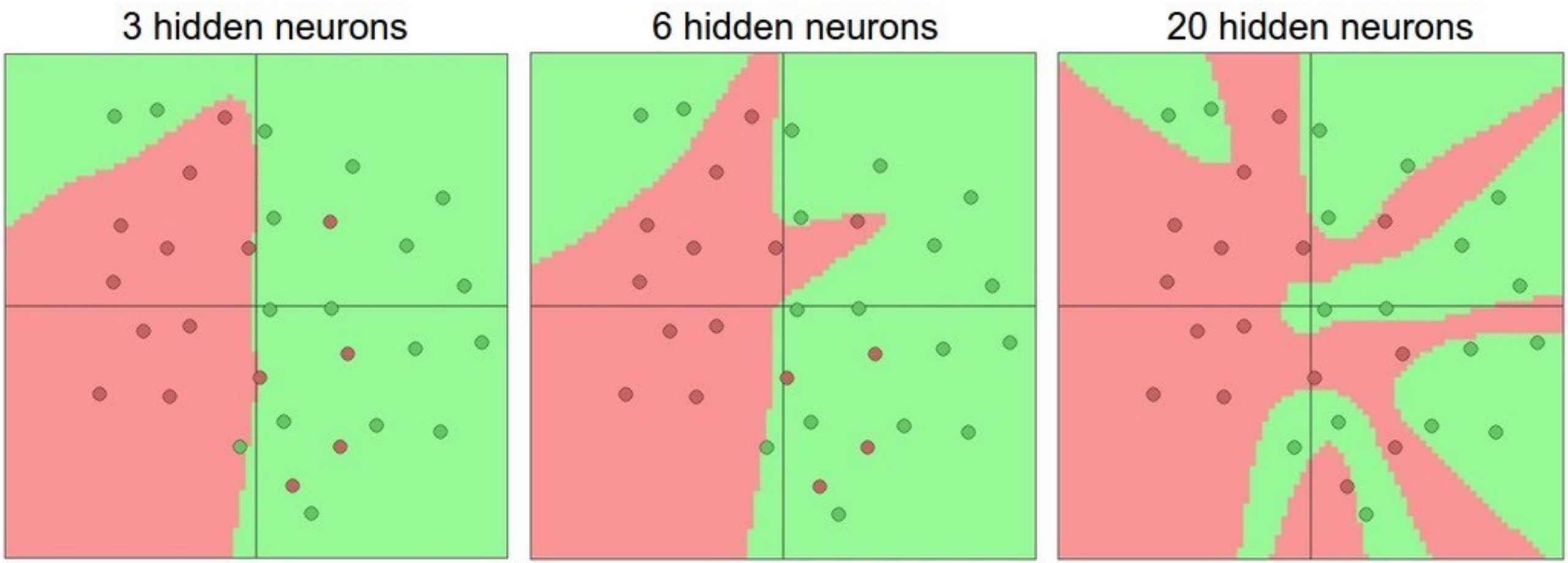
Neural networks



Comments

- Assignment 3 code released
  - implement classification algorithms
  - use kernels for census dataset
- Thought questions 3 due this week
- Mini-project: hopefully you have started

Example: logistic regression versus neural network

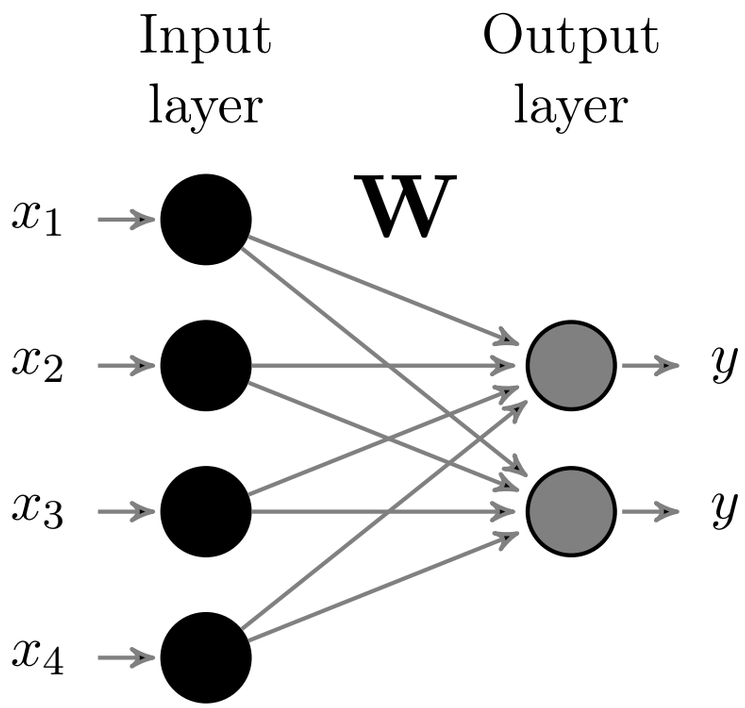
- Both try to predict p(y = 1 | x)
- Logistic regression learns W such that

