{x}))$$

- Can derive gradient descent rules to train multi-layer networks

Sigmoid Units vs Perceptron

- Sigmoid units provide “soft” threshold
- Perceptrons provide “hard” threshold
- Expressive power is the same: limited to linearly separable instances

Neuron: Popular Activation Functions

- Sigmoid function

- Hyperbolic tangent function

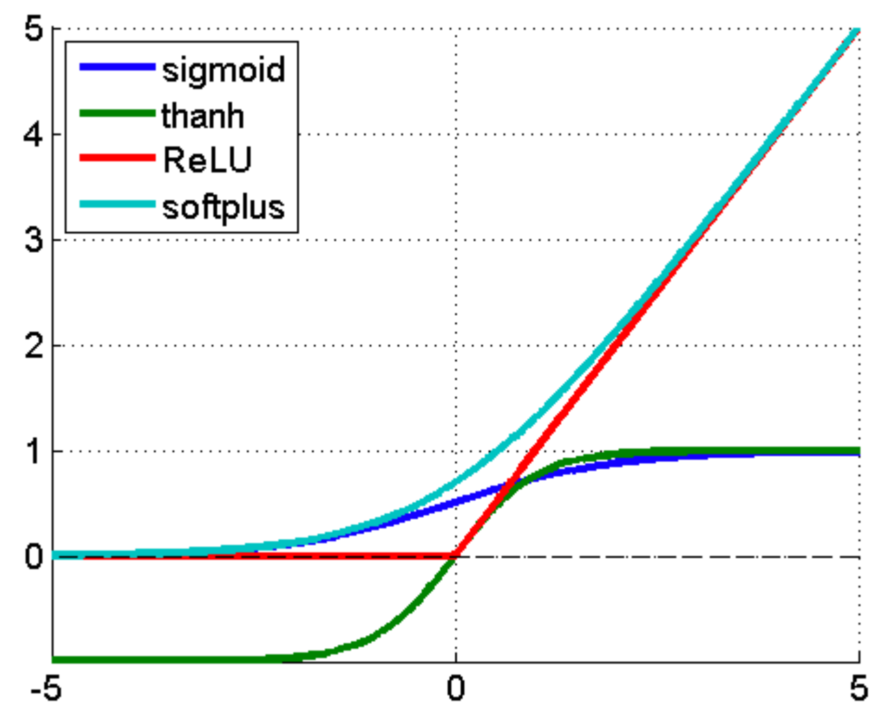
$$f(\mathbf{x}) = \sinh(\mathbf{x})/\cosh(\mathbf{x})$$

- Rectified linear unit (ReLU)

$$f(\mathbf{x}) = \max(0, \mathbf{x})$$

- Softplus

$$f(\mathbf{x}) = \log(1 + \exp(\mathbf{x}))$$



https://imiloainf.wordpress.com/2013/11/06/rectifier-nonlinearities/

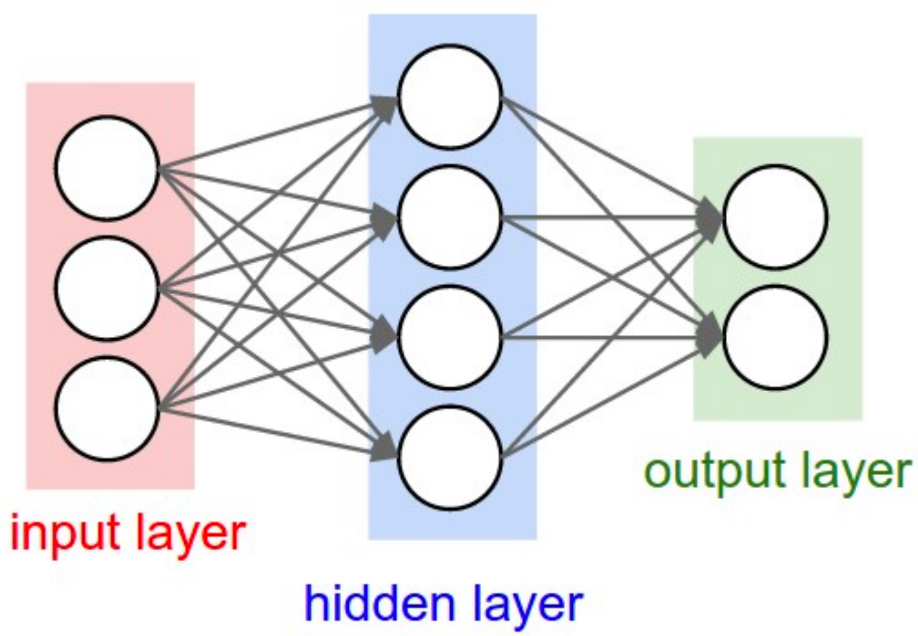
Neural Networks

Neural Networks

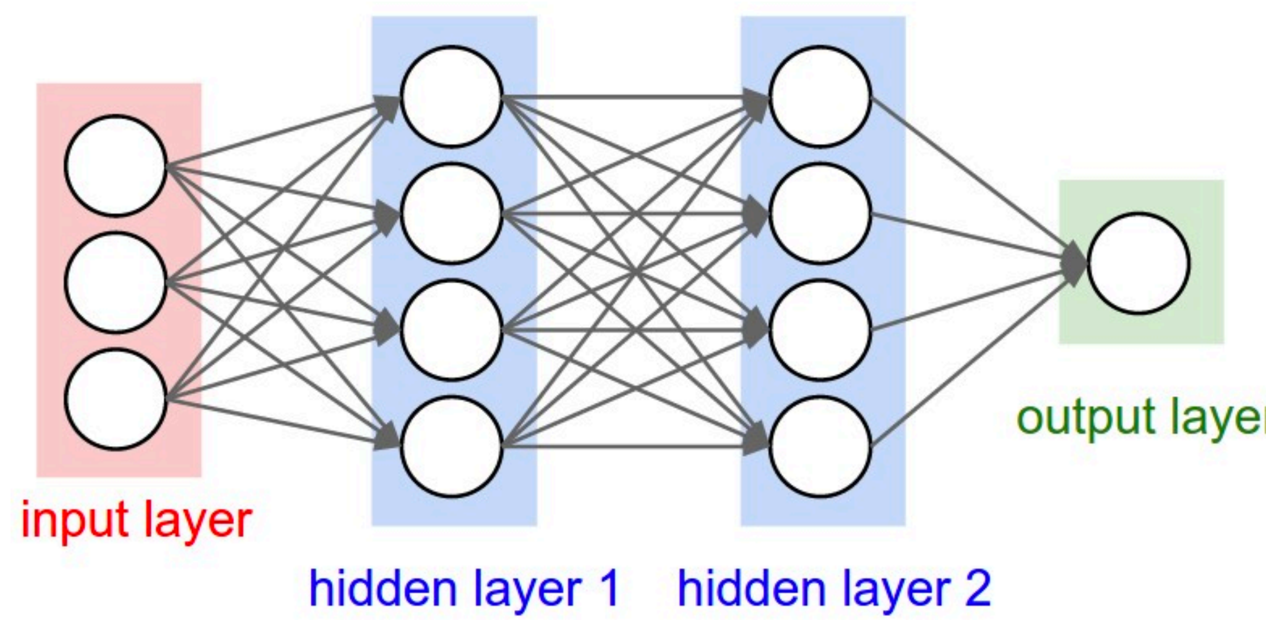
- Collection of neurons that are connected in an acyclic graph
  - Outputs of some neurons become inputs to other neurons
  - Compute non-linear decision boundaries
- Often organized into distinct layers of neurons
- AKA Artificial Neural Networks (ANN) or Multi-Layer Perceptrons (MLP)

Neural Networks: Architectures

2-layer neural network



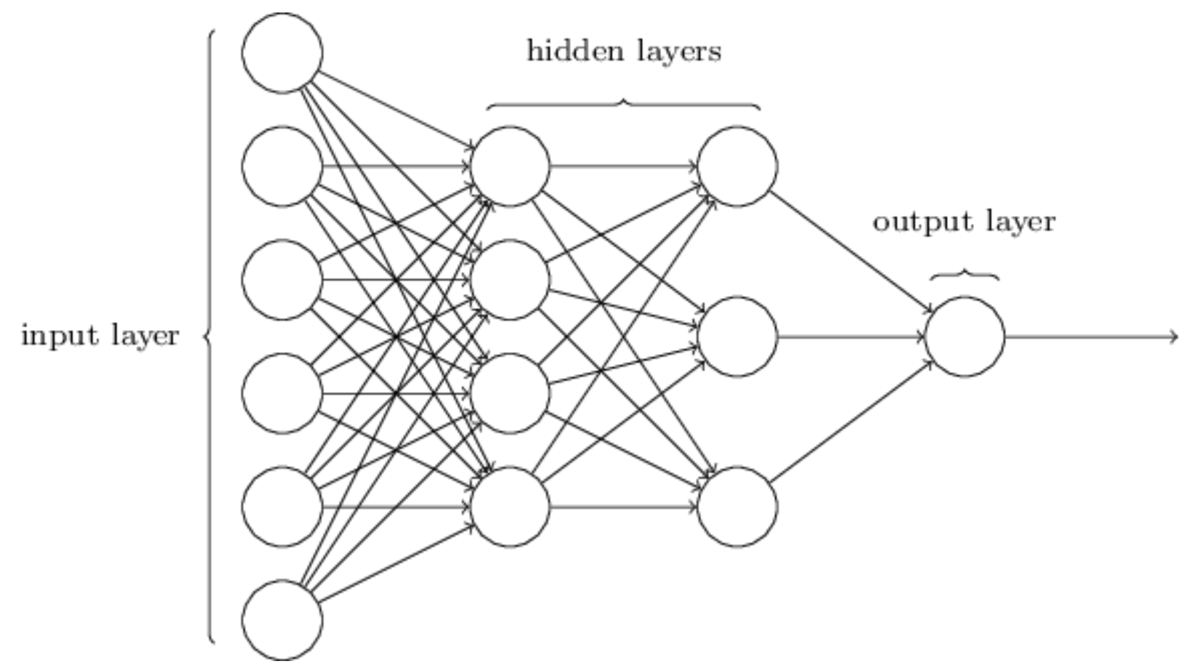
3-layer neural network



Naming convention doesn't count input layer

MLP: Feedforward Neural Network

- Composition of neurons with sigmoid activation function
- Typically, each unit of layer t is connected to every unit of the previous layer t - 1 only
- No cross-connections between units in the same layer

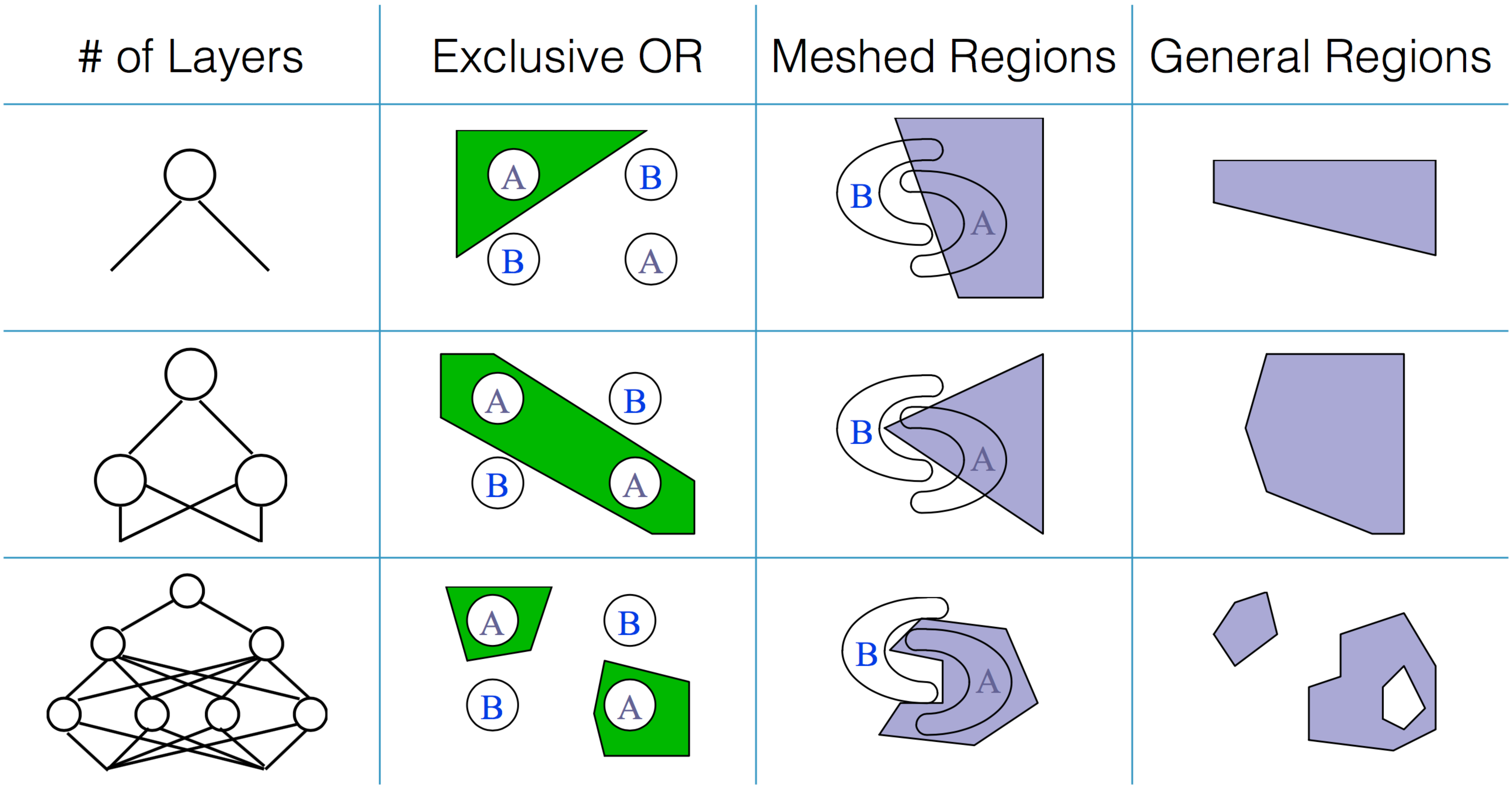


http://neuralnetworksanddeeplearning.com/chap1.html

MLP: Expressiveness

- Single sigmoid neuron has same representational power as a perceptron: Boolean AND, OR, NOT, but not XOR
- Every boolean function can be represented by a network with a single hidden layer, but may require exponential number of hidden units compared to inputs
- Every bounded continuous function can be approximated by a network with one, sufficiently hidden layer
- Any function can be approximated by a network with two hidden layers

MLP: Layer Comparison



MLP: Prediction

- Single forward pass to predict for a new sample
- For each layer
  - Compute the output of all neurons in the layer
  - Copy this output as inputs to the next layer and repeat until at the output layer

