CS 760: Machine Learning
Neural Networks Continued

University of Wisconsin-Madison

Outline

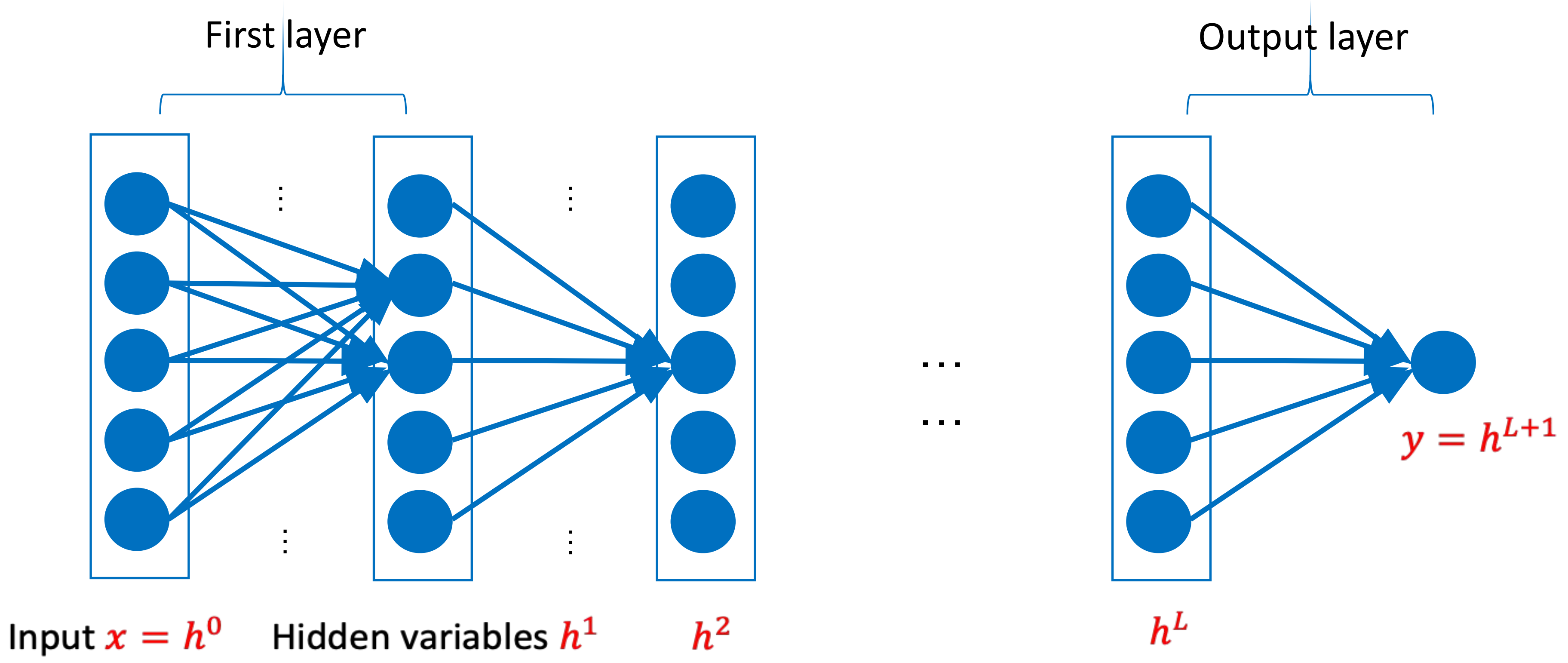
- **Review: Neural Networks**
  - Introduction, Setup, Components, Activations
- **Review: Training Neural Networks**
  - SGD, Computing Gradients, Backpropagation
- **Regularization**
  - Review, Penalties, Augmentation, Deep Net Approaches

Outline

- **Review: Neural Networks**
  - Introduction, Setup, Components, Activations
- **Review: Training Neural Networks**
  - SGD, Computing Gradients, Backpropagation
- **Regularization**
  - Review, Penalties, Augmentation, Deep Net Approaches

Neural Network Components

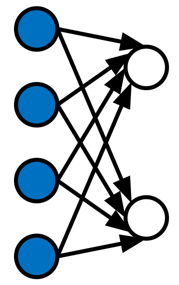
- An $(L+1)$-layer network



Feature Encoding for NNs

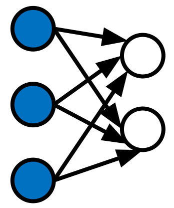
- Nominal features usually a one hot encoding

$$A = \begin{bmatrix}1\\0\\0\\0\end{bmatrix} \quad C = \begin{bmatrix}0\\1\\0\\0\end{bmatrix} \quad G = \begin{bmatrix}0\\0\\1\\0\end{bmatrix} \quad T = \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$$



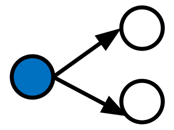
- Ordinal features: use a *thermometer* encoding

$$\text{small}= \begin{bmatrix}1\\0\\0\end{bmatrix} \quad \text{medium}= \begin{bmatrix}1\\1\\0\end{bmatrix} \quad \text{large}= \begin{bmatrix}1\\1\\1\end{bmatrix}$$



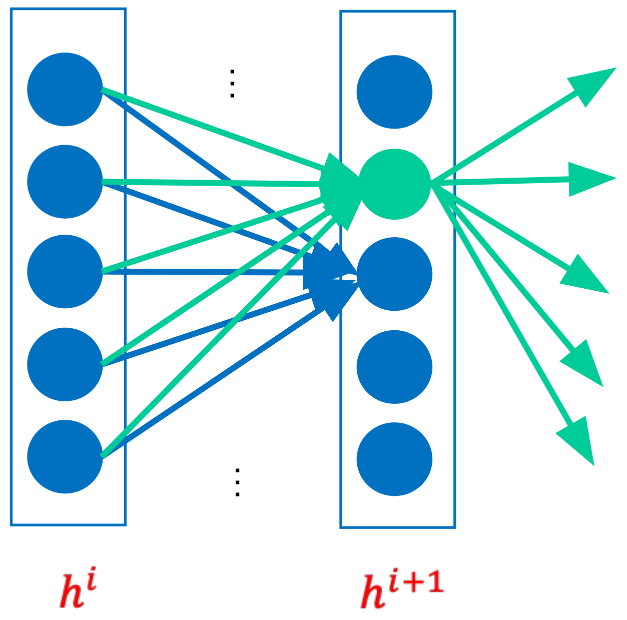
- Real-valued features use individual input units (may want to scale/normalize them first though)

$$\text{precipitation} = \begin{bmatrix}0.68\end{bmatrix}$$



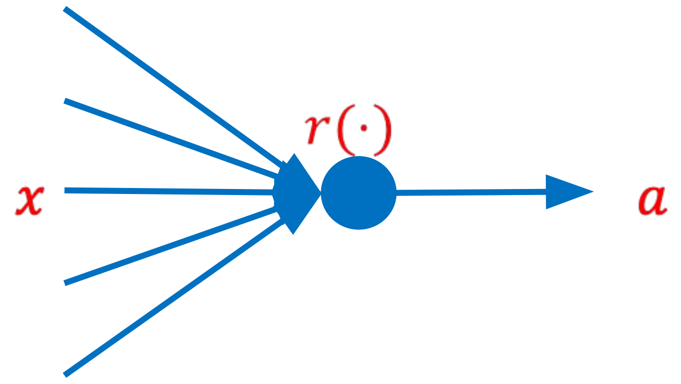
Hidden Layers

- Neuron takes weighted linear combination of the previous representation layer
    - Outputs one value for the next layer

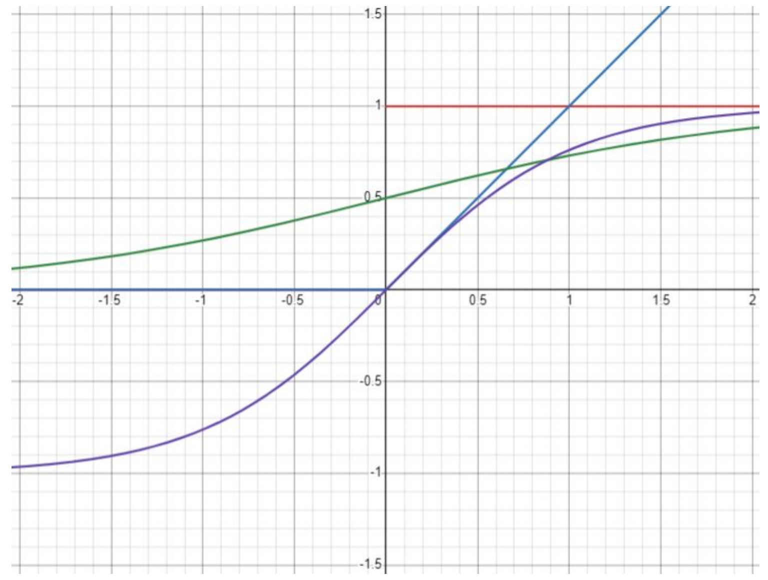


Hidden Layers

- Outputs $a = r(w^T x + b)$



- Typical activation function r
  - threshold $\mathrm{h}(z) = 1_{\{z \geq 0\}}$
  - ReLU $\mathrm{ReLU}(z) = z \cdot t(z) = \max\{0, z\}$
  - sigmoid $\sigma(z) = 1/(1 + \exp(-z))$
  - hyperbolic tangent $\tanh(z) = 2\sigma(2z) - 1$



- Why not **linear activation** functions?
  - Model would be linear.

More on Activations

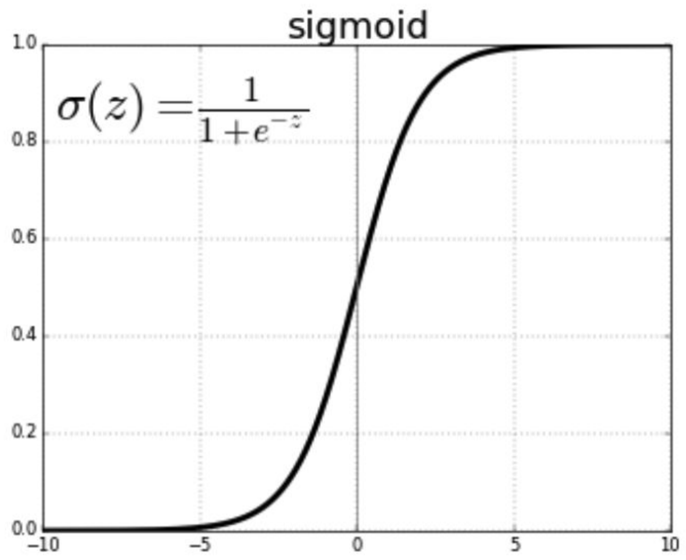
- Outputs $a = r(w^T x + b)$

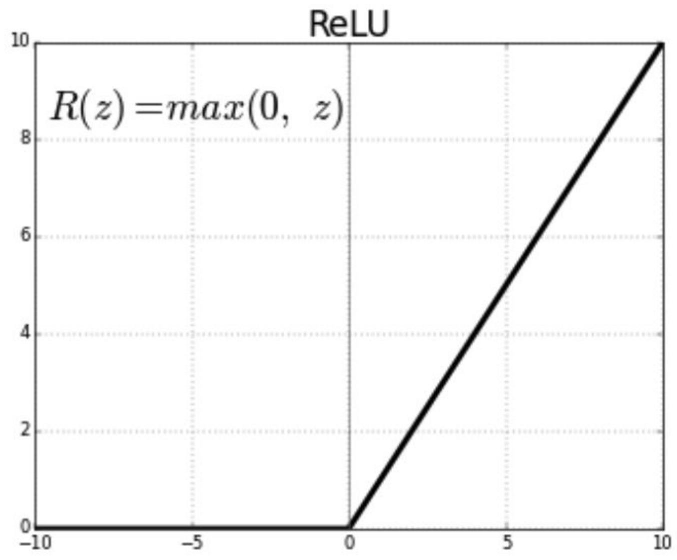
  Bias

  Activation

  Weight

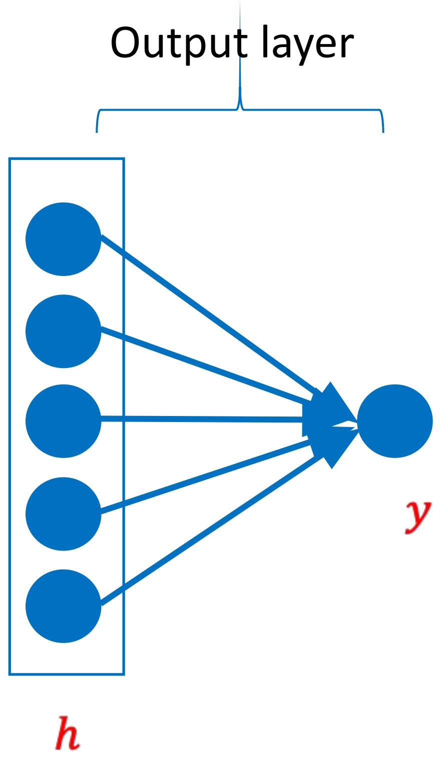
- Consider **gradients**… saturating vs. nonsaturating