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

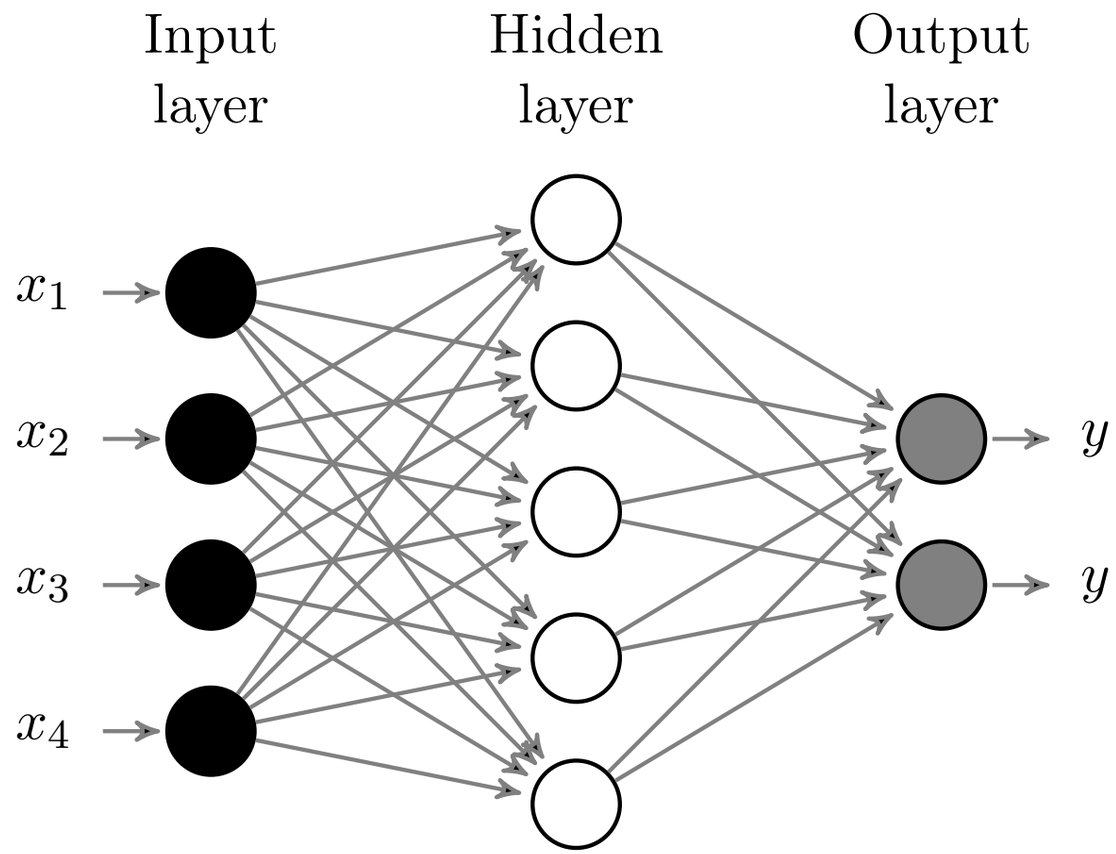
- Neural network learns W1 and W2 such that

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

No representation learning vs. neural network



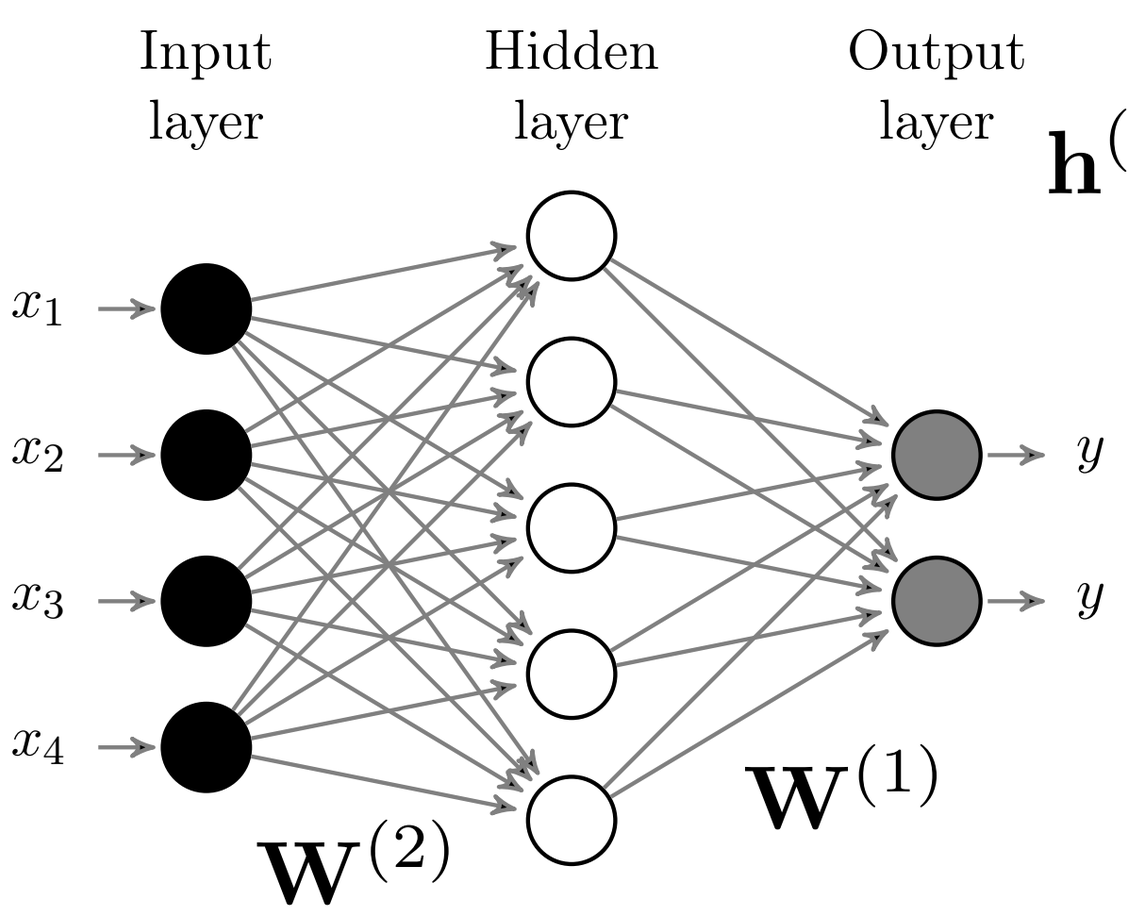
GLM
(e.g. logistic regression)



Two-layer neural network

What are the representational capabilities of neural nets?