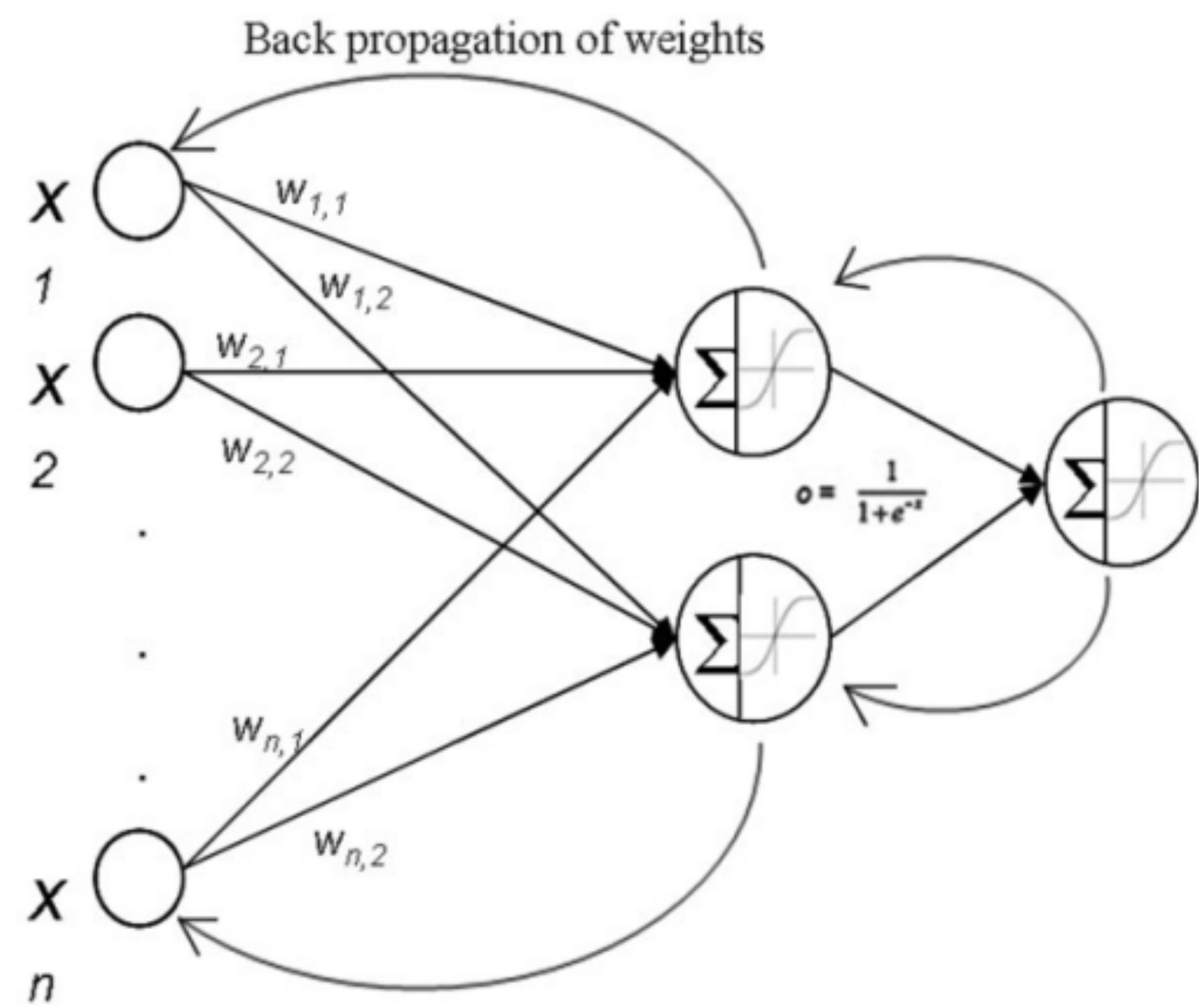
MLP: Learning Weights

- Assume the network structure (units and connections) is given
- Learning problem is finding good set of weights
- Answer: Backpropogation = gradient descent + chain rule

Backpropogation

Backpropogation Algorithm

- Method of training neural network via gradient descent
- Calculate error at output layer for each training example
- Propagate errors backward through the network and update the weights accordingly

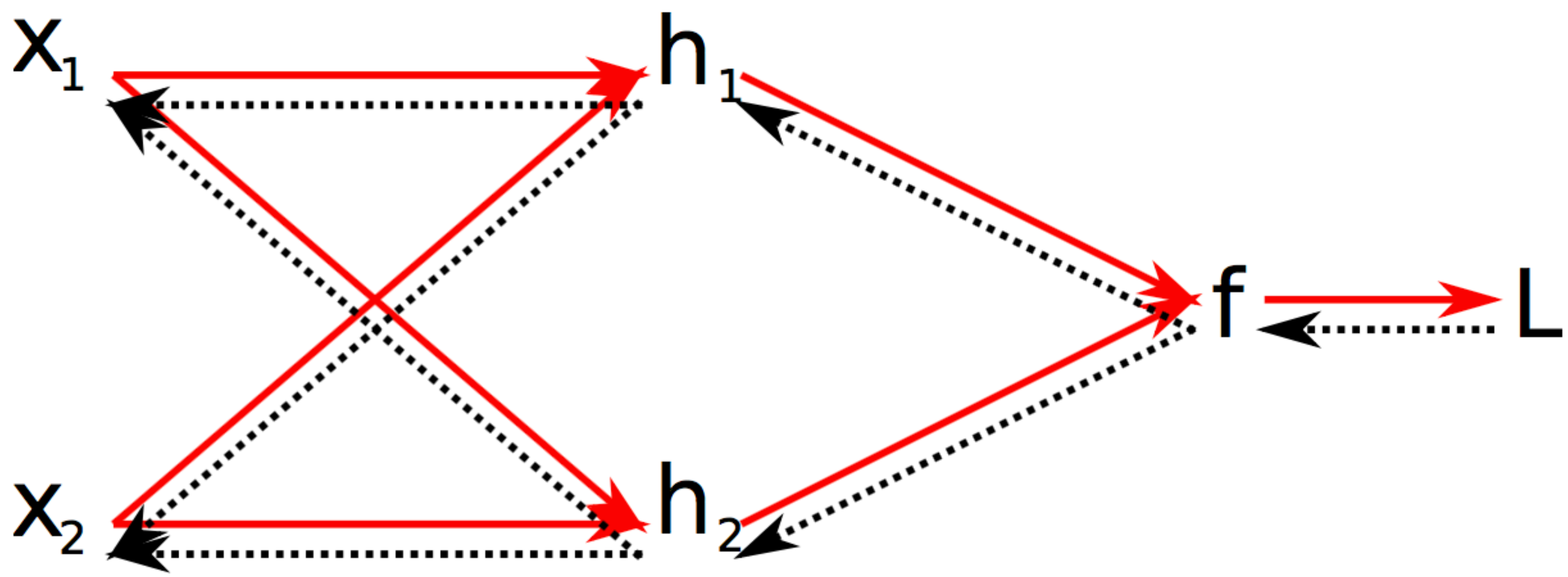


https://openi.nlm.nih.gov/imgs/512/121/2716495/PMC2716495_bcr2257-1.png

Backpropogation Algorithm

- Assume fully connected network (all units in layer k are connected to all units in layer k+1)
  - N input units ($x_1, \ldots, x_N$)
  - One hidden layer with M hidden units ($h_1, \ldots, h_M$)
  - One output unit (f)
- Loss function: squared error

Backpropogation: Forward Pass



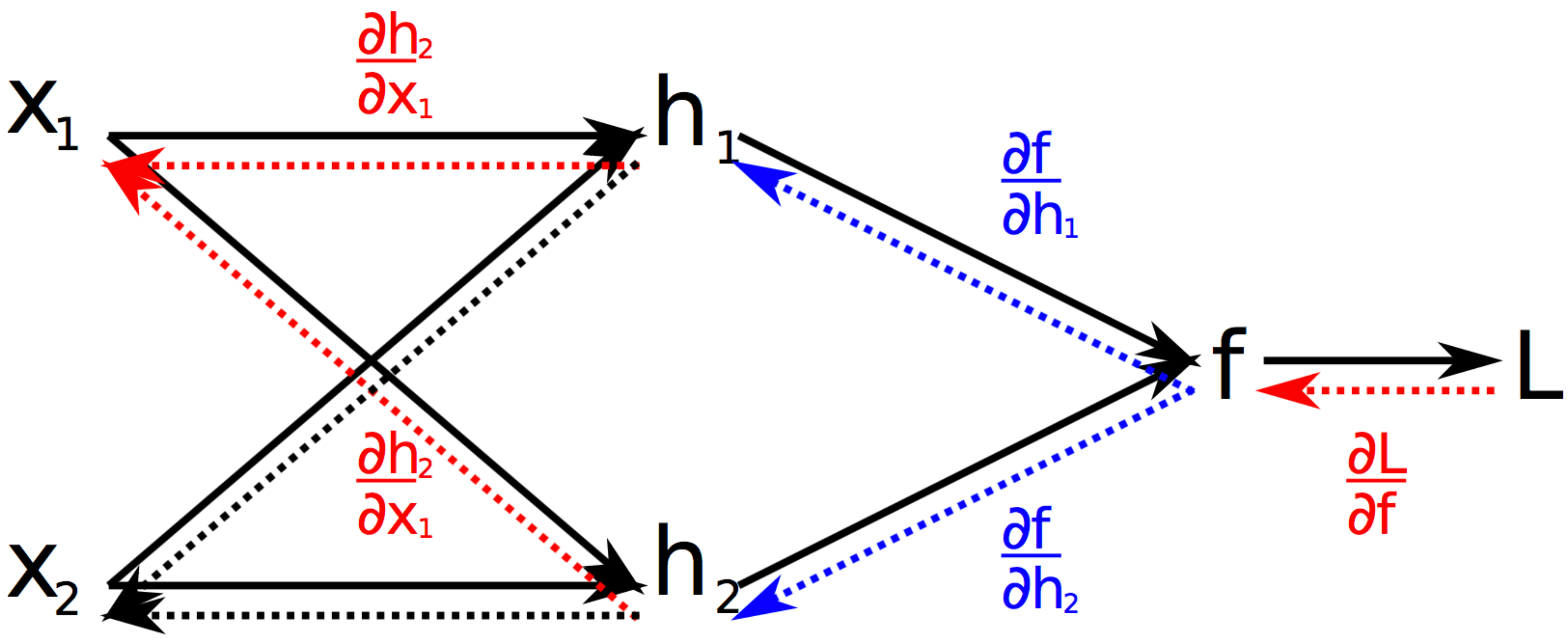
- Forward computation

$$L(f(h_1(\mathbf{x}_1, \cdots, \mathbf{x}_N, \boldsymbol{\theta}_{h_1}), \cdots, h_M(\mathbf{x}_1, \cdots, \mathbf{x}_N, \boldsymbol{\theta}_{h_1}), \boldsymbol{\theta}_f, y)$$

- MLP with single hidden layer

$$L(\mathbf{x}, y, \boldsymbol{\theta}) = \frac{1}{2}(y - \mathbf{U}^\top \phi(\mathbf{W}^\top x))^2$$

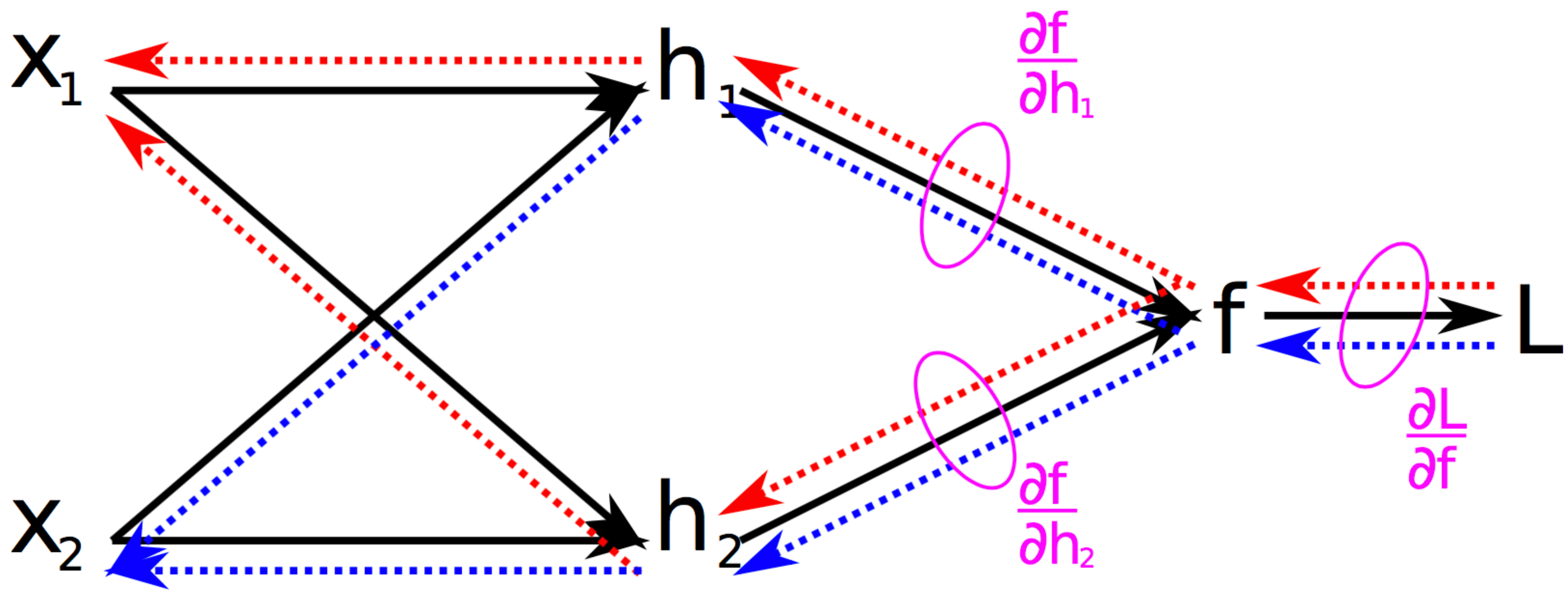
Backpropogation: Chain Rule



Chain rule of derivatives:

$$\frac{\partial L}{\partial x_1} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial x_1} = \frac{\partial L}{\partial f}\left(\frac{\partial f}{\partial h_1}\frac{\partial h_1}{\partial x_1} + \frac{\partial f}{\partial h_2}\frac{\partial h_2}{\partial x_1}\right)$$