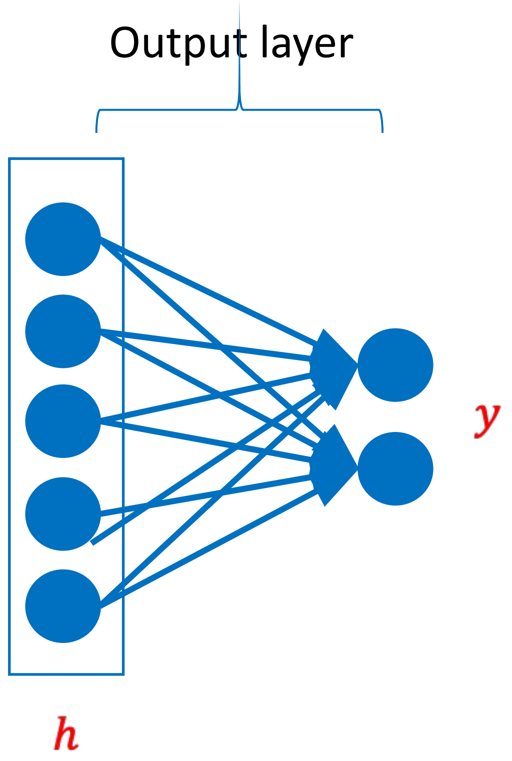




Output Layer: Examples

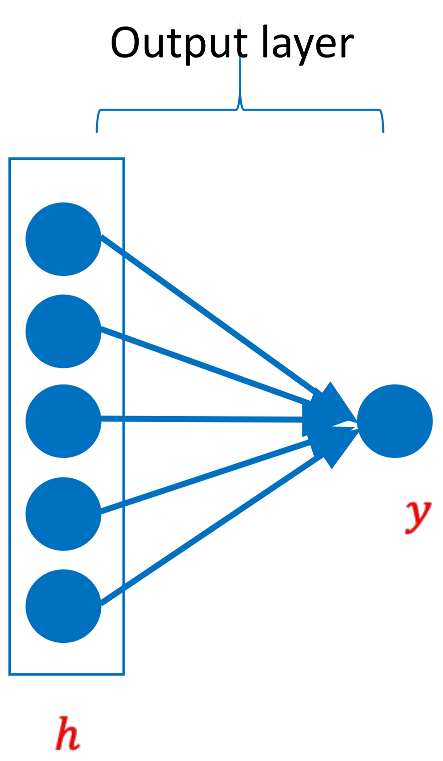
- Regression: $y = w^T h + b$
  - Linear units: no nonlinearity
- Multi-dimensional regression: $y = W^T h + b$
  - Linear units: no nonlinearity

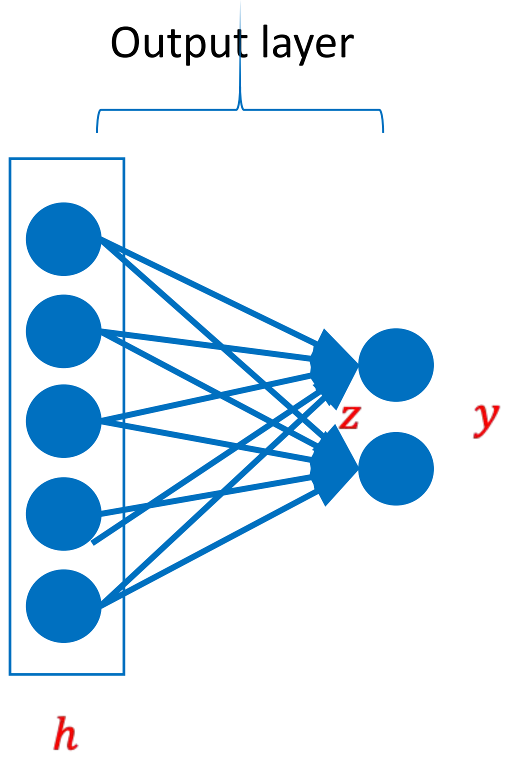




Output Layer: Examples

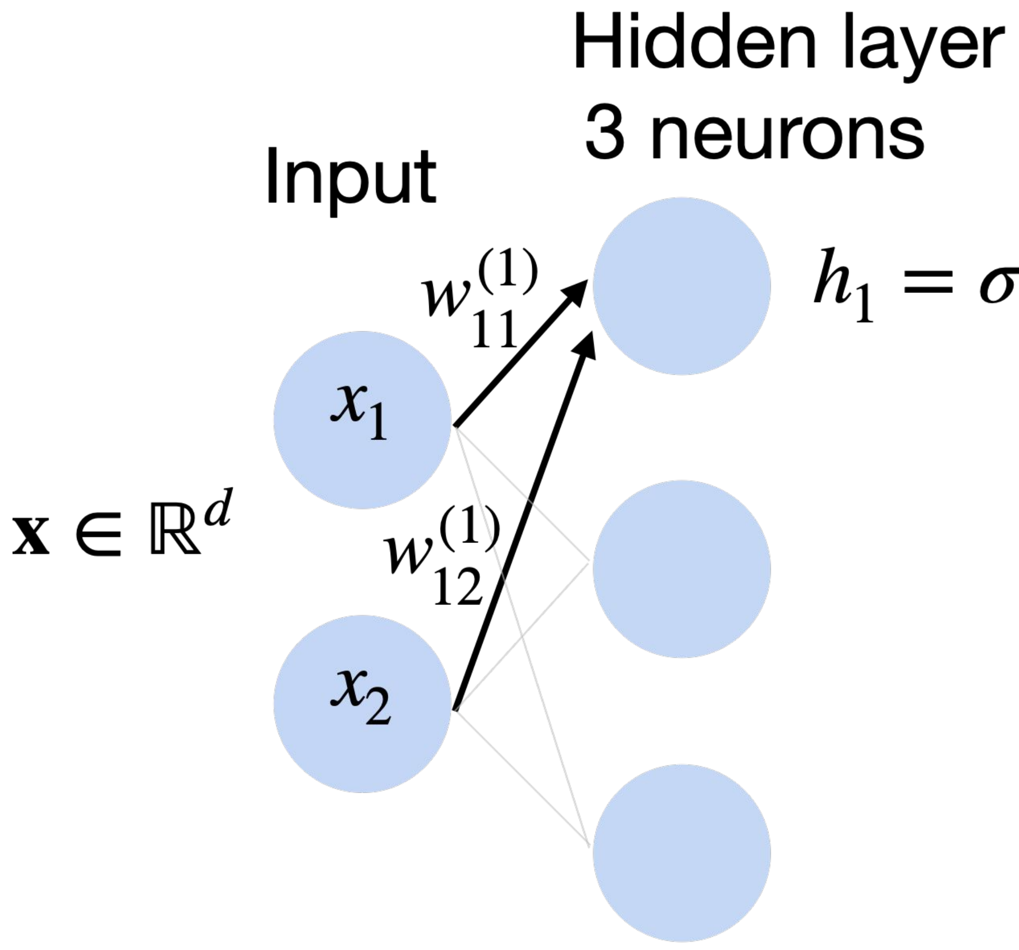
- Binary classification: $y = \sigma(w^T h + b)$
  - Corresponds to using logistic regression on h
- Multiclass classification:
  - $y = \mathrm{softmax}(z)$ where $z = W^T h + b$





MLPs: Multilayer Perceptron

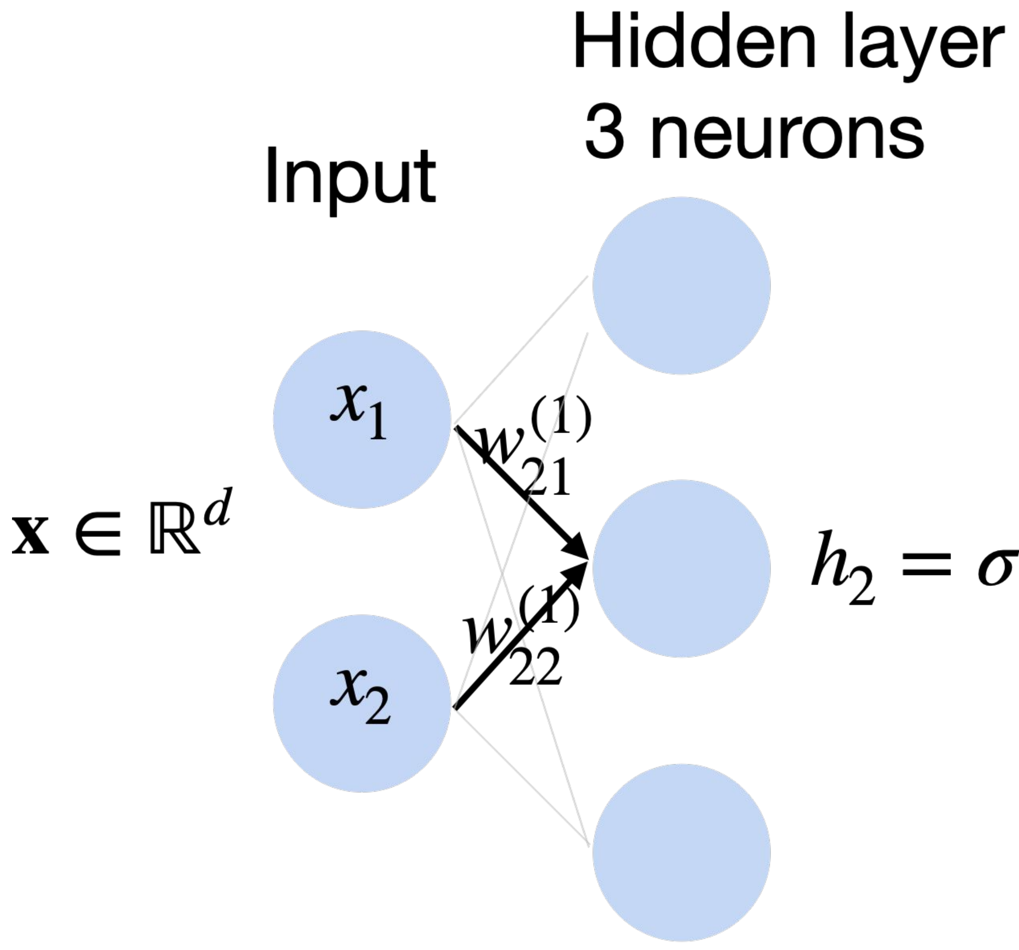
- **Ex**: 1 hidden layer, 1 output layer: depth 2



