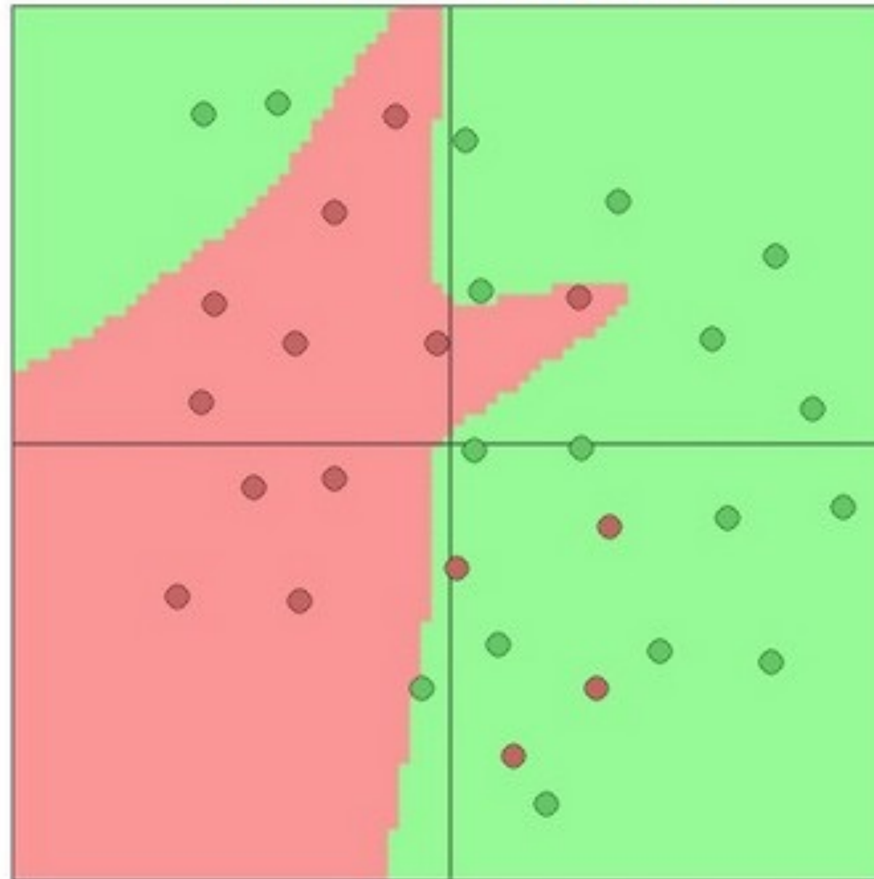
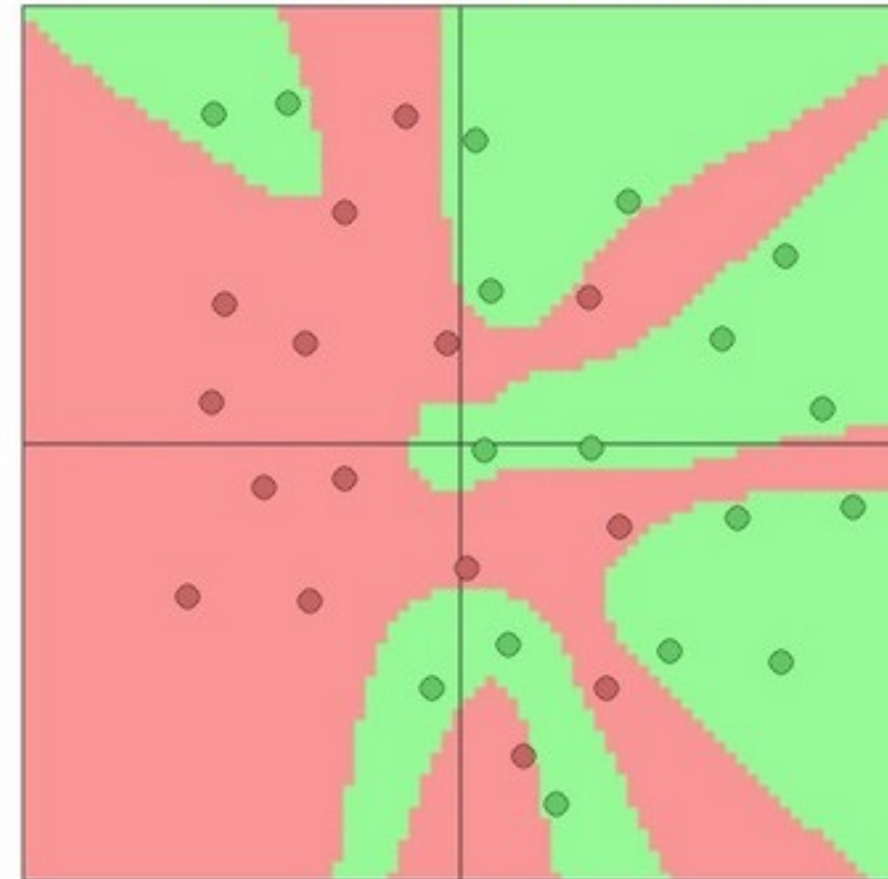
# Neural networks



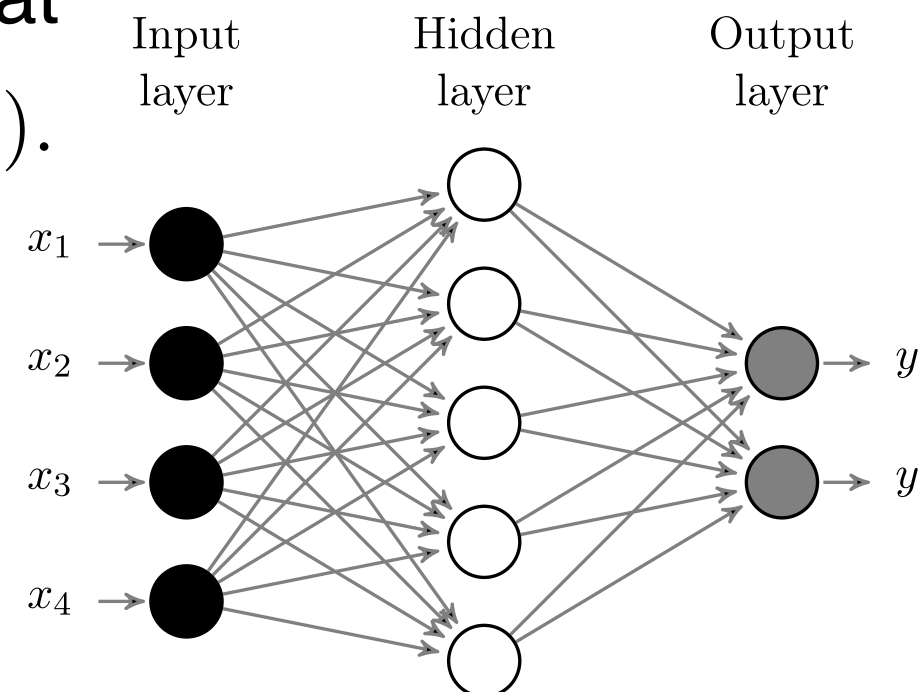
# Example: logistic regression and using a neural network

- The goal is still to predict p(y = 1 | x)
  - But now want this to be a more general nonlinear function of x
- Logistic regression learns W such that

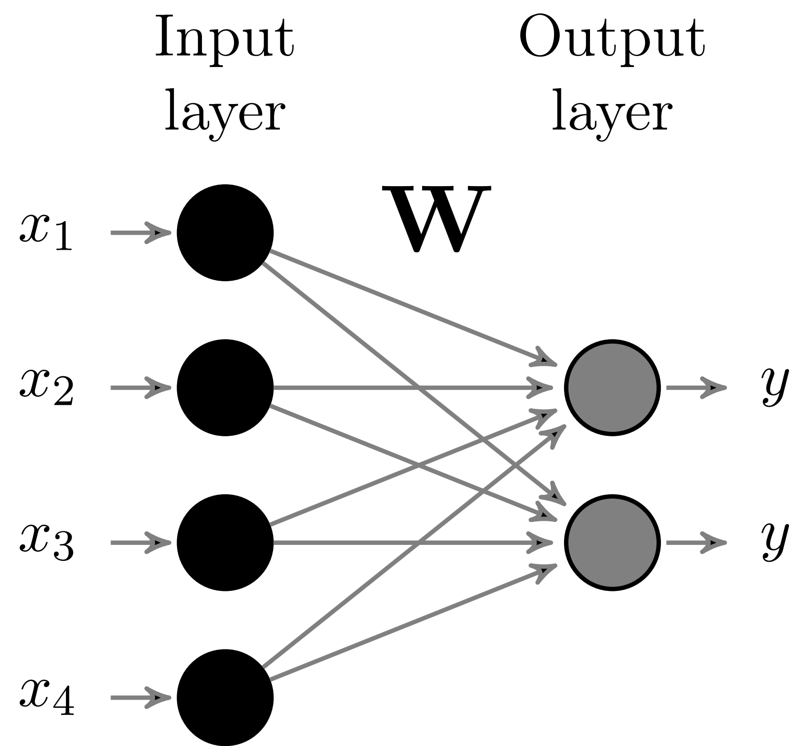
$$f(\mathbf{xW}) = \sigma(\mathbf{xW}) = p(y = 1|\mathbf{x})$$