Backpropogation: Shared Derivative



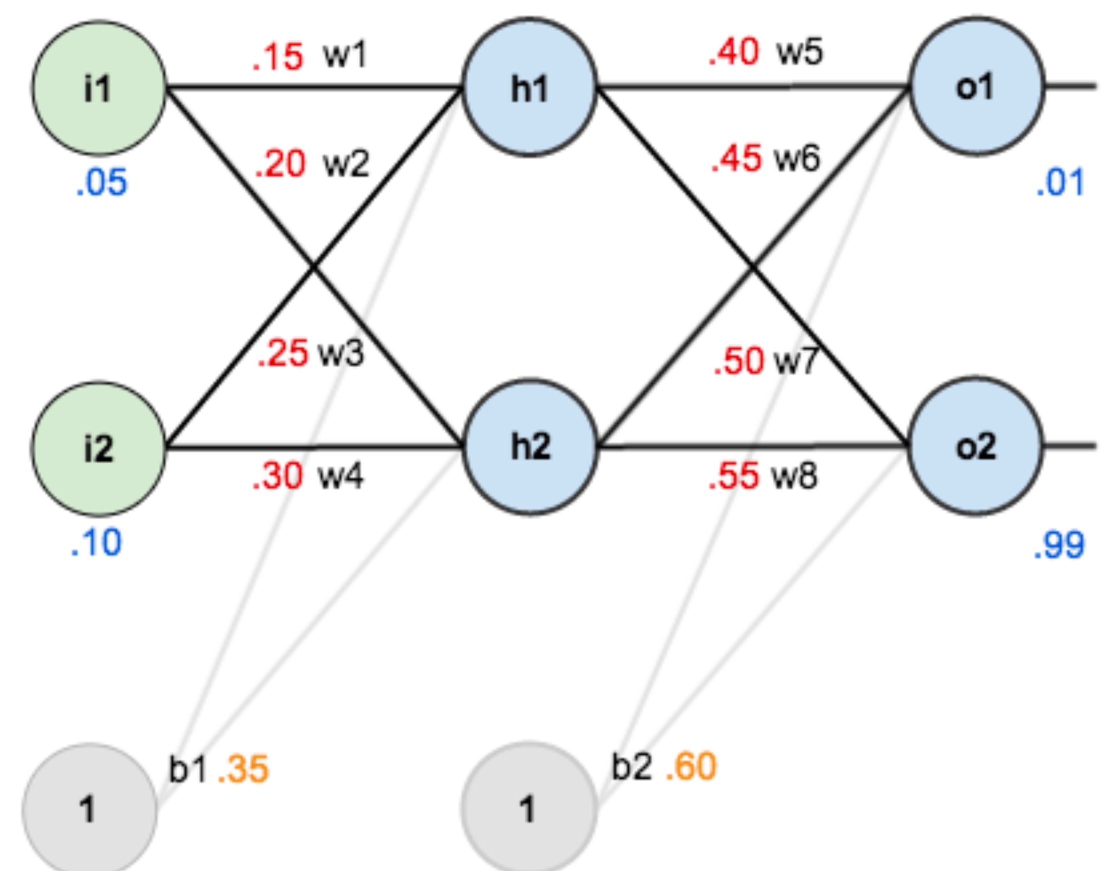
Local derivatives are *shared*:

$$\frac{\partial L}{\partial x_1} = \frac{\partial L}{\partial f}\left(\frac{\partial f}{\partial h_1}\frac{\partial h_1}{\partial x_1} + \frac{\partial f}{\partial h_2}\frac{\partial h_2}{\partial x_1}\right)$$

$$\frac{\partial L}{\partial x_2} = \frac{\partial L}{\partial f}\left(\frac{\partial f}{\partial h_1}\frac{\partial h_1}{\partial x_2} + \frac{\partial f}{\partial h_2}\frac{\partial h_2}{\partial x_2}\right)$$

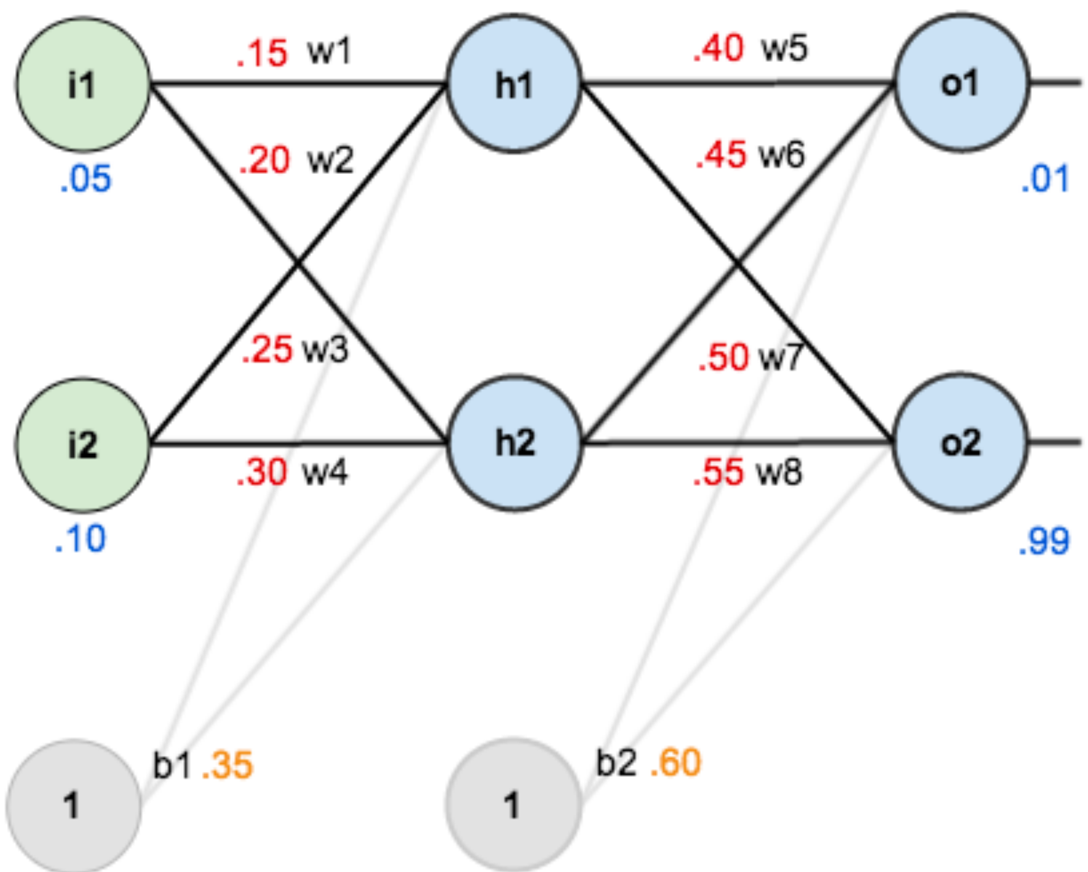
Example: Backpropogation

- Simple neural network with two inputs, two hidden neurons and two output neurons
- Activation function is sigmoid function
- Imagine single training set with inputs (0.05, 0.10) and want output to be 0.01 and 0.09 and want to minimize squared error



https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

Example: Forward Pass



$$h1 = w1 \times i1 + w2 \times i2$$

$$h2 = w3 \times i1 + w4 \times i2$$

$$o1 = 1/(1 + \exp -(w5 \times h1 + w6 \times h2))$$

$$o2 = 1/(1 + \exp -(w7 \times h1 + w8 \times h2))$$

$$e_{\hat{o}1} = \frac{1}{2}(o1 - \hat{o}1)^2 = 0.274811083$$

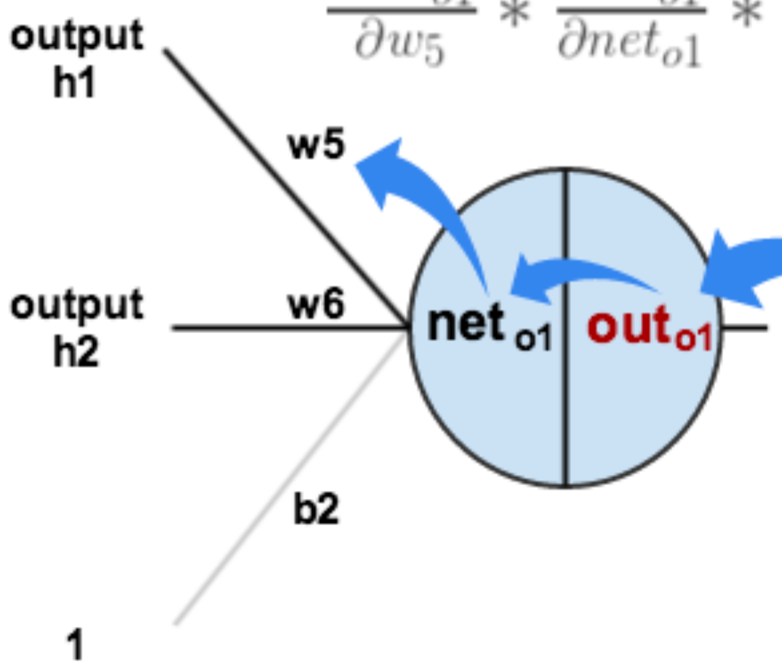
$$e_{\hat{o}2} = 0.023560026$$

$$e_{\text{total}} = e_{\hat{o}1} + e_{\hat{o}2}$$

https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

Example: Backward Pass

$$\frac{\partial net_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{total}}{\partial w_5}$$



$$E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^2$$

$$E_{total} = E_{o1} + E_{o2}$$

$$\text{out}_{o1} = 1/1 + \exp(-\text{net}_{o1})$$

$$\frac{\partial \text{out}_{o1}}{\partial \text{net}_o 1} = \text{out}_{o1}(1 - \text{out}_{o1})$$

$$\text{net}_{o1} = w5 \times \text{out}_{h1} + w6 \times \text{out}_{h2} + b2$$

$$\frac{\partial \text{net}_{o1}}{\partial w5} = \text{out}_{h1} + 0 + 0$$

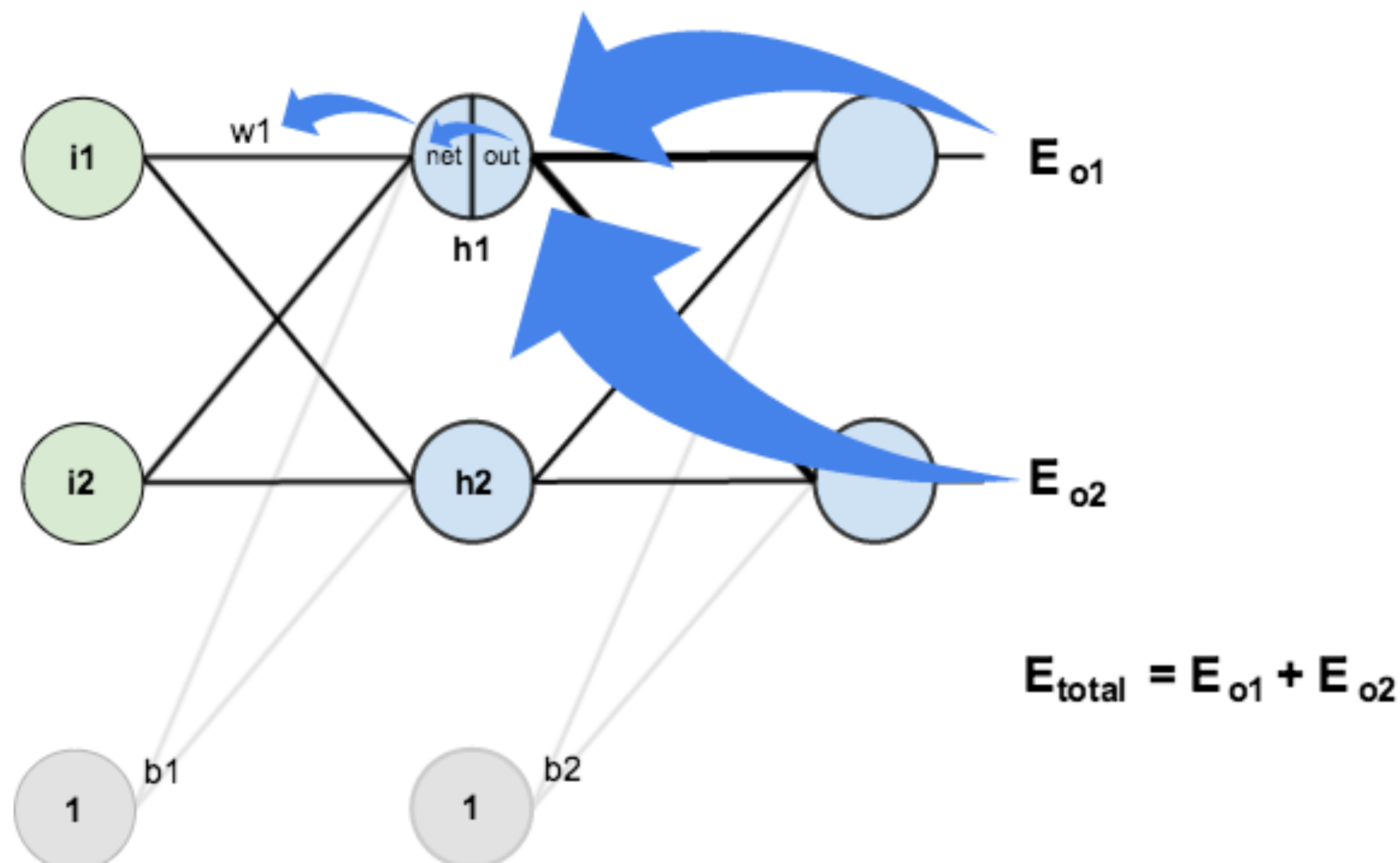
$$w5^{+} = w5 - \eta \frac{\partial e_{\text{total}}}{\partial w5}$$

https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

Example: Backward Pass

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$



https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

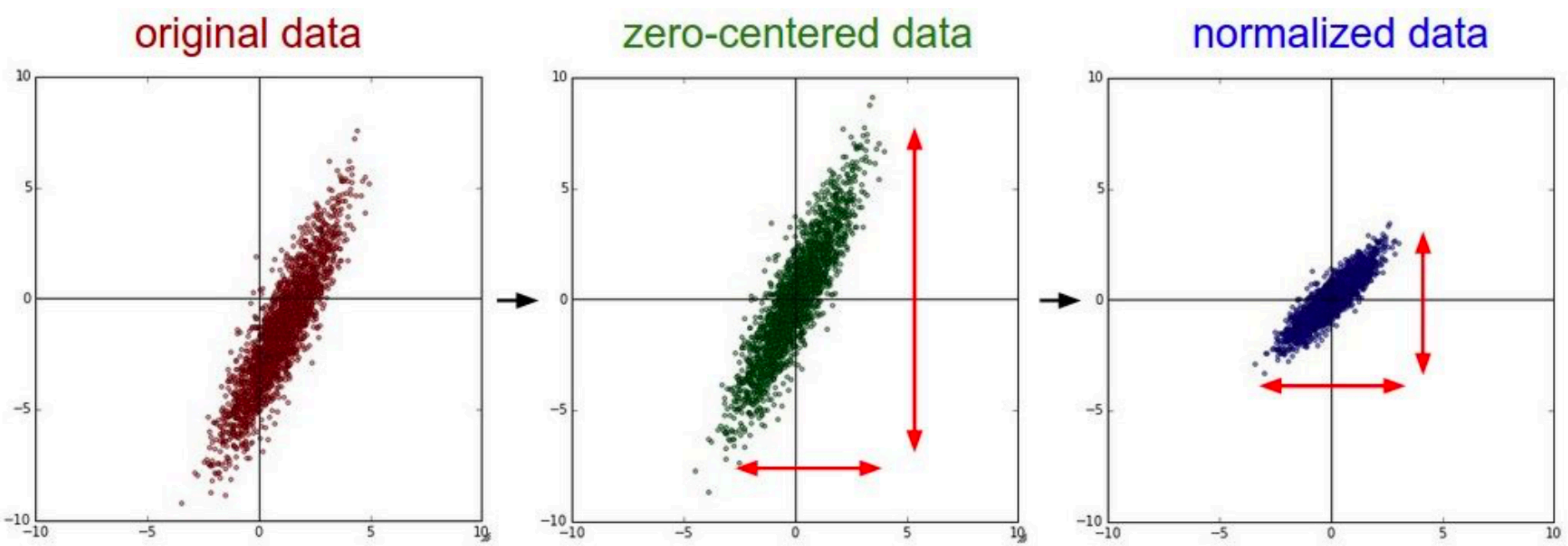
Backpropogation: Practical Considerations

- Do we need to pre-process the training data? If so, how?
- How do we choose the initial weights?
- How do we choose an appropriate learning rate?
- Are some activation functions better than others?