$$h_1 = \sigma\left(\sum_{i=1}^{d} x_i w_{1i}^{(1)} + b_1\right)$$

MLPs: Multilayer Perceptron

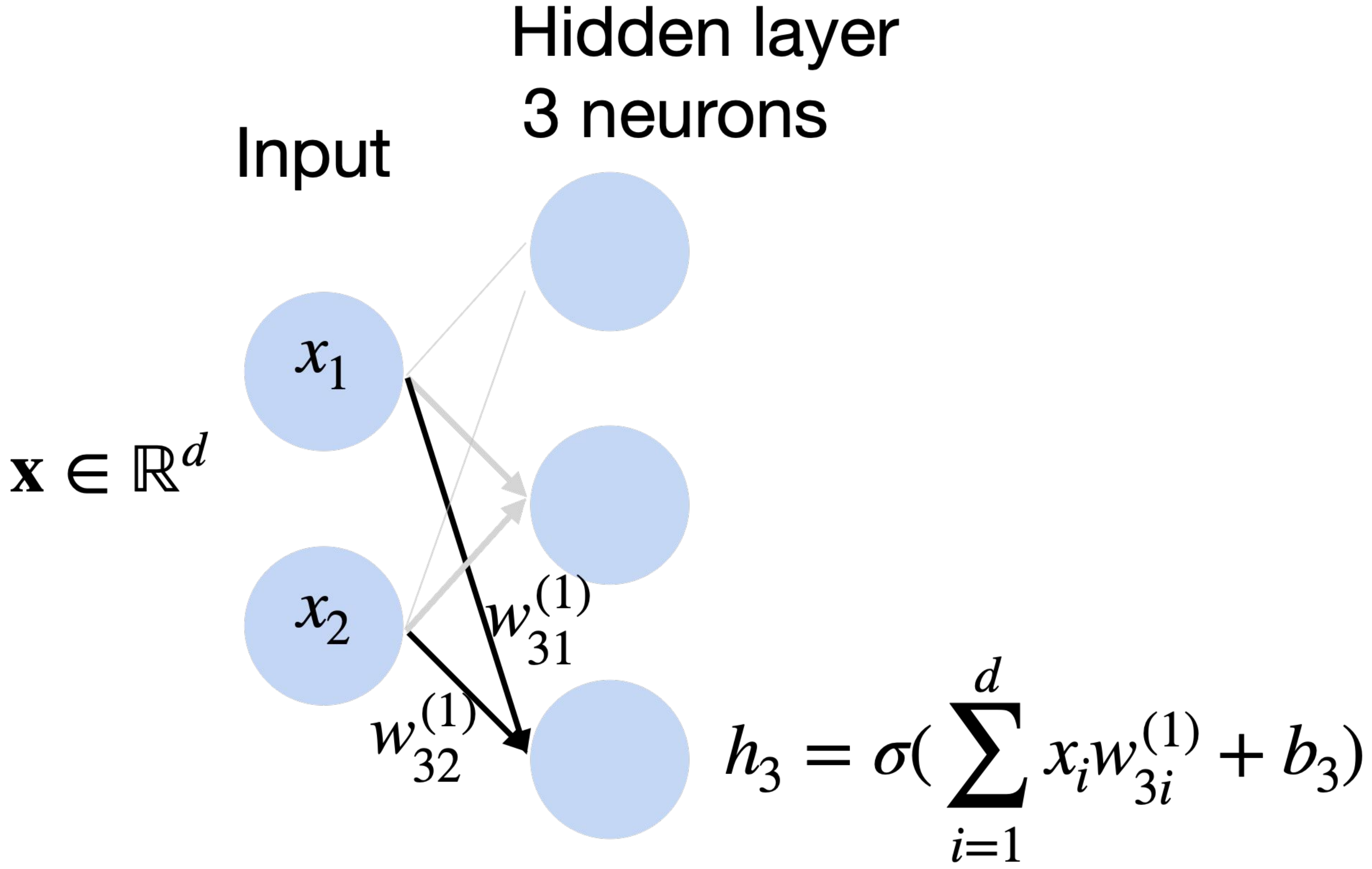
- **Ex**: 1 hidden layer, 1 output layer: depth 2



$$h_2 = \sigma(\sum_{i=1}^{d} x_i w_{2i}^{(1)} + b_2)$$

MLPs: Multilayer Perceptron

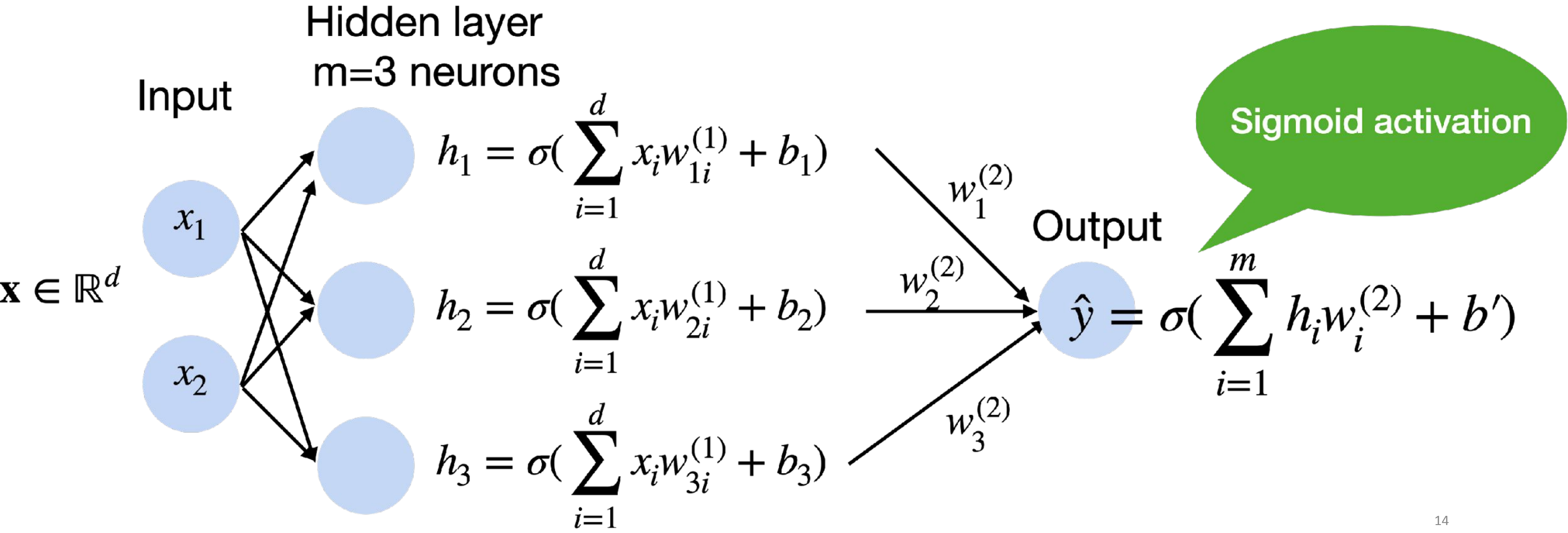
- **Ex**: 1 hidden layer, 1 output layer: depth 2



$$h_3 = \sigma\left(\sum_{i=1}^{d} x_i w_{3i}^{(1)} + b_3\right)$$

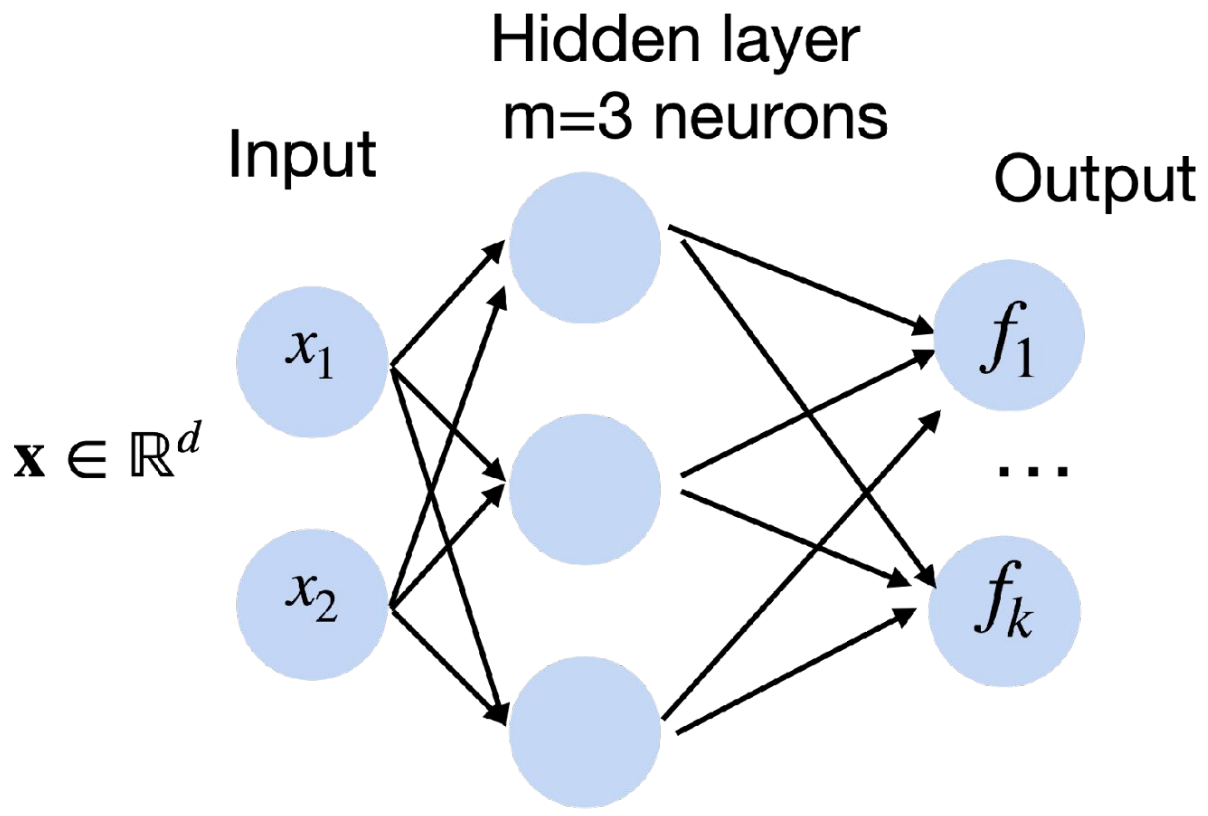
MLPs: Multilayer Perceptron

- **Ex**: 1 hidden layer, 1 output layer: depth 2


Hidden layer
m=3 neurons
Input
x_1
x_2
$\mathbf{x} \in \mathbb{R}^d$
$h_1 = \sigma(\sum_{i=1}^{d} x_i w_{1i}^{(1)} + b_1)$
$h_2 = \sigma(\sum_{i=1}^{d} x_i w_{2i}^{(1)} + b_2)$
$h_3 = \sigma(\sum_{i=1}^{d} x_i w_{3i}^{(1)} + b_3)$
$w_1^{(2)}$
$w_2^{(2)}$
$w_3^{(2)}$
Output
$\hat{y} = \sigma(\sum_{i=1}^{m} h_i w_i^{(2)} + b')$
Sigmoid activation

Multiclass Classification Output

- Create k output units
- Use softmax (just like logistic regression)



$$p(y \mid \mathbf{x}) = \text{softmax}(f)$$

$$= \frac{\exp f_y(x)}{\sum_i^k \exp f_i(x)}$$

Break & Quiz

Q: Which of the following activation functions are typically **NOT** used in practice, and why not?

-

a. sigmoid function $\sigma(z) = {}^{1}/_{1+\exp(-z)}$

b. hyperbolic tangent $\tanh(z) = 2\sigma(z) - 1$

c. threshold function $H(z) = 1_{z \geq 0}$

d. linear (identity) function z

e. Rectified linear unit $\mathrm{ReLU}(z) = zH(z) = \max\{z, 0\}$

Q: Which of the following activation functions are **typically** NOT used in practice, and why not?

- a. sigmoid function $\sigma(z) = {}^{1}/_{1+\exp(-z)}$
- b. hyperbolic tangent $\tanh(z) = 2\sigma(z) - 1$
- c. threshold function $H(z) = 1_{z\geq 0}$ $\Leftarrow$ hard to optimize (derivative=0 everywhere except z=0, where it is undefined)
- d. linear (identity) function z $\Leftarrow$ composition of linear functions is no more expressive than a single linear function
- e. Rectified linear unit $\mathrm{ReLU}(z) = zH(z) = \max\{z, 0\}$

Outline

- **Review: Neural Networks**
  - Introduction, Setup, Components, Activations
- **Review: Training Neural Networks**
  - SGD, Computing Gradients, Backpropagation
- **Regularization**
  - Review, Penalties, Augmentation, Deep Net Approaches

Training Neural Networks

Training is done in the usual way: pick a loss and optimize it

- **Example**: 2 scalar weights

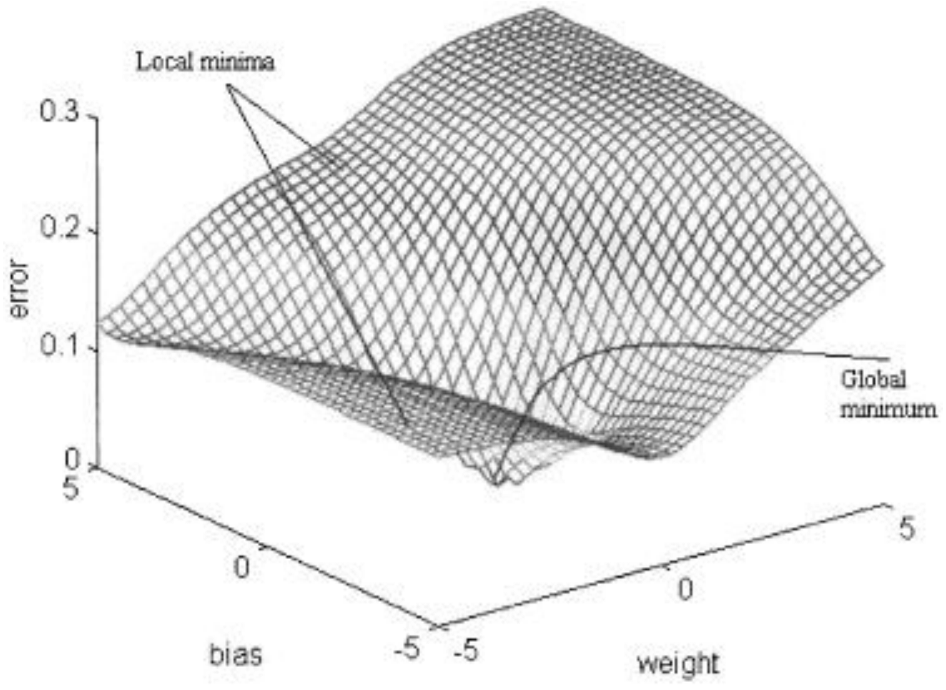


figure from Cho & Chow, *Neurocomputing* 1999

Training Neural Networks with SGD

Algorithm:

- Input dataset $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$
- Initialize weights
- Until stopping criterion is met:
  - For each training point $(x^{(i)}, y^{(i)})$ do
    - Compute prediction: $\hat{y}^{(i)} = f_w(x^{(i)})$ $\longleftarrow$ **Forward Pass**
    - Compute loss: $L^{(i)} = L(\hat{y}^{(i)}, y^{(i)})$ $\longleftarrow$ e.g. negative log-likelihood (NLL) loss $L(\hat{y}, y) = -y \log \hat{y} - (1-y)\log(1-\hat{y})$
    - Compute gradient: $\nabla_w L^{(i)} = \left(\partial_{w_1} L^{(i)}, \partial_{w_2} L^{(i)}, \ldots, \partial_{w_m} L^{(i)}\right)^{\top}$ $\longleftarrow$ **Backward Pass**
    - Update weights: $w \leftarrow w - \alpha \nabla_w L^{(i)}$ $\longleftarrow$ SGD step

Training Neural Networks with **minibatch** SGD

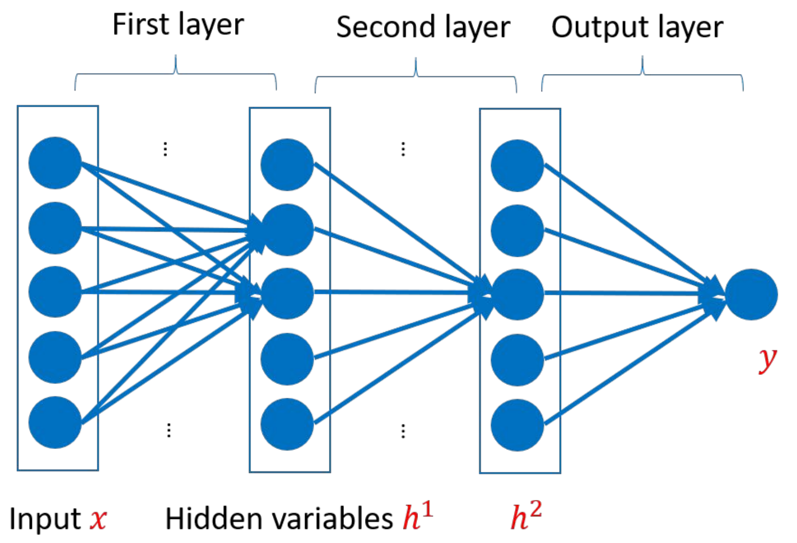
Algorithm:

- Input dataset $D = \left\{\left(x^{(1)}, y^{(1)}\right), \ldots, \left(x^{(n)}, y^{(n)}\right)\right\}$
- Initialize weights
- Until stopping criterion is met:
  - Sample a **batch of b training points** $i_1, \ldots, i_b$
  - Compute predictions: $\left\{\hat{y}^{(i_1)}, \ldots, \hat{y}^{(i_b)}\right\} = \left\{f_w\left(x^{(i_1)}\right), \ldots, f_w\left(x^{(i_b)}\right)\right\}$
  - Compute avg. loss: $L^{(i_1,\ldots,i_b)} = \frac{1}{b}\sum_{j=1}^{b} L(\hat{y}^{(i_j)}, y^{(i_j)})$
  - Compute gradient: $\nabla_w L^{(i_1,\ldots,i_b)} = \left(\partial_{w_1} L^{(i_1,\ldots,i_b)}, \ldots, \partial_{w_m} L^{(i_1,\ldots,i_b)}\right)^\top$
  - Update weights: $w \leftarrow w - \alpha \nabla_w L^{(i_1,\ldots,i_b)}$

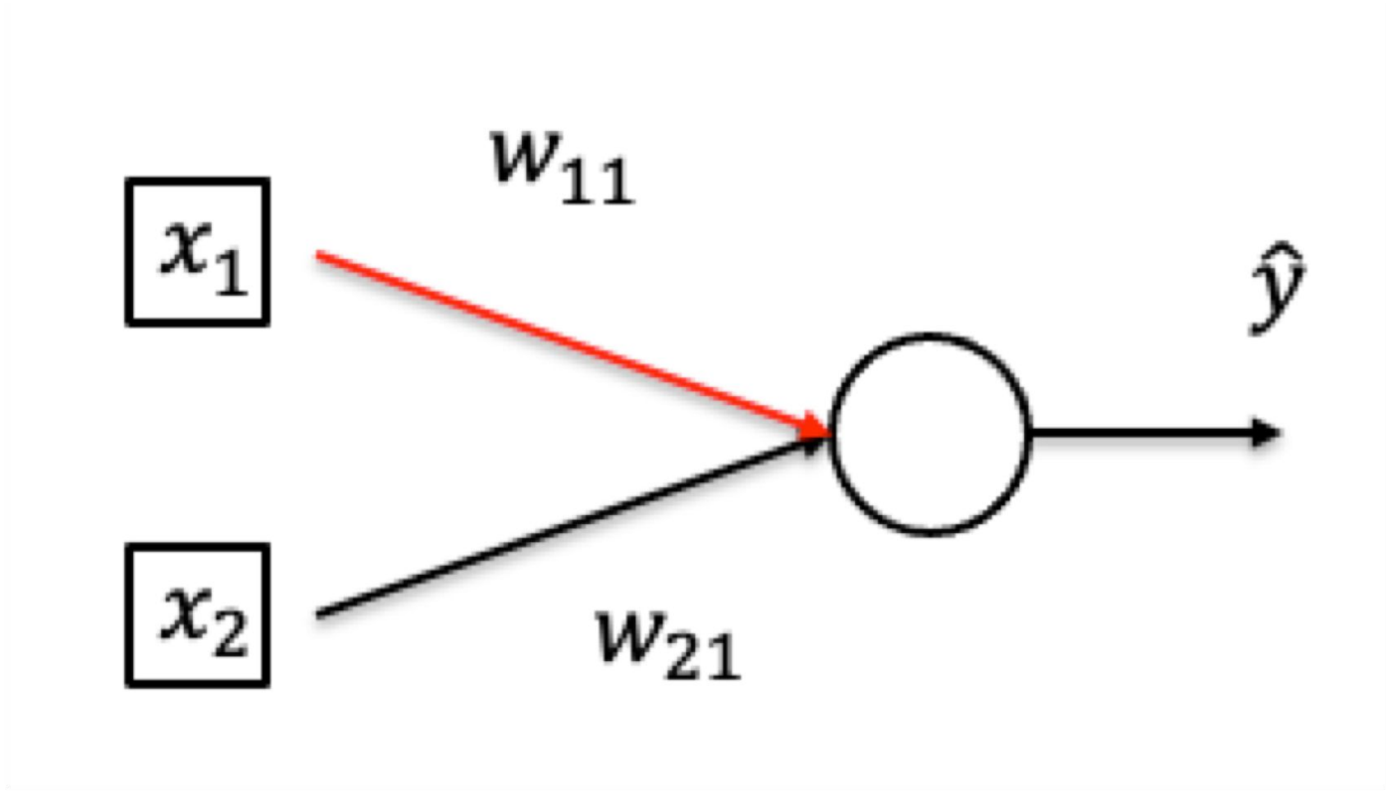
Training Neural Networks: Chain Rule

- Will need to compute terms like: $\frac{\partial L}{\partial w_1}$
  - But, L is a composition of:
    - Loss with output y
    - Output itself a composition of softmax with outer layer
    - Outer layer a combination of outputs from previous layer
    - Outputs from prev. layer a composition of activations and linear functions…
- Need the **chain rule**!
  - Suppose $L = L(g_1, \ldots, g_k) \quad g_j = g_j(w_1, \ldots, w_p)$
  - Then,

$$\frac{\partial L}{\partial w_i} = \sum_{j=1}^{k} \frac{\partial L}{\partial g_j} \frac{\partial g_j}{\partial w_i}$$

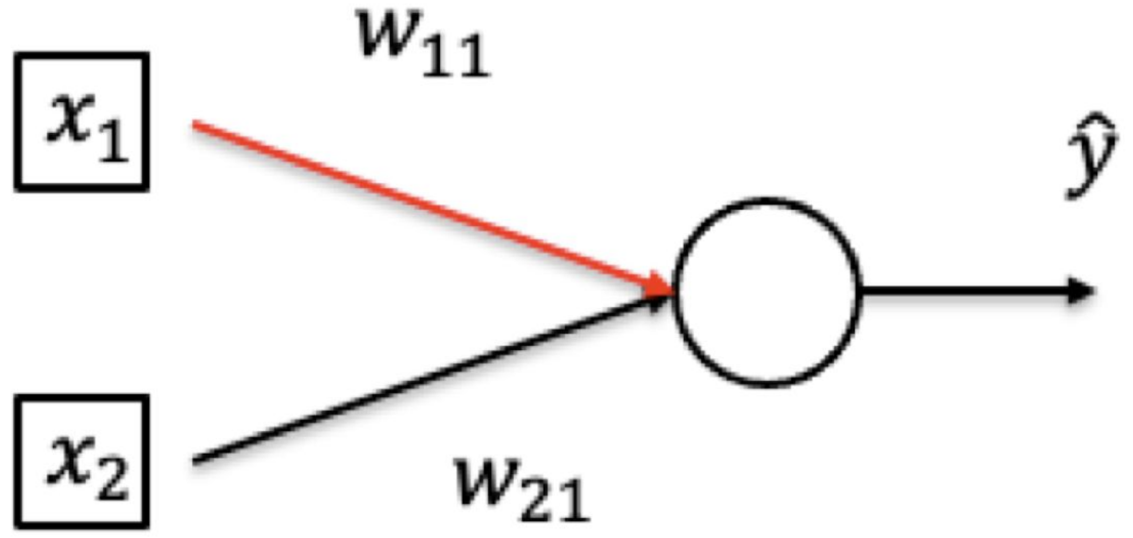


Computing Gradients



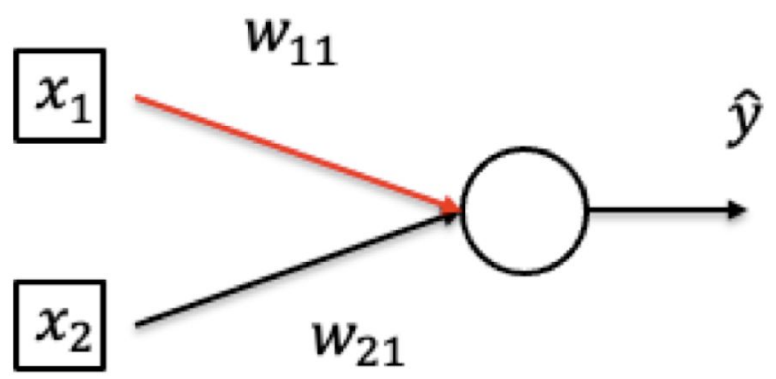
Want to compute $\frac{\partial \ell(\mathbf{x}, y)}{\partial w_{11}}$

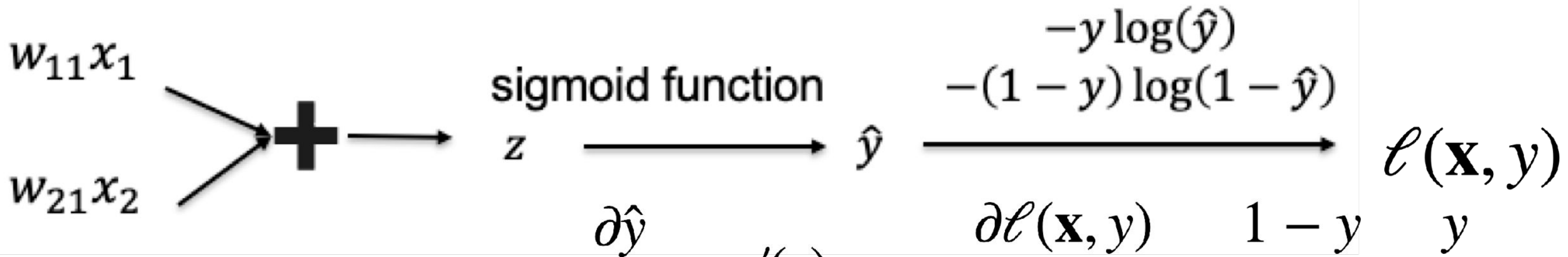
Computing Gradients



$$\begin{tikzcd}
w_{11}x_1 \arrow[dr] & & & & \\
& + \arrow[r] & z \arrow[r, "\text{sigmoid function}"] & \hat{y} \arrow[r, "\overbrace{-y\log(\hat{y}) - (1-y)\log(1-\hat{y})}^{\text{negative log-likelihood (NLL) loss}}"] & \ell(\mathbf{x}, y) \\
w_{21}x_2 \arrow[ur] & & & &
\end{tikzcd}$$