- Neural network learns W1 and W2 such that

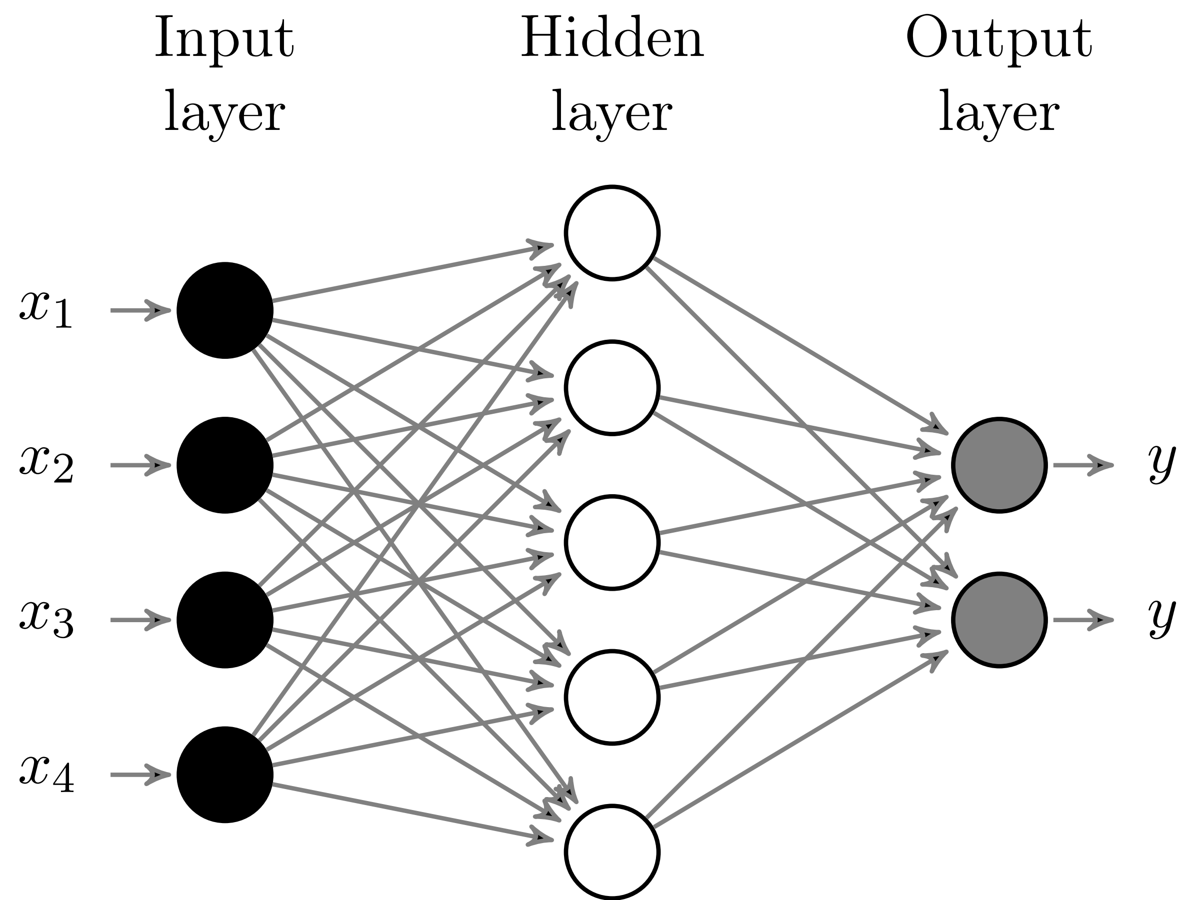
$$p(y = 1|\mathbf{x}) = \sigma(\mathbf{hW}^{(1)}) = \sigma(\sigma(\mathbf{xW}^{(2)})\mathbf{W}^{(1)}).$$



# No representation learning vs. neural network




GLM
(e.g. logistic regression)

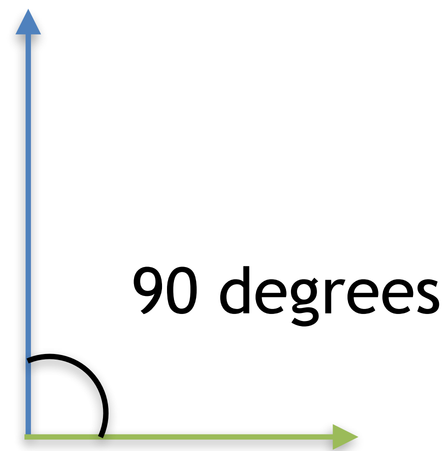



Two-layer neural network

# An aside: Orthogonality

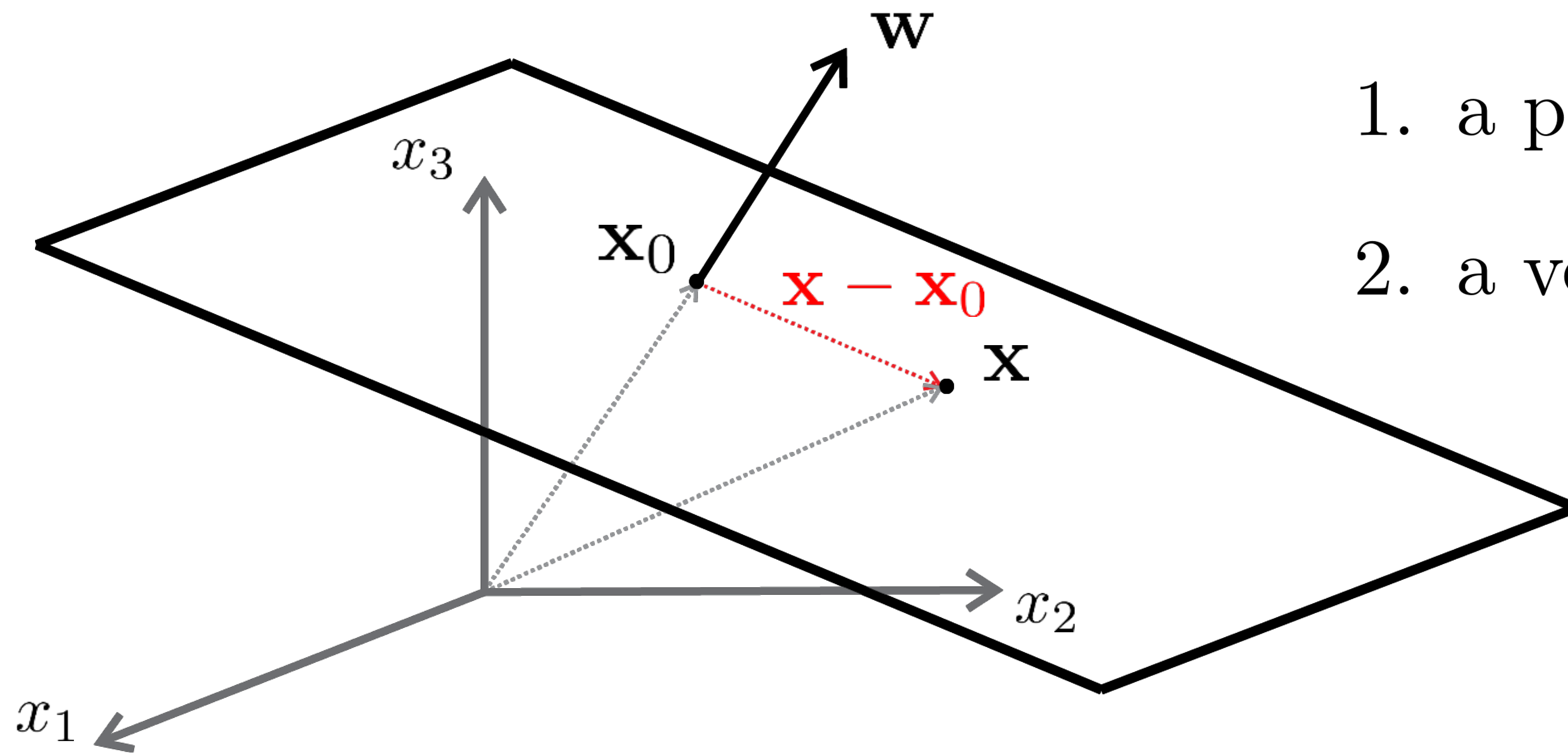
- Two points are orthogonal if dot product is 0
- Cosine similarity: theta angle between w and x

$$\mathbf{w}^\top \mathbf{x} = \|\mathbf{w}\| \|\mathbf{x}\| \cos(\theta)$$




cos(0 degrees) = 0

# Equation of the Plane




A plane is defined using:

1. a point $\mathbf{x}_0$ lying in the plane
2. a vector $\mathbf{w}$ normal to the plane

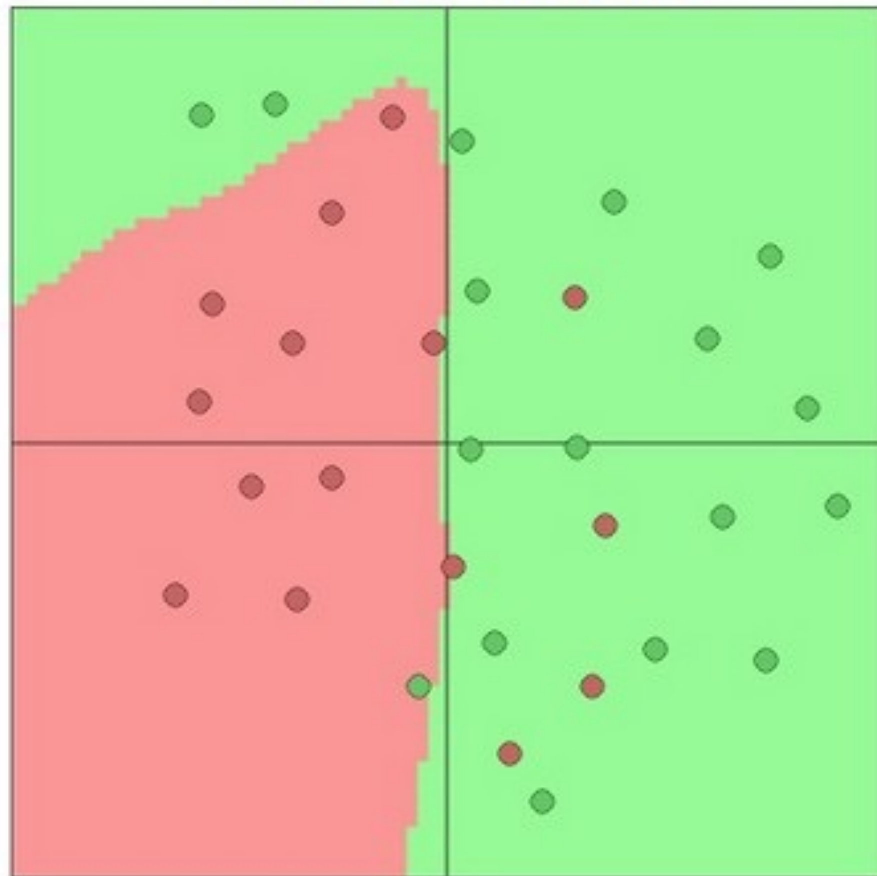
Let $\mathbf{x}$ be on the plane defined by $\mathbf{w}$ and $\mathbf{x}_0$:

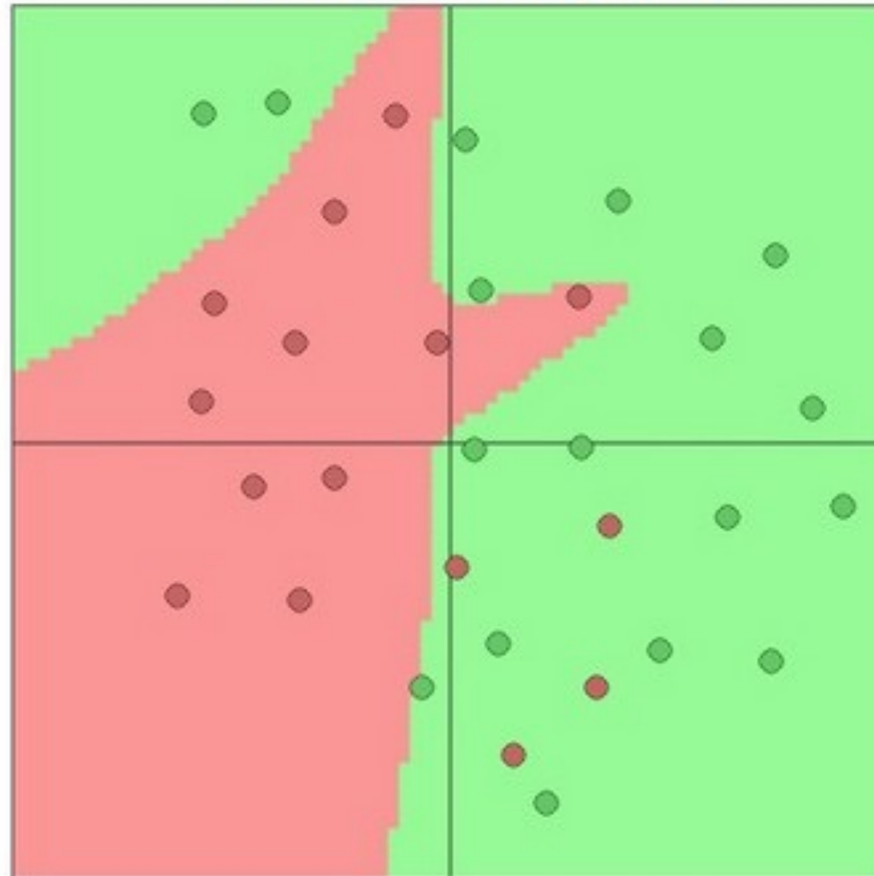
$$\mathbf{w}^T(\mathbf{x} - \mathbf{x}_0) = 0$$

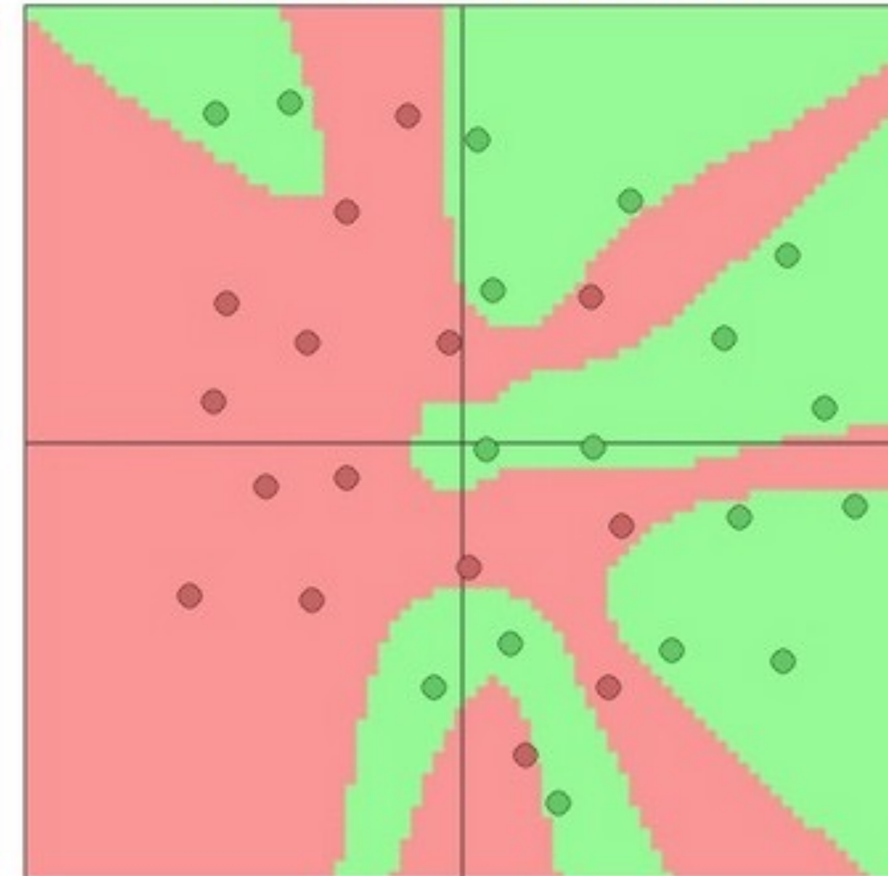
$$\mathbf{w}^T\mathbf{x} - \mathbf{w}^T\mathbf{x}_0 = 0$$

$$\boxed{\mathbf{w}^T\mathbf{x} + w_0 = 0}$$

# Nonlinear decision surface




NN uses cross-entropy and sigmoid on last layer; it still learns a linear plane, just in a different space (higher-dimensional space)

 * from http://cs231n.github.io/neural-networks-1/; see that page for a nice discussion on neural nets

# Maximum likelihood problem

- The goal is to still to find parameters (i.e., all the weights in the network) that maximize the likelihood of the data
- What is p(y | x), for our NN?