Pre-processing Data

- In principle, can use any raw input-output data
- Pre-process can help learning
  - Rescale continuous features: normalize to zero mean and standard deviation of 1
  - De-correlate data: remove correlated features and transformed data with diagonal covariance matrix
  - Whiten data: convert diagonal covariance matrix to identity matrix so all eigenvalues are the same

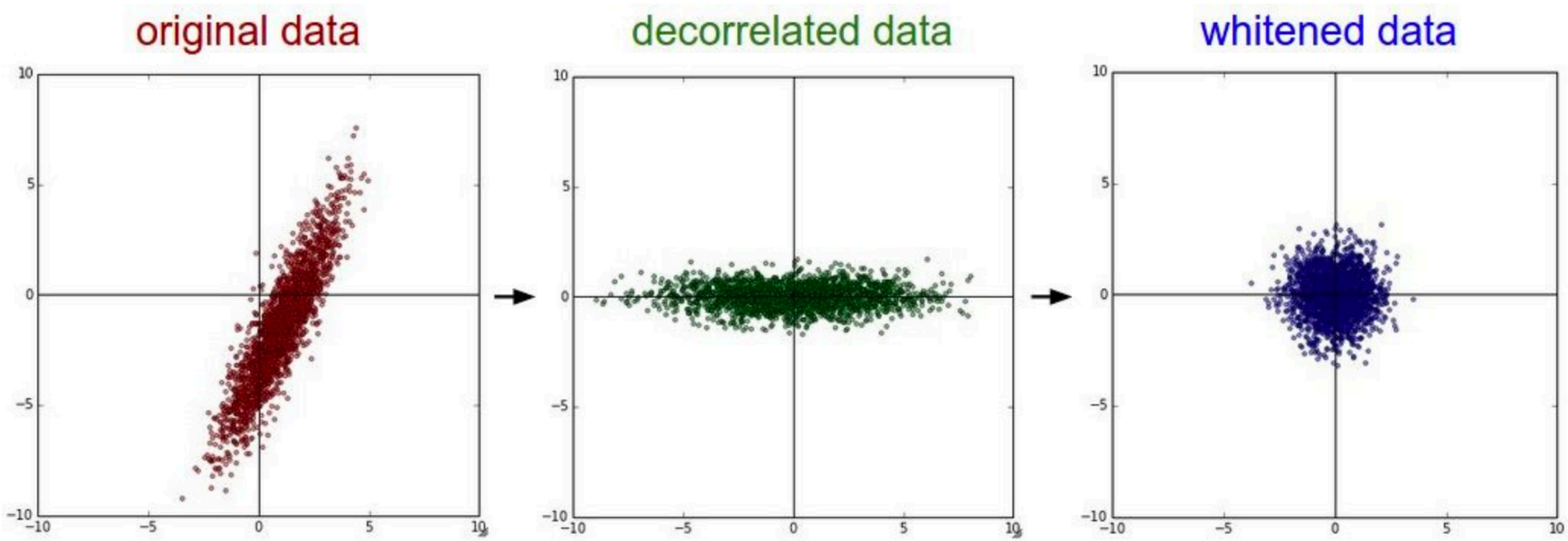
Pre-processing Data




```
X -= np.mean(X, axis = 0)
```

```
X /= np.std(X, axis = 0)
```

Pre-processing Data



(data has diagonal covariance matrix)

(covariance matrix is the identity matrix)

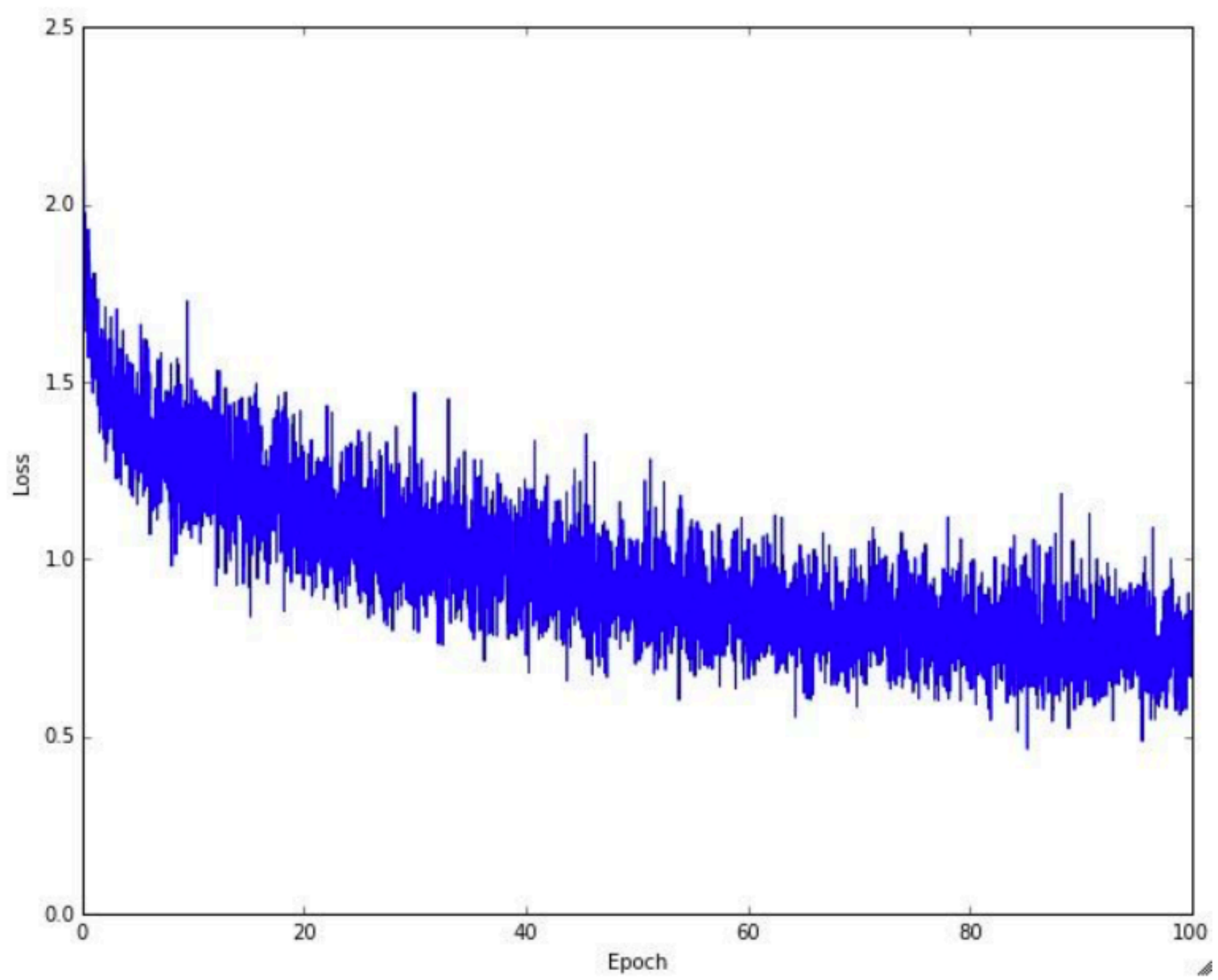
Choosing Initial Weights

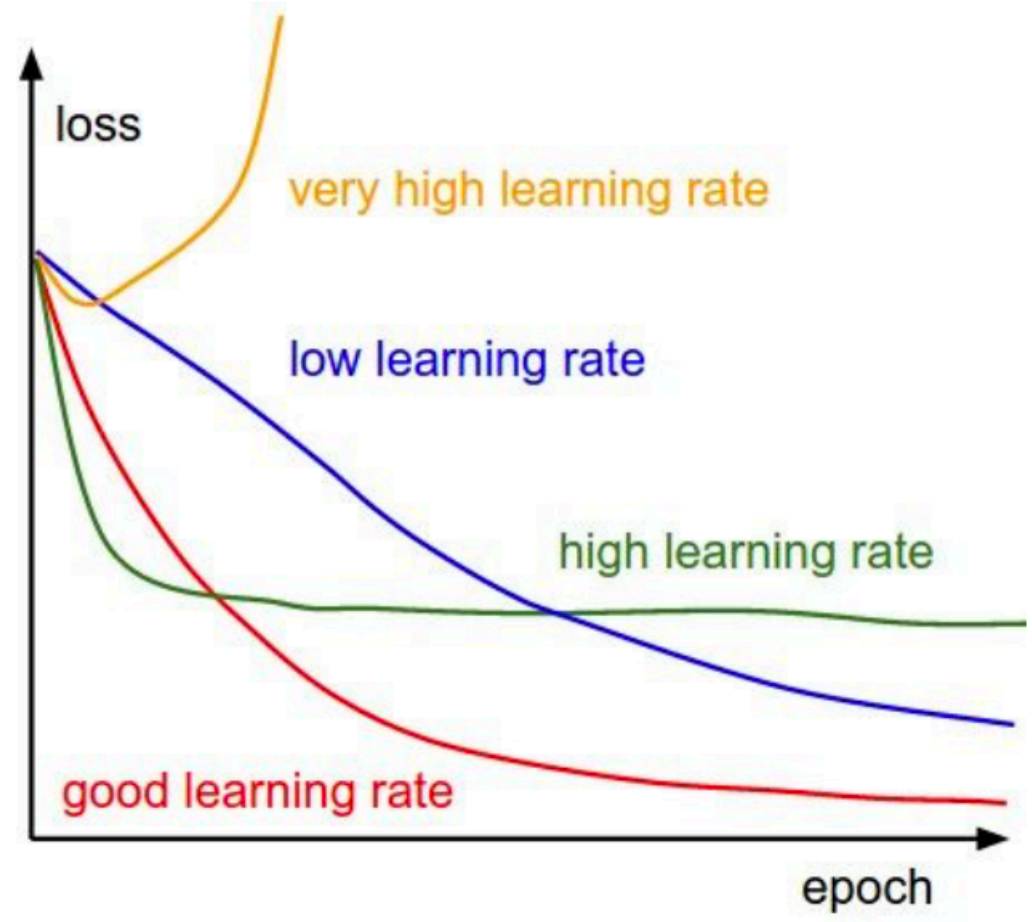
- All weights are treated the same way using gradient descent —> do not initialize with the same values
- Generally start off weights with small random values that do not cause saturation
  - Works okay for small networks
- Proper initialization is an active area of research

Choosing Learning Rate

- If learning rate is too small, it will take a long time to get anywhere near the minimum of the error function
- If learning rate is too large, the weight updates will overshoot the error minimum and weights will oscillate or even diverge
- Solution: Babysit the learning process at the beginning for small portion of training data

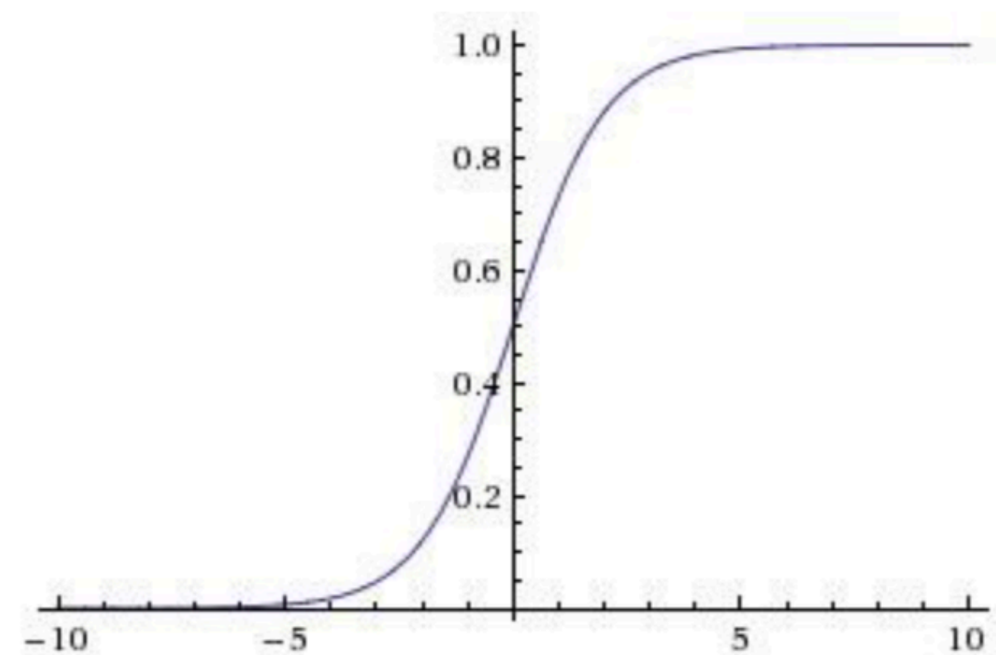
Choosing Learning Rate





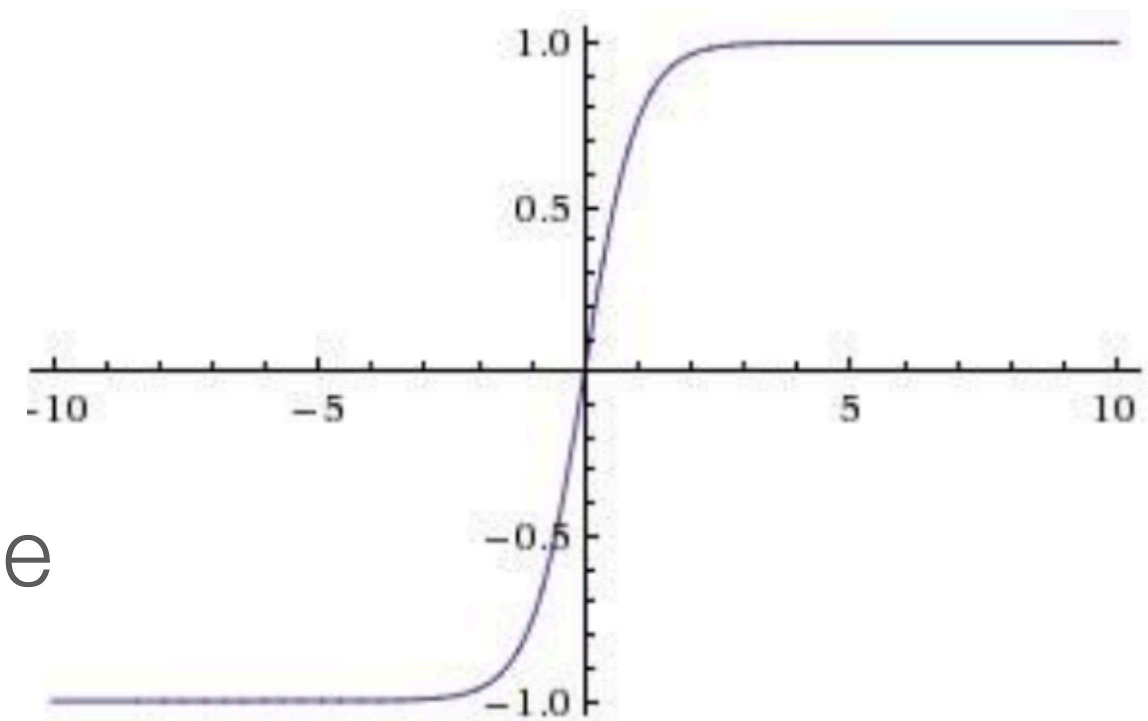
Activation Functions: Sigmoid

- Squashes numbers to [0, 1]
- Popular due to nice interpretation as a saturating "firing rate" of neuron
- (Con) Saturated neurons "kill" the gradients
- (Con) Sigmoid outputs not zero-centered
- (Con) Exponential expensive to compute