Computing Gradients

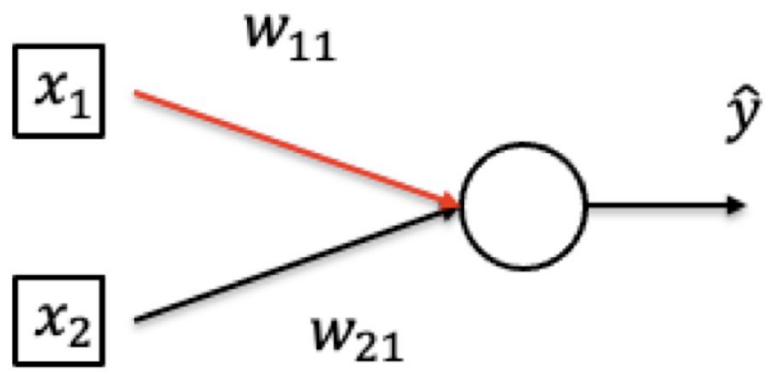




$$\frac{\partial \hat{y}}{\partial z} = \sigma'(z) \qquad \frac{\partial \ell(\mathbf{x}, y)}{\partial \hat{y}} = \frac{1-y}{1-\hat{y}} - \frac{y}{\hat{y}}$$

By chain rule: $$\frac{\partial l}{\partial w_{11}} = \frac{\partial l}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial w_{11}}$$

Computing Gradients

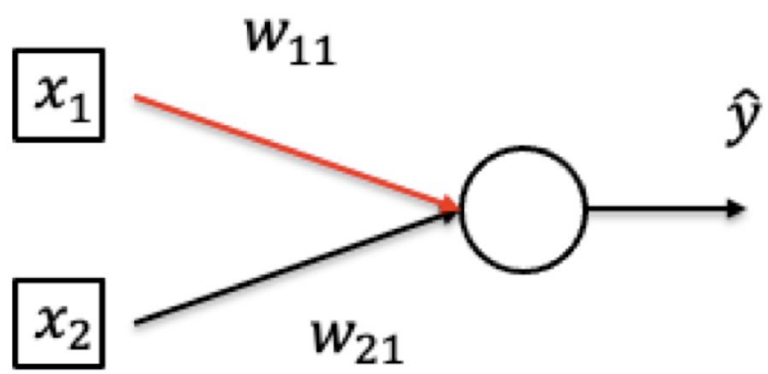


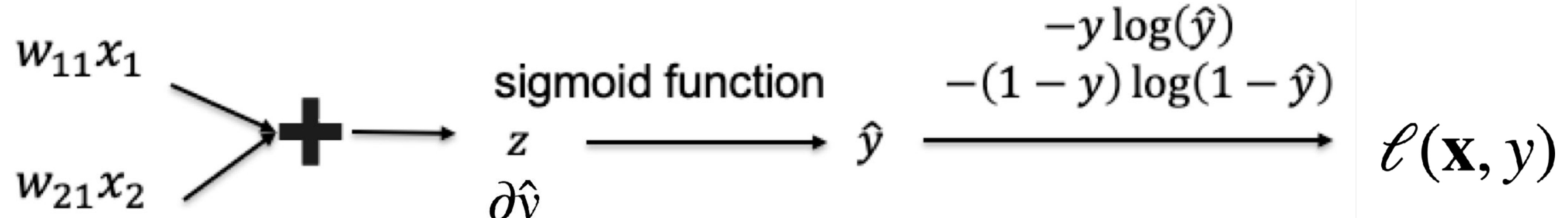
$$\begin{tikzcd} w_{11}x_1 \arrow[dr] & & & & \\ & + \arrow[r] & z \arrow[r, "\text{sigmoid function}"] & \hat{y} \arrow[r, "-y\log(\hat{y}) - (1-y)\log(1-\hat{y})"] & \ell(\mathbf{x}, y) \\ w_{21}x_2 \arrow[ur] & & & & \end{tikzcd}$$

$$\frac{\partial \hat{y}}{\partial z} = \sigma'(z) \qquad \frac{\partial \ell(\mathbf{x}, y)}{\partial \hat{y}} = \frac{1-y}{1-\hat{y}} - \frac{y}{\hat{y}}$$

By chain rule: $$\frac{\partial l}{\partial w_{11}} = \frac{\partial l}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} x_1$$

Computing Gradients



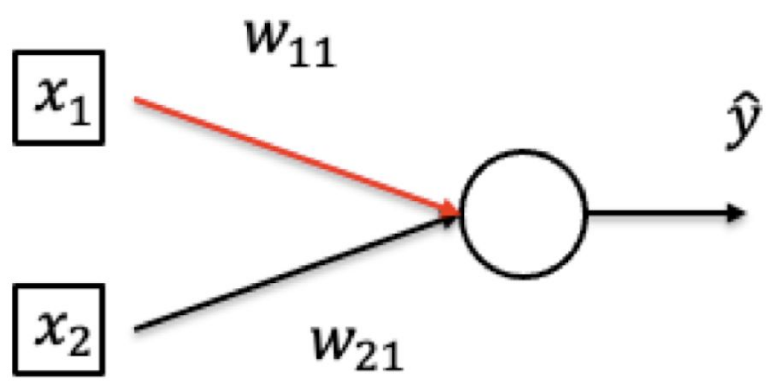


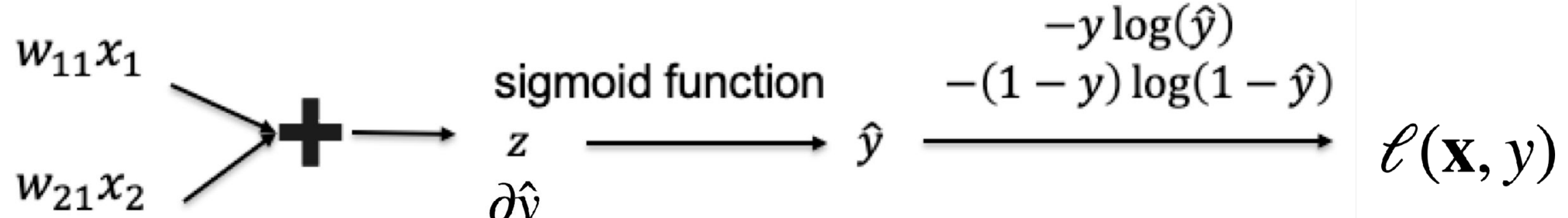
$$\frac{\partial \hat{y}}{\partial z} = \sigma'(z) = \sigma(z)(1-\sigma(z))$$

By chain rule:

$$\frac{\partial l}{\partial w_{11}} = \frac{\partial l}{\partial \hat{y}} \; \hat{y}(1-\hat{y})x_1$$

Computing Gradients

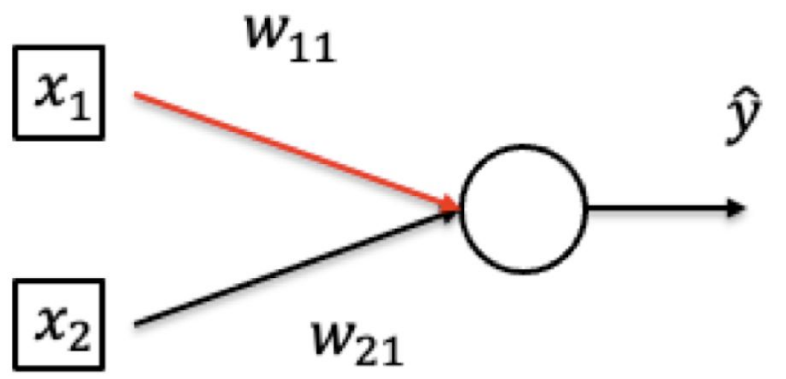




$$\frac{\partial \hat{y}}{\partial z} = \sigma'(z) = \sigma(z)(1-\sigma(z))$$

By chain rule: $$\frac{\partial l}{\partial w_{11}} = \left(\frac{1-y}{1-\hat{y}} - \frac{y}{\hat{y}}\right)\hat{y}(1-\hat{y})x_1$$

Computing Gradients

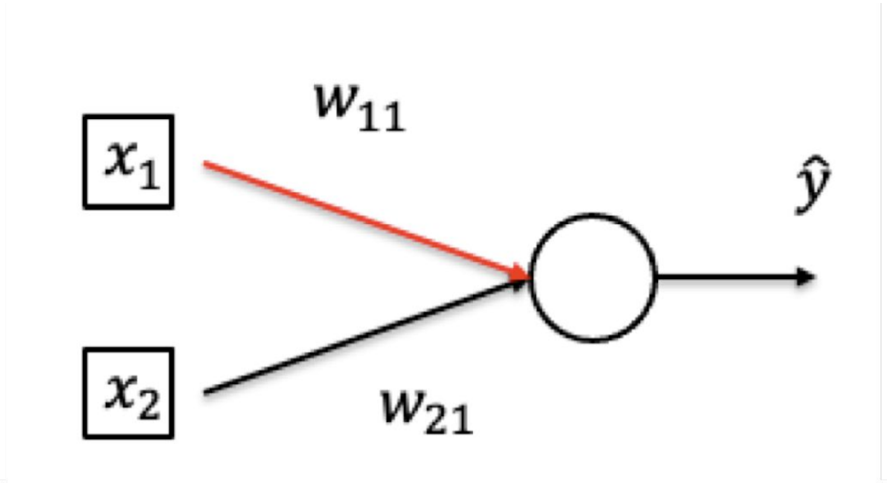


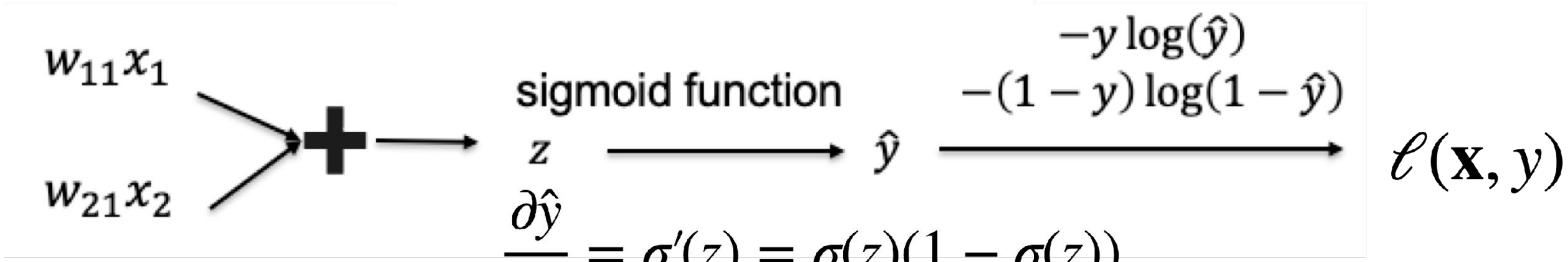
$$\begin{tikzcd} w_{11}x_1 \arrow[dr] & & & & \\ & + \arrow[r] & z \arrow[r, "\text{sigmoid function}"] & \hat{y} \arrow[r, "-y\log(\hat{y}) - (1-y)\log(1-\hat{y})"] & \ell(\mathbf{x}, y) \\ w_{21}x_2 \arrow[ur] & & & & \end{tikzcd}$$