$$E[Y|x] = NN(\mathbf{x}) = f_1(f_2(\mathbf{x}\mathbf{W}^{(2)})\mathbf{W}^{(1)})$$

$$\text{e.g., mean of Gaussian, variance } \sigma^2 \text{ still a fixed value}$$

$$\text{e.g., Bernoulli parameter } p(y=1|x) = E[Y|x]$$

$$p = NN(\mathbf{x}) = f_1(f_2(\mathbf{x}\mathbf{W}^{(2)})\mathbf{W}^{(1)})$$

$$\text{Gaussian: } \sum_{i=1}^{n} (p_i - y_i)^2$$

$$\text{Bernoulli: } \sum_{i=1}^{n} \text{Cross-Entropy}(p_i, y_i)$$

# What if removed one connection (i.e., not fully connected)?

$$\boldsymbol{\delta}_k^{(1)} = \hat{y}_k - y_k$$

$$\frac{\partial}{\partial \mathbf{W}_{jk}^{(1)}} = \boldsymbol{\delta}_k^{(1)} \mathbf{h}_j$$

Fully connected update

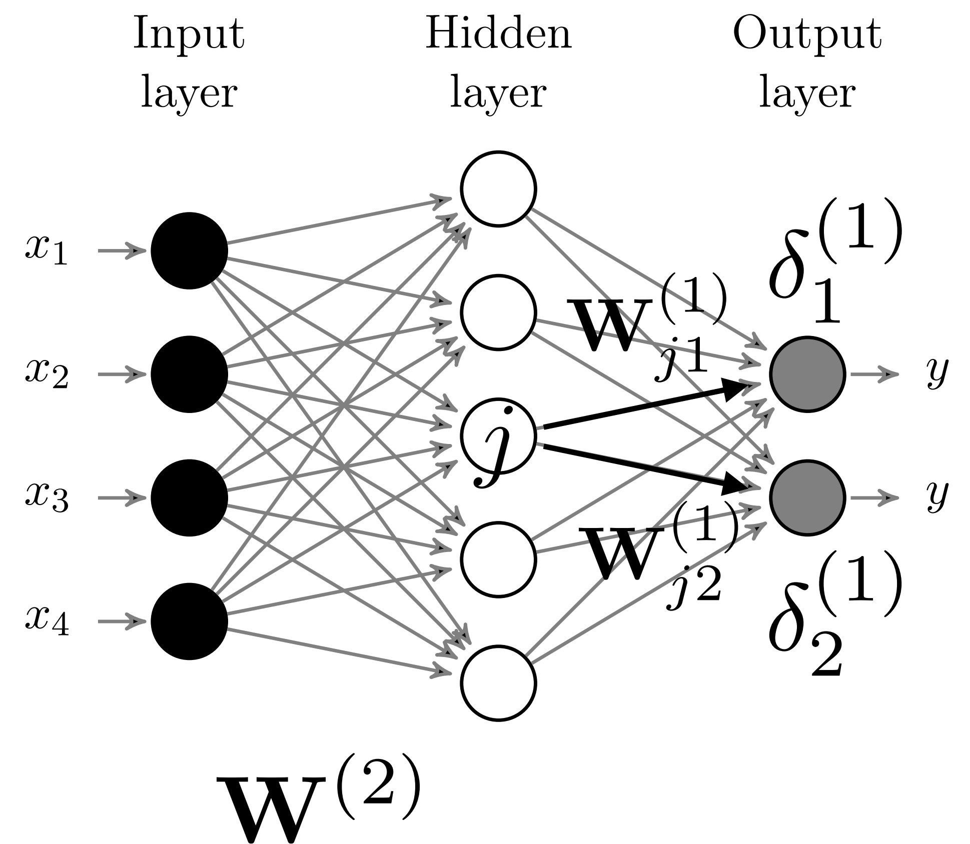
$$\boldsymbol{\delta}_j^{(2)} = \left( \mathbf{W}_{j:}^{(1)} \boldsymbol{\delta}^{(1)} \right) \mathbf{h}_j (1 - \mathbf{h}_j)$$

$$\frac{\partial}{\partial \mathbf{W}_{ij}^{(2)}} = \boldsymbol{\delta}_j^{(2)} \mathbf{x}_i$$

---

$\mathbf{W}_{j1}^{(1)}$ no longer exists, so no update to it

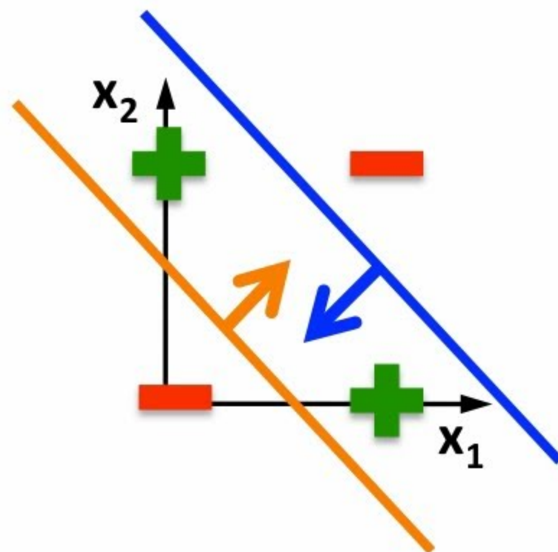
$$\delta_j^{(2)} = (\mathbf{W}_{j2}^{(1)} \delta_2^{(1)}) \mathbf{h}_j (1 - \mathbf{h}_j)$$

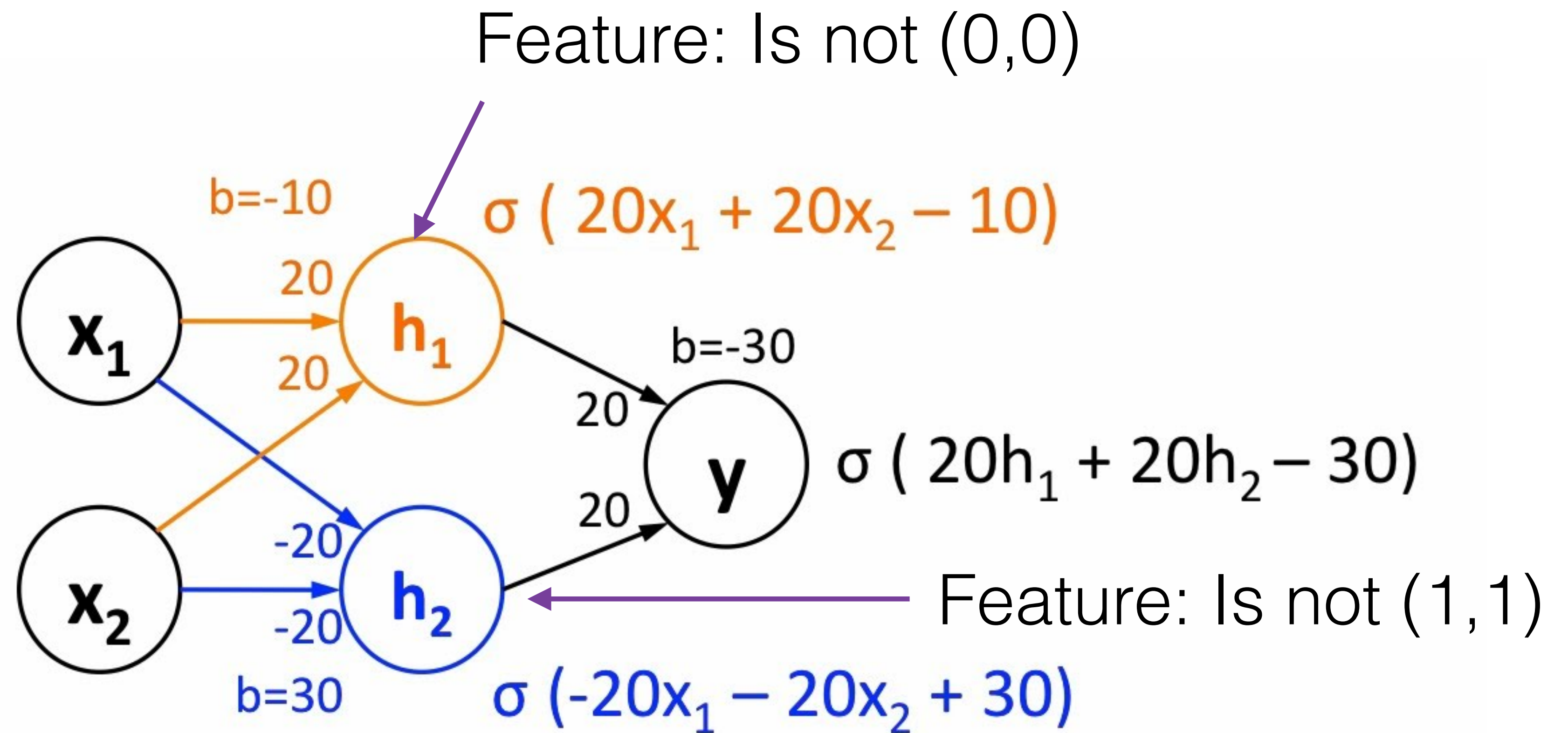


# Recap

- Neural networks let us learn a nonlinear representation phi(x)
  - instead of using a fixed representation, like kernels
- We derived a gradient descent update to learn these reps
- What can NNs really learn?
- How do we optimize them in practice?