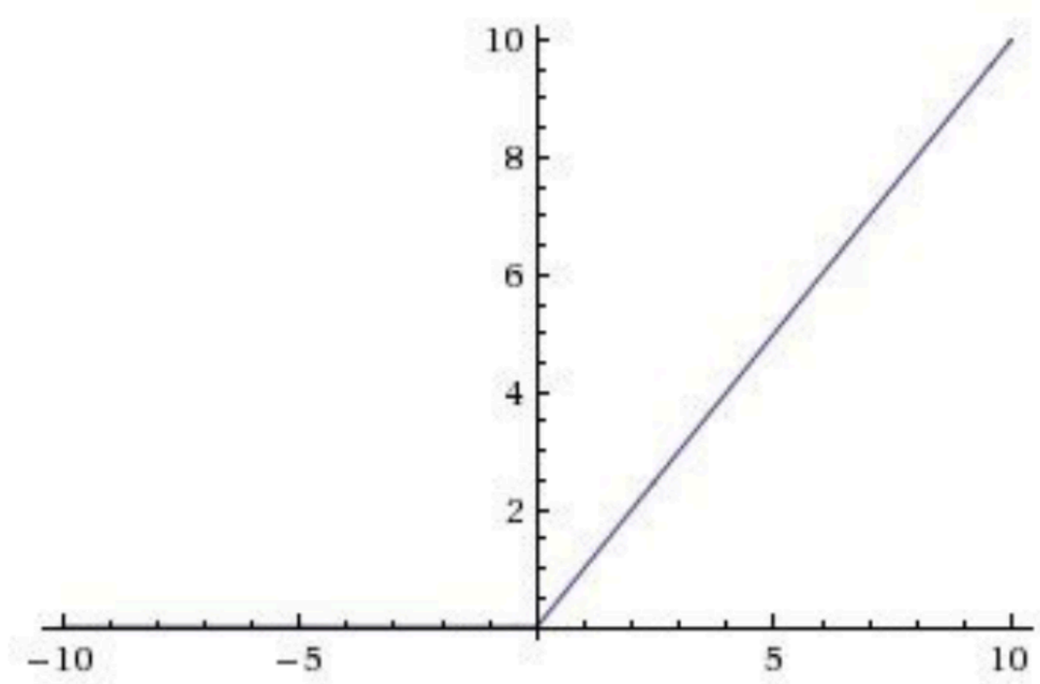
Activation Functions: Tanh

- Squashed numbers to [-1, 1]
- Zero-centered
- (Con) Saturated neurons "kill" the gradients



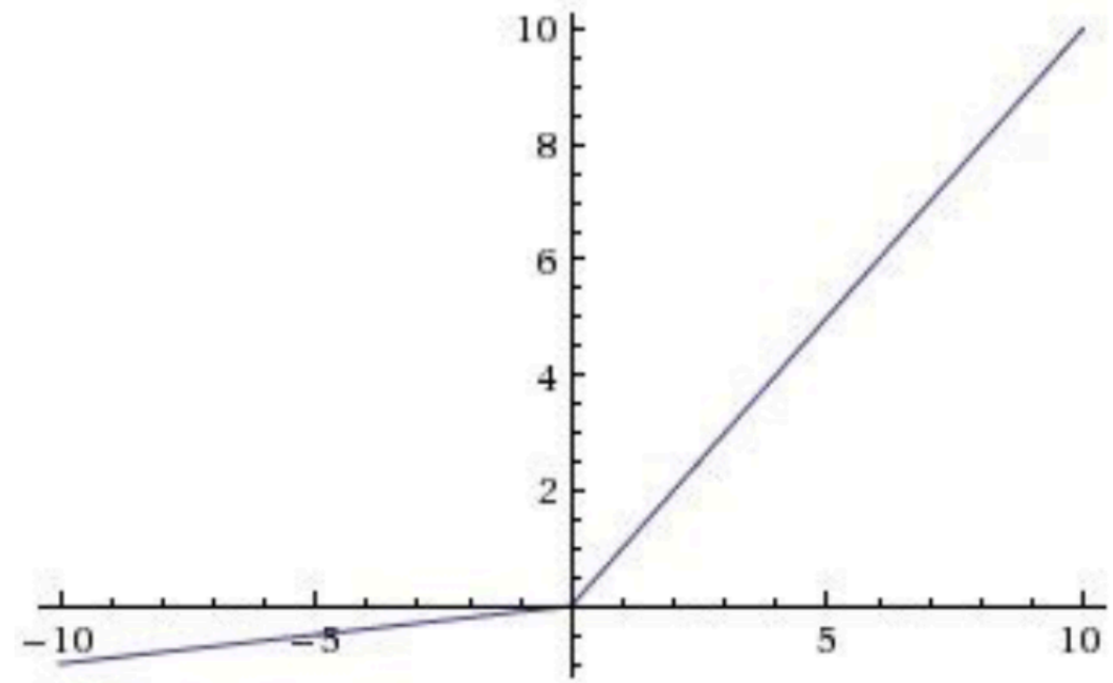
Activation Functions: ReLU

- Does not saturate in positive region
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g., 6x)
- (Con) Not zero-centered
- (Con) What is the gradient for negative region?



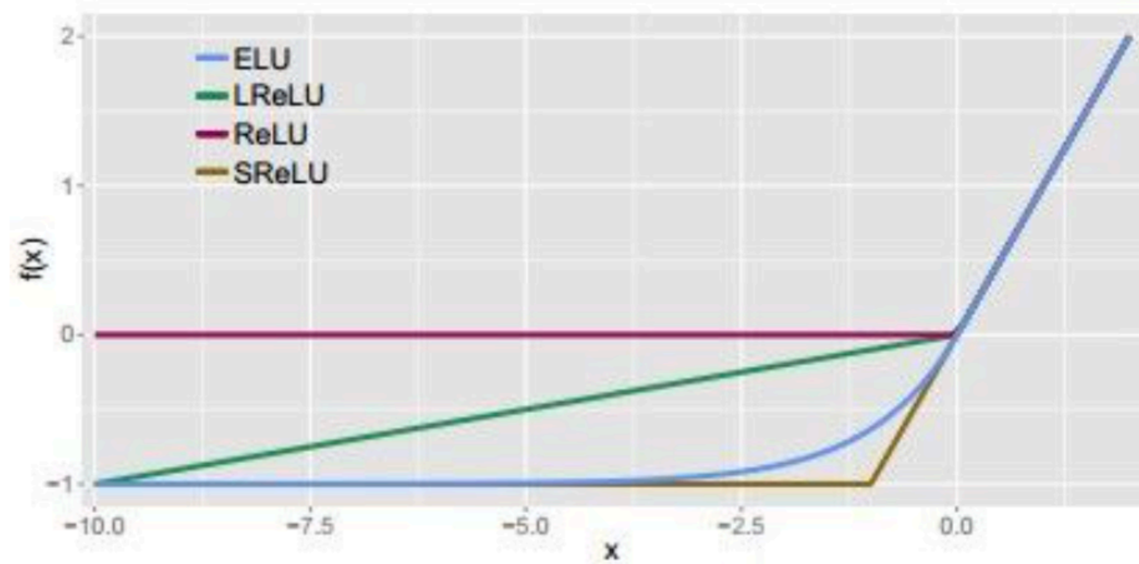
Activation Functions: Leaky ReLU

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh
- Will not "die"



Activation Functions: ELU

- All benefits of ReLU
- Does not die
- Closer to zero-mean outputs
- (Con) Requires exp() computation



$$f(x) = \begin{cases} x & x > 0 \\ \alpha(\exp(x - 1) & x \leq 0 \end{cases}$$

Activation Functions: In Practice

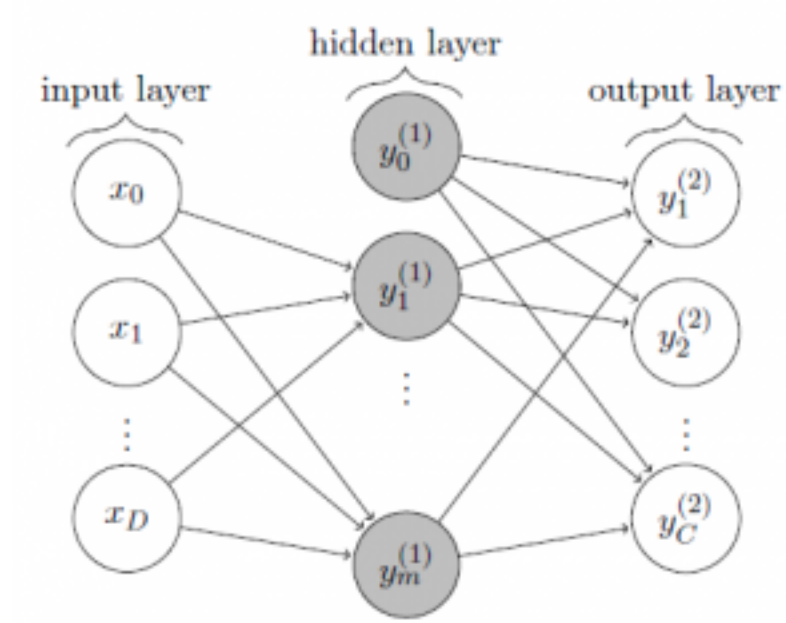
- Use ReLU and be careful with the learning rates
- Try out Leaky ReLU / ELU
- Try out tanh but don’t expect too much
- Don’t use sigmoid

MNIST Dataset

- Scanned 28 x 28 greyscale images of handwritten digits
- Training data
  - 60,000 images
  - 250 people
- Test Data
  - 10,000 images
  - Different 250 people

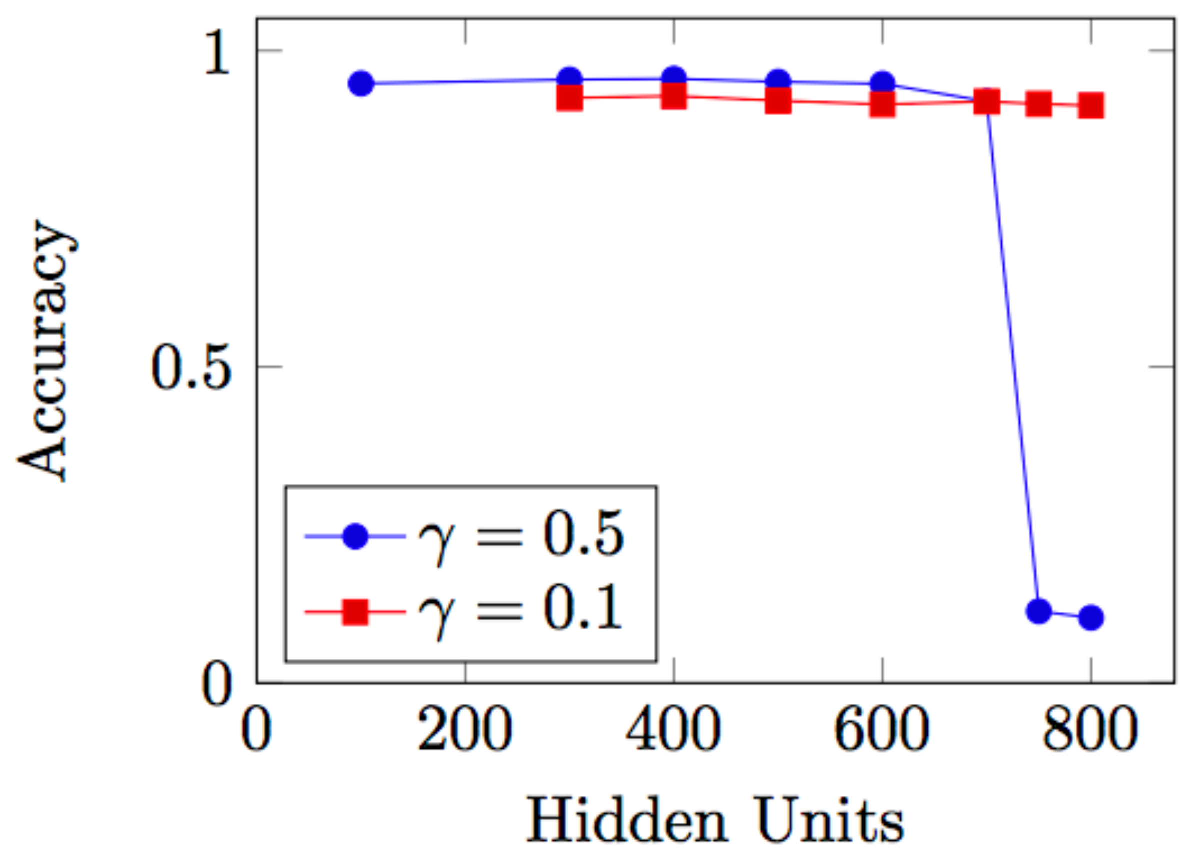
Experiment: 2 Layer Perceptron

- 784 input units, variable number of hidden units, and 10 output units
- Activation function = logistic sigmoid
- Sum of squared error function
- Stochastic variant of mini-batch training

