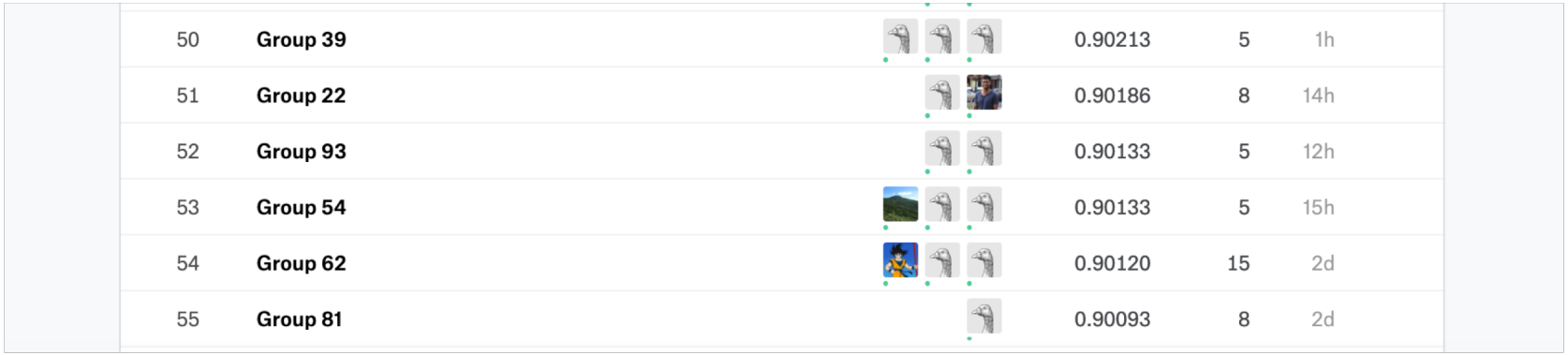
# COMP 551 - Applied Machine Learning Lecture 13 — Neural networks

William L. Hamilton

(with slides and content from Joelle Pineau)

# MiniProject 2



| | | | | |
|---|---|---|---|---|
| 50 | Group 39 | 0.90213 | 5 | 1h |
| 51 | Group 22 | 0.90186 | 8 | 14h |
| 52 | Group 93 | 0.90133 | 5 | 12h |
| 53 | Group 54 | 0.90133 | 5 | 15h |
| 54 | Group 62 | 0.90120 | 15 | 2d |
| 55 | Group 81 | 0.90093 | 8 | 2d |

- Most groups have >29/30 on the competition part!
- Note the grading formula:

$$\text{Score} = \frac{\text{Your score} - \text{Random baseline}}{\text{Prof. benchmark} - \text{Random baseline}}$$

- An accuracy of 0.9 is approximately 29/30!
- Only the top-3 groups get a bonus, so use your time wisely!

# MiniProject 2

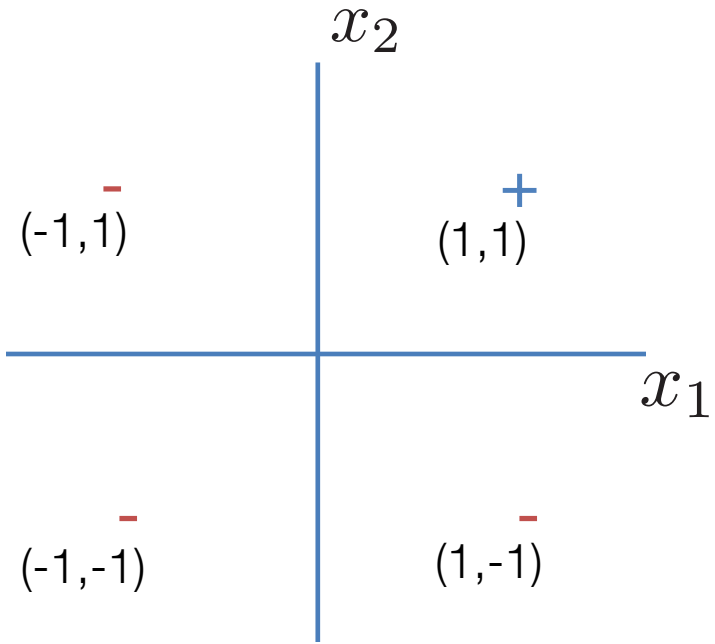
- Reports will be graded on the following scale:
  - Perfect (roughly top-1% of the class): 60/60
  - Outstanding quality (roughly top-10% of the class): 55/60
  - Good quality (roughly top 30% of the class): 50/60
  - Average quality: 45/60
  - Significantly below average quality but still reasonable: 40/60
  - Serious issues (e.g., very hard to understand or incomplete): 30/60
- We will provide a set of strengths and areas of improvement for reach write-up.
- The report for MiniProject 3 will be graded on the same scale, but relative to the quality of the MiniProject 2 reports. I.e., if the entire class improves on MiniProject 3, then everyone gets higher grades.
- Our plan is to release the best MiniProject 2 reports as a reference.

# Quiz 5 discussion

- We are given data from a logical AND function.
- We want to fit a hard SVM:

$$\hat{y} = \text{sign}(\mathbf{w}^\top \mathbf{x} + b)$$

- The hard SVM decision boundary is the one the maximizes the margin.



# Quiz 5 discussion

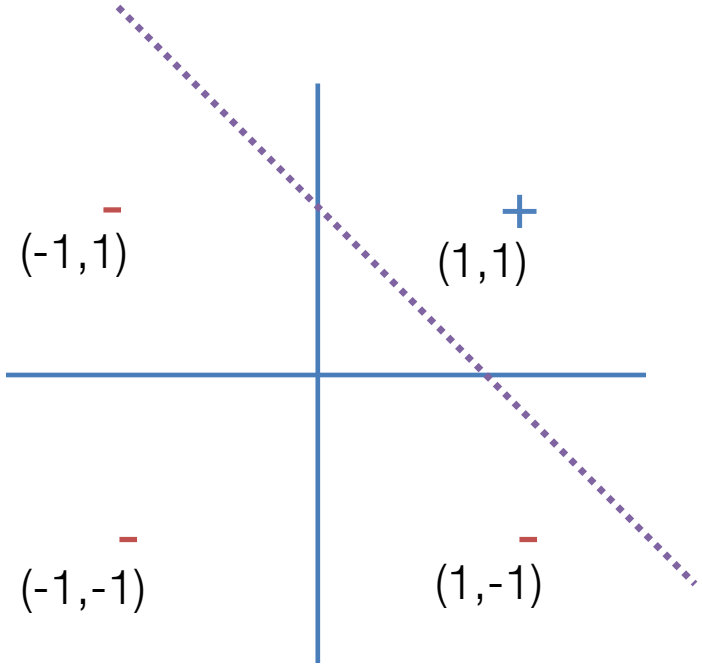
- No need for a convex optimization solver…
- The line that goes through (0,1) and (1,0) seems like a good candidate.
  - In slope-intercept form:

$$x_2 = -x_1 + 1$$

  - But remember that $\mathbf{w}$ in our SVM equations specifies the **normal** to the decision plane, so rearrange:

$$x_1 + x_2 - 1 = 0$$

$$\mathbf{w} = [1, 1], b = -1$$



# Quiz 5 discussion

- No need for a convex optimization solver…
- The line that goes through (0,1) and (1,0) seems like a good candidate.
- It classifies all the points correctly:

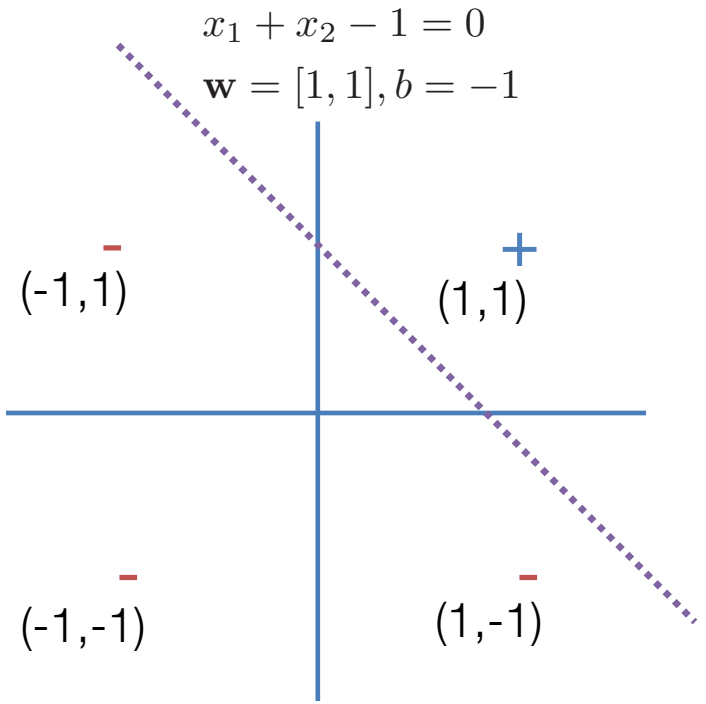
$$\hat{y} = \text{sign}(\mathbf{w}^\top \mathbf{x} + b)$$