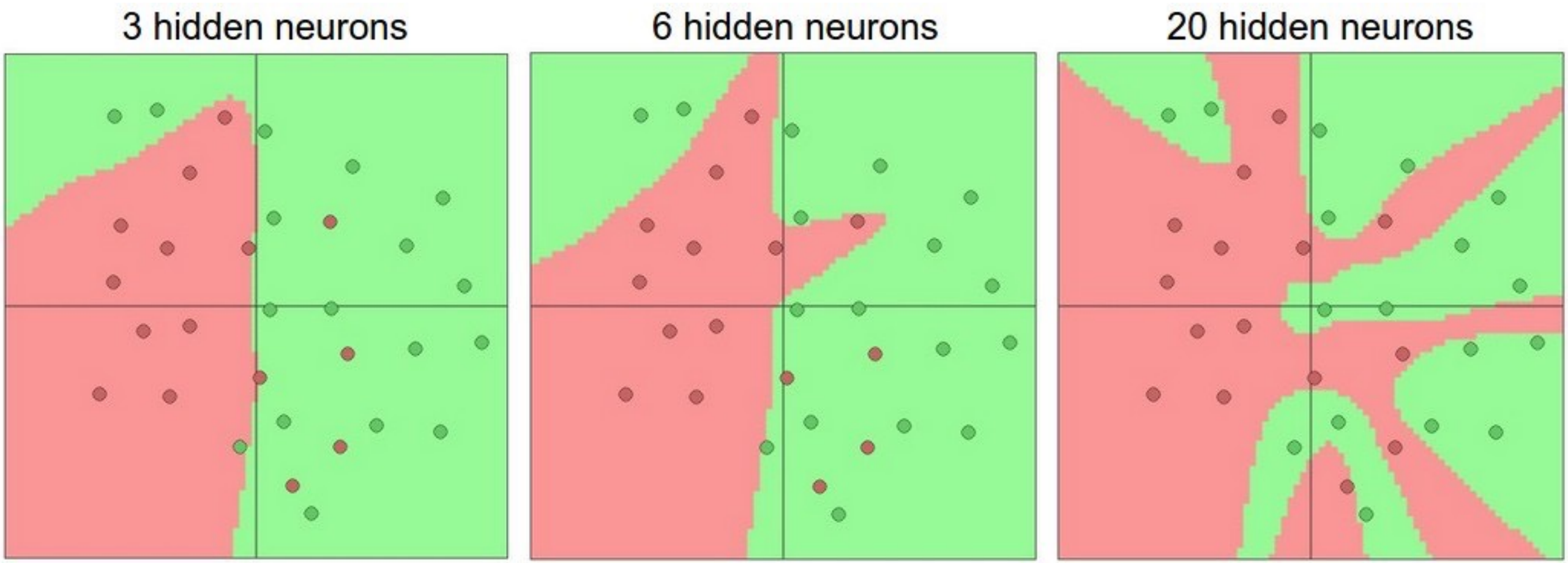
- Single hidden-layer neural networks with sigmoid transfer can represent any continuous function on a bounded space within epsilon accuracy, for a large enough number of hidden nodes
  - see Cybenko, 1989: “Approximation by Superpositions of a Sigmoidal Function”



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

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

Nonlinear decision surface



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

How do we learn the parameters to the neural network?

- In linear regression and logistic regression, learned parameters by specifying an objective and minimizing using gradient descent

- We do the exact same thing with neural networks; the only difference is that our function class is more complex

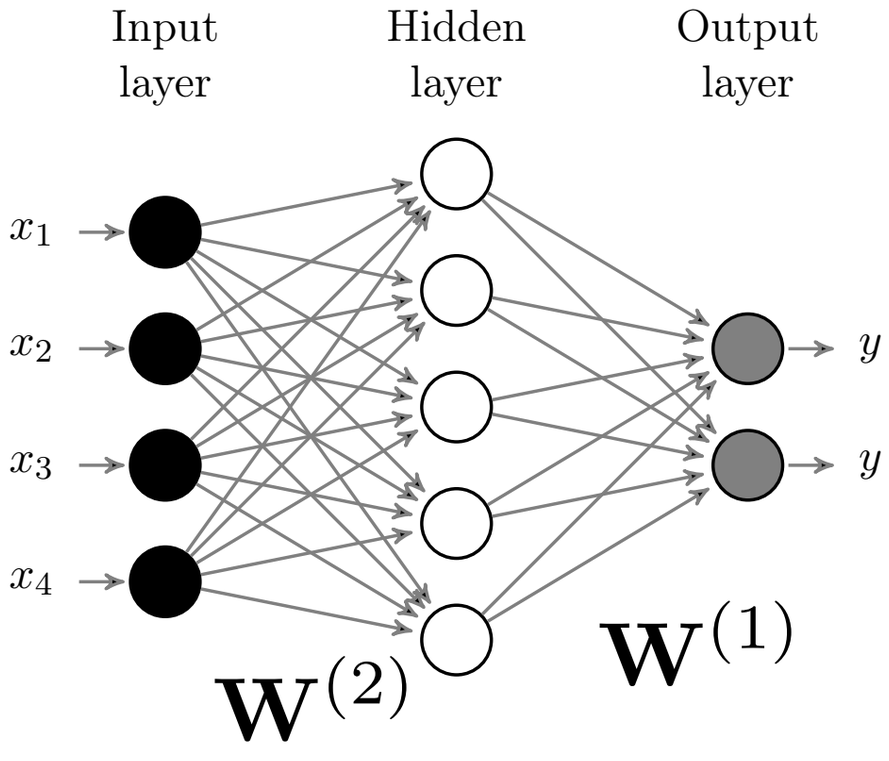
- Need to derive a gradient descent update for W1 and W2
  - reasonably straightforward, indexing just a bit of a pain

Example for p(ylx) Bernoulli

$$L(\hat{y}, y) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$$

$$f_2(\mathbf{x}\mathbf{W}^{(2)}_{:j}) = \sigma(\mathbf{x}\mathbf{W}^{(2)}_{:j}) = \frac{1}{1 + \exp(-\mathbf{x}\mathbf{W}^{(2)}_{:j})}$$

$$f_1(\mathbf{h}\mathbf{W}^{(1)}_{:k}) = \sigma(\mathbf{h}\mathbf{W}^{(1)}_{:k}) = \frac{1}{1 + \exp(-\mathbf{h}\mathbf{W}^{(1)}_{:k})}$$



Forward propagation

- First have to compute all the required components to produce the prediction yhat, so that we can measure the error
- Forward propagation simply means starting from inputs to compute hidden layers to then finally output a prediction
  - i.e., simply means evaluating the function f(x) that is the NN
- A fancy name for a straightforward concept
  - naming things is useful, but can obfuscate simple concepts

Backward propagation

- Once have output prediction yhat (and all intermediate layers), can now compute gradient

- The gradient computed for the weights on the output layer contains some shared components with the weights for the hidden layer

- This shared component is computed for output weights W1

- Instead of recomputing it for W2, that work is passed to the computation of the gradient of W2 (propagated backwards)

Example for Bernoulli (cont)

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

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

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

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

$$\longrightarrow \quad \mathbf{W}_{jk}^{(1)} \leftarrow \mathbf{W}_{jk}^{(1)} - \alpha \frac{\partial}{\mathbf{W}_{jk}^{(1)}}$$

$$\mathbf{W}_{ij}^{(2)} \leftarrow \mathbf{W}_{ij}^{(2)} - \alpha \frac{\partial}{\mathbf{W}_{ij}^{(2)}}$$

Whiteboard

- Derivation of back-propagation for two layers
- Exercise: single hidden-layer with no activation function
- Exercise: what if not fully connected?
- Disclaimer: understanding NNs, what works and doesn’t, is an in-progress research question; some of what I tell you will be hypotheses, and not as concrete as some of the previous foundations of ML

Comments (Nov 2)