$$\frac{\partial \hat{y}}{\partial z} = \sigma'(z) = \sigma(z)(1 - \sigma(z))$$

By chain rule: $$\frac{\partial l}{\partial w_{11}} = (\hat{y} - y)x_1$$

Computing Gradients



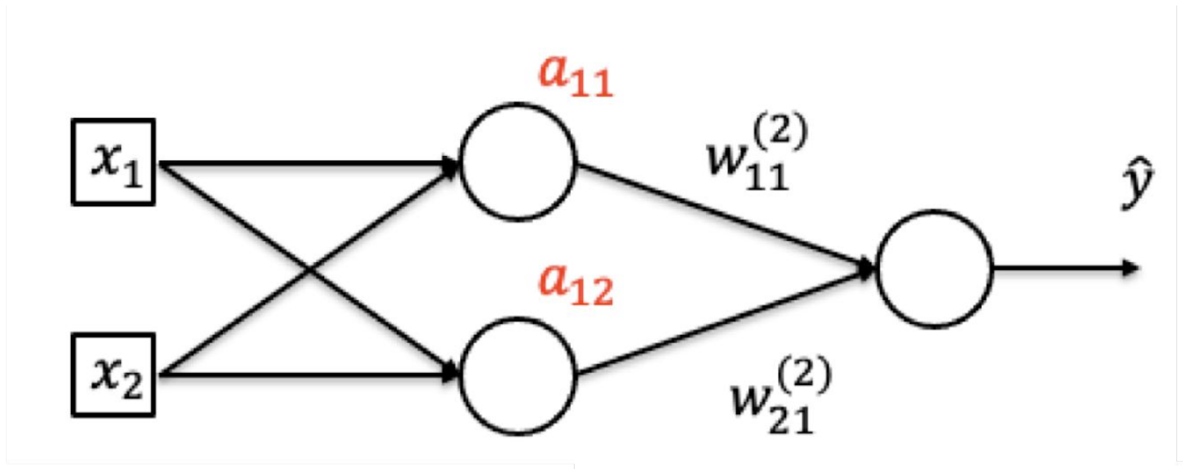


$$\frac{\partial \hat{y}}{\partial z} = \sigma'(z) = \sigma(z)(1-\sigma(z))$$

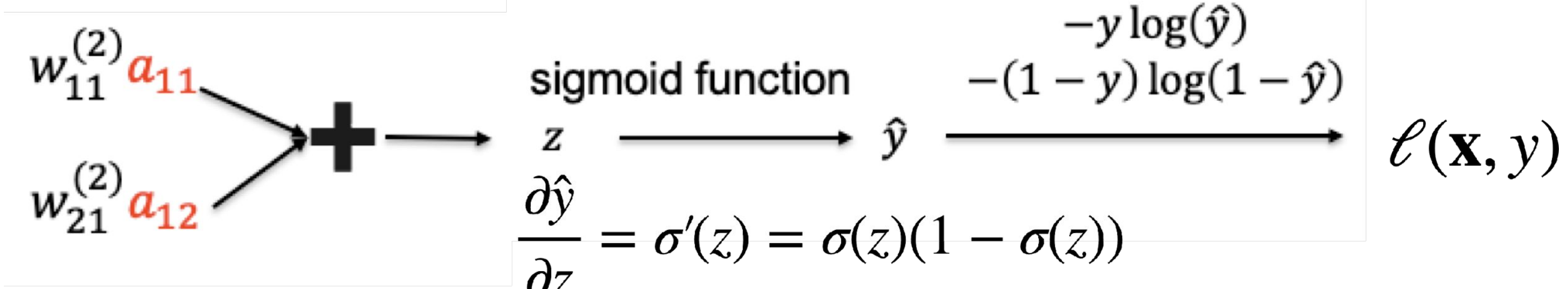
By chain rule:

$$\frac{\partial l}{\partial x_1} = \frac{\partial l}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} w_{11} = (\hat{y} - y) w_{11}$$

Computing Gradients: More Layers



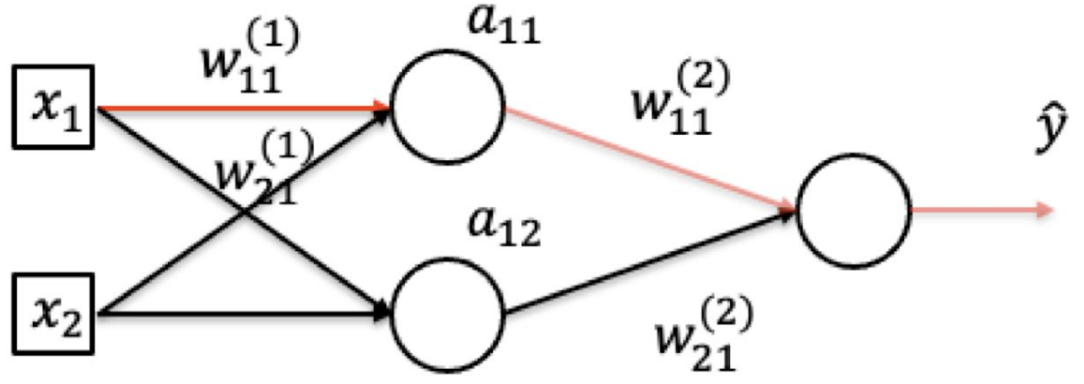
Make it deeper



$$\frac{\partial \hat{y}}{\partial z} = \sigma'(z) = \sigma(z)(1-\sigma(z))$$

By chain rule: $$\frac{\partial l}{\partial a_{11}} = (\hat{y}-y)w_{11}^{(2)}, \quad \frac{\partial l}{\partial a_{12}} = (\hat{y}-y)w_{21}^{(2)}$$

Computing Gradients: More Layers

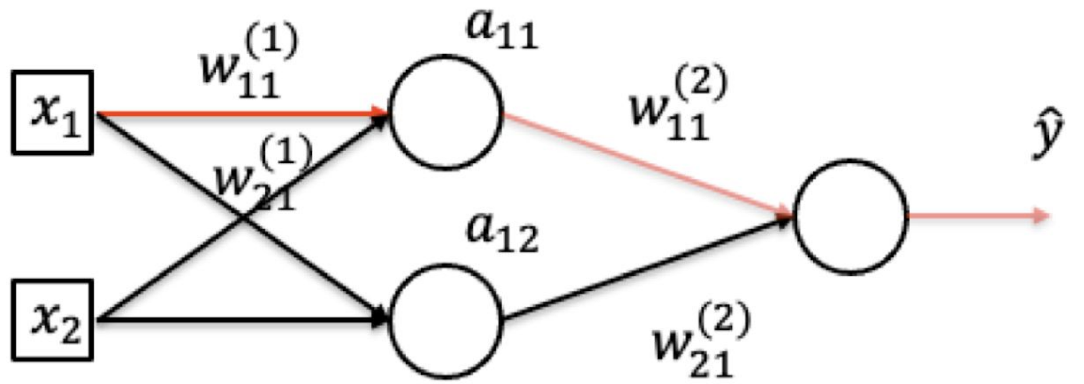


$$\begin{tikzcd}
w_{11}^{(1)} x_1 \arrow[dr] & & & & \\
& + \arrow[r] & z_{11} \arrow[r, "\sigma(z_{11})"] & a_{11} \arrow[r] & l(\boldsymbol{x}, y) \\
w_{21}^{(1)} x_2 \arrow[ur] & & & &
\end{tikzcd}$$

$$\frac{\partial a_{11}}{\partial z_{11}} = \sigma'(z_{11}) \qquad \frac{\partial l}{\partial a_{11}} = (\hat{y} - y) w_{11}^{(2)}$$

By chain rule: $$\frac{\partial l}{\partial w_{11}} = \frac{\partial l}{\partial a_{11}} \frac{\partial a_{11}}{\partial w_{11}^{(1)}} = (\hat{y} - y) w_{11}^{(2)} \frac{\partial a_{11}}{\partial w_{11}^{(1)}}$$

Computing Gradients: More Layers

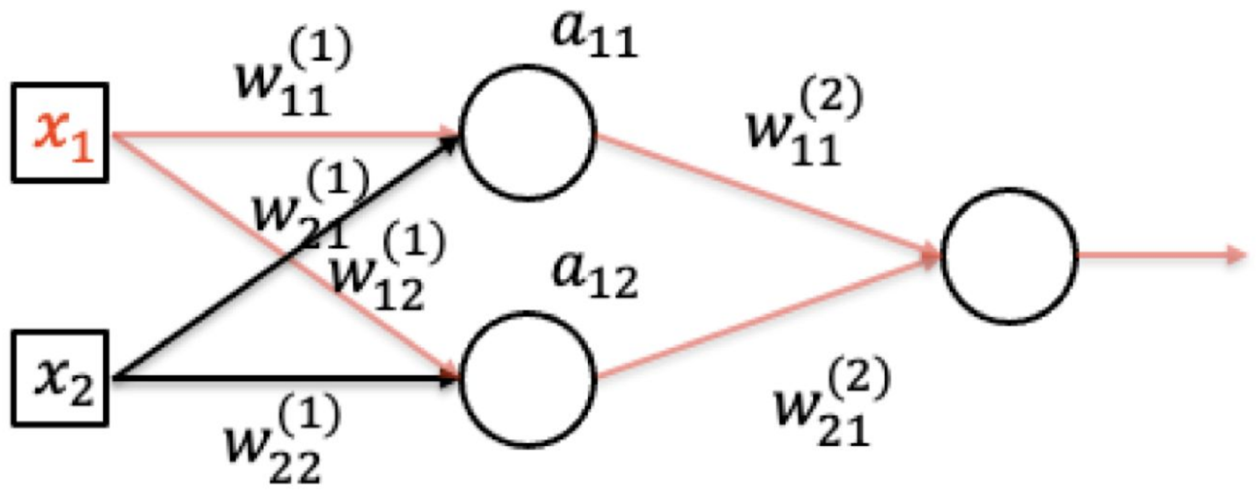


$$\begin{tikzcd}
w_{11}^{(1)} x_1 \arrow[dr] & & & & \\
 & + \arrow[r] & z_{11} \arrow[r, "\sigma(z_{11})"] & a_{11} \arrow[r] & l(\boldsymbol{x}, y) \\
w_{21}^{(1)} x_2 \arrow[ur] & & & &
\end{tikzcd}$$

$$\frac{\partial a_{11}}{\partial z_{11}} = \sigma'(z_{11}) \qquad \frac{\partial l}{\partial a_{11}} = (\hat{y} - y) w_{11}^{(2)}$$

By chain rule: $$\frac{\partial l}{\partial w_{11}} = \frac{\partial l}{\partial a_{11}} \frac{\partial a_{11}}{\partial w_{11}^{(1)}} = (\hat{y} - y) w_{11}^{(2)} a_{11} (1 - a_{11}) x_1$$

Computing Gradients: More Layers



$$\begin{tikzcd} w_{11}^{(1)}x_1 \arrow[dr] & & & & \\ & + \arrow[r] & z_{11} \arrow[r, "\sigma(z_{11})"] & a_{11} \arrow[r] & l(\boldsymbol{x}, y) \\ w_{21}^{(1)}x_2 \arrow[ur] & & & & \end{tikzcd}$$

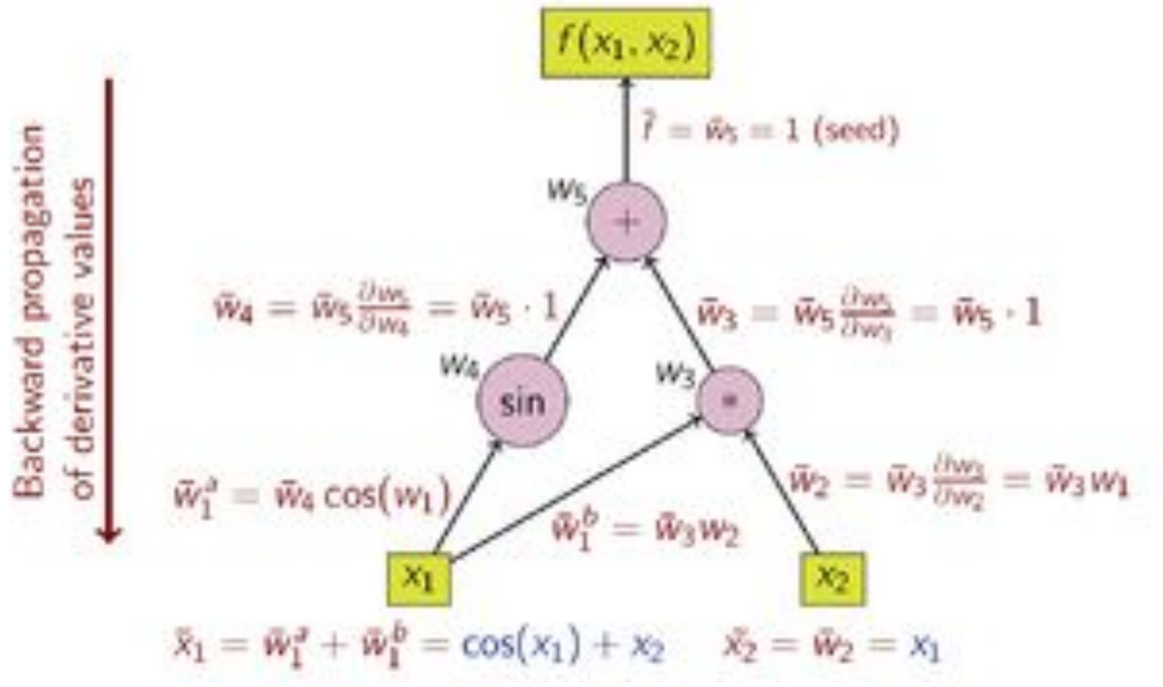
$$\frac{\partial a_{11}}{\partial z_{11}} = \sigma'(z_{11}) \qquad \frac{\partial l}{\partial a_{11}} = (\hat{y} - y)w_{11}^{(2)}$$