$$\text{sign}(1 + 1 - 1) = \text{sign}(1) = 1$$

$$\text{sign}(1 - 1 - 1) = \text{sign}(-1) = -1$$

$$\text{sign}(-1 + 1 - 1) = \text{sign}(-1) = -1$$

$$\text{sign}(-1 - 1 - 1) = \text{sign}(-2) = -1$$

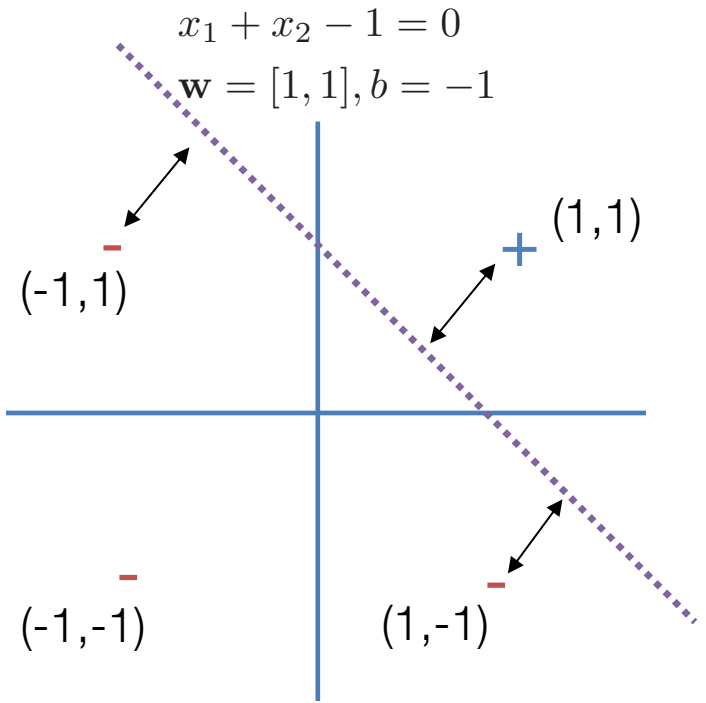


# Quiz 5 discussion

- No need for a convex optimization solver…
- The line that goes through (0,1) and (1,0) seems like a good candidate.
- Now we can calculate the distance between the line and the nearest points (i.e., the candidate support vectors):

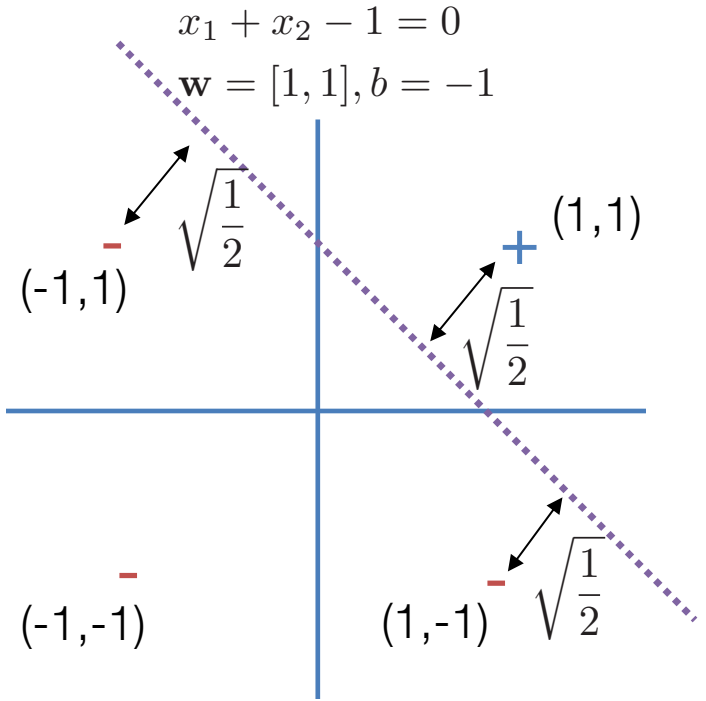
$$d_{\mathbf{w},b}(\mathbf{x}) = \frac{w_1 x_1 + w_2 x_2 + b}{\sqrt{w_1^2 + w_2^2}}$$

$$d_{\mathbf{w},b}([1,1]) = \sqrt{\frac{1}{2}}, d_{\mathbf{w},b}([1,-1]) = \sqrt{\frac{1}{2}}, d_{\mathbf{w},b}([-1,1]) = \sqrt{\frac{1}{2}}$$



# Quiz 5 discussion

- The distances are all the same!
- So we have indeed found the maximum margin hyperplane because any modification would make it closer to one of the three support vectors!



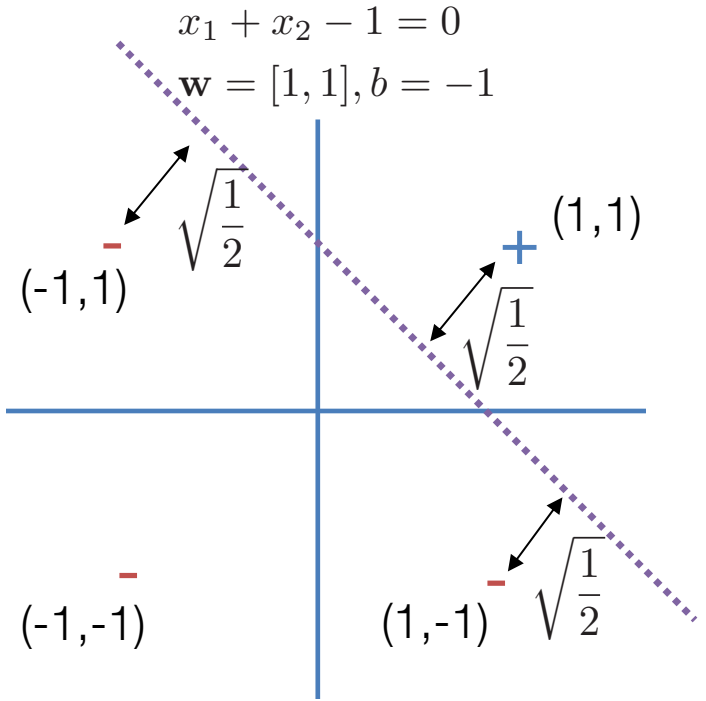
# Quiz 5 discussion

- What is the margin? $M = \sqrt{\frac{1}{2}}$?

  or

  $M = 2\sqrt{\frac{1}{2}}$?

- Different conventions: In the Bishop book $\mathbf{M}$ is the distance to the nearest point. In other sources (e.g., Wikipedia, Lecture 10) $\mathbf{M}$ is twice the distance to the nearest point.
- We will accept both or be unambiguous.

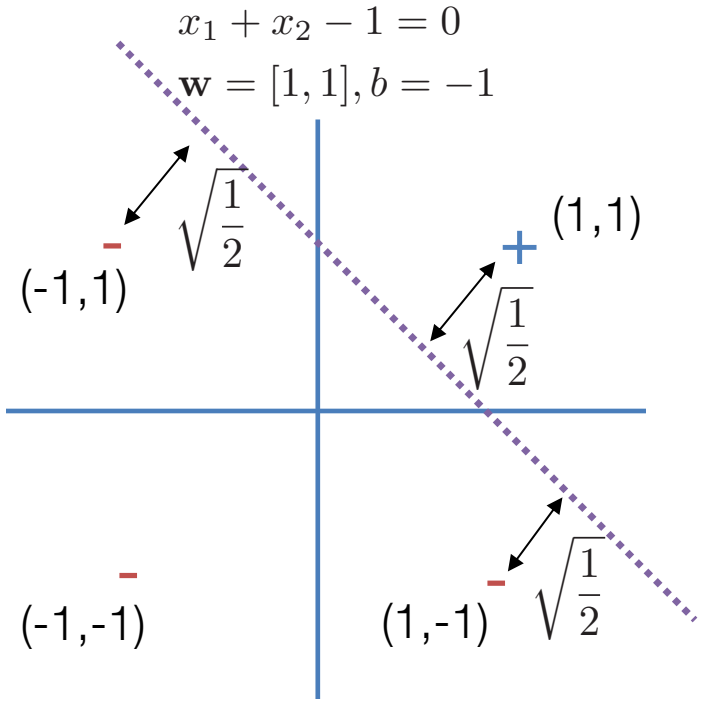


# Quiz 5 discussion

- Note that if we use the definition of the SVM optimization with
  - Minimize $||w||$
    with respect to $w$
    subject to $y_i \mathbf{w}^T \mathbf{x}_i \geq 1$
  - and $||\mathbf{w}||M = 1$