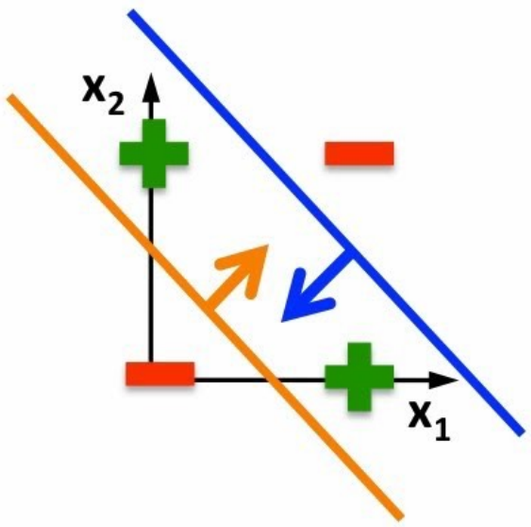
- Thought question deadline extended by 2 days
- Peng posted a few specifications for Assignment 3
- If you'd like, grab some Halloween candy at the front

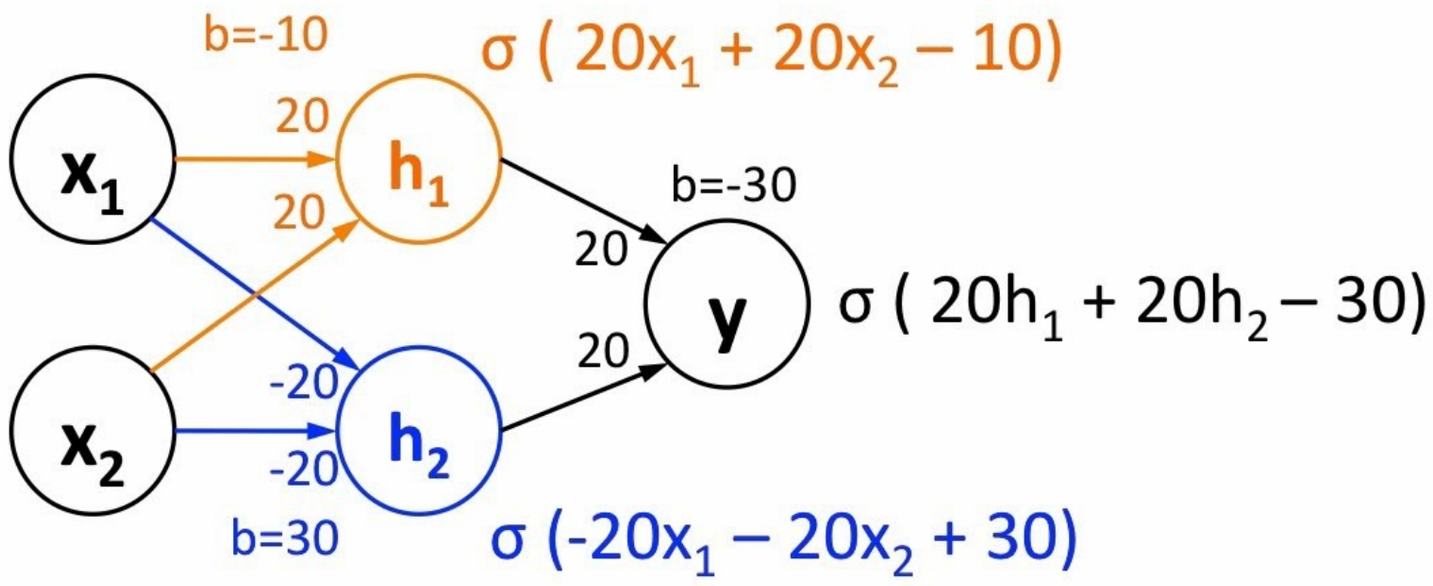
Recap

- 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

Linear classifiers cannot solve this



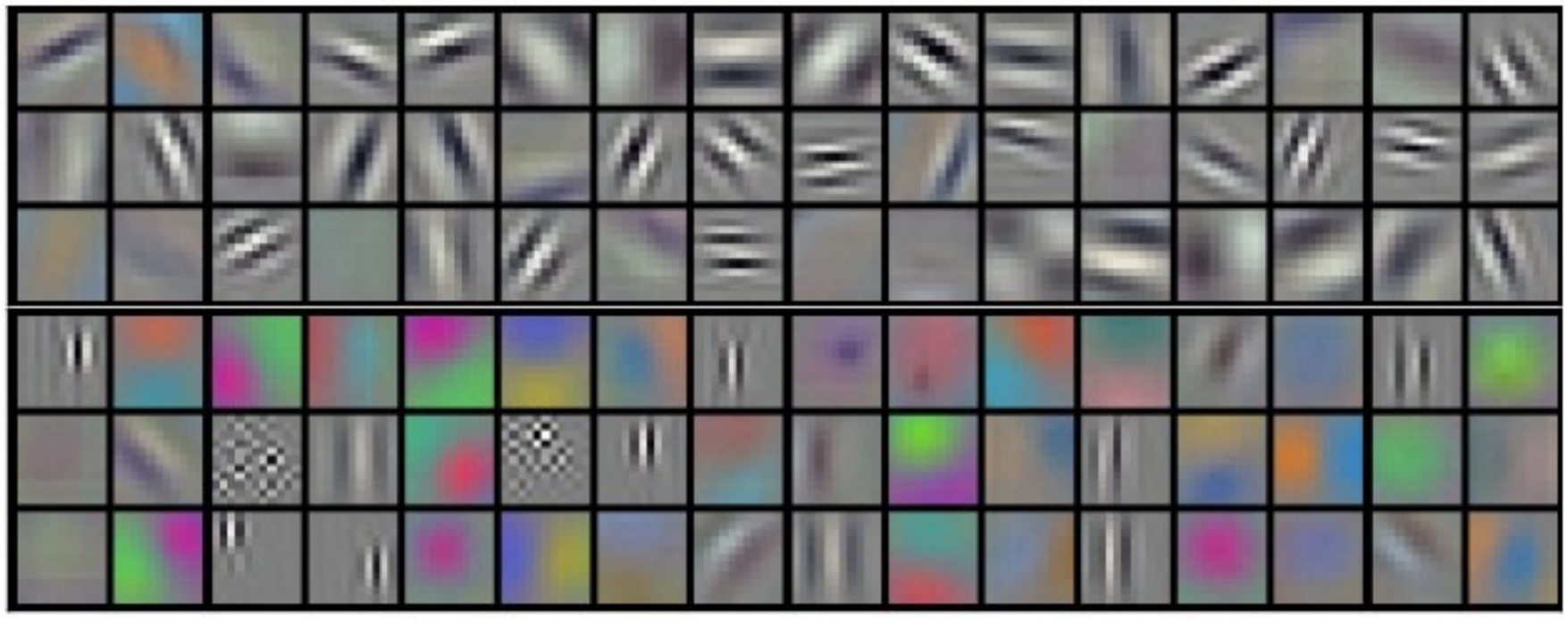


σ(20*0 + 20*0 – 10) ≈ 0 σ (-20*0 – 20*0 + 30) ≈ 1 σ (20*0 + 20*1 – 30) ≈ 0
σ(20*1 + 20*1 – 10) ≈ 1 σ (-20*1 – 20*1 + 30) ≈ 0 σ (20*1 + 20*0 – 30) ≈ 0
σ(20*0 + 20*1 – 10) ≈ 1 σ (-20*0 – 20*1 + 30) ≈ 1 σ (20*1 + 20*1 – 30) ≈ 1
σ(20*1 + 20*0 – 10) ≈ 1 σ (-20*1 – 20*0 + 30) ≈ 1 σ (20*1 + 20*1 – 30) ≈ 1

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