# Simple example of representational capabilities: XOR



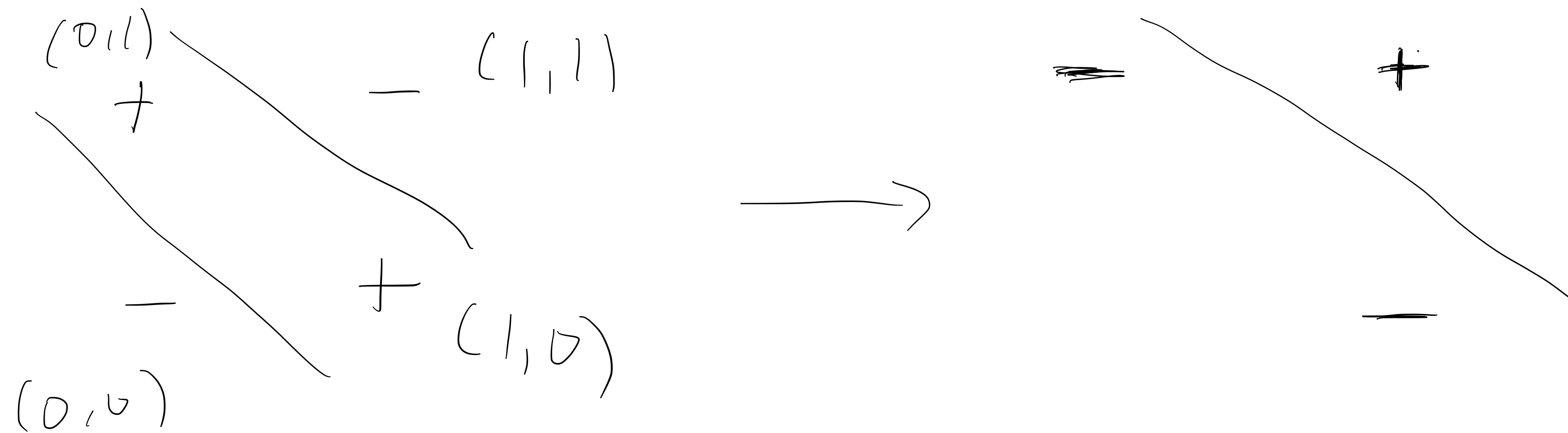




| | | |
|---|---|---|
| $\sigma(20*0 + 20*0 - 10) \approx 0$ | $\sigma(-20*0 - 20*0 + 30) \approx 1$ | $\sigma(20*0 + 20*1 - 30) \approx 0$ |
| $\sigma(20*1 + 20*1 - 10) \approx 1$ | $\sigma(-20*1 - 20*1 + 30) \approx 0$ | $\sigma(20*1 + 20*0 - 30) \approx 0$ |
| $\sigma(20*0 + 20*1 - 10) \approx 1$ | $\sigma(-20*0 - 20*1 + 30) \approx 1$ | $\sigma(20*1 + 20*1 - 30) \approx 1$ |
| $\sigma(20*1 + 20*0 - 10) \approx 1$ | $\sigma(-20*1 - 20*0 + 30) \approx 1$ | $\sigma(20*1 + 20*1 - 30) \approx 1$ |

Copyright © 2014 Victor Lavrenko

*see nice video Neural Networks 6: solving XOR with a hidden layer
: https://www.youtube.com/watch?v=kNPGXgzxoHw

# Linearly separable now




New features: (Not (0,0), Not (1,1))

(0,0) → (0,1)

(1,1) → (1,0)

(0,1) → (1,1)

(1,0) → (1,1)

Linearly separable in new space.

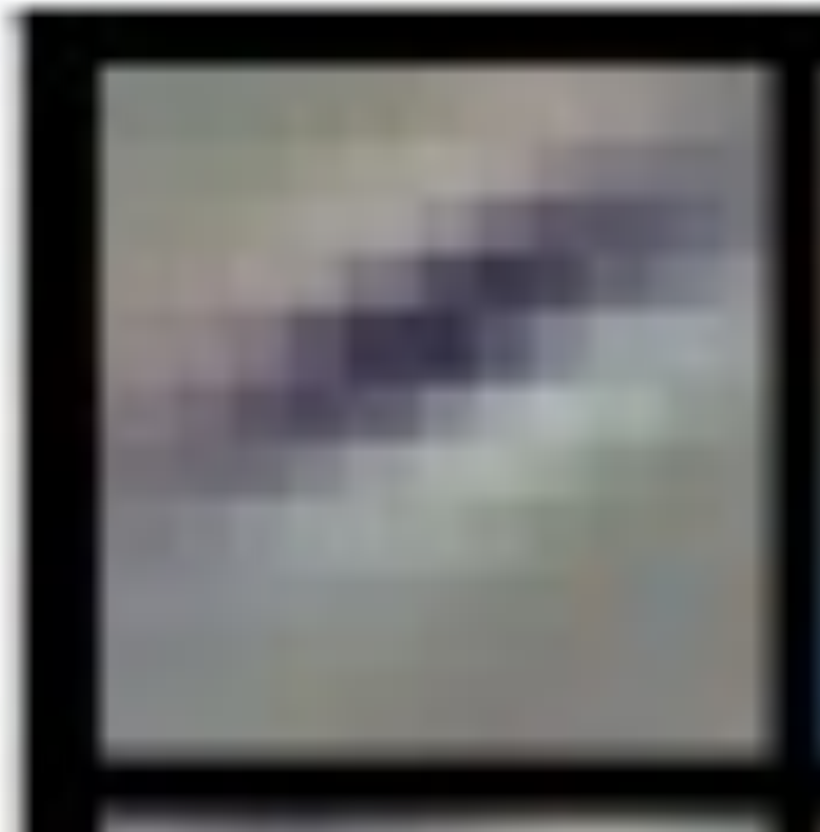
# One layer can act like a filter

- Dot-product with input x, and a weight vector w, can emphasize or filter parts of x
  - e.g., imagine x is an image, and w is zero everywhere except one small patch in the corner. It will pick out the magnitude of pixels in that small patch

*awesome overview: http://cs231n.github.io/convolutional-networks/

# Zooming in

- Dot-product with input x, and a weight vector w, can emphasize or filter parts of x



Input square in image (linearized): $\mathbf{x}$

Dot product with filter: $\mathbf{w}$

$\mathbf{x}^\top \mathbf{w}$ also represents similarity!

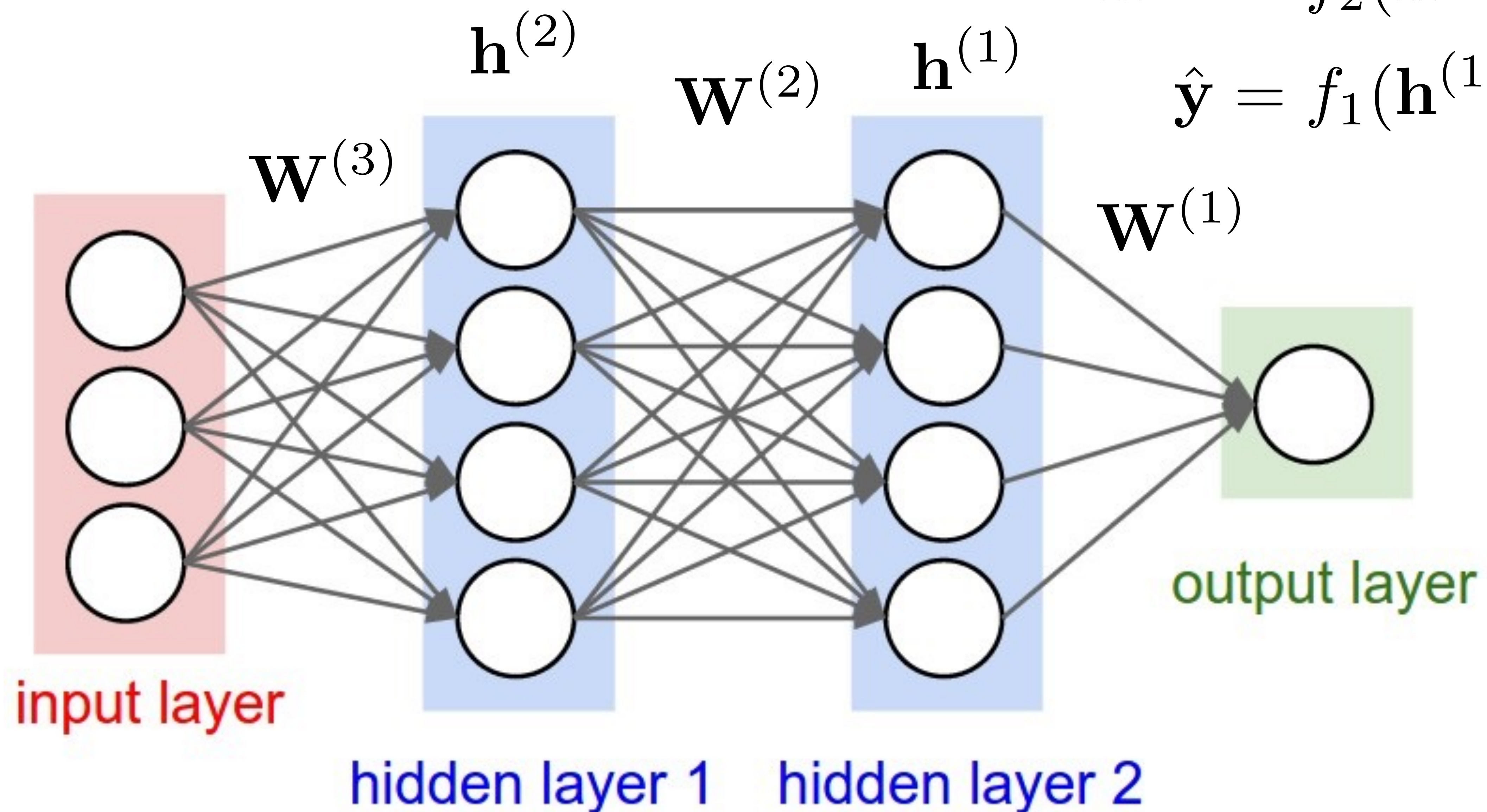
# Multi-layer neural network

What is phi(x) here?