By chain rule:

$$\boxed{\frac{\partial l}{\partial x_1} = \frac{\partial l}{\partial a_{11}}\frac{\partial a_{11}}{\partial x_1} + \frac{\partial l}{\partial a_{12}}\frac{\partial a_{12}}{\partial x_1}}$$

Backpropagation

- So to compute derivative w.r.t specific weights we **propagate** loss information **back** through the network
- Today we do this by automatic differentiation (**autodiff**) for arbitrarily complex computation graphs
- Go backwards from top to bottom, recursively computing gradients



Backward propagation of derivative values
$f(x_1, x_2)$
$\bar{f} = \bar{w}_5 = 1$ (seed)
w_5
+
$\bar{w}_4 = \bar{w}_5 \frac{\partial w_5}{\partial w_4} = \bar{w}_5 \cdot 1$
$\bar{w}_3 = \bar{w}_5 \frac{\partial w_5}{\partial w_3} = \bar{w}_5 \cdot 1$
w_4
sin
w_3
*
$\bar{w}_1^a = \bar{w}_4 \cos(w_1)$
$\bar{w}_2 = \bar{w}_3 \frac{\partial w_3}{\partial w_2} = \bar{w}_3 w_1$
$\bar{w}_1^b = \bar{w}_3 w_2$
x_1
x_2
$\bar{x}_1 = \bar{w}_1^a + \bar{w}_1^b = \cos(x_1) + x_2$
$\bar{x}_2 = \bar{w}_2 = x_1$

Wiki

Break & Quiz

True or False: Backprop is an optimization algorithm.

a. **False;** some optimization algorithms use the output of backprop.

True or False: Backprop is a weight-updating algorithm

a. **False;** gradient-based optimization algorithms like SGD use the output of backprop to update the weights.

True or False: Backprop is a gradient-computation algorithm

a. True

Outline

- **Review: Neural Networks**
  - Introduction, Setup, Components, Activations
- **Review: Training Neural Networks**
  - SGD, Computing Gradients, Backpropagation
- **Regularization**
  - Review, Penalties, Augmentation, Deep Net Approaches

Review: Regularization

Any method to **prevent overfitting** or **help optimization**

So far, we've seen one approach: penalty-based regularization

$$\underset{\theta}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} loss_i(\theta) + \lambda R(\theta)$$

- $R(\theta) = \frac{1}{2} ||\theta||_2^2$ (Ridge)
- $R(\theta) = ||\theta||_1$ (LASSO)

Penalty-based regularization of convex losses

Helps **prevent overfitting** by reducing the hypothesis space

- equivalent to constraint formulation for convex penalties R:

$$\underset{\theta}{\operatorname{argmin}} \frac{1}{n}\sum_{i=1}^{n} loss_i(\theta) + \lambda R(\theta) = \underset{R(\theta)\le\tau}{\operatorname{argmin}} \frac{1}{n}\sum_{i=1}^{n} loss_i(\theta)$$

- LASSO known to yield **sparse** parameters θ

Ridge **helps optimization** by making the objective **strongly convex**

- gives objective positive curvature at every point
- gradient descent on strongly convex + smooth losses has suboptimality $O\left((1-\kappa)^T\right)$ after T iterations, vs. $O(1/T)$ with regular convexity

Ridge (ℓ_2) regularization of general losses

- Gradient of regularized objective

$$\nabla \hat{L}_R(\theta) = \nabla \hat{L}(\theta) + \lambda\theta$$

- Gradient descent update

$$\theta \leftarrow \theta - \eta \nabla \hat{L}_R(\theta) = \theta - \eta \, \nabla \hat{L}(\theta) - \eta\lambda\theta$$

$$= (1 - \eta\lambda)\theta - \eta \, \nabla \hat{L}(\theta)$$

- In words, **weight decay**

ℓ_1 regularization of general losses

- Gradient of regularized objective

$$\nabla \hat{L}_R(\theta) = \nabla \hat{L}(\theta) + \lambda \mathrm{sign}(\theta)$$

where $\mathbf{sign}$ applies to each element in θ

- Gradient descent update

$$\theta \leftarrow \theta - \eta \nabla \hat{L}_R(\theta) = \theta - \eta\, \nabla \hat{L}(\theta) - \eta \lambda \mathrm{sign}(\theta)$$

- Effects like sparsity less well-understood than convex case

Data Augmentation

Augmentation: transform + add new samples to dataset

- Transformations: based on domain
- Idea: build **invariances** into the model
  - Ex: if all images have same alignment, model learns to use it
- Keep the label the same!

Data Augmentation: Examples

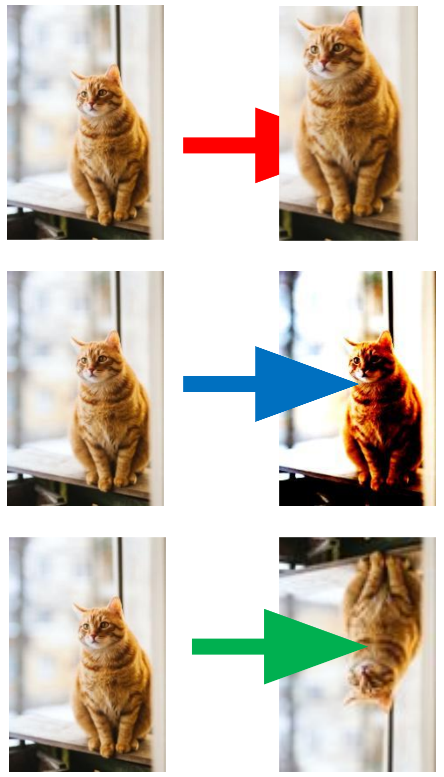
Examples of transformations for images

- **Crop** (and zoom)
- **Color** (change contrast/brightness)
- **Rotations+** (translate, stretch, shear, etc.)

Many more possibilities. Combine as well!

Q: how to know what performs best?

A: cross-validation



Data Augmentation: Other Domains

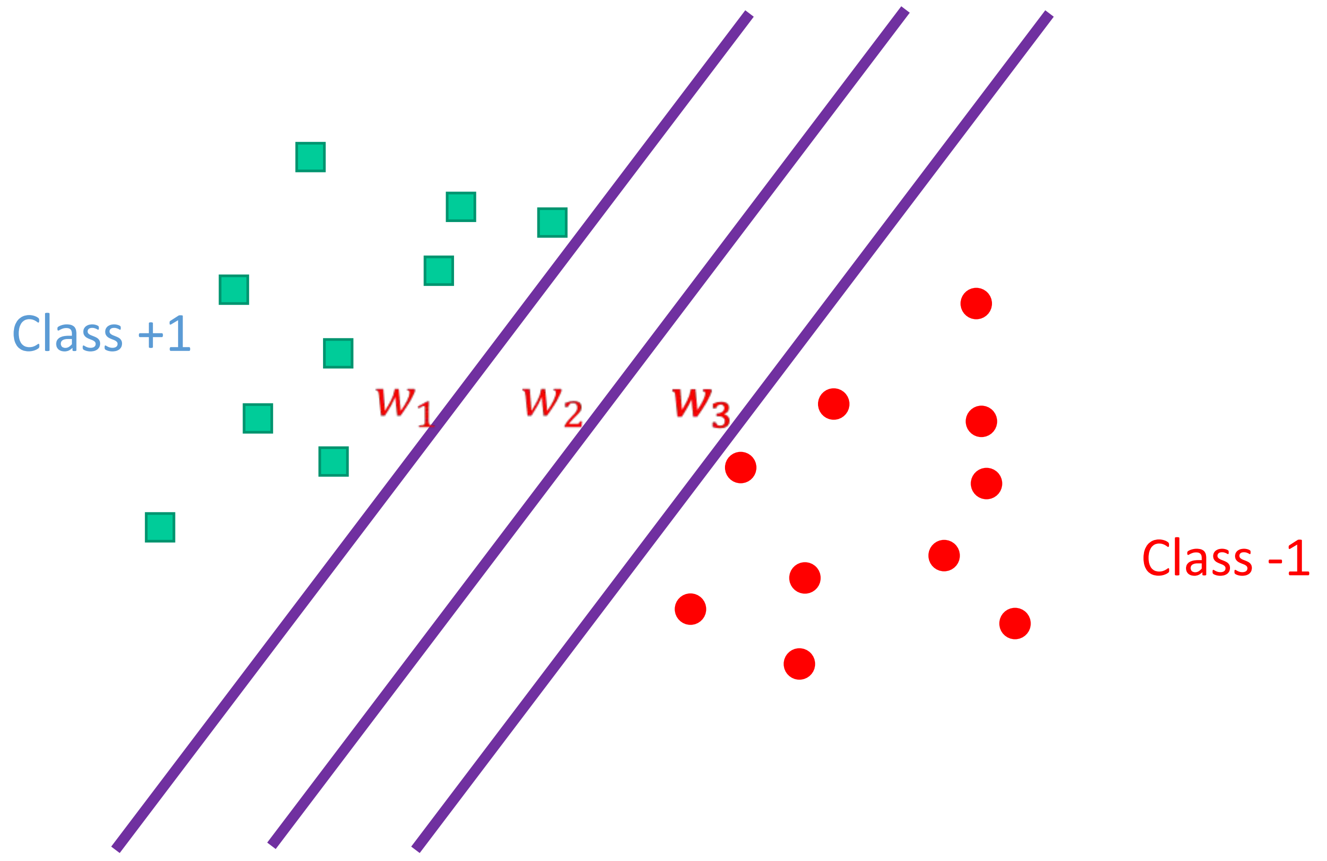
Not just for image data. For example, on text:

- Substitution
  - E.g. “It is a **great** day” → “It is a **wonderful** day”
  - Use a thesaurus for particular words
  - Or use a model. Pre-trained word embeddings, language models

- Back-translation
  - “Given the low budget and production limitations, this movie is very good.”
    → “There are few budget items and production limitations to make this film a really good one”

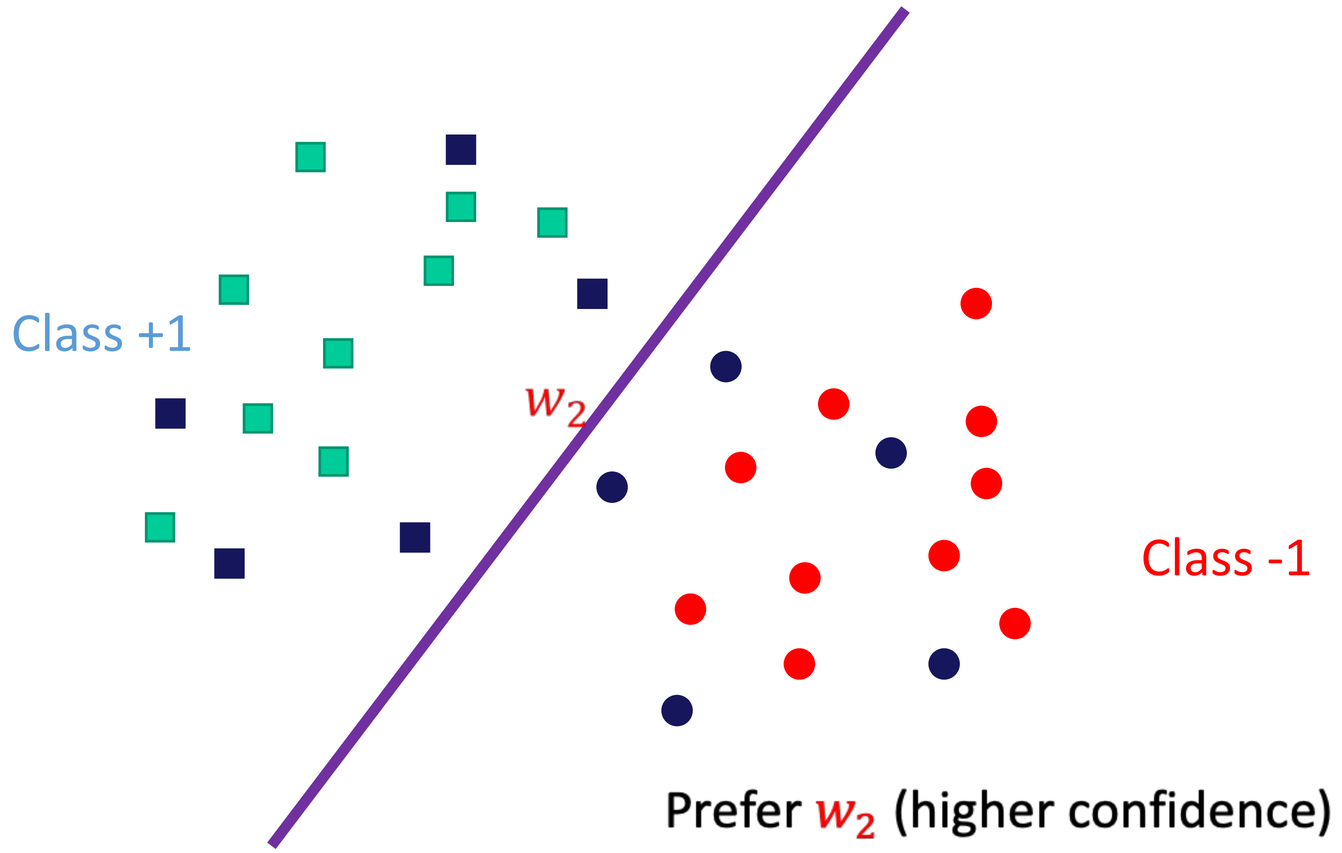
Adding Noise

- What if we have many solutions?