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/

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 l x), for our NN?

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

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

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

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

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

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

Gradient descent procedure for NNs

- Compute delta for all nodes in the last layer

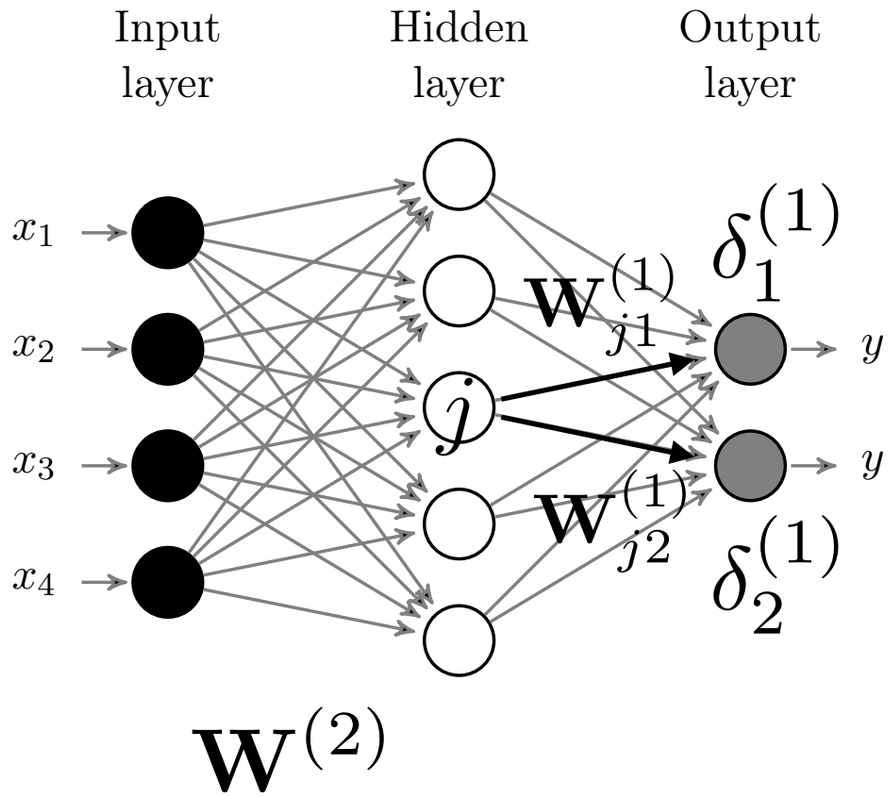
- This delta gets passed back to the nodes on previous layer (that influenced it), weighted by the weights leading into the node with delta

