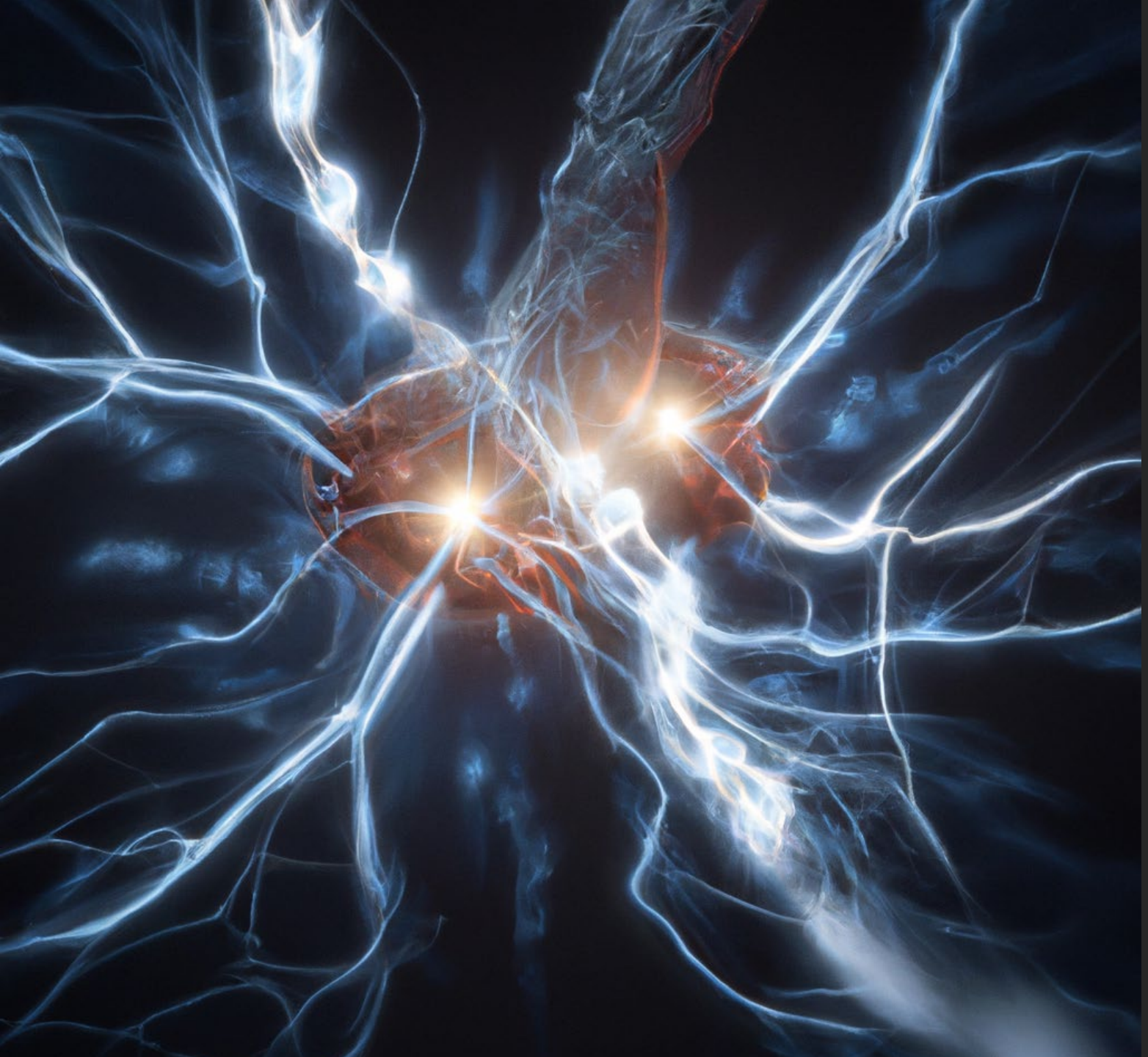


MLPs and Backprop

Applied Machine Learning
Derek Hoiem

Dall-E

Review from the practice questions

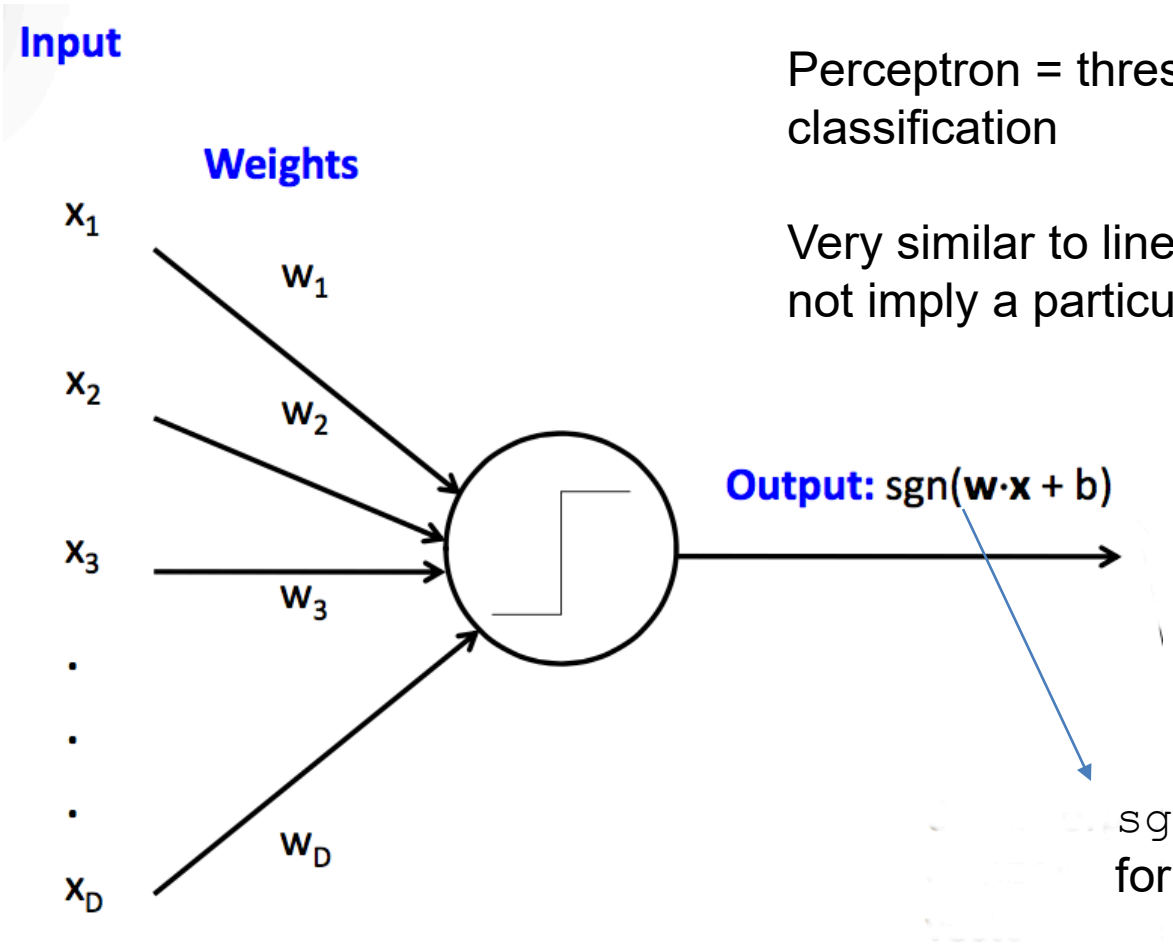
True or False:

- Unlike SVM, linear logistic regression loss always adds a non-zero penalty over all training data points.
- The PEGASUS algorithm computes the gradient for the optimization algorithm using only one sample out of the training data points – instead of using the whole dataset – thus increasing its computational efficiency.
- PEGASUS has the disadvantage that the larger the training dataset, the slower it can be optimized to reach a particular test error.

Multi-layer Perceptrons (MLPs)

- What is a perceptron
- What is an MLP
  - Layers
  - Activations
  - Losses
- How do we optimize with SGD and back-propagation

Perceptron



Perceptron = thresholded linear prediction model for classification

Very similar to linear logistic regression, though perceptron does not imply a particular error or training objective

`sgn` returns -1 for negative inputs and +1 for positive inputs

Fig source: CS 440

Perceptron Update Rule

Prediction: $f(\boldsymbol{x}) = w_0 x_0 + w_1 x_1 + \ldots w_m x_m + b$

Error: $E(\boldsymbol{x}) = (f(\boldsymbol{x}) - y)^2$

prediction target

Update w_i: take a step to decrease $E(\boldsymbol{x})$

$$\frac{\partial E(\boldsymbol{x})}{\partial w_i} = 2(f(\boldsymbol{x}) - y)\left[\frac{\partial (f(\boldsymbol{x}) - y)}{\partial w_i}\right]$$

$$\frac{\partial E(\boldsymbol{x})}{\partial w_i} = 2(f(\boldsymbol{x}) - y)x_i$$

$$w_i = w_i - \eta(f(\boldsymbol{x}) - y)x_i$$

Make error *lower* Learning rate

Chain Rule:
$h(x) = f(g(x))$, then
$h'(x) = f'\big(g(x)\big)g'(x)$

(the 2 is folded into the learning rate)