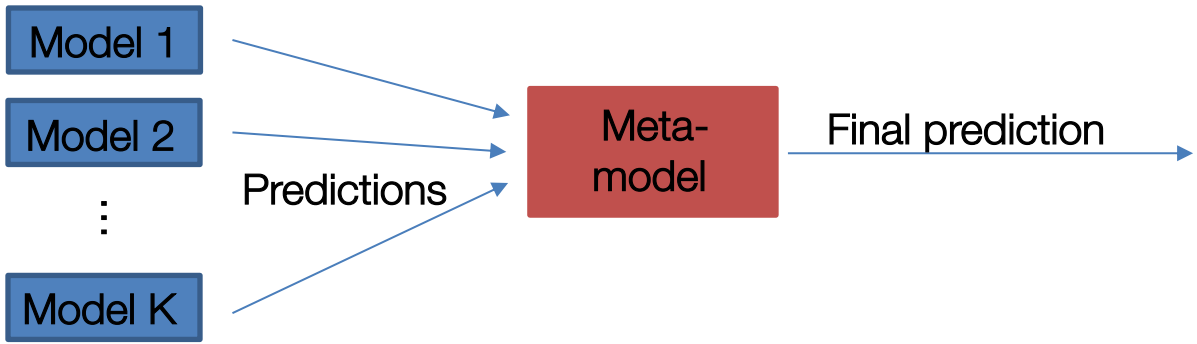
- Then Bishop's definition is more consistent,

  i.e., $||\mathbf{w}|| = \sqrt{\frac{1}{2}}$

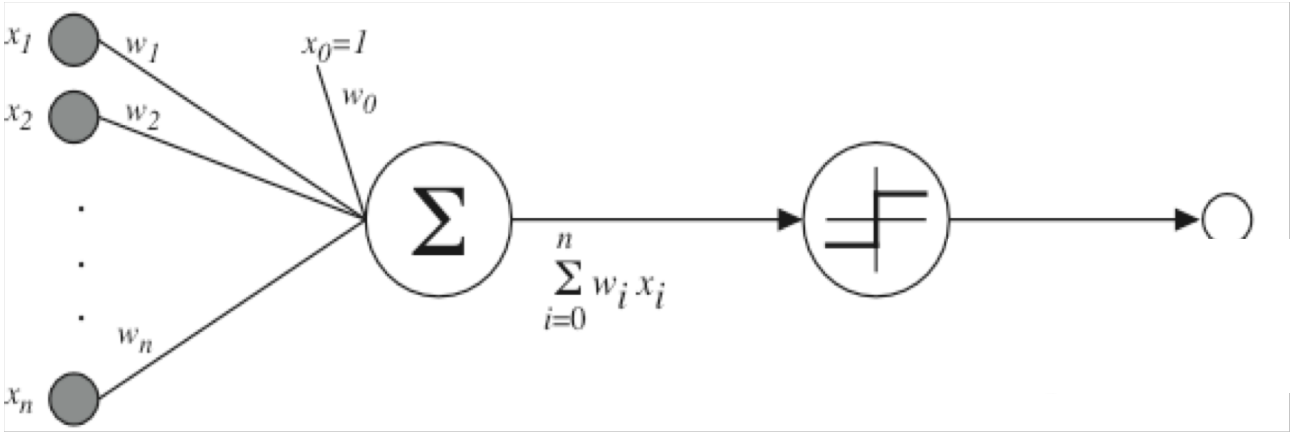


# Recall the idea of stacking

- Train K distinct base models, $h_k$, $k=1...K$, on the training set $<x_i, y_i>$, $i=1...n$
- Make a new training set where the new features are the predictions of the base models: $<h_1(x_i), h_2(x_i), ..., h_K(x_i), y_i>$, $i=1...n$
  - (Can also add the original feature information, $x_i$, to the new training set)
- Train a meta-model $m$ on this new training set.
  - (Possibly train multiple meta-models and repeat the process.)



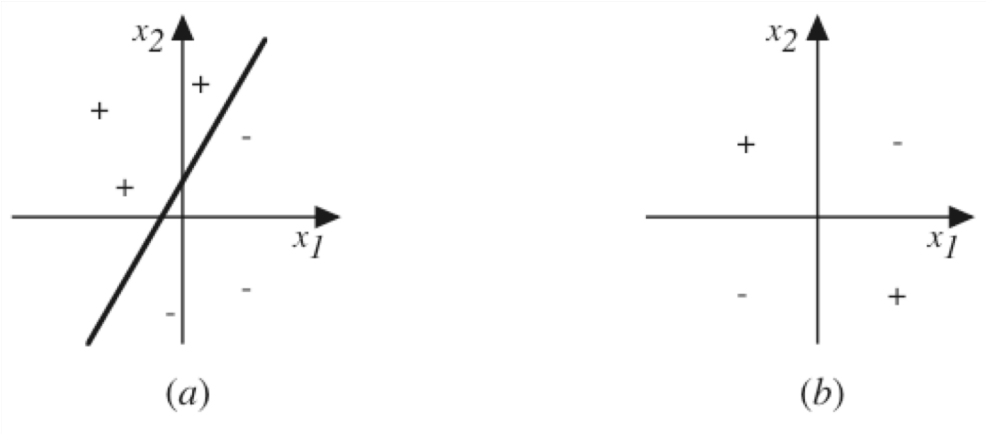
# Recall the perceptron



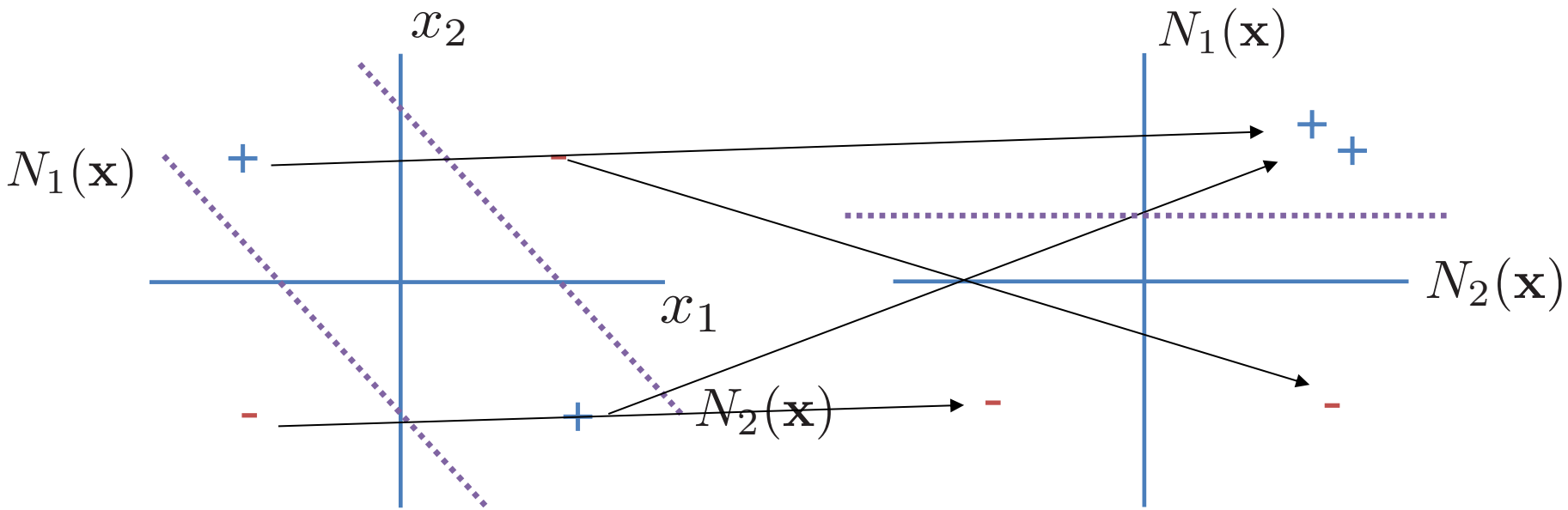

$$h_{\mathbf{w}}(\mathbf{x}) = \text{sgn}(\mathbf{x} \cdot \mathbf{w}) = \begin{cases} +1 & \text{if } \mathbf{x} \cdot \mathbf{w} > 0 \\ -1 & \text{otherwise} \end{cases}$$

# Decision surface of a perceptron

- Single perceptron can represent linear boundaries.
- To represent non-linearly separate functions (e.g. XOR), we could use a network of stacked perceptron-like elements.
- If we connect perceptrons into networks, the error surface for the network is not differentiable (because of the hard threshold).



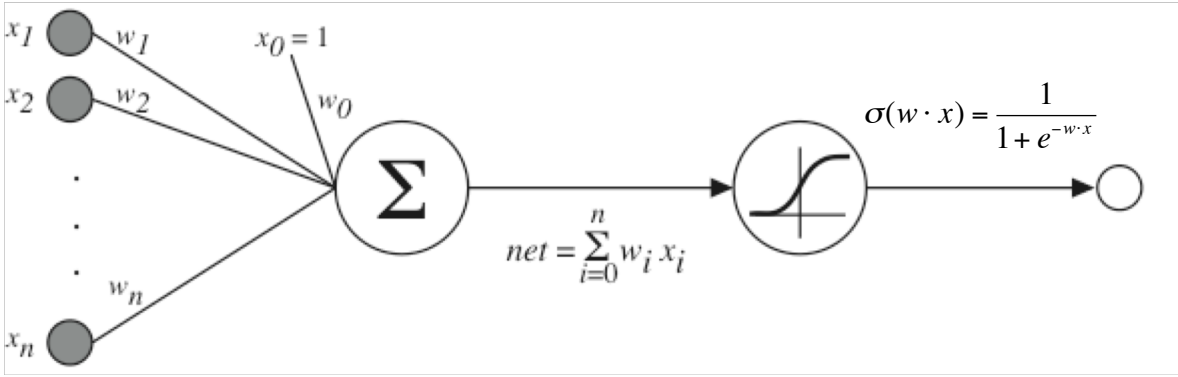
# Example: A network representing XOR




1) Run two perceptrons ($N_1$ and $N_2$) on the original dataset and get the decision boundaries above

2) New dataset defined by the output of $N_1$ and $N_2$ is linearly separable!