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

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

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

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



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

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

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

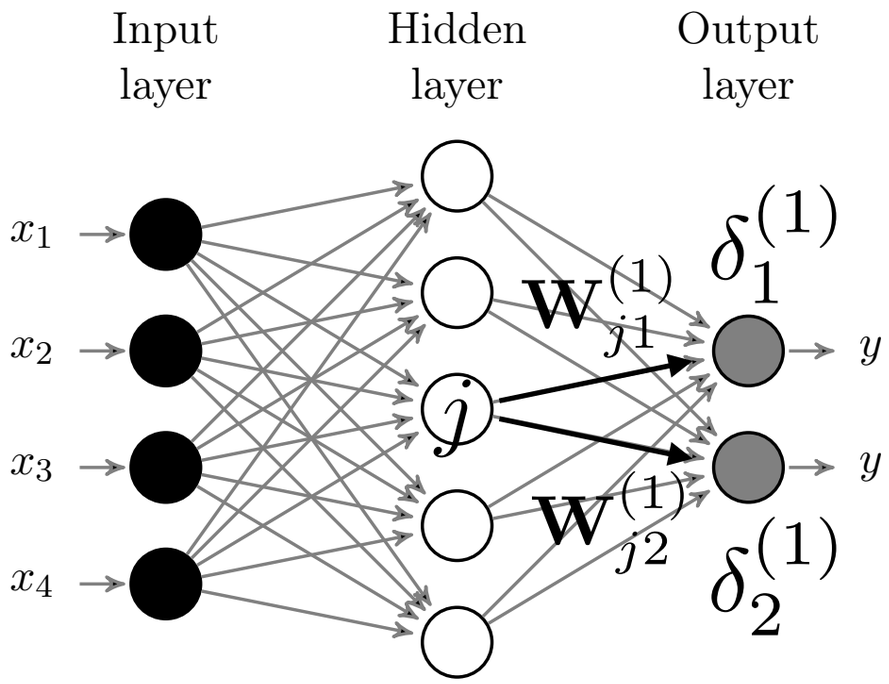
Fully connected update

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

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

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

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



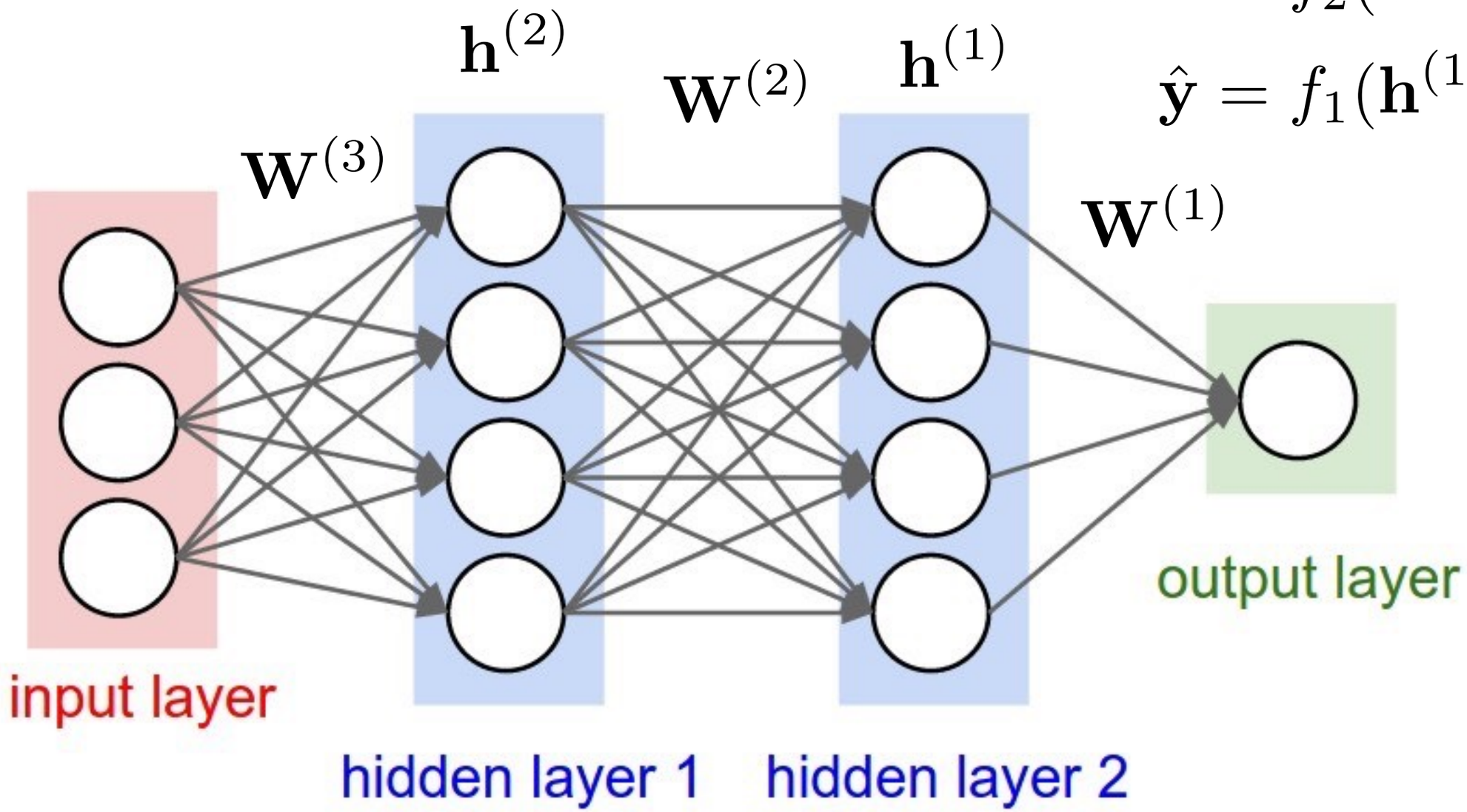
Multi-layer neural network

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?

- Mostly the utility of more layers is an empirical observation; not a lot of theory to support the importance of depth

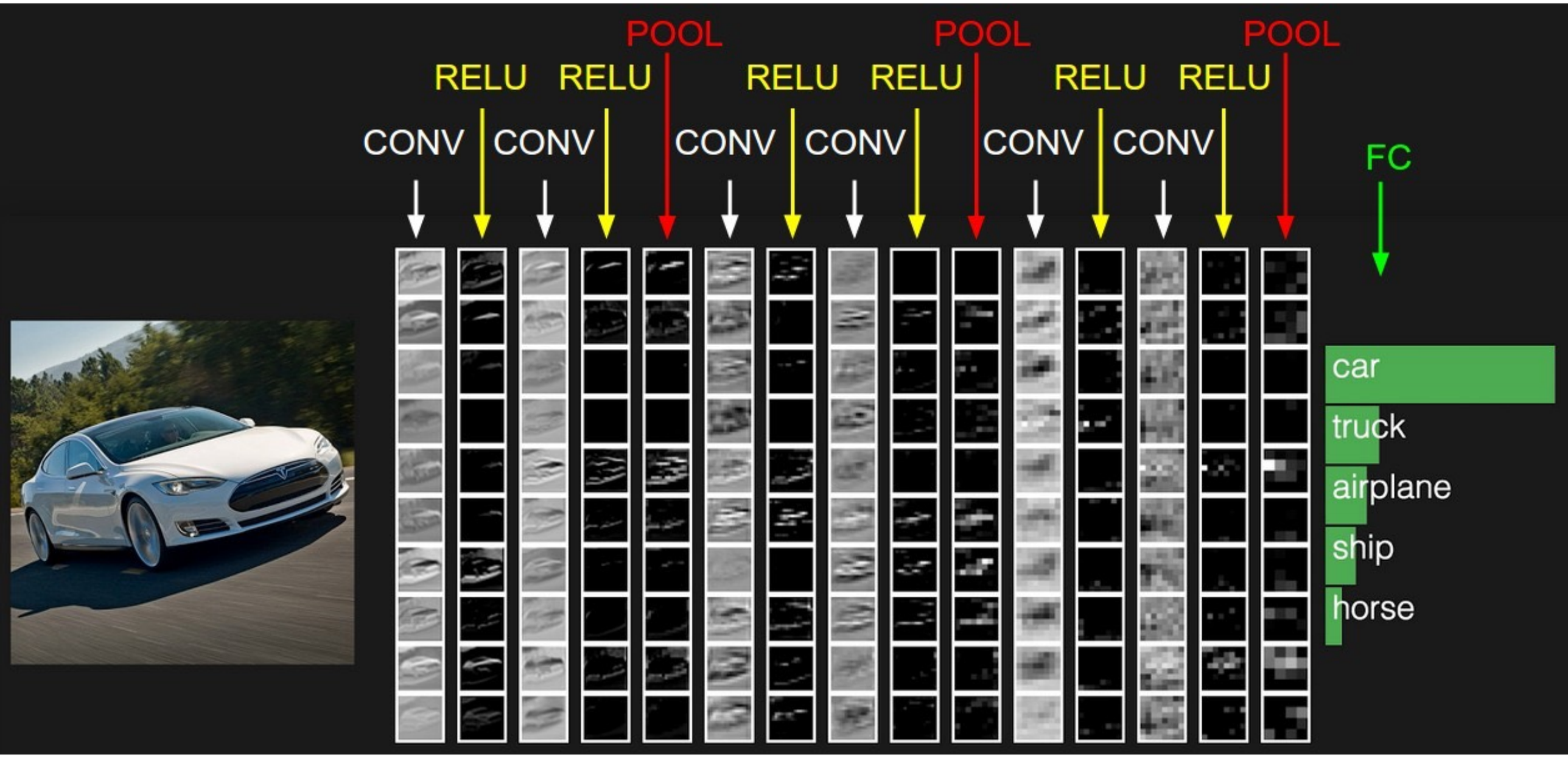
- Depth has shown to be particularly important for convolutional neural networks
  - each convolutional layer summarizes the previous layer, providing a hierarchical structure where depth is intuitively useful