Perceptron Optimization by SGD

Randomly initialize weights, e.g. w ~ Gaus(mu=0, std=0.05)

For each iteration t:

Split data into batches

$\eta = 0.1/t$

For each batch X_b:

For each weight w_i:

$$w_i = w_i - \eta \frac{1}{|X_b|} \sum_{\boldsymbol{x}_n \in X_b} (f(\boldsymbol{x}_n) - y_n) x_{ni}$$

With different loss, the update changes accordingly

Logistic loss:

$$f(\boldsymbol{x}) = w_0 x_0 + w_1 x_1 + \dots w_m x_m + b$$

$$P(y|\boldsymbol{x}) = \frac{1}{1+\exp(-yf(x))}, y \in \{-1,1\}$$

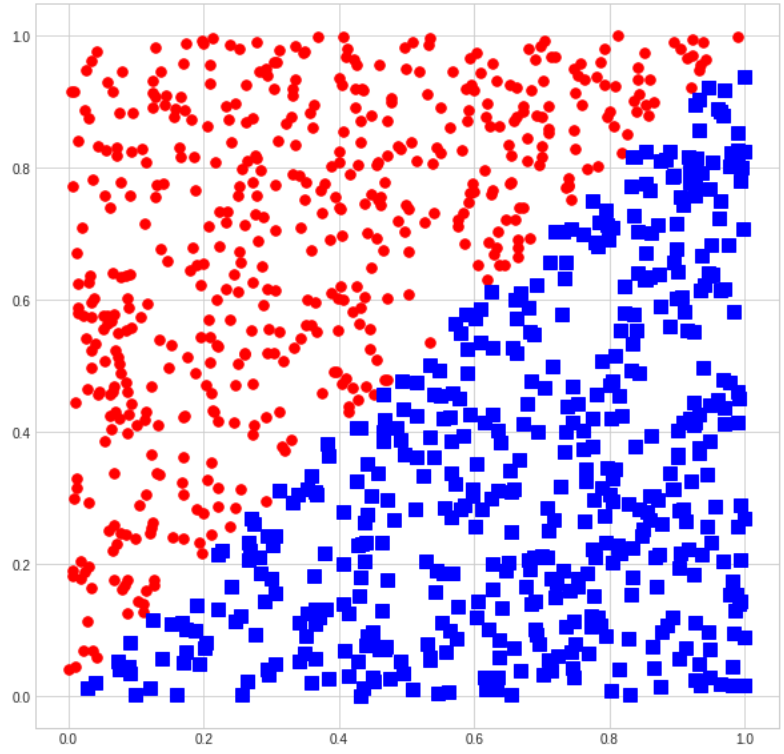
$$E(\boldsymbol{x}) = -\log P(y|\boldsymbol{x})$$

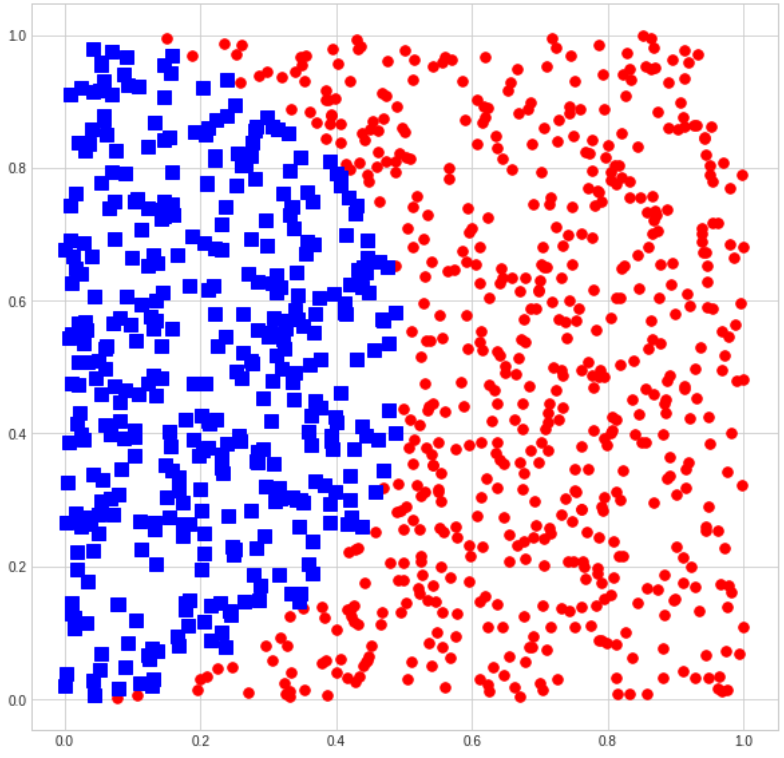
$$w_i = w_i + \eta \frac{1}{|X_b|} \sum_{\boldsymbol{x}_n \in X_b} y_n x_{ni} (1 - P(y = y_n | x_n))$$