$$\mathbf{h}^{(2)} = f_3(\mathbf{x}\mathbf{W}^{(3)})$$

$$\mathbf{h}^{(1)} = f_2(\mathbf{h}^{(2)}\mathbf{W}^{(2)})$$

$$\hat{\mathbf{y}} = f_1(\mathbf{h}^{(1)}\mathbf{W}^{(1)})$$




* from http://cs231n.github.io/neural-networks-1/; see that page for a nice discussion on neural nets

# What about more layers?

- Can consider the first N-1 layers to learn the new representation of x: phi(x)
  - this new representation is informed by prediction accuracy, unlike a fixed representation

- The last layer learns a generalized linear model on phi(x) to predict E[Y | x]: f(< phi(x), w > )

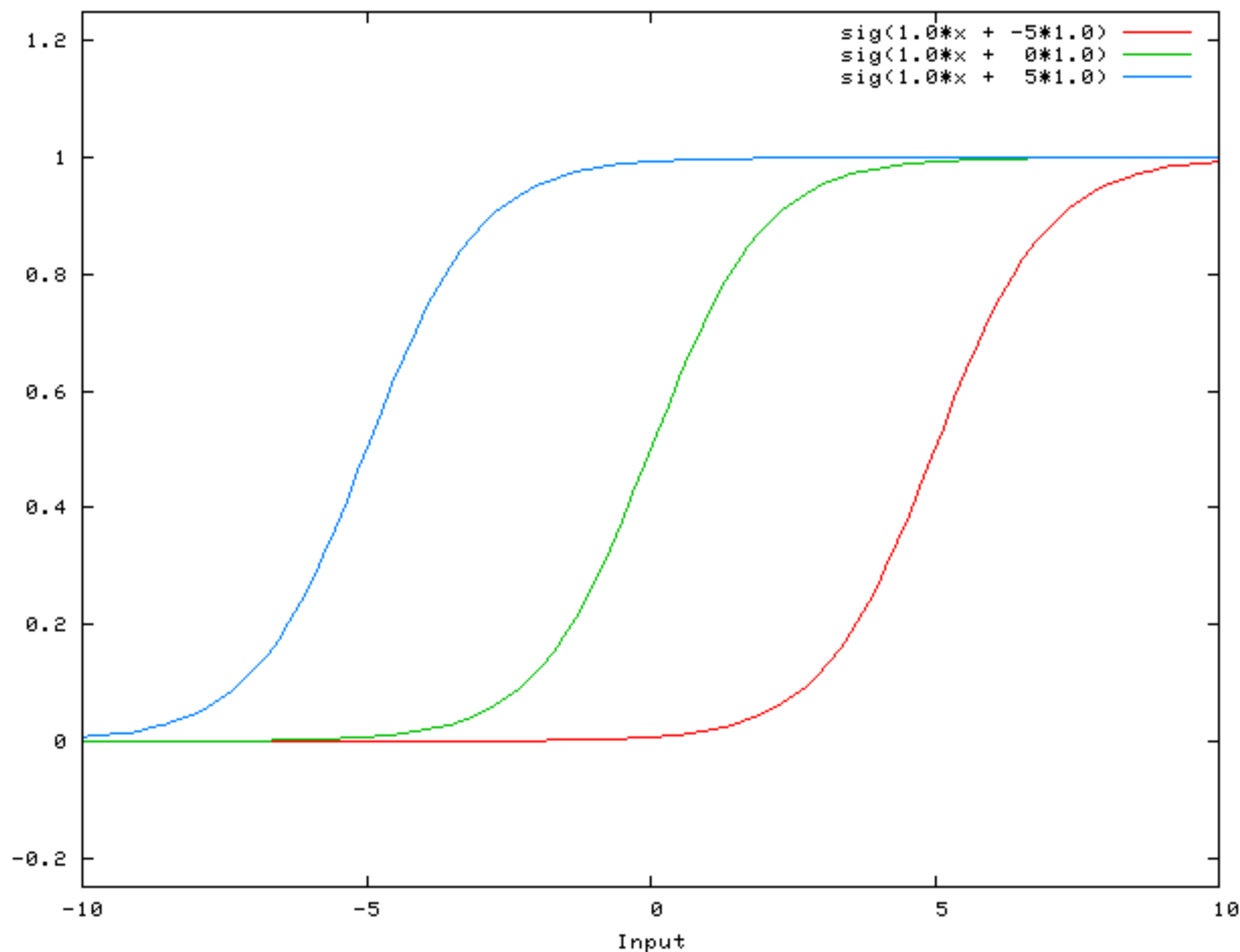
- As with previous generalizations, this last layer can:
  - use any generalized linear model transfer and loss
  - can have multivariate output y
  - can use regularizers
  - can use different costs per sample

# Theory to support depth?

- The utility of more layers has been primarily an empirical observation; more theory now to support the utility of depth
  - Though still new
- Depth has shown to be particularly important for convolutional neural networks
  - each convolutional layer summarizes the previous layer, providing a hierarchical structure where depth is intuitively useful
- See: “Learning Functions: When Is Deep Better Than Shallow ”https://arxiv.org/abs/1603.00988
- See for example: “Do Deep Nets Really Need to be Deep?” https://arxiv.org/abs/1312.6184

# Exercise: Bias unit and adding a column of ones to GLMs

- This provides the same outcome as for linear regression
- g(E[y I x]) = x w —> bias unit in x with coefficient w0 shifts the function left or right