# Recall the sigmoid function




Sigmoid provide "soft threshold", whereas perceptron provides "hard threshold"

- It has the following nice property: $\frac{d\sigma(z)}{dz} = \sigma(z)(1 - \sigma(z))$

We can derive a **gradient descent rule** to train:

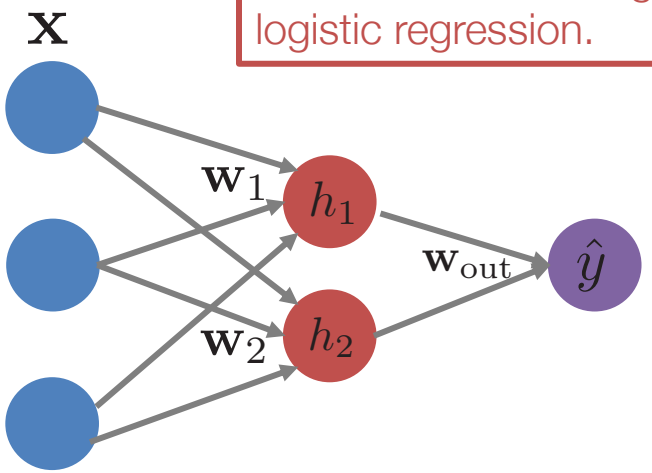
- One sigmoid unit -> multi-layer networks of sigmoid units.

# Feed-forward neural networks

- We are stacking simple models with sigmoid output functions.
  - (I.e., basically stacking logistic regression models)
- "Hidden" units are the output of the sigmoid/logistic models in the stack.
- However, unlike stacking in an ensemble, we want to **jointly train the entire "network"**

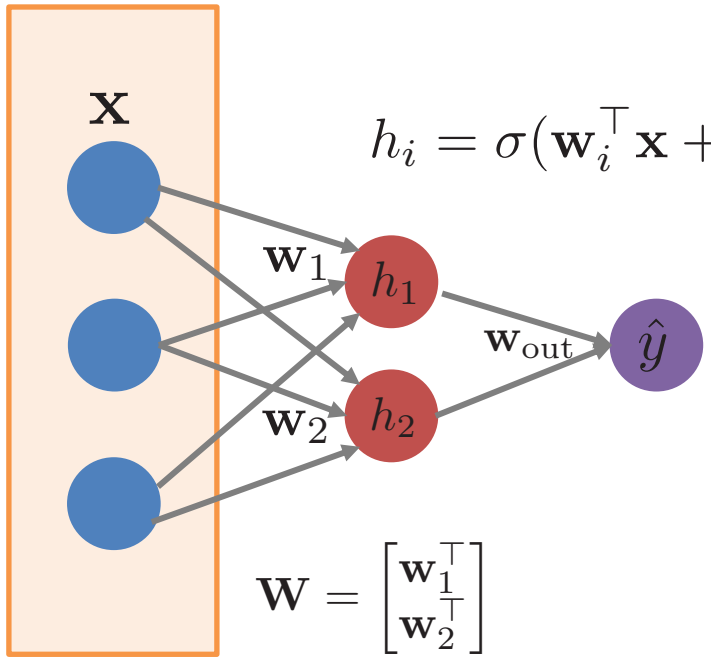
$$h_i = \sigma(\mathbf{w}_i^\top \mathbf{x} + b_i), \forall i$$

Hidden units are linear + sigmoid activation, i.e., analogous to logistic regression.



# Feed-forward neural networks

Input data (or input units)




$$h_i = \sigma(\mathbf{w}_i^\top \mathbf{x} + b_i), \forall i$$

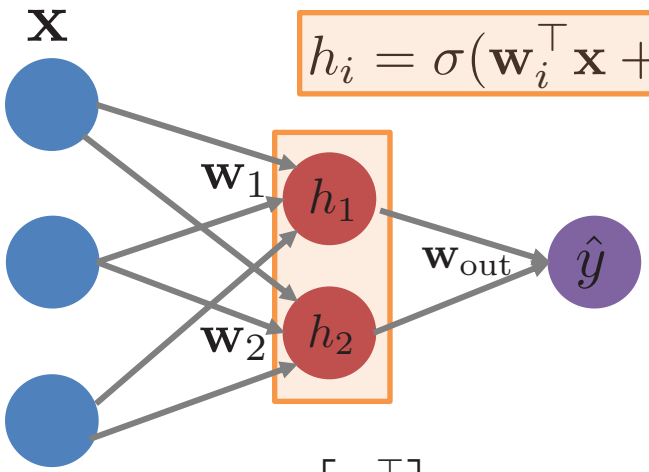
$$\hat{y} = \phi_{\text{out}}(\mathbf{w}_{\text{out}}^\top \mathbf{x} + b_{\text{out}})$$

$$\mathbf{W} = \begin{bmatrix} \mathbf{w}_1^\top \\ \mathbf{w}_2^\top \end{bmatrix}$$

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

# Feed-forward neural networks

Hidden units are linear function + sigmoid applied to input.



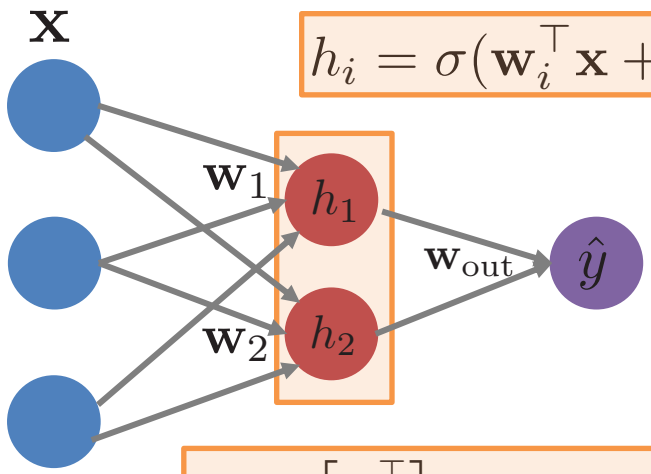

$$\hat{y} = \phi_{\text{out}}(\mathbf{w}_{\text{out}}^\top \mathbf{x} + b_{\text{out}})$$

$$\mathbf{W} = \begin{bmatrix} \mathbf{w}_1^\top \\ \mathbf{w}_2^\top \end{bmatrix}$$

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

# Feed-forward neural networks

**Matrix notation:** We can combine the hidden units together into a vector and their weights into a matrix




$$h_i = \sigma(\mathbf{w}_i^\top \mathbf{x} + b_i), \forall i$$

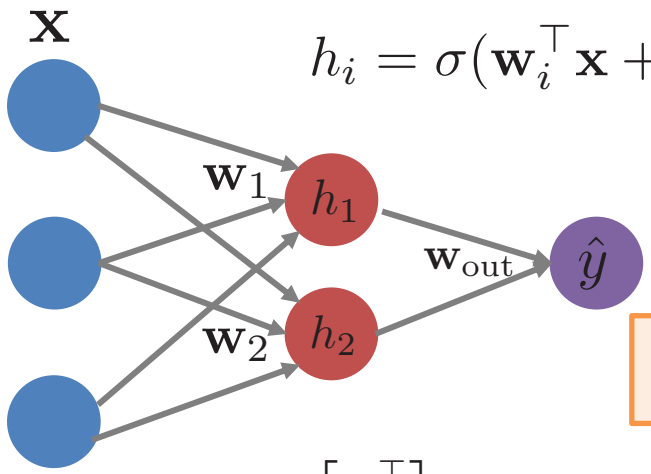
$$\hat{y} = \phi_{\text{out}}(\mathbf{w}_{\text{out}}^\top \mathbf{x} + b_{\text{out}})$$

$$\mathbf{W} = \begin{bmatrix} \mathbf{w}_1^\top \\ \mathbf{w}_2^\top \end{bmatrix}$$

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

# Feed-forward neural networks

**Output unit:** Linear function of the hidden units followed by an "activation function", $\phi_{out}$.




$$h_i = \sigma(\mathbf{w}_i^\top \mathbf{x} + b_i), \forall i$$

$$\hat{y} = \phi_{\text{out}}(\mathbf{w}_{\text{out}}^\top \mathbf{x} + b_{\text{out}})$$

$$\mathbf{W} = \begin{bmatrix} \mathbf{w}_1^\top \\ \mathbf{w}_2^\top \end{bmatrix}$$

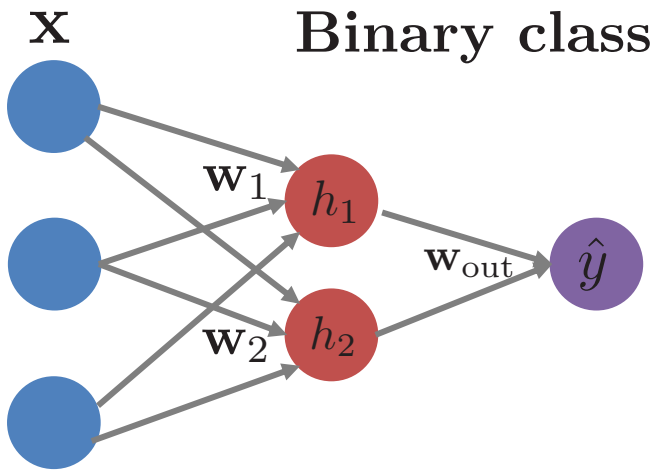
$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