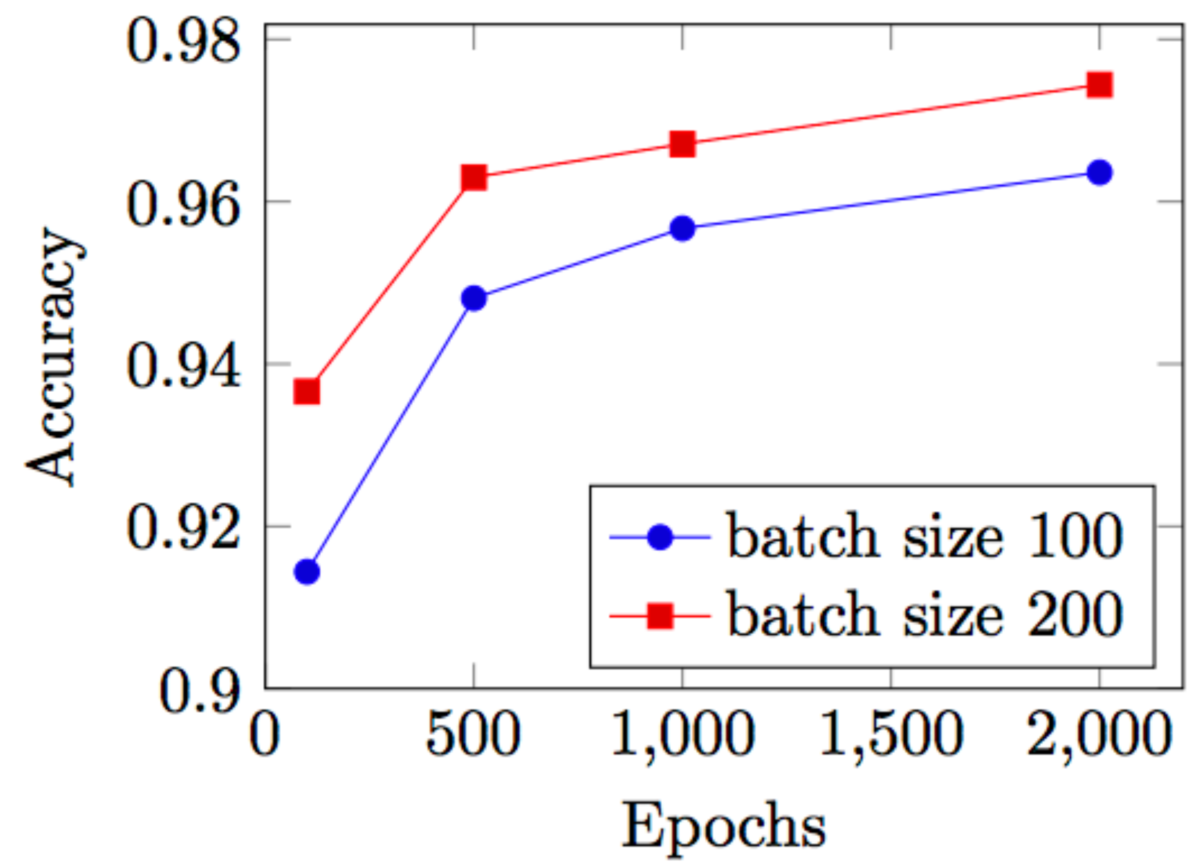


http://davidstutz.de/recognizing-handwritten-digits-mnist-dataset-twolayer-perceptron

Experiment: 2 Layer Perceptron



(a) 500 epochs with batch size 100.

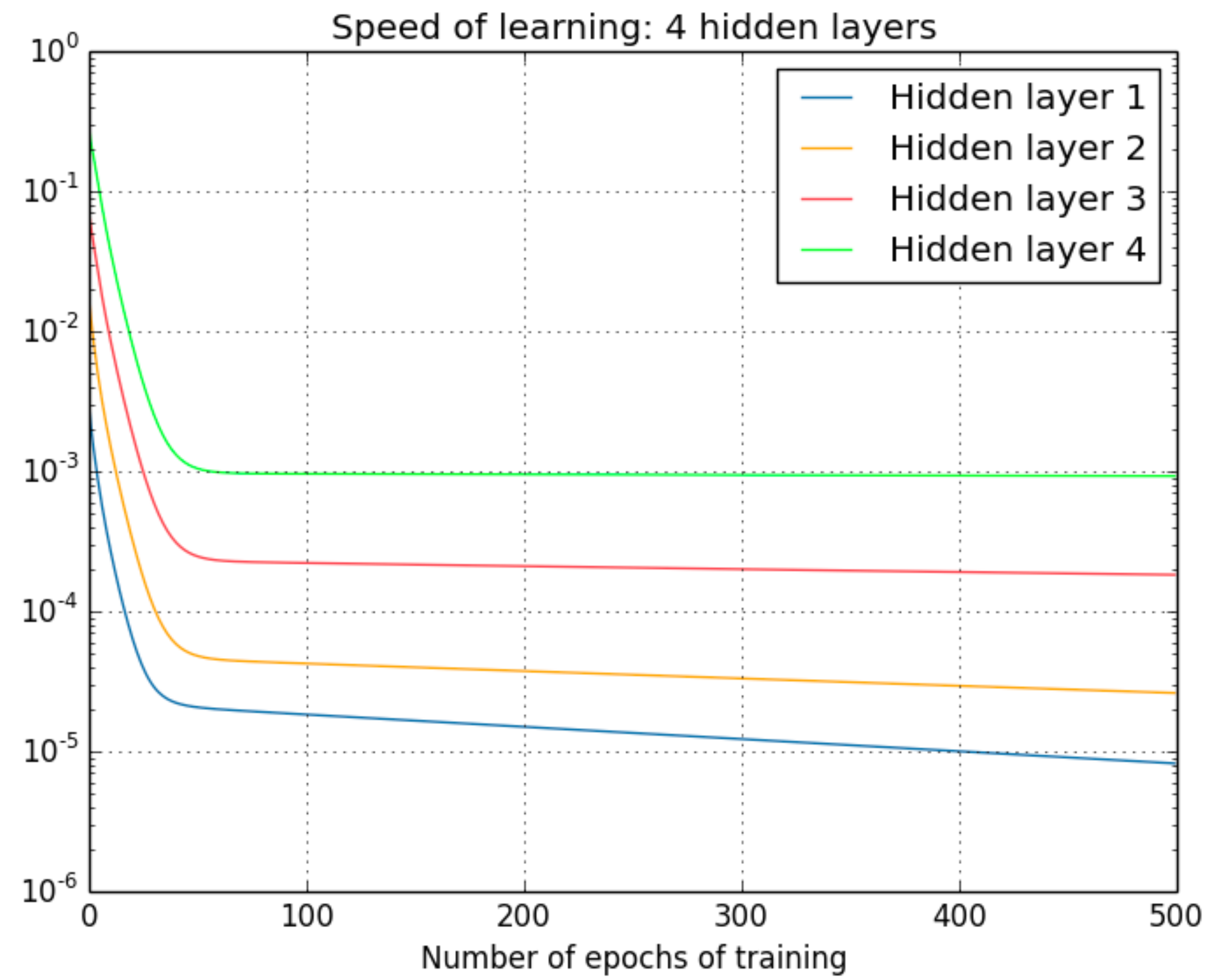


(b) 500 epochs with learning rate $\gamma = 0.5$.

http://davidstutz.de/wordpress/wp-content/uploads/2014/03/seminar.pdf

Obstacles to Deep MLPs

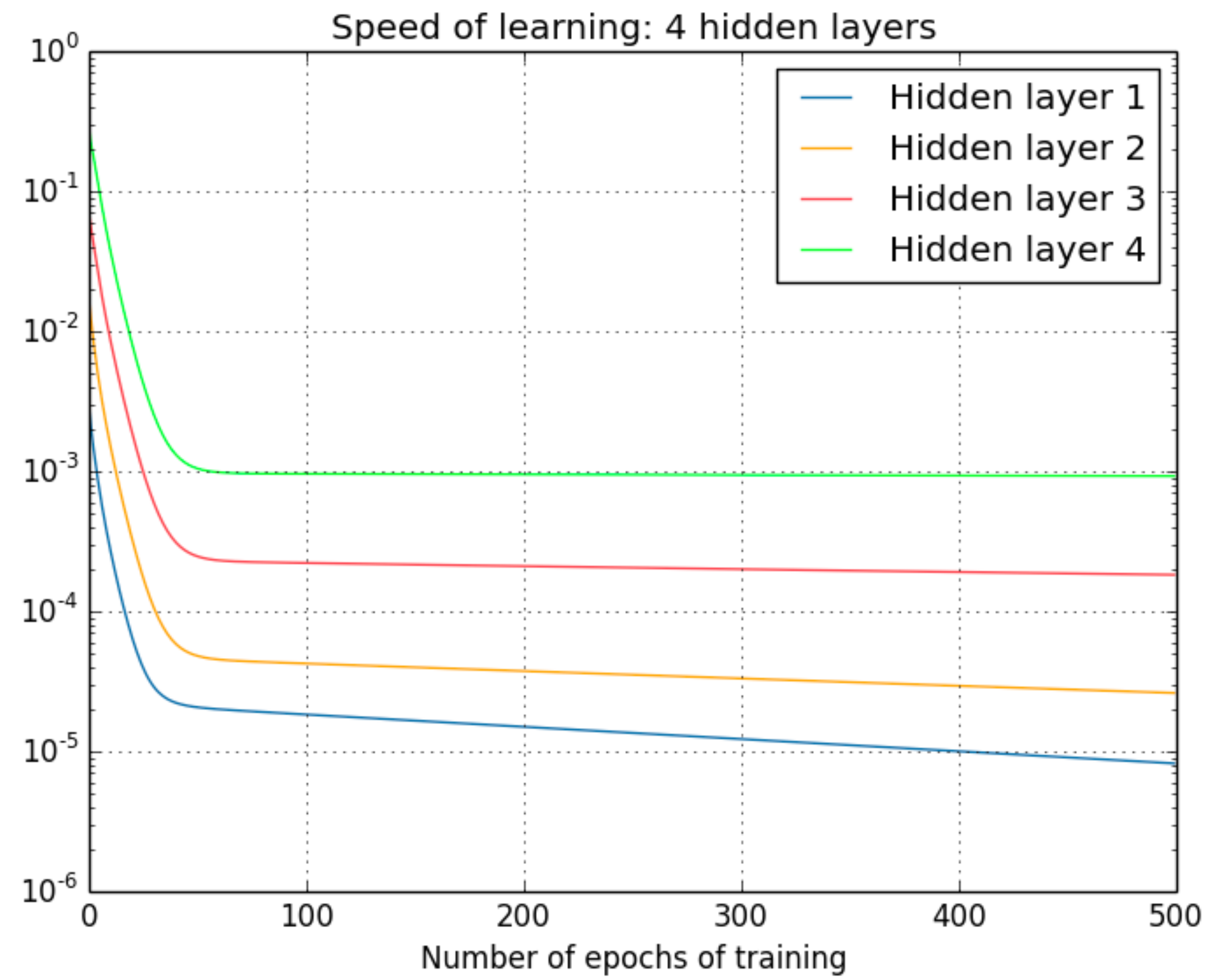
- Requires lots of labeled training data
- Computationally extremely expensive
  - Vanishing & unstable gradients
  - Training can be slow and get stuck in local minimum



http://neuralnetworksanddeeplearning.com/chap5.html

Obstacles to Deep MLPs

- Difficult to tune
  - Choice of architecture (layers + activation function)
  - Learning algorithm
  - Hyperparameters



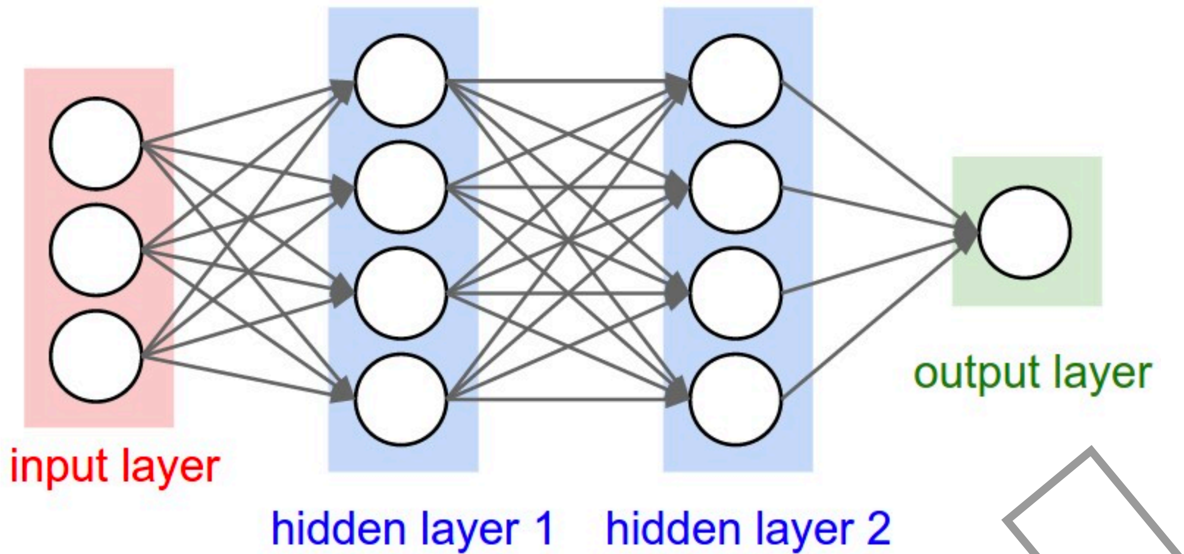
http://neuralnetworksanddeeplearning.com/chap5.html

Convolutional Neural Networks (CNN)

- Specialized neural network for processing known, grid-like topology
  - Powerful model for image, speech recognition
  - LeNet helped propel field of deep learning in 1988
- Use convolution instead of general matrix multiplication in one of its layers

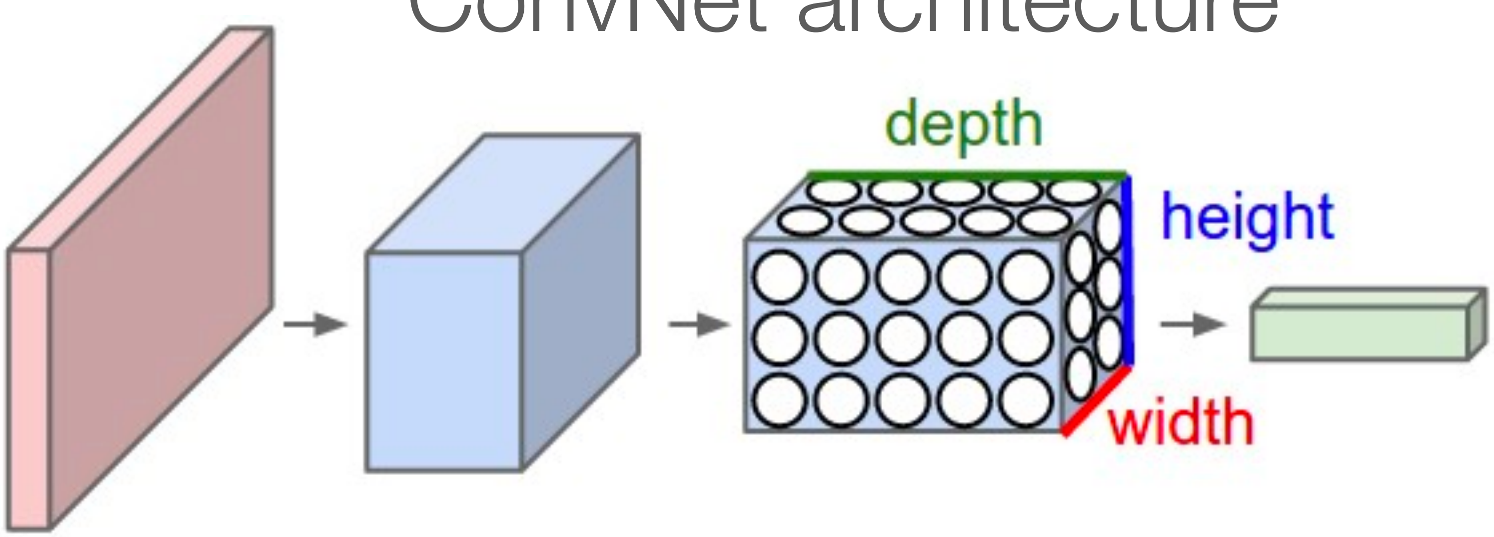
CNN: Comparison with NN

3-layer neural network



Regular NN does not scale well to full images — think about 200 x 200 x 3 = 120,000 weights at first layer