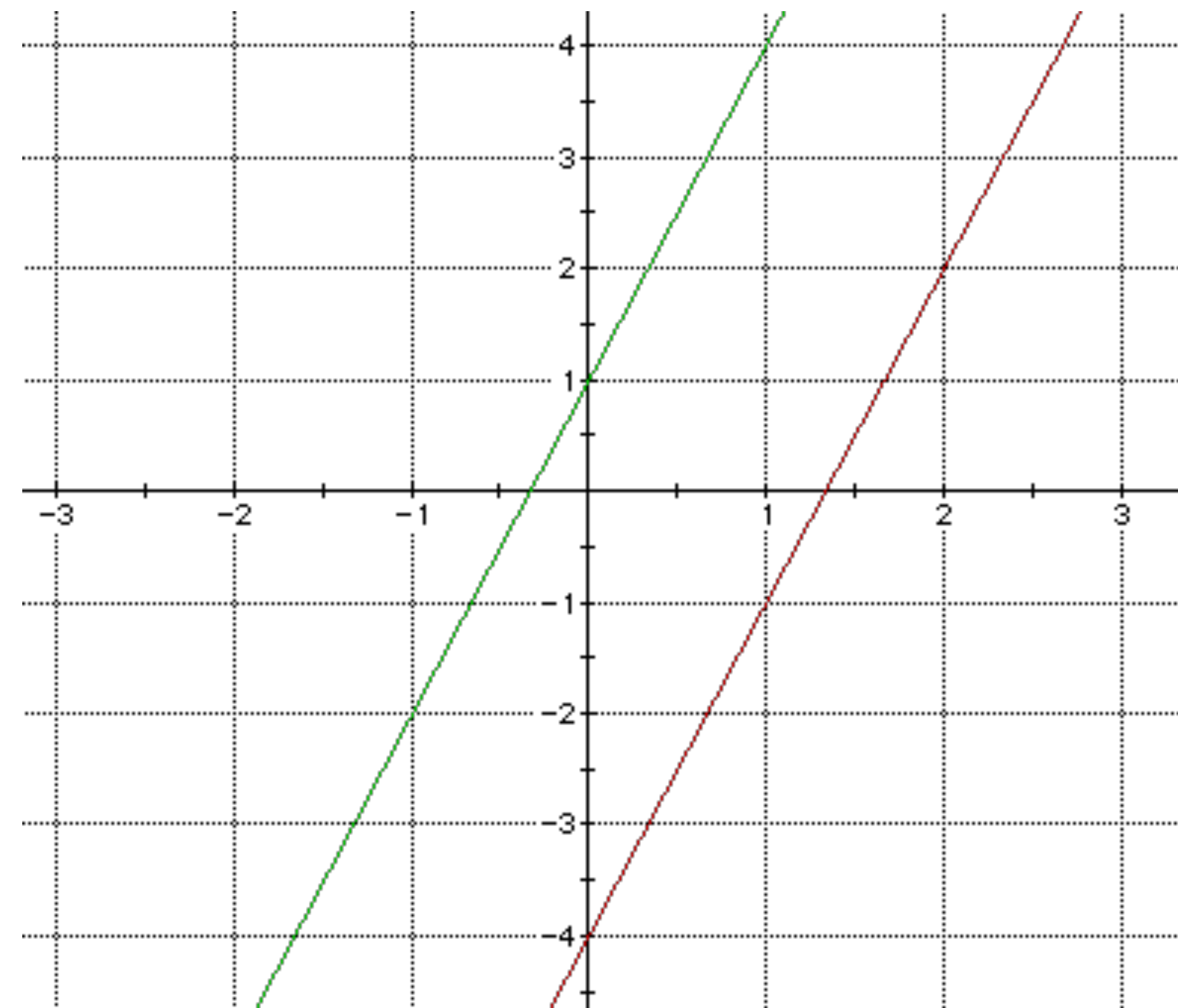




*Figure from http://stackoverflow.com/questions/2480650/role-of-bias-in-neural-networks

# Exercise: bias unit

- Assume we pick a sigmoid activation
- What does it mean to add a bias unit to the input?
  - can shift the sigmoid curve left or right, just like before, for the first hidden layer
- What does it mean to add a bias unit for an interior layer?
  - can shift the sigmoid curve left or right for the next layer, without having to rely on previous layer to carefully adjust
- What does it mean to add a bias unit to the last layer (the last hidden layer before predicting y)?
  - yup, you guessed it, still the same reason

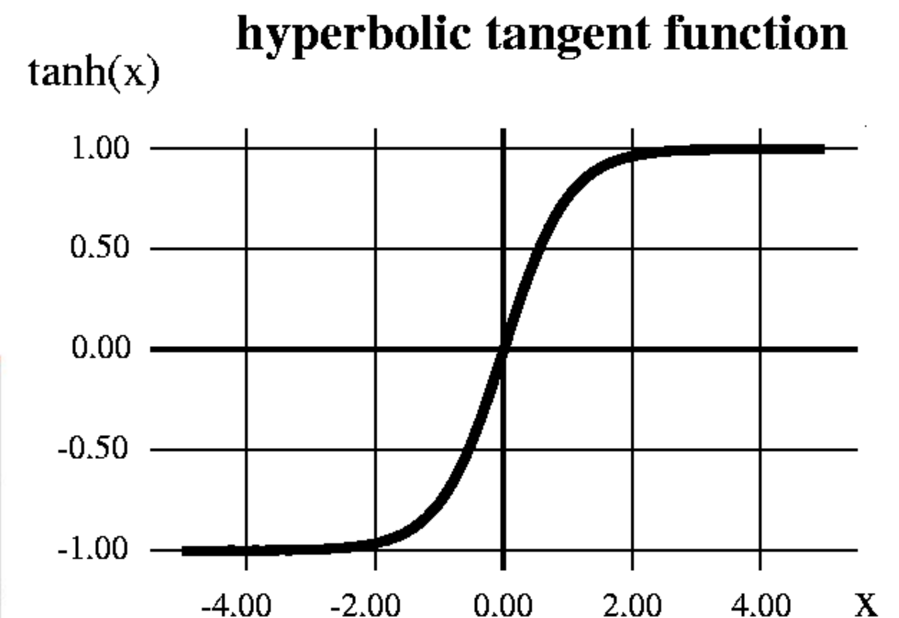
# Structural choices

- The number of hidden layers
- The number of hidden nodes in each layer
- The activation functions
- How connected each layer is (maybe not fully connected)
- …
- The network structure simply indicates which variables influence other variables (contribute to their construction); can imagine many different architectures

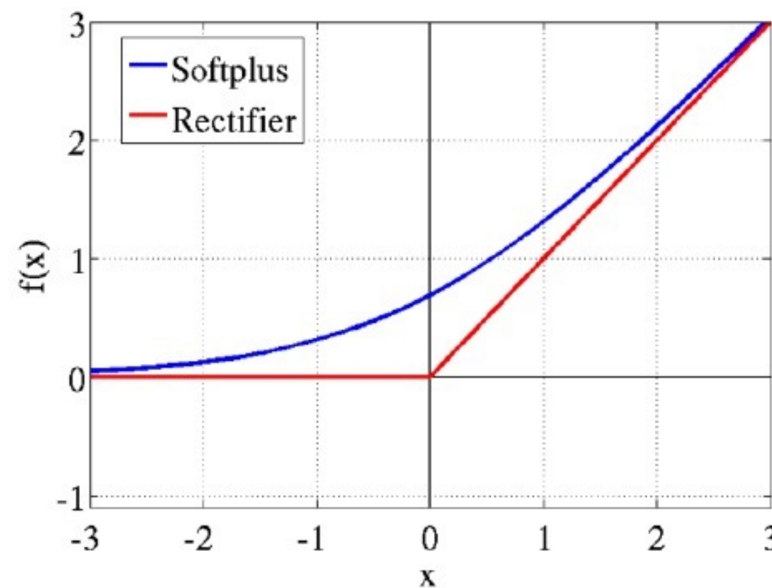
# Tanh and rectified linear

- Two more popular transfers are tanh and rectified linear
- Tanh is balanced around 0, which seems to help learning
  - usually preferred to sigmoid

$$\tanh(\theta) = \frac{\exp(\theta) - \exp(-\theta)}{\exp(\theta) + \exp(-\theta)}$$

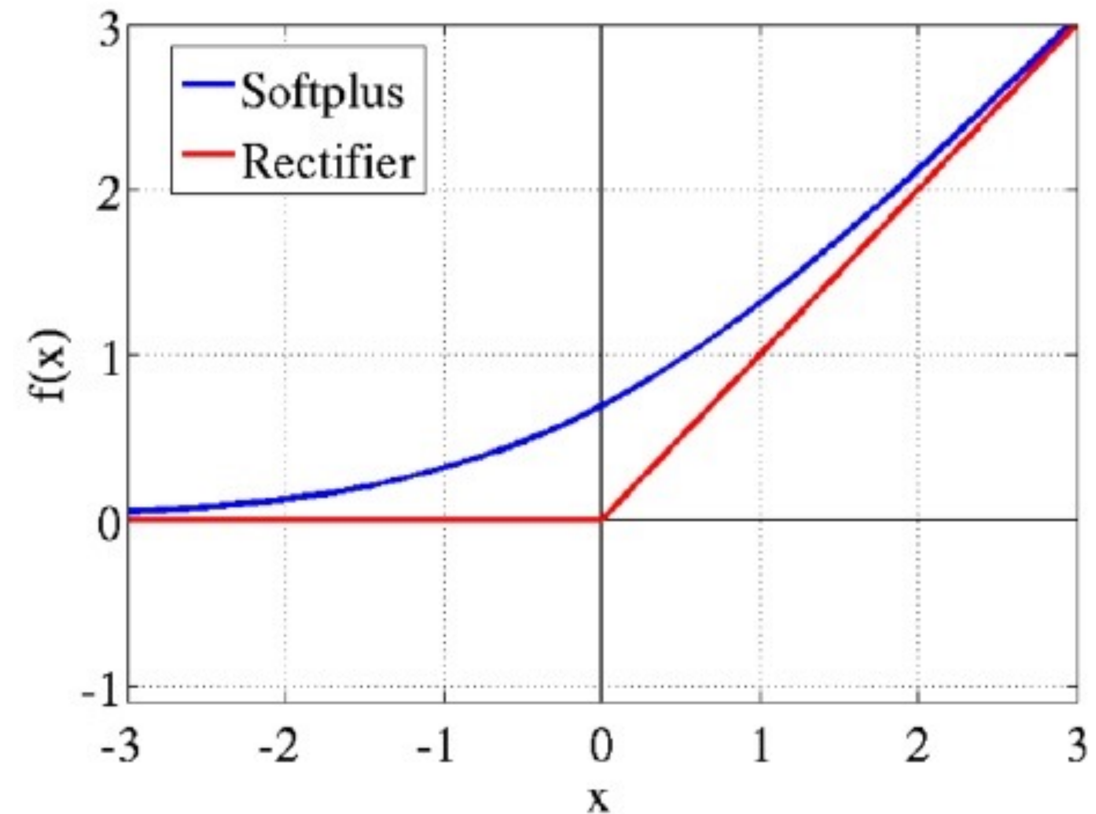



- Rectified linear




- Binary threshold function (perceptron): less used,
  - some notes for this approach: http://www.cs.indiana.edu/~predrag/classes/2015springb555/9.pdf

# Rectified linear unit (ReLU)




- Rectified(x) = max(0, x)
  - Non-differentiable point at 0
  - Commonly gradient is 0 for x <= 0, else 1
- Recall our variable is $\theta = \mathbf{x}^\top \mathbf{w}$