# Feed-forward neural networks

The activation function on the output depends on the task (e.g., regression or classification)

$$\textbf{Regression} : \phi_{\text{out}}(z) = z$$

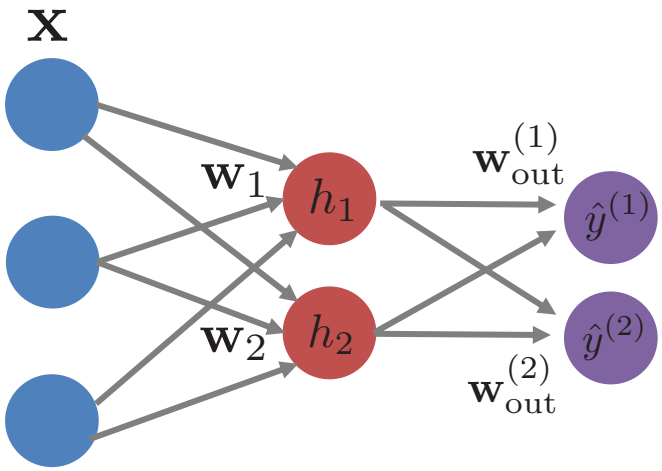
$$\textbf{Binary classification} : \phi_{\text{out}}(z) = \sigma(z)$$




$$\hat{y} = \phi_{\text{out}}(\mathbf{w}_{\text{out}}^\top \mathbf{x} + b_{\text{out}})$$

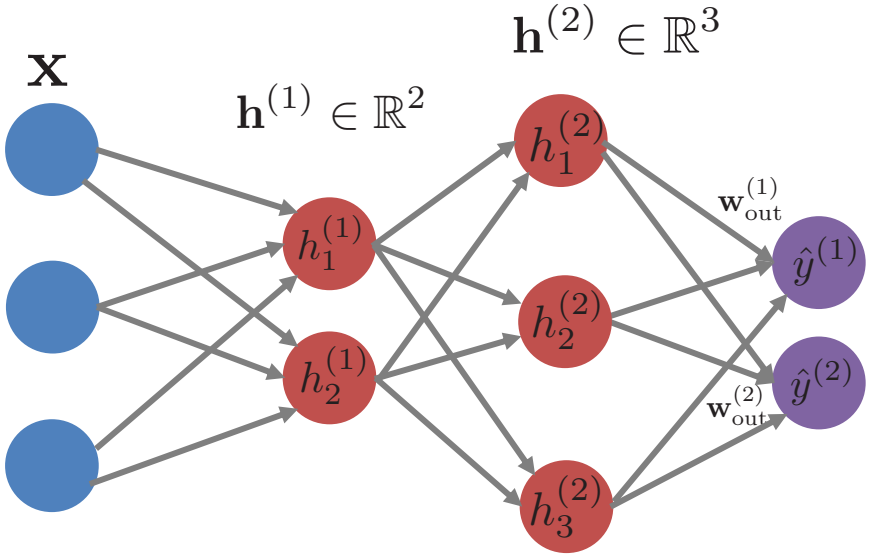
# Feed-forward neural networks

- It is possible to have multiple output units.
- E.g., for multi-label classification.



# Feed-forward neural networks

- It is possible to stack more than one hidden layer.
- This is known as the "depth" of the network.

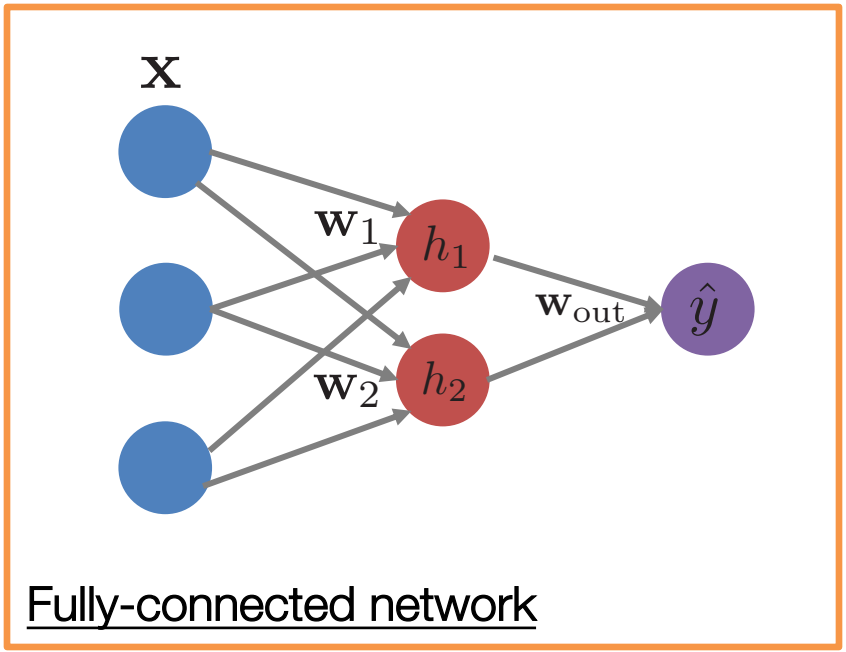



$$\mathbf{h}^{(1)} = \sigma(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \rightarrow \mathbf{h}^{(2)} = \sigma(\mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}) \rightarrow \hat{\mathbf{y}} = \phi_{out}(\mathbf{W}_{out}\mathbf{h}^{(2)} + \mathbf{b}_{out})$$

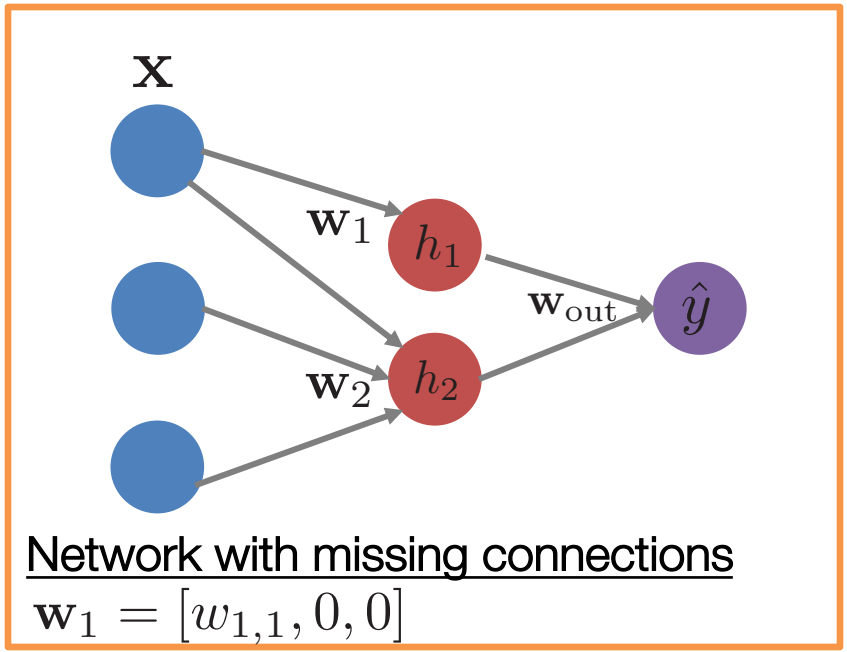
# Why this name?

- In feed-forward networks the output of units in layer $j$ become input to the units in layers $j+1$.
- No cross-connection between units in the same layer.
- No backward ("recurrent") connections from layers downstream
- In fully-connected networks, all units in layer $j$ provide input to all units in layer $j+1$.

# Fully-connected networks




Fully-connected network




Network with missing connections

$\mathbf{w}_1 = [w_{1,1}, 0, 0]$

Fully connected networks are far more common!

# Feed-forward neural networks

- In general, we have an **input layer, H hidden layers, and an output layer.**
- Computing the output is called running the "forward pass":

$$\mathbf{h}^0 = \mathbf{x}$$

for i=1...H:

$$\mathbf{h}^{(i)} = \sigma(\mathbf{W}^{(i)}\mathbf{h}^{(i-1)} + \mathbf{b}^{(i)})$$

$$\hat{\mathbf{y}} = \phi_{\text{out}}(\mathbf{W}_{\text{out}}\mathbf{h}^{(H)} + \mathbf{b}_{\text{out}})$$

Initialize

Compute each hidden layer sequentially

Compute the output

# Learning in feed-forward neural networks

- Assume the network structure (units + connections) is given.
- The learning problem is finding a good set of weights to minimize the error at the output of the network.
- Approach: gradient descent, because the form of the hypothesis formed by the network is:
  - Differentiable! Because of the choice of sigmoid units.
  - Very complex! Hence direct computation of the optimal weights is not possible.