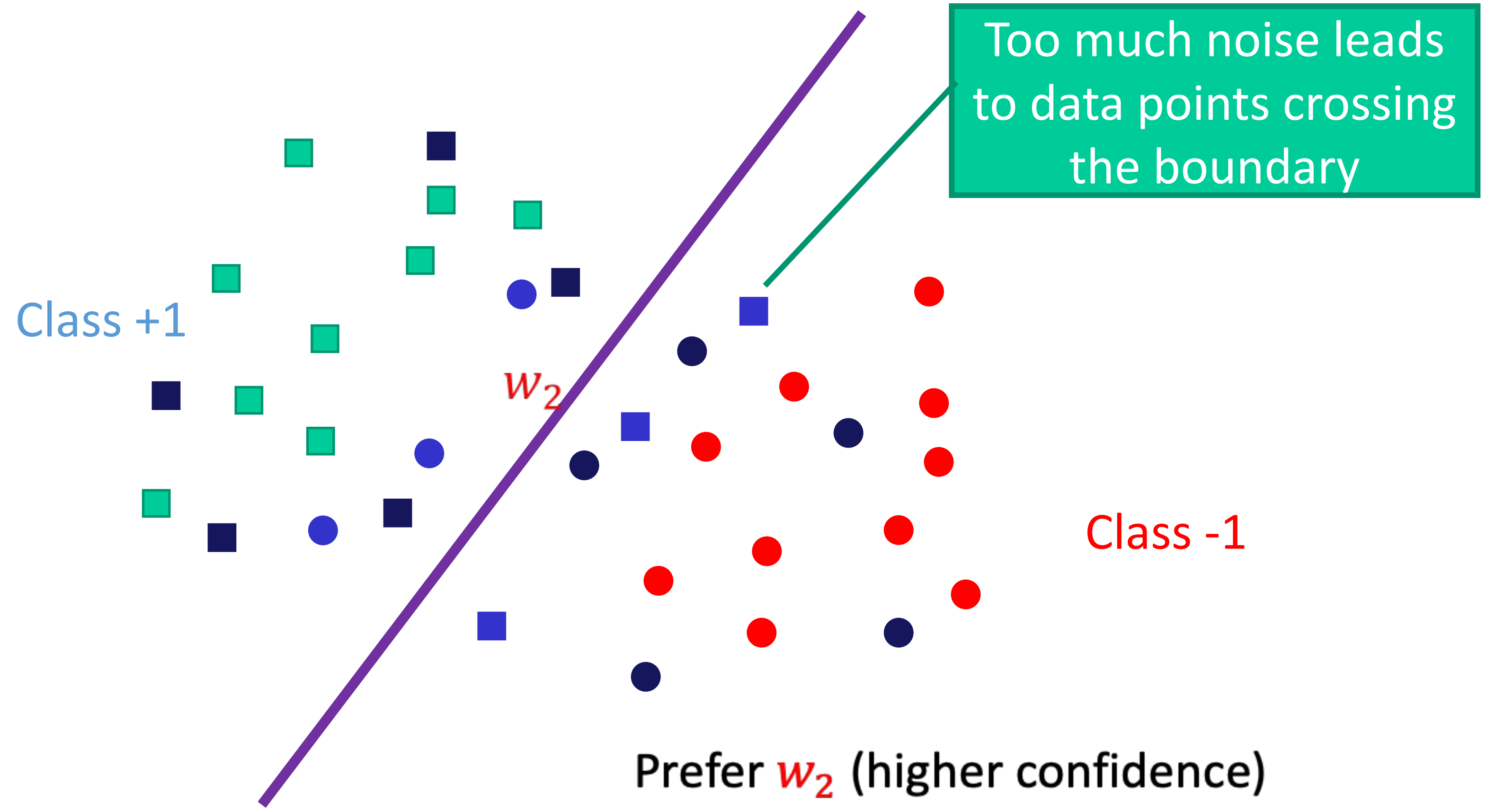
Adding Noise

- Adding some amount of noise helps us pick solution:



Adding Noise

- Too much: hurts instead


Too much noise leads to data points crossing the boundary
Class +1
w_2
Class -1
Prefer w_2 (higher confidence)

Adding Noise: Equivalence to Weight Decay

- Suppose the hypothesis is $f(x) = w^T x$, noise is $\epsilon \sim N(0, \lambda I)$
- After adding noise, the loss is

$$L(f) = \mathbb{E}_{x,y,\epsilon}[f(x+\epsilon) - y]^2 = \mathbb{E}_{x,y,\epsilon}[f(x) + w^T\epsilon - y]^2$$

$$L(f) = \mathbb{E}_{x,y,\epsilon}[f(x) - y]^2 + 2\mathbb{E}_{x,y,\epsilon}[w^T\epsilon(f(x) - y)] + \mathbb{E}_{x,y,\epsilon}[w^T\epsilon]^2$$

$$L(f) = \mathbb{E}_{x,y,\epsilon}[f(x) - y]^2 + \lambda\left|\left|w\right|\right|^2$$

Early Stopping

Idea: don't train to too small training error

- limits the volume of parameter space reachable from the initialization

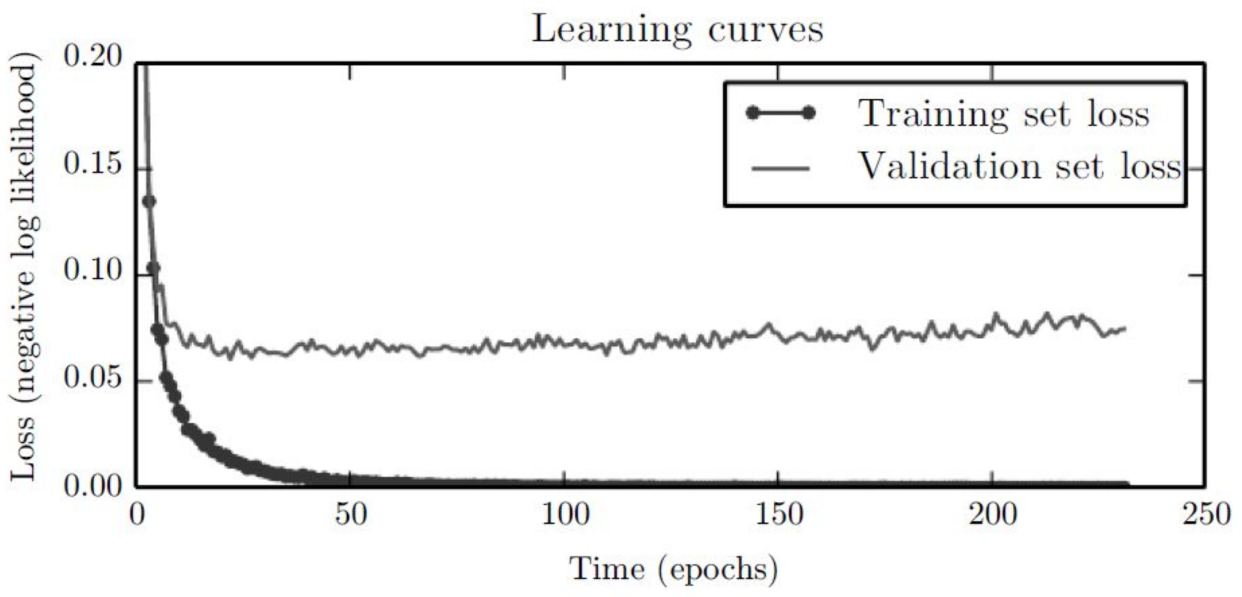


Figure from *Deep Learning*, Goodfellow, Bengio and Courville

Practically: when training, also output validation error

- every time validation error improved, store a copy of the weights
- when validation error not improved for some time, stop
- return the copy of the weights stored

Dropout

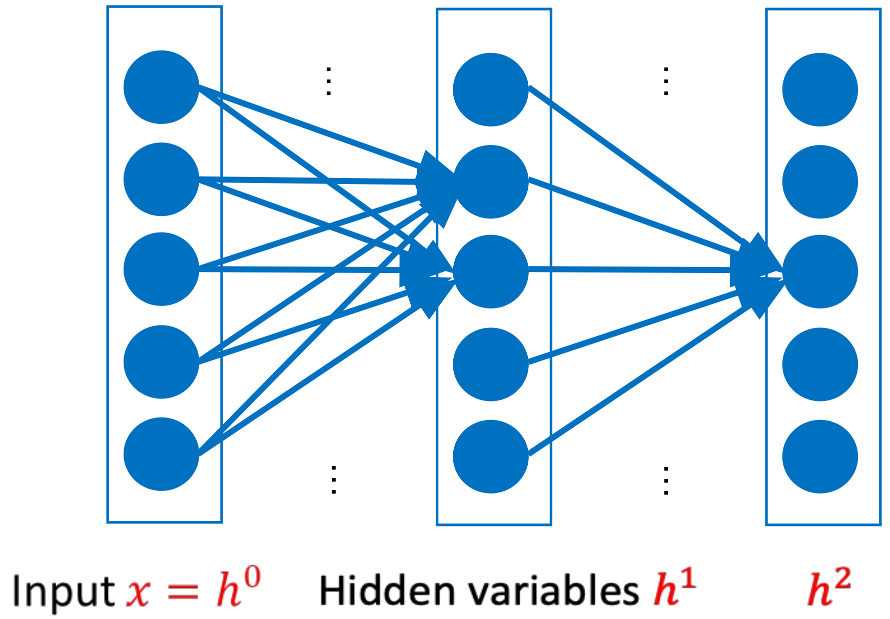
Basic idea: randomly select a subset of weights to update

- In each update step
  - Randomly sample a different binary mask to all the input and hidden units
  - Multiply the mask bits with the units and do the update as usual
- At test time: use all of the weights
- Typical dropout probabilities:
  - 0.2 for inputs weights
  - 0.5 for hidden units

Batch Normalization

Basic idea: standardize inputs to each layer of a neural network

- recall: it is often beneficial to standardize the ith **input** feature x_i by passing $(x_i - \mu_i)/\sigma_i$ to the model; statistics μ_i and σ_i are computed on the training data
- **batch normalization** standardizes the outputs h^L of each layer L before passing them to layer L+1, with **statistics computed on each batch**



Supposed to help optimization, but its mechanism is poorly understood

Summary of regularization

Linear models dominated by **well-understood** penalty-based approaches such as Ridge and LASSO

Neural network regularization is less understood:

- some (weight decay, data augmentation) are applicable to or adapted from linear models
- some (dropout, batch-norm) specific to deep nets
- usually selected via a mix of **trial-and-error** + **formal validation**

Thanks Everyone!

Some of the slides in these lectures have been adapted/borrowed from materials developed by Misha Khodak, Mark Craven, David Page, Jude Shavlik, Tom Mitchell, Nina Balcan, Elad Hazan, Tom Dietterich, Pedro Domingos, Jerry Zhu, Yingyu Liang, Volodymyr Kuleshov, Sharon Li, Fred Sala