- See for example: “Do Deep Nets Really Need to be Deep?” https://arxiv.org/abs/1312.6184

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

Example of hidden layers in a deeper network



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

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

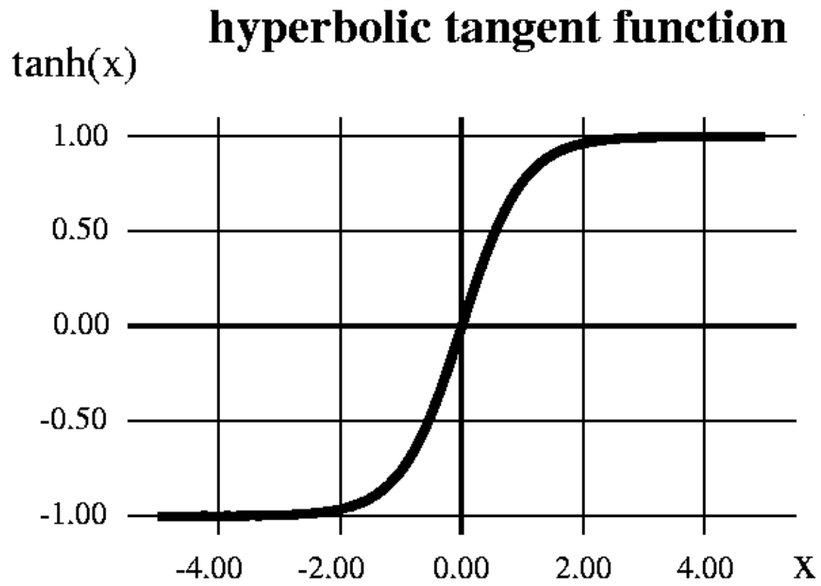
Tanh and rectified linear

- Two more popular transfers are tanh and rectified linear

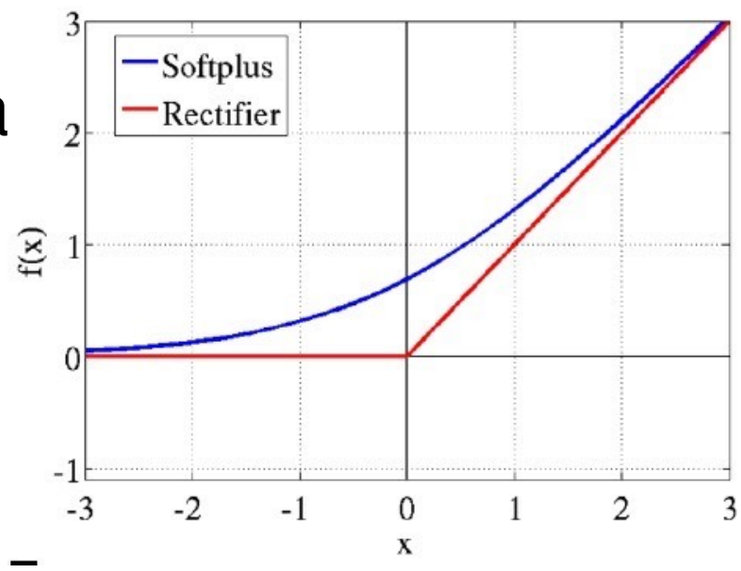
- Tanh is balanced around 0, which seems to help learning

  - usually preferred to sigmoid

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



- Rectified linea

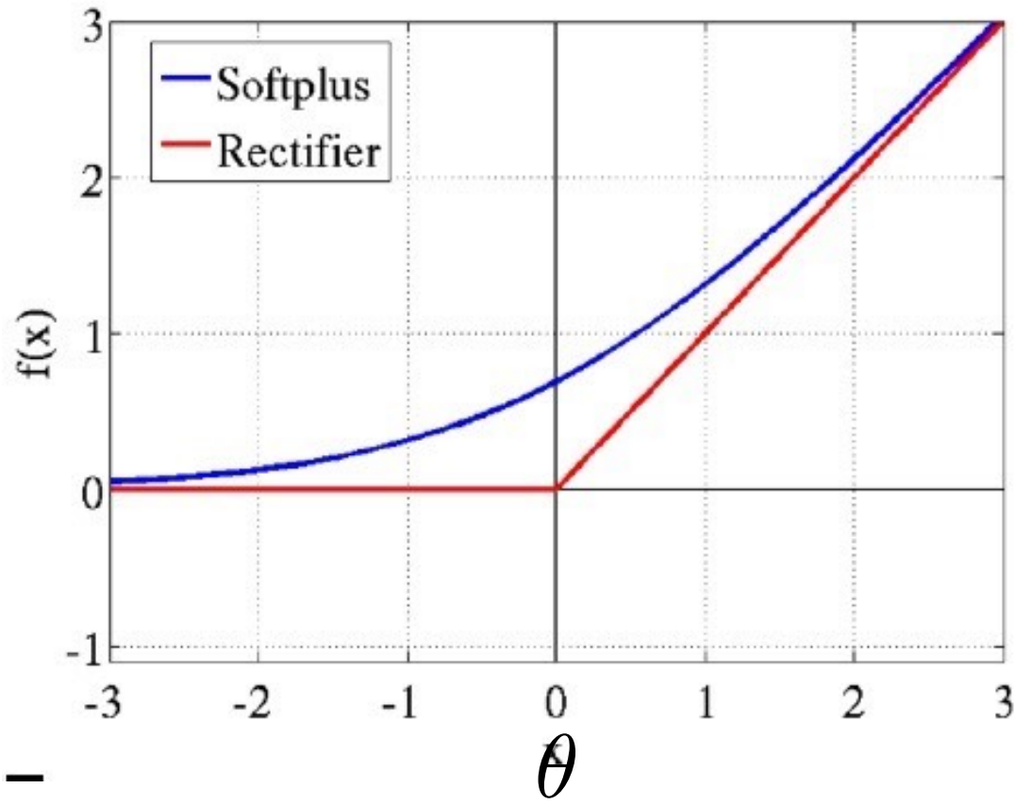


- Binary threshold function (perceptron): less used,

  - 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
- Softplus(x) = ln(1+e^{x})
- Recall our variable is $\theta = \mathbf{x}^\top \mathbf{w}$



- Common strategy: still use sigmoid (or tanh) with cross-entropy in the last output layer, and use rectified linear units in the interior

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

- Derived gradient descent update for two-layers
  - and natural extension to more layers
- 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?)