# Gradient-descent preliminaries for NN

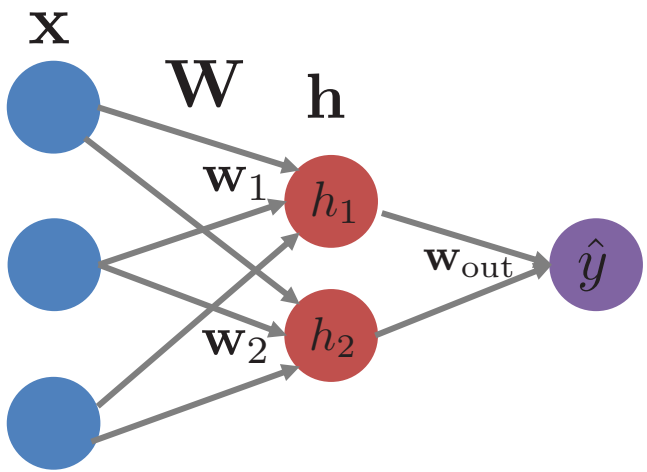
- Take regression as a simple case (i.e., the $\mathbf{y}$ values are one-dimensional and real-valued).
- Assume we have a fully-connected network with one hidden layer.
- We want to compute the weight update after seeing **a single training example** $\mathbf{<x, y>}$.
- We are using the squared loss: $J(y, \hat{y}) = \frac{1}{2}(\hat{y} - y)^2$

# Gradient-descent update for the **output** node

$$\frac{\partial J}{\partial \mathbf{w}_{\text{out}}} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \mathbf{w}_{\text{out}}}$$

Apply the chain rule

Basic Neural Net




$$\mathbf{W} = \begin{bmatrix} \mathbf{w}_1^\top \\ \mathbf{w}_2^\top \end{bmatrix}$$

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

$$\hat{y} = \mathbf{w}_{\text{out}}^\top \mathbf{h} + b_{\text{out}}$$

# Gradient-descent update for the **output** node

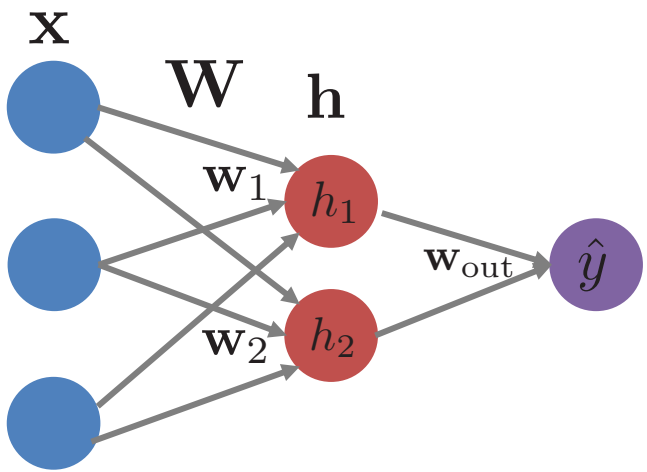
$$\frac{\partial J}{\partial \mathbf{w}_{\text{out}}} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \mathbf{w}_{\text{out}}}$$

$$= (\hat{y} - y) \frac{\partial \hat{y}}{\partial \mathbf{w}_{\text{out}}}$$

Recall that:

$$J(y, \hat{y}) = \frac{1}{2}(\hat{y} - y)^2$$

Basic Neural Net




$$\mathbf{W} = \begin{bmatrix} \mathbf{w}_1^\top \\ \mathbf{w}_2^\top \end{bmatrix}$$

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

$$\hat{y} = \mathbf{w}_{\text{out}}^\top \mathbf{h} + b_{\text{out}}$$

# Gradient-descent update for the **output** node

$$\frac{\partial J}{\partial \mathbf{w}_{\text{out}}} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \mathbf{w}_{\text{out}}}$$

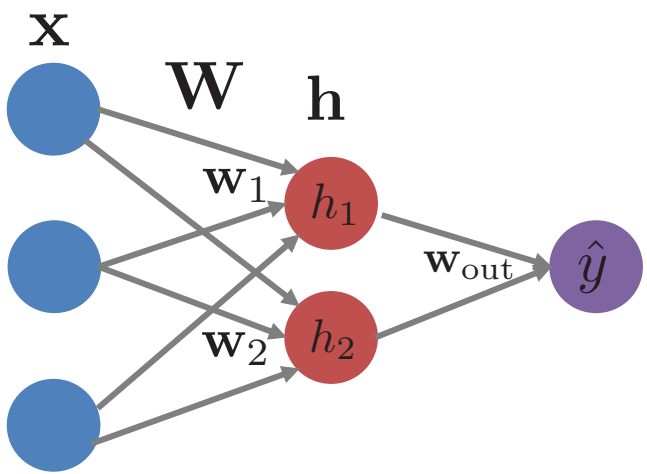
$$= (\hat{y} - y) \frac{\partial \hat{y}}{\partial \mathbf{w}_{\text{out}}}$$

$$= (\hat{y} - y) \frac{\partial \left( \mathbf{w}_{\text{out}} \mathbf{h} + b_{\text{out}} \right)}{\partial \mathbf{w}_{\text{out}}}$$

Recall that:

$$\hat{y} = \mathbf{w}_{\text{out}}^\top \mathbf{h} + b_{\text{out}}$$

Basic Neural Net




$$\mathbf{W} = \begin{bmatrix} \mathbf{w}_1^\top \\ \mathbf{w}_2^\top \end{bmatrix}$$

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

$$\hat{y} = \mathbf{w}_{\text{out}}^\top \mathbf{h} + b_{\text{out}}$$

# Gradient-descent update for the **output** node

$$\frac{\partial J}{\partial \mathbf{w}_{\text{out}}} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \mathbf{w}_{\text{out}}}$$

$$= (\hat{y} - y) \frac{\partial \hat{y}}{\partial \mathbf{w}_{\text{out}}}$$

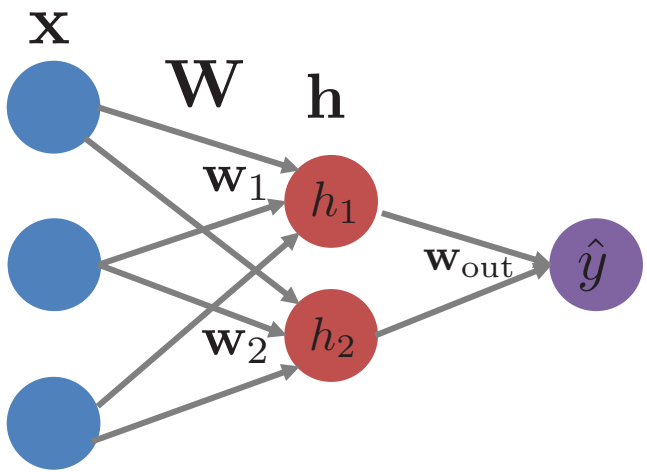
$$= (\hat{y} - y) \frac{\partial \left( \mathbf{w}_{\text{out}} \mathbf{h} + b_{\text{out}} \right)}{\partial \mathbf{w}_{\text{out}}}$$

$$= (\hat{y} - y) \mathbf{h}$$

$$= \delta_{out} \mathbf{h}$$

We can think of this of this a the "error signal" at the output node.

Basic Neural Net




$$\mathbf{W} = \begin{bmatrix} \mathbf{w}_1^\top \\ \mathbf{w}_2^\top \end{bmatrix}$$

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

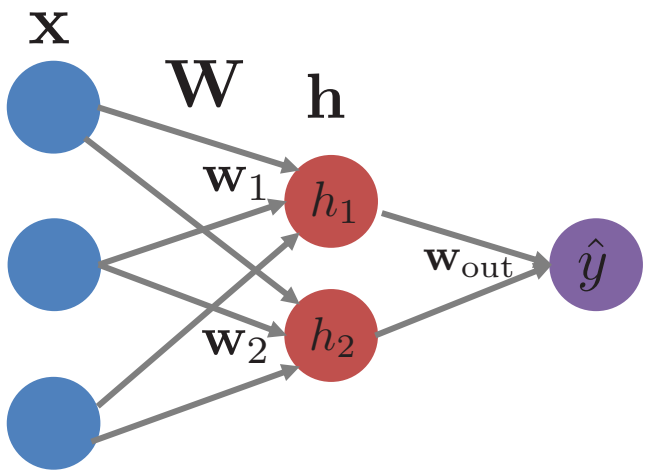
$$\hat{y} = \mathbf{w}_{\text{out}}^\top \mathbf{h} + b_{\text{out}}$$

# Gradient-descent update for the **hidden** node

$$\frac{\partial J}{\partial \mathbf{w}_i}$$

We want to determine the derivative of the error w.r.t. to the weights of the hidden node.