ConvNet architecture

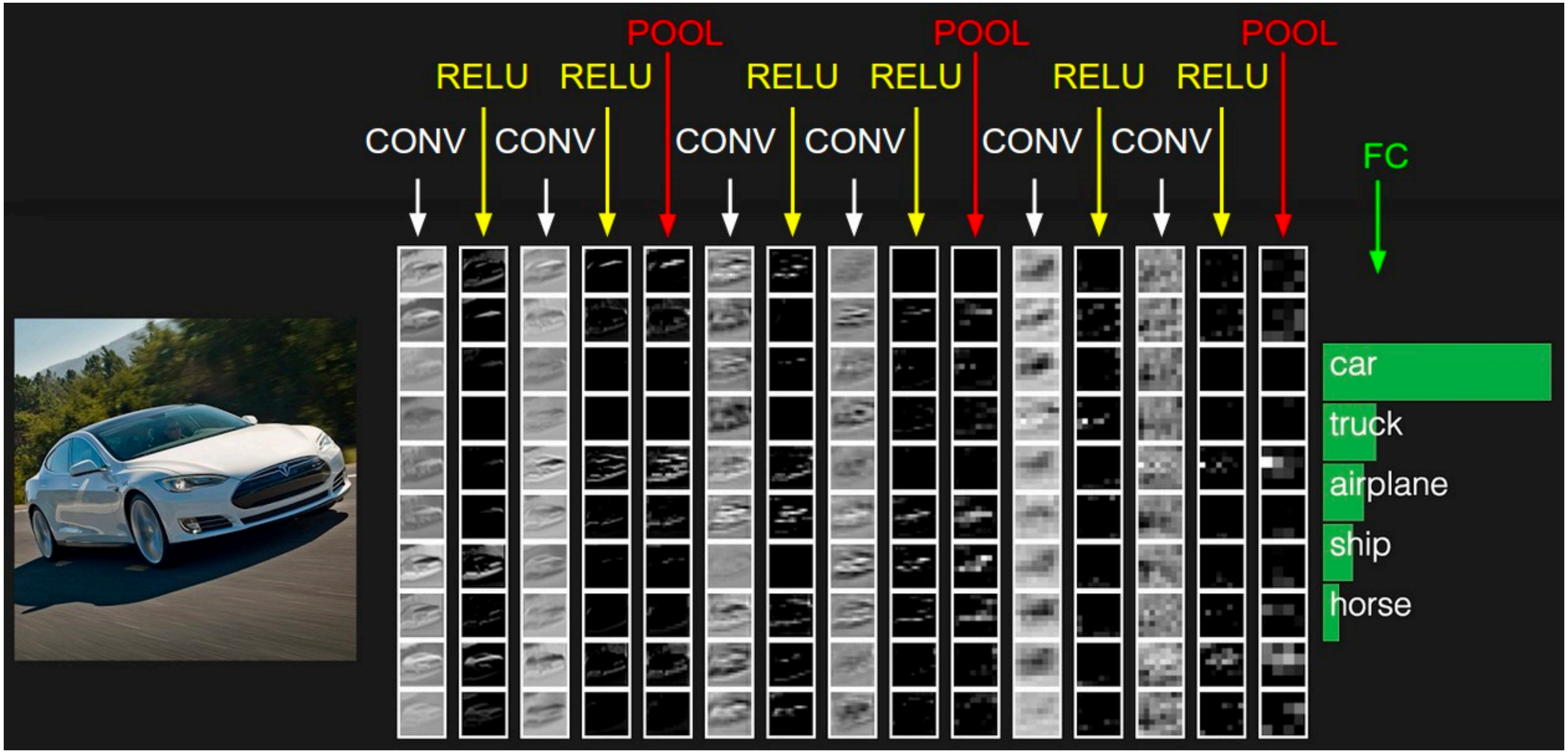


Constrain architecture to look at width, height, depth and avoid fully-connected network

CNN: Four Main Layers

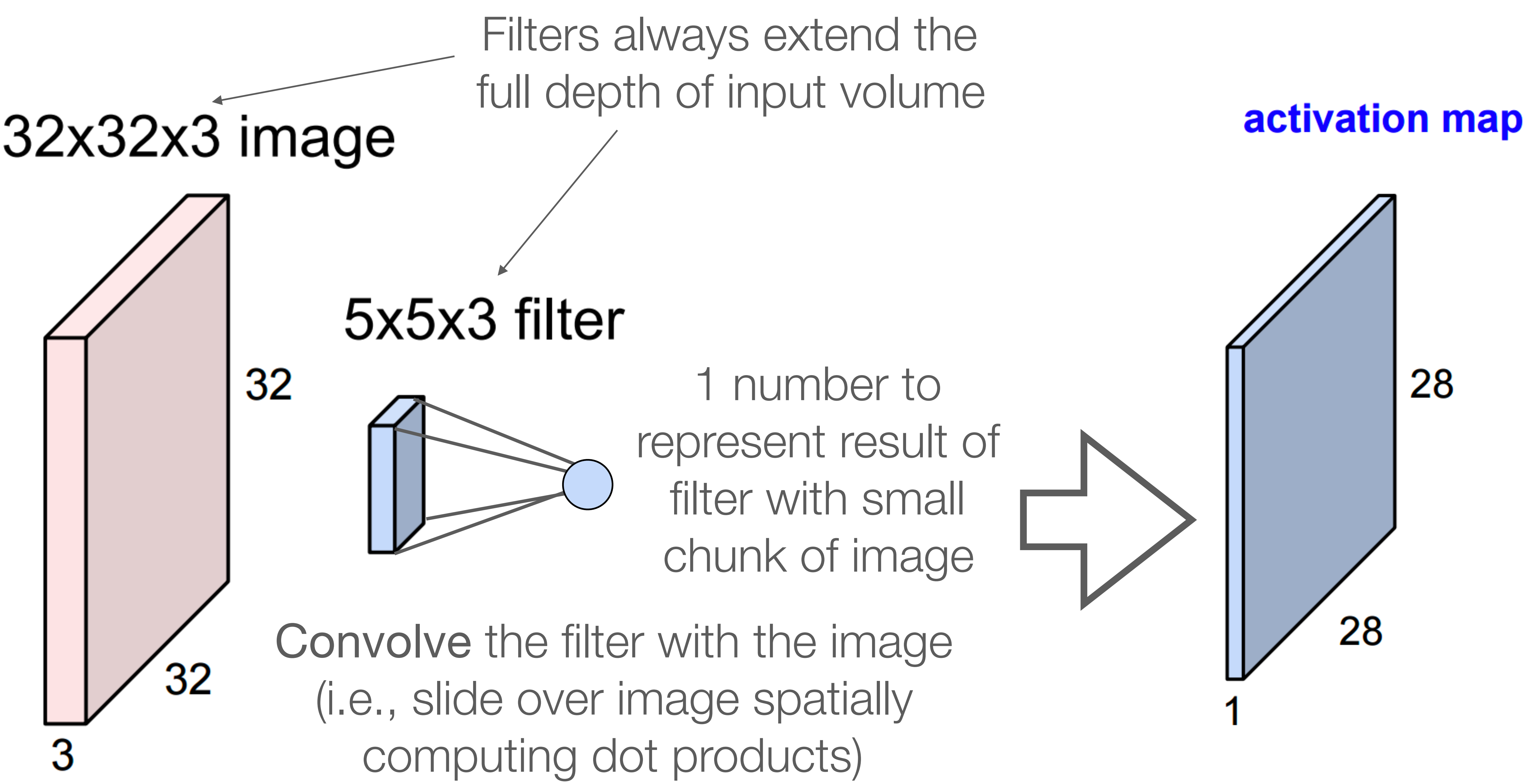
- Convolutional layer — output neurons that are connected to local regions in the input
- ReLU layer — elementwise activation function
- Pooling layer — perform a downsampling operation along the spatial dimensions
- Fully-connected layer — same as regular neural networks

CNN: Example

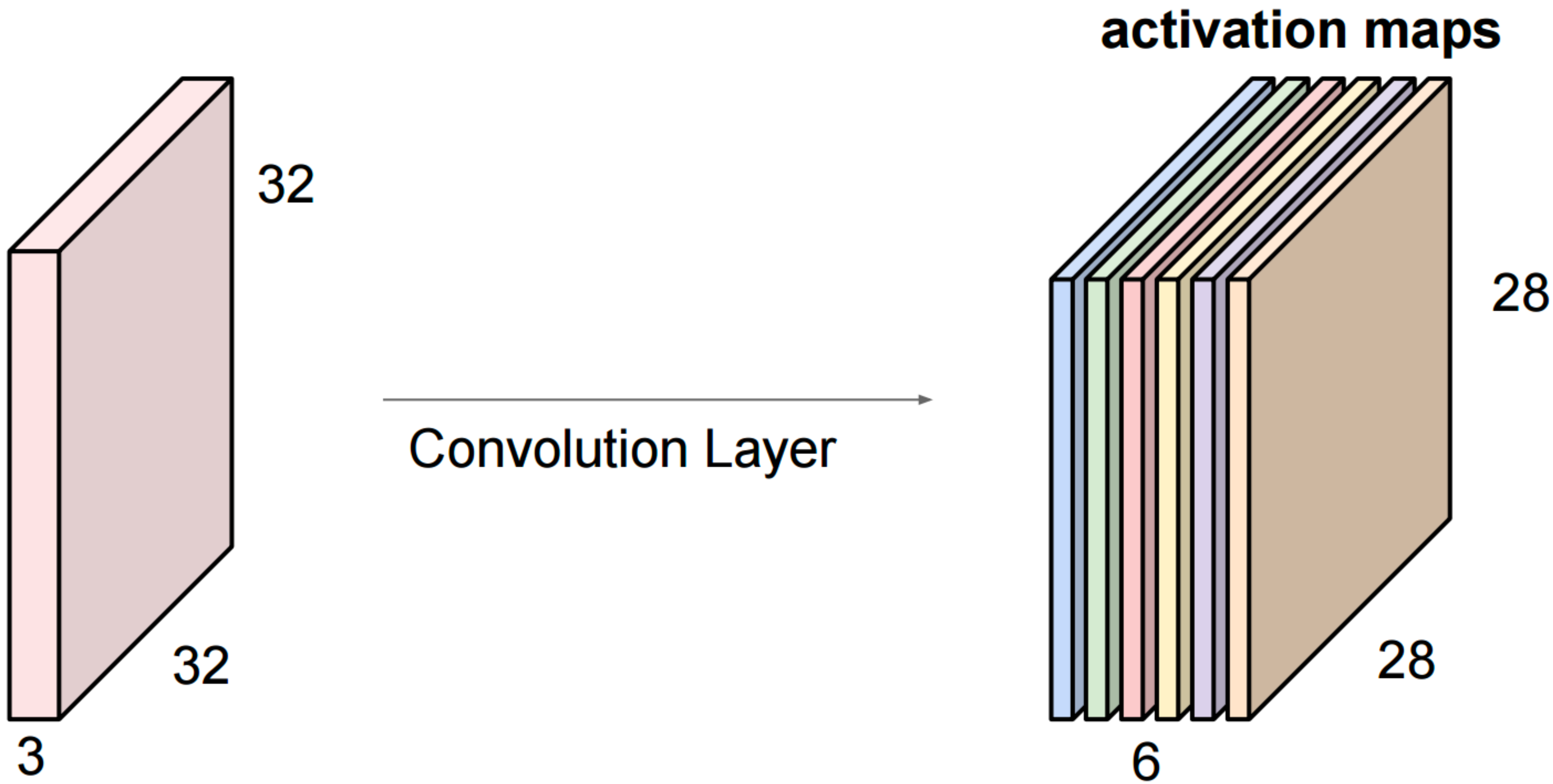


POOL
POOL
POOL
RELU
RELU
RELU
RELU
RELU
RELU
CONV
CONV
CONV
CONV
CONV
CONV
FC
car
truck
airplane
ship
horse

CNN: Convolution Layer



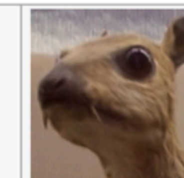
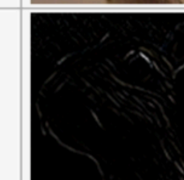
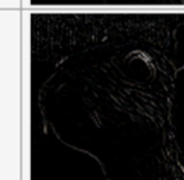
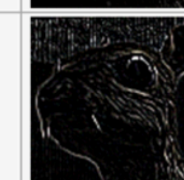
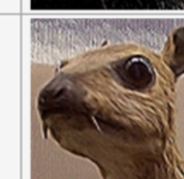
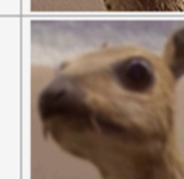
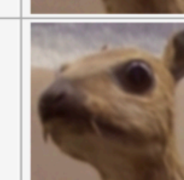
CNN: Convolution Layer



Stacking up multiple filters yields “new image”