$\theta$

- Common strategy: still use sigmoid (or tanh) with cross-entropy in the last output layer, and use rectified linear units in the interior
- Variants of ReLu: Softplus(x) = ln(1+e^{x}), Leaky Relu

# Exercise: changing from sigmoid to tanh

- Let's revisit the two-layer update.

$$\boldsymbol{\delta}_j^{(2)} = \left(\mathbf{W}_{j:}^{(1)} \boldsymbol{\delta}^{(1)}\right) \frac{\partial f_2(\boldsymbol{\theta}_j^{(2)})}{\partial \boldsymbol{\theta}_j^{(2)}}$$

$$\boldsymbol{\delta}_j^{(2)} = \left(\mathbf{W}_{j:}^{(1)} \boldsymbol{\delta}^{(1)}\right) \mathbf{h}_j (1 - \mathbf{h}_j)$$

$$\frac{\partial}{\partial \mathbf{W}_{ij}^{(2)}} = \boldsymbol{\delta}_j^{(2)} \mathbf{x}_i$$

- How does it change if we instead use f_2 = tanh, for the activation on the first layer?
  - recall: the derivative of tanh(theta) is 1-tanh^2(theta)

$$\delta_j^{(2)} = (\mathbf{W}_{j:}^{(1)} \boldsymbol{\delta}^{(1)})(1 - \mathbf{h}_j^2)$$

# Exercise: changing from sigmoid to ReLU

- Let's revisit the two-layer update.

$$\boldsymbol{\delta}_j^{(2)} = \left(\mathbf{W}_{j:}^{(1)}\boldsymbol{\delta}^{(1)}\right)\frac{\partial f_2(\boldsymbol{\theta}_j^{(2)})}{\partial \boldsymbol{\theta}_j^{(2)}}$$

$$\boldsymbol{\delta}_j^{(2)} = \left(\mathbf{W}_{j:}^{(1)}\boldsymbol{\delta}^{(1)}\right)\mathbf{h}_j(1-\mathbf{h}_j)$$

$$\frac{\partial}{\partial \mathbf{W}_{ij}^{(2)}} = \boldsymbol{\delta}_j^{(2)}\mathbf{x}_i$$

- How does it change if we instead use f_2 = relu, for the activation on the first layer?
    - recall: the derivative of relu(theta) = max(0, theta) is 1 or 0

$$\delta_j^{(2)} = (\mathbf{W}_{j:}^{(1)}\boldsymbol{\delta}^{(1)})\text{Indicator}(\mathbf{h}_j > 0)$$

# Why so careful with l1 and not ReLU?

- For L1 (Lasso) used proximal operators for non-differentiable function to ensure convergence

- Why so uncareful with ReLUs?

- One answer: it seems to work

- Hypothesis: if gradient pushing input to ReLU to zero, then overshooting non-differentiable point ok —> the output value is still 0!

# How do we select the loss function and activations?

- How do we select the loss function?
  - Loss is only defined for the last layer —> we use generalized linear models
- How do we select activations?
  - activation on last layer determined by GLM
  - for interior activations, its an art to decide what to use

# Optimization choices

- The objective is still (mostly) smooth, but is no longer convex; is this a problem?
  - Can still use gradient descent approaches, but might get stuck in local minima or saddle points —> the chosen optimization approaches care more about getting out of such solutions
  - The initialization matters more (why?)

# Initialization

- One of the key aspects that have made NNs work is better initialization strategies

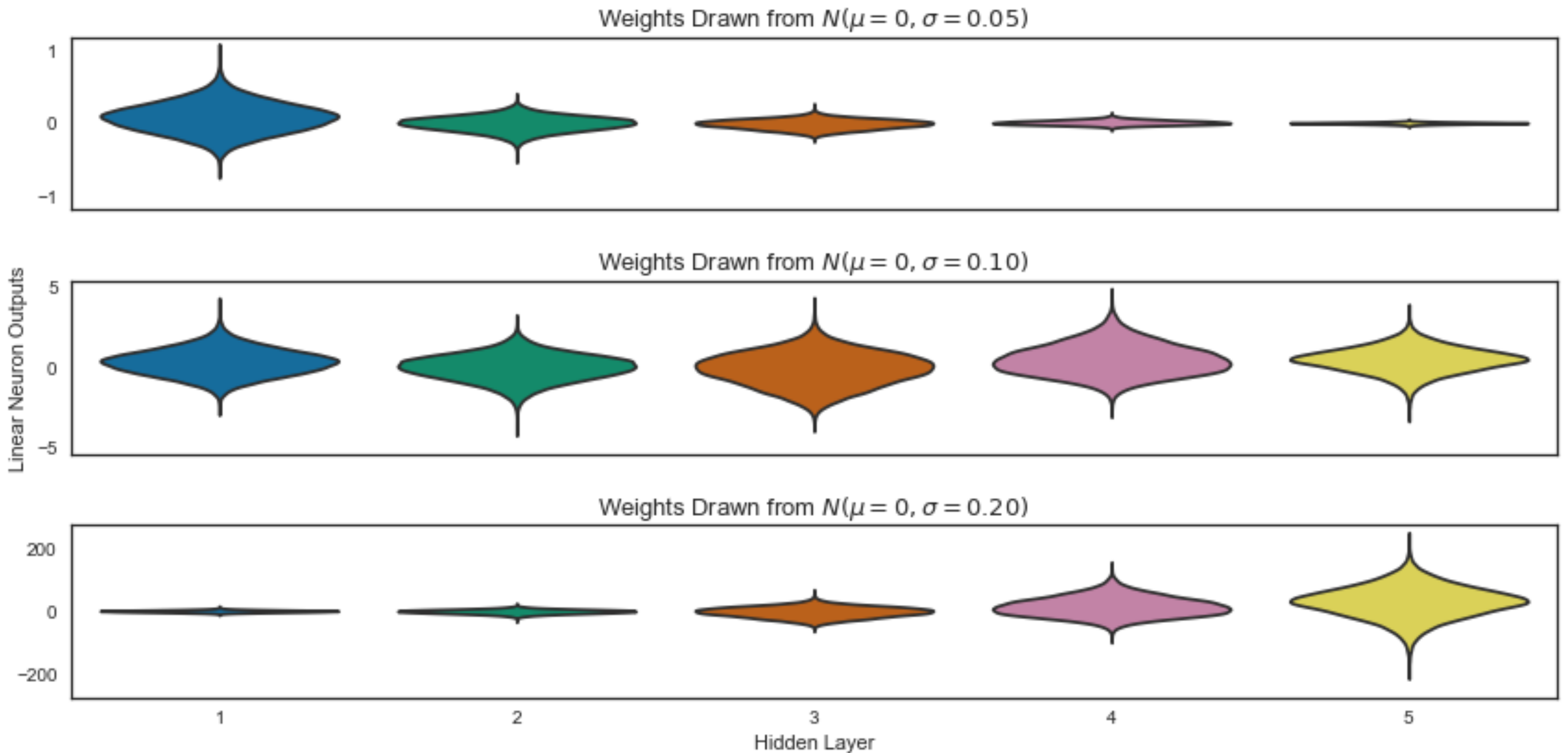
- Imagine could initialize really close to the true solution
  - wouldn’t that be great! We would just need to iterate a small number of steps and be done

- In general, where we initialize from can significantly impact the number of steps and the final solution
  - initialization affects how close we are to a good solution
  - initializations affects the function surface in that local region; flat function surfaces can be bad

# Modern initialization strategies

- Maintain consistent variance of gradients throughout the network, to ensure that gradients do not go to zero in earlier layers
  - if activations become zero, they start to filter some of the gradient that is being passed backwards
  - if activations get very large, they magnify gradients and cause instability
- See the paper: “Understanding the difficulty of training deep feedforward neural networks”, Glorot and Bengio

# Impact of initialization




Activations of the hidden layers after one batch of 1000 MNIST images are passed through the NN (5 hidden layers, 100 nodes each, linear activation)

*image from https://intoli.com/blog/neural-network-initialization/

# Selecting step sizes

- Can select a single stepsize for the entire network
  - That's a hard parameter to tune
- Much better to select an individual stepsize for each parameter
  - a vector stepsizes
- Quasi-second order algorithms also work for NNs
  - Adadelta and RMSProp
  - Adam and AMSGrad

# Exercise: overfitting

- Imagine someone gave you a kernel representation with 1000 prototypes
  - representation is likely sparse: only a small number of features in phi(x) are active (the rest are near zero)
- Imagine you learned an NN, with one hidden layer of size 1000
- Which do you think might be more prone to overfitting?
- Is it just about number of parameters? What if use a linear activation function?

# Strategies to avoid overfitting

- Early stopping
  - keep a validation set, a subset of the training set
  - after each epoch, check if accuracy has levelled off on the validation set; if so, stop training
  - uses test accuracy rather than checking the objective is minimized
- Dropout
- Other regularizers
- New idea (counter-intuitive): make your network really big