Basic Neural Net




$$\mathbf{W} = \begin{bmatrix} \mathbf{w}_1^\top \\ \mathbf{w}_2^\top \end{bmatrix}$$

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

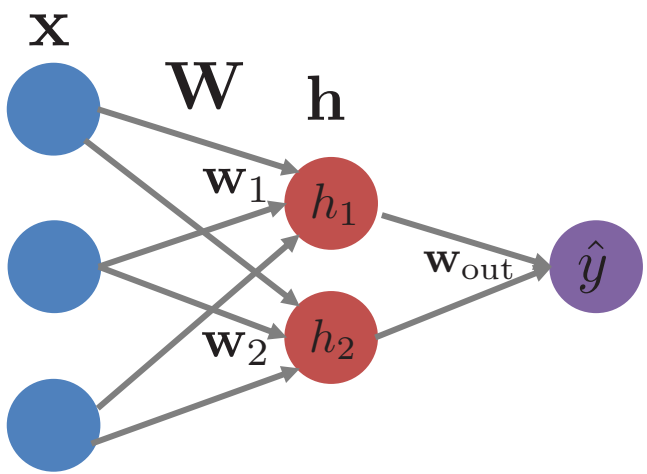
$$\hat{y} = \mathbf{w}_{\text{out}}^\top \mathbf{h} + b_{\text{out}}$$

# Gradient-descent update for the **hidden** node

$$\frac{\partial J}{\partial \mathbf{w}_i} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \mathbf{w}_j}$$

Again, apply the chain rule

Basic Neural Net




$$\mathbf{W} = \begin{bmatrix} \mathbf{w}_1^\top \\ \mathbf{w}_2^\top \end{bmatrix}$$

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

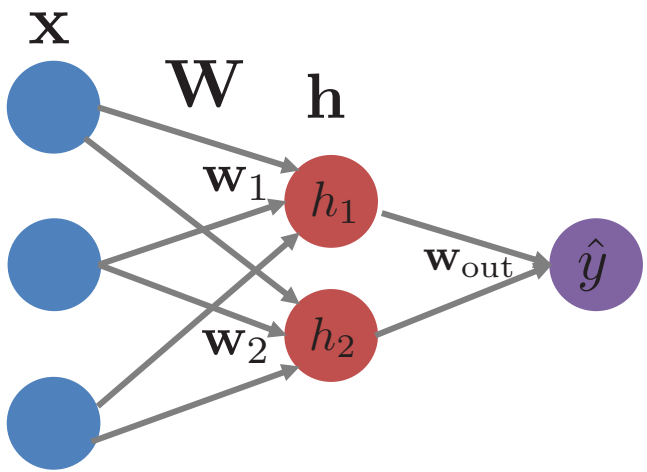
$$\hat{y} = \mathbf{w}_{\text{out}}^\top \mathbf{h} + b_{\text{out}}$$

# Gradient-descent update for the **hidden** node

$$\frac{\partial J}{\partial \mathbf{w}_i} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \mathbf{w}_j}$$
$$= \delta_{\text{out}} \frac{\partial \hat{y}}{\partial \mathbf{w}_j}$$

We already compute the error at the output node, so we can just substitute this in.

Basic Neural Net




$$\mathbf{W} = \begin{bmatrix} \mathbf{w}_1^\top \\ \mathbf{w}_2^\top \end{bmatrix}$$

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

$$\hat{y} = \mathbf{w}_{\text{out}}^\top \mathbf{h} + b_{\text{out}}$$

# Gradient-descent update for the **hidden** node

$$\frac{\partial J}{\partial \mathbf{w}_i} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \mathbf{w}_j}$$

$$= \delta_{\text{out}} \frac{\partial \hat{y}}{\partial \mathbf{w}_j}$$

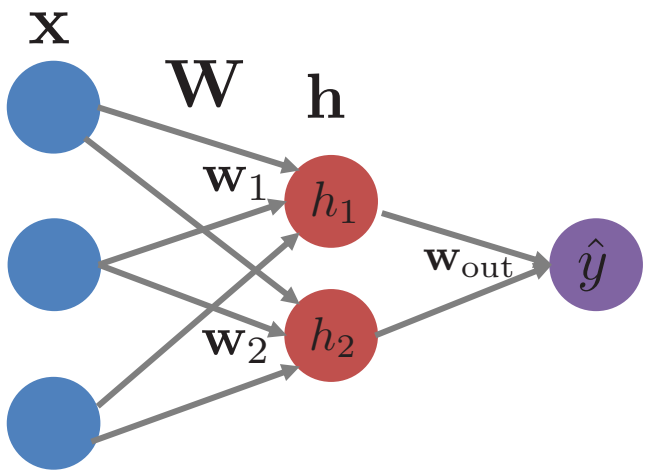
$$= \delta_{\text{out}} \frac{\partial \hat{y}}{\partial h_j} \frac{\partial h_j}{\partial \mathbf{w}_j}$$

Recall that:

$$h_i = \sigma(\mathbf{w}_i^\top \mathbf{x} + b_i), \forall i$$

And again, apply the chain rule….

Basic Neural Net




$$\mathbf{W} = \begin{bmatrix} \mathbf{w}_1^\top \\ \mathbf{w}_2^\top \end{bmatrix}$$

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

$$\hat{y} = \mathbf{w}_{\text{out}}^\top \mathbf{h} + b_{\text{out}}$$

# Gradient-descent update for the **hidden** node

$$\frac{\partial J}{\partial \mathbf{w}_i} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \mathbf{w}_j}$$

$$= \delta_{\text{out}} \frac{\partial \hat{y}}{\partial \mathbf{w}_j}$$

$$= \delta_{\text{out}} \frac{\partial \hat{y}}{\partial h_j} \frac{\partial h_j}{\partial \mathbf{w}_j}$$

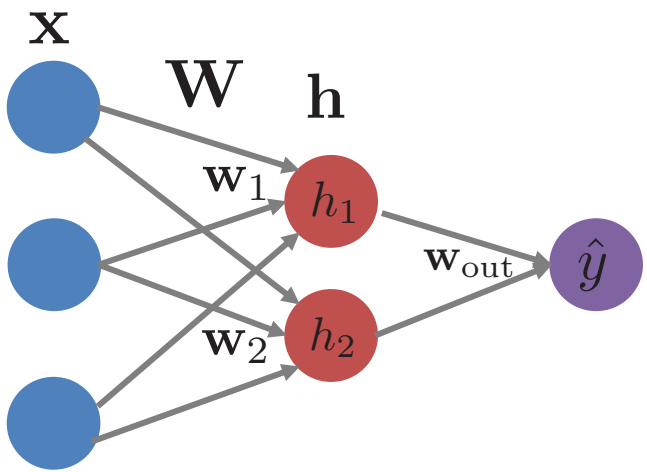
$$= \delta_{out} w_{\text{out},j} \frac{\partial h_j}{\partial \mathbf{w}_j}$$

Recall that

$$\hat{y} = \mathbf{w}_{\text{out}}^\top \mathbf{h} + b_{\text{out}}$$

and note that the j'th hidden node only interacts with the j'th value in $\mathbf{w}_{\text{out}}$

Basic Neural Net




$$\mathbf{W} = \begin{bmatrix} \mathbf{w}_1^\top \\ \mathbf{w}_2^\top \end{bmatrix}$$

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

$$\hat{y} = \mathbf{w}_{\text{out}}^\top \mathbf{h} + b_{\text{out}}$$

# Gradient-descent update for the **hidden** node

$$\frac{\partial J}{\partial \mathbf{w}_i} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \mathbf{w}_j}$$

$$= \delta_{\text{out}} \frac{\partial \hat{y}}{\partial \mathbf{w}_j}$$

$$= \delta_{\text{out}} \frac{\partial \hat{y}}{\partial h_j} \frac{\partial h_j}{\partial \mathbf{w}_j}$$

$$= \delta_{out} w_{\text{out},j} \frac{\partial h_j}{\partial \mathbf{w}_j}$$

$$= \delta_{out} w_{\text{out},j} \sigma(\mathbf{w}_j^\top \mathbf{x} + b)(1 - \sigma(\mathbf{w}_j^\top \mathbf{x} + b))\mathbf{x}$$

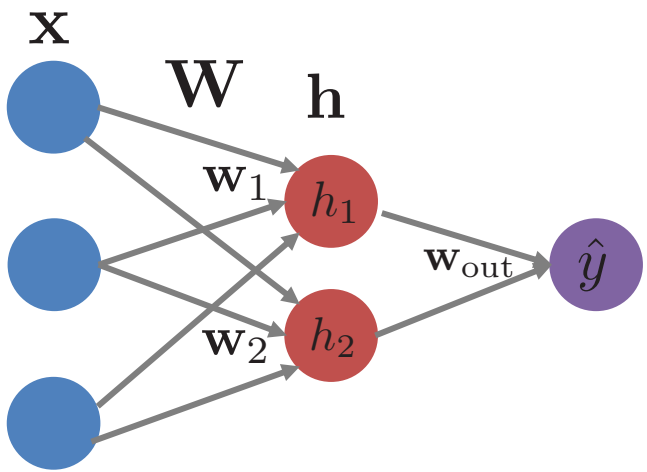
Recall that

$$h_i = \sigma(\mathbf{w}_i^\top \mathbf{x} + b_i), \forall i$$

and the identity

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

Basic Neural Net




$$\mathbf{W} = \begin{bmatrix} \mathbf{w}_1^\top \\ \mathbf{w}_2^\top \end{bmatrix}$$

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

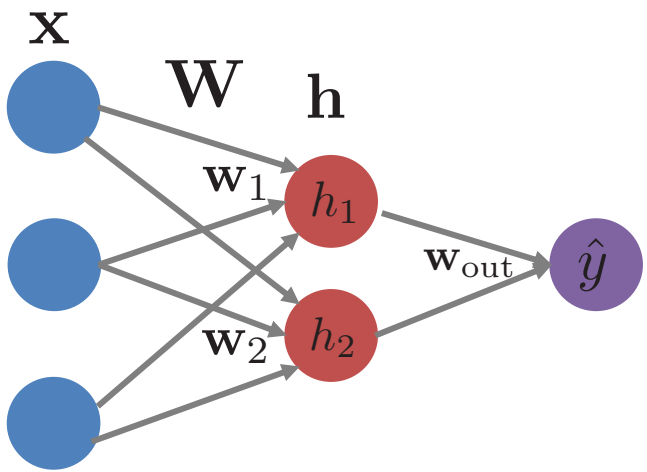
$$\hat{y} = \mathbf{w}_{\text{out}}^\top \mathbf{h} + b_{\text{out}}$$

# Gradient-descent update for the **hidden** node

$$
\begin{aligned}
\frac{\partial J}{\partial \mathbf{w}_i} &= \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \mathbf{w}_j} \\
&= \delta_{\text{out}} \frac{\partial \hat{y}}{\partial \mathbf{w}_j} \\
&= \delta_{\text{out}} \frac{\partial \hat{y}}{\partial h_j} \frac{\partial h_j}{\partial \mathbf{w}_j} \\
&= \delta_{out} w_{\text{out},j} \frac{\partial h_j}{\partial \mathbf{w}_j} \\
&= \delta_{out} w_{\text{out},j} \sigma(\mathbf{w}_j^\top \mathbf{x} + b)(1 - \sigma(\mathbf{w}_j^\top \mathbf{x} + b))\mathbf{x} \\
&= \delta_{h_j} \mathbf{x}
\end{aligned}
$$

We can think of this of this a the "error signal" at the hidden node.