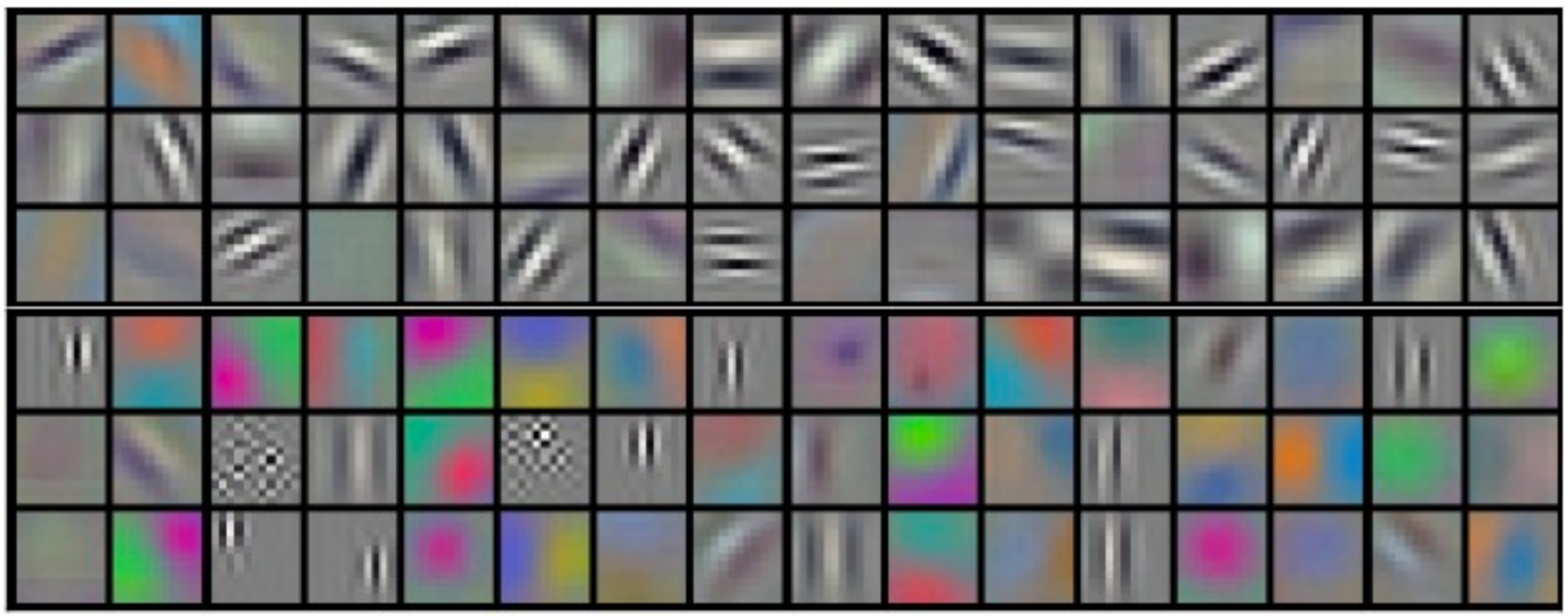
CNN: Convolution Filters

- Filters act as feature detectors from original image
- Network will learn filters that active when they see some type of visual feature (e.g., edge of some orientation, blotch of some color, etc.)
- Only need to learn the weights of the filters

Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

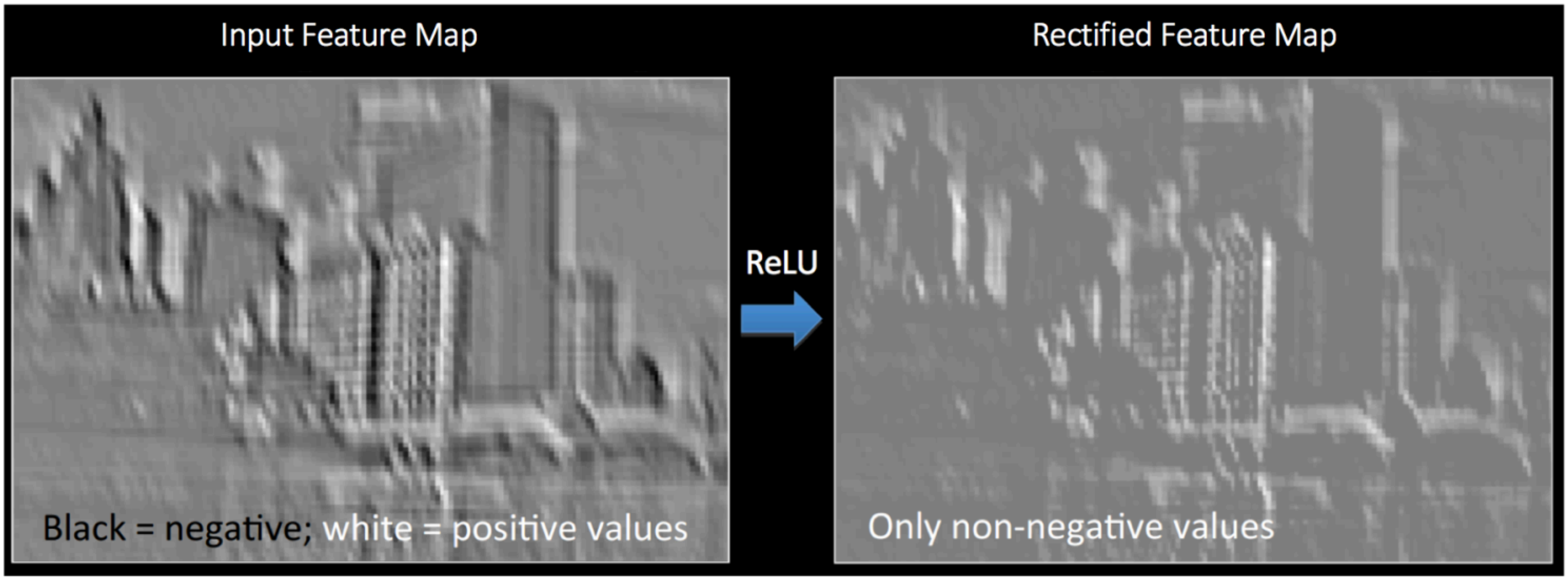
https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/

CNN: Example Filters



https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/

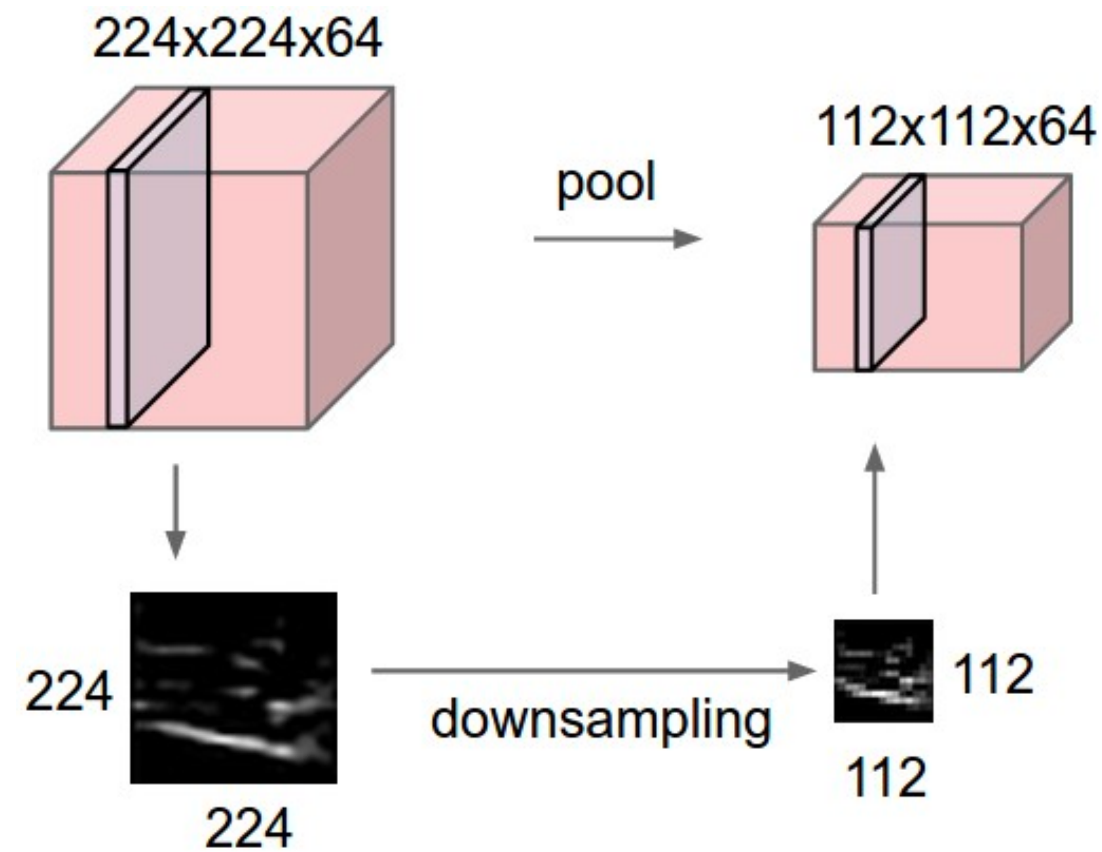
CNN: ReLU



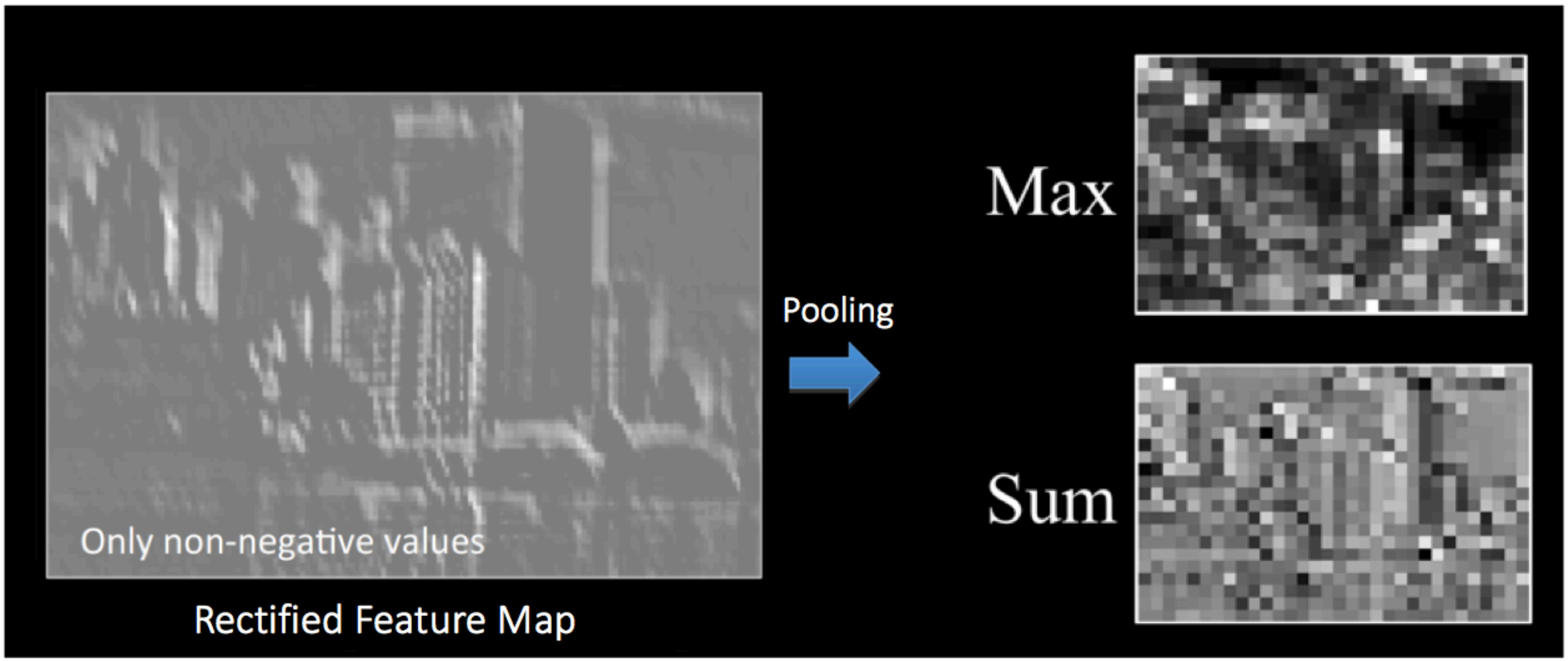
Used after every convolution operation and introduces non-linearity

CNN: Pooling Layer

- Make representations smaller and more manageable
- Helps control overfitting
- Operates over each activation map independently
- Common use of max pooling (take max of spatial neighborhood)



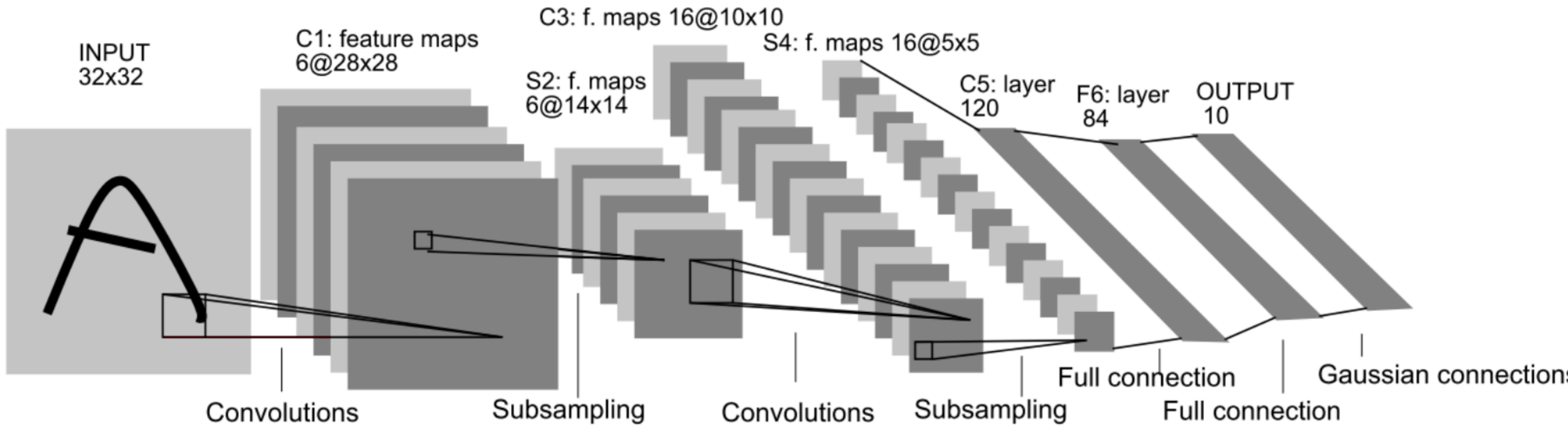
CNN: Pooling Example



CNN: Fully-Connected Layer

- Traditional MLP using softmax activation function
  - Generalization of logistic function to multi-class problem
  - Output probabilities for each class that sum to 1
- Output of convolutional and pooling layers represent high-level features

LeNet 5 [LeCun et al., 1998]



- 32 x 32 pixel with largest character 20 x 20
- Black and white pixel values are normalized to get mean of 0, standard deviation of 1
- Output layer uses 10 RBF (radial basis activation function), one for each digit

CNN: MNIST Dataset Results

- Original dataset (60,000 images)
  - Test error = 0.95%
- Distorted dataset (540,000 artificial distortions + 60,000 images)
  - Test error = 0.8%

Misclassified examples

Why is CNN Successful?

Compared to standard feedforward neural networks with similarly-sized (5-7) layers

- CNNS have much fewer connections and parameters —> easier to train
- Traditional fully-connected neural network is almost impossible to train when initialized randomly
- Theoretically-best performance is likely only slightly worse than vanilla neural networks

Neural Networks: When to Consider

- Noisy data
- Training time is unimportant
- Form of target function is unknown or very complex
- Human readability of results is unimportant