Exercise: Updating with new samples

- How might we incorporate new samples, into our learned neural network model?
- What if the world is non-stationary, say the distribution drifts?
  - Do you expect this to be more or less reactive than updating with new samples in logistic regression?

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 nodes become zero, they start to filter some of the gradient that is being passed backwards

- See the paper: “Understanding the difficulty of training deep feedforward neural networks”, Glorot and Bengio

Impact of initialization

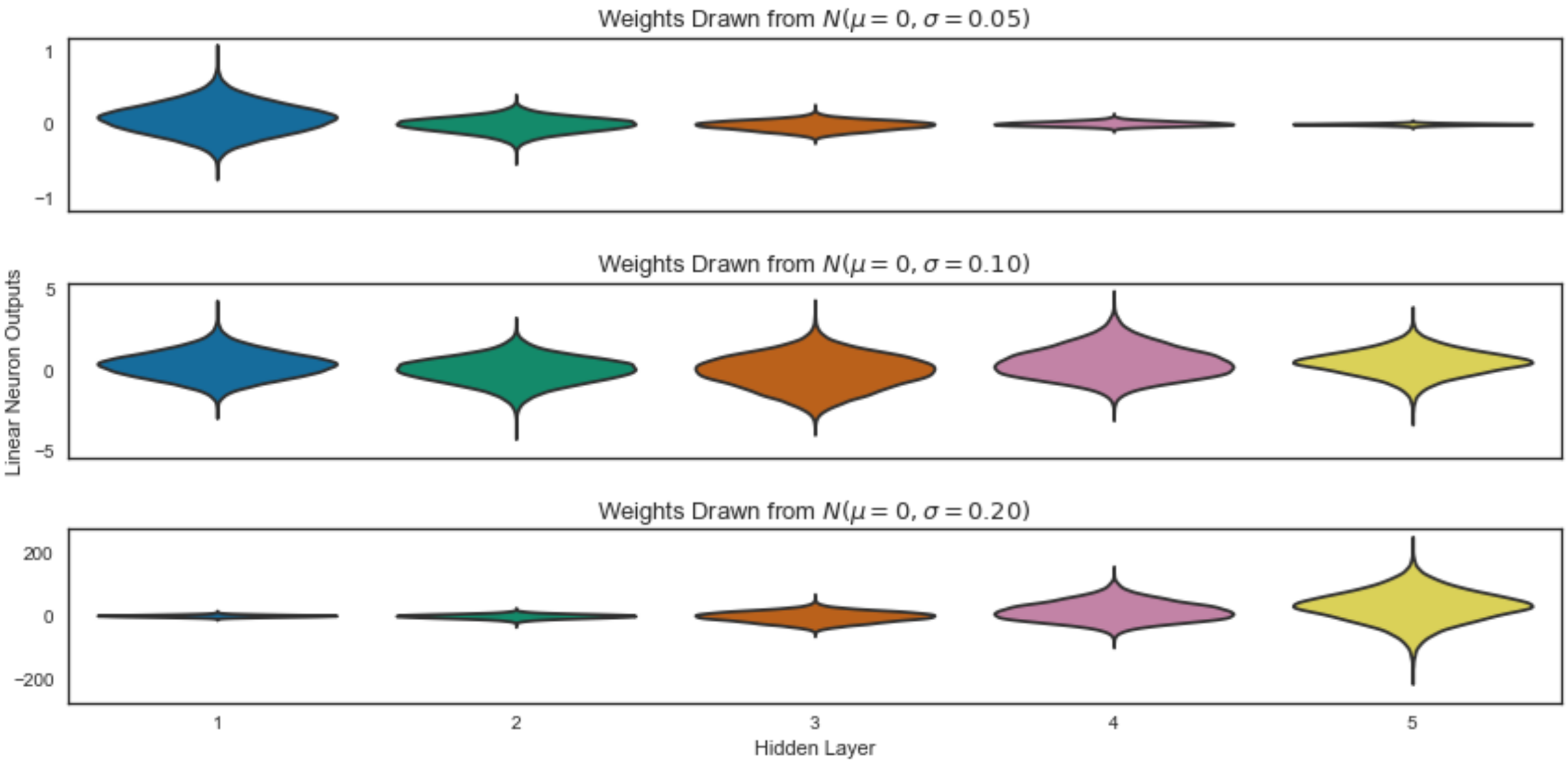
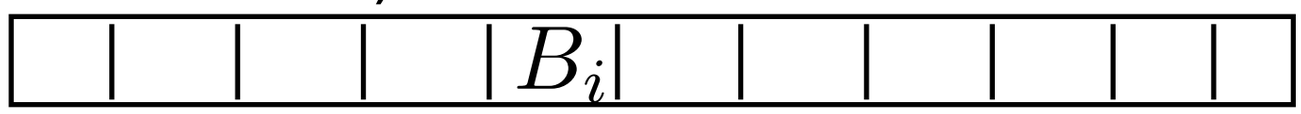


*image from https://intoli.com/blog/neural-network-initialization/

Gradient descent approaches

- Commonly use stochastic gradient descent (SGD) or mini-batch SGD, for a relatively small mini-batch size of say 32

- Mini-batch: update weights using an averaged gradient over a subset of 32 samples (a mini batch B)



- Approach: for one epoch (iterating over the dataset once)
  - SGD with one sample: $\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \nabla c_i(\mathbf{w}_t)$
  - SGD with mini-batch: $\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \sum_{j \in B_i} \nabla c_j(\mathbf{w}_t)$
- If n = 1000, mini-batch b = 10, how many iterations for SGD and mini-batch SGD within one epoch?

Why SGD? Why not batch gradient descent?

- Same reasons as before: computationally wasteful to estimate gradient for the entire dataset, only improves direction minorly over a sample average of the gradients for a much smaller mini-batch

- SGD can more easily jump past saddle points in the objective

- SGD helps prevent overfitting, since it does not converge exactly to minimizers (since we never set step sizes that carefully)

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
  - Adam

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

Whiteboard

- Linear neural network
- Auto-encoder
- Matrix factorization