Basic Neural Net




$$\mathbf{W} = \begin{bmatrix} \mathbf{w}_1^\top \\ \mathbf{w}_2^\top \end{bmatrix}$$

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

$$\hat{y} = \mathbf{w}_{\text{out}}^\top \mathbf{h} + b_{\text{out}}$$

# Gradient-descent update for the **hidden** node

$$\frac{\partial J}{\partial \mathbf{w}_i} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \mathbf{w}_j}$$

$$= \delta_{\text{out}} \frac{\partial \hat{y}}{\partial \mathbf{w}_j}$$

$$= \delta_{\text{out}} \frac{\partial \hat{y}}{\partial h_j} \frac{\partial h_j}{\partial \mathbf{w}_j}$$

$$= \delta_{out} w_{\text{out},j} \frac{\partial h_j}{\partial \mathbf{w}_j}$$

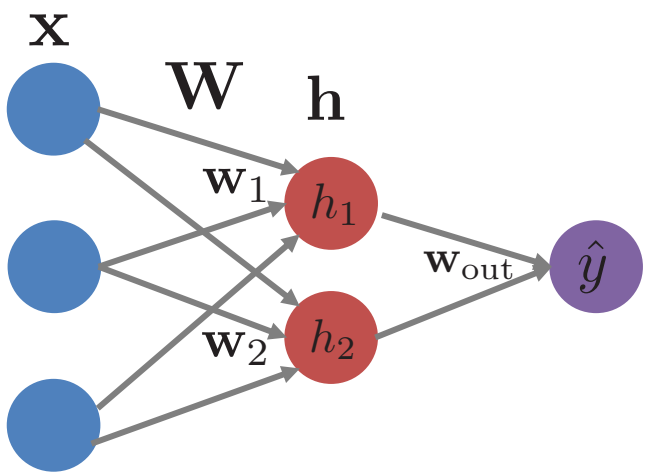
$$= \delta_{out} w_{\text{out},j} \sigma(\mathbf{w}_j^\top \mathbf{x} + b)(1 - \sigma(\mathbf{w}_j^\top \mathbf{x} + b))\mathbf{x}$$

$$= \delta_{h_j} \mathbf{x}$$

The error at the hidden node is a function of the error at the output, and we are "propagating" this error backwards through the network.

We can think of this of this a the "error signal" at the hidden node.

Basic Neural Net




$$\mathbf{W} = \begin{bmatrix} \mathbf{w}_1^\top \\ \mathbf{w}_2^\top \end{bmatrix}$$

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

$$\hat{y} = \mathbf{w}_{\text{out}}^\top \mathbf{h} + b_{\text{out}}$$

# Stochastic gradient descent

- Initialize all weights to small random numbers.
- Repeat until convergence:
  - Pick a training example, **x**.
  - Feed example through network to compute output y.
  - For the output unit, compute the correction:

$$\frac{\partial J}{\partial \mathbf{w}_{\text{out}}} = \delta_{\text{out}} \mathbf{x}$$

  - For each hidden unit *j*, compute its share of the correction:

$$\frac{\partial J}{\partial \mathbf{w}_j} = \delta_{\text{out}} w_{out,j} \sigma(\mathbf{w}_j^\top \mathbf{x} + b)(1 - \sigma(\mathbf{w}_j^\top \mathbf{x} + b))\mathbf{x}$$

  - Update each network weight:

$$\mathbf{w}_j = \mathbf{w}_j - \alpha \frac{\partial J}{\partial \mathbf{w}_j} \ \forall j, \qquad \mathbf{w}_{out} = \mathbf{w}_{out} - \alpha \frac{\partial J}{\partial \mathbf{w}_{out}}$$

Initialization

Forward pass

Backpro-pagation

Gradient descent

# Organizing the training data

- **Stochastic gradient descent**: Compute error on a single example at a time (as in previous slide).

- **Batch gradient descent**: Compute error on all examples.
  - Loop through the training data, accumulating weight changes.
  - Update all weights and repeat.

- **Mini-batch gradient descent**: Compute error on small subset.
  - Randomly select a "mini-batch" (i.e. subset of training examples).
  - Calculate error on mini-batch, apply to update weights, and repeat.

# Expressiveness of feed-forward NN

A neural network with no hidden layers?

- Same representational power as logistic/linear regression or a perceptron; Boolean AND, OR, NOT, but not XOR.

# Expressiveness of feed-forward NN

A neural network with no hidden layers?

- Same representational power as logistic/linear regression or a perceptron; Boolean AND, OR, NOT, but not XOR.

A neural network with a single hidden layer?

- Can represent every boolean function, but might require a number of hidden units that is exponential in the number of inputs.
- Every bounded continuous function can be approximated with arbitrary precision by a boolean function.

# Expressiveness of feed-forward NN

A neural network with no hidden layers?

- Same representational power as logistic/linear regression or a perceptron; Boolean AND, OR, NOT, but not XOR.

A neural network with a single hidden layer?

- Can represent every boolean function, but might require a number of hidden units that is exponential in the number of inputs.
- Every bounded continuous function can be approximated with arbitrary precision by a boolean function.

A neural network with two hidden layers?

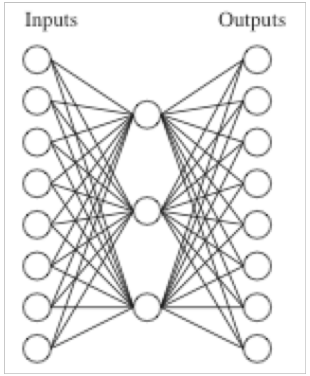
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers.

# Learning the identity function

| Input | | Output |
|---|---|---|
| 10000000 | → | 10000000 |
| 01000000 | → | 01000000 |
| 00100000 | → | 00100000 |
| 00010000 | → | 00010000 |
| 00001000 | → | 00001000 |
| 00000100 | → | 00000100 |
| 00000010 | → | 00000010 |
| 00000001 | → | 00000001 |

# Learning the identity function

- Neural network structure:

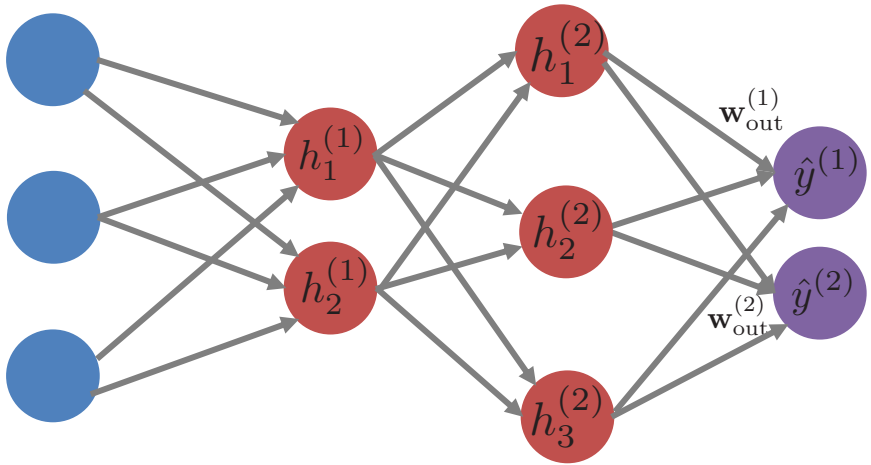



- Learned hidden layer weights:

| Input | | Hidden Layer | | | | Output |
|---|---|---|---|---|---|---|
| 10000000 | → | .89 | .04 | .08 | → | 10000000 |
| 01000000 | → | .15 | .99 | .99 | → | 01000000 |
| 00100000 | → | .01 | .97 | .27 | → | 00100000 |
| 00010000 | → | .99 | .97 | .71 | → | 00010000 |
| 00001000 | → | .03 | .05 | .02 | → | 00001000 |
| 00000100 | → | .01 | .11 | .88 | → | 00000100 |
| 00000010 | → | .80 | .01 | .98 | → | 00000010 |
| 00000001 | → | .60 | .94 | .01 | → | 00000001 |

# Generalizing the feed-forward NN

- Can use arbitrary output activation functions.
- In practice, we do not necessarily need to use a sigmoid activation in the hidden layer.
- We can make networks as deep as we want.
- We can add regularization.
- But how to compute these nasty derivatives..? (Next lecture!)




$$\mathbf{h}^{(i)} = \phi_i(\mathbf{W}^{(i)}\mathbf{h}^{(i-1)} + \mathbf{b}^{(i)})$$

Can be an arbitrary **non-linear** activation function