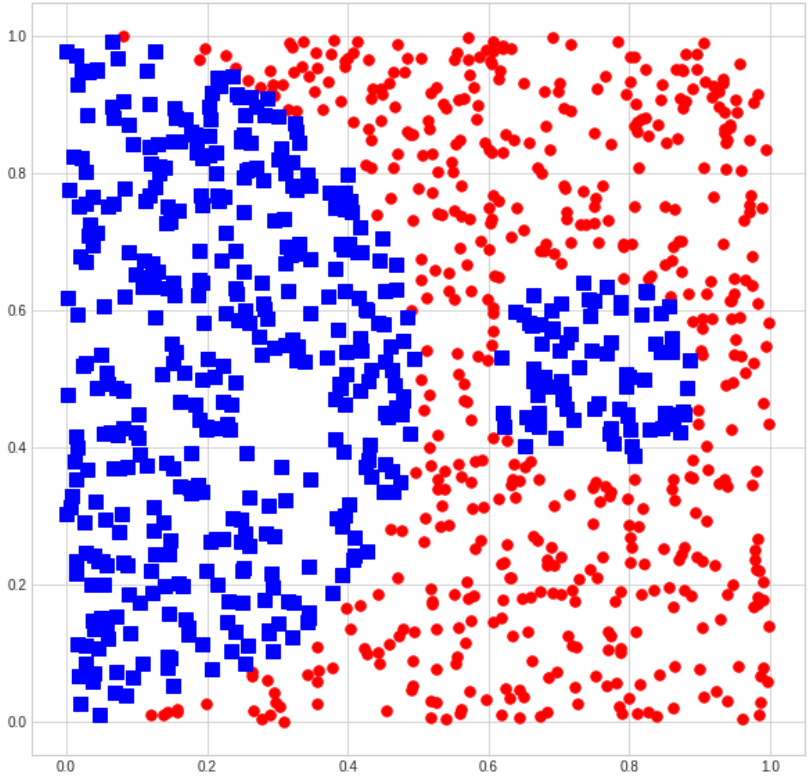
decrease –logP(y|x) → increase logP(y|x)

Is a perceptron enough?

Which of these can a perceptron solve (fit with zero training error)?







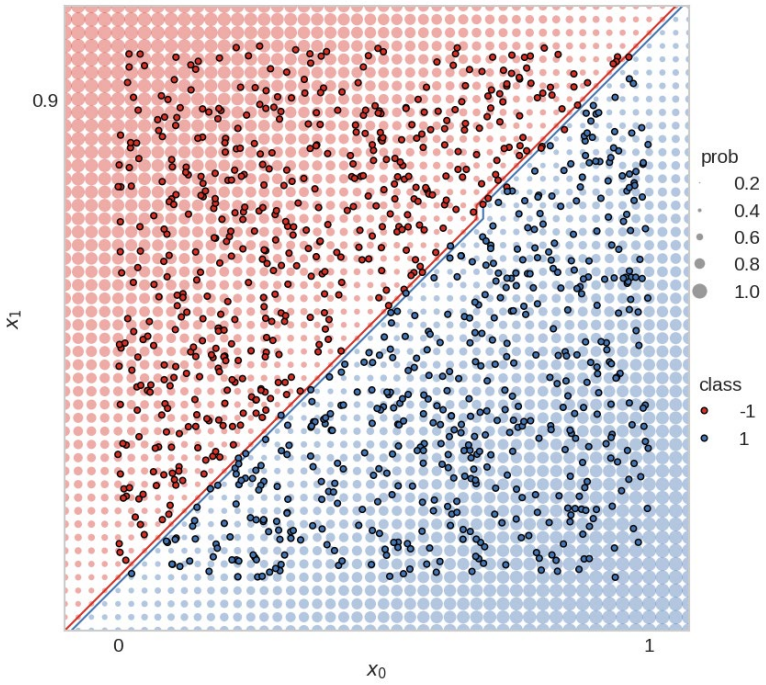
Demo

https://colab.research.google.com/drive/1nKNJyolqgzW53Rz59M2BZtyQM8bbrExb?usp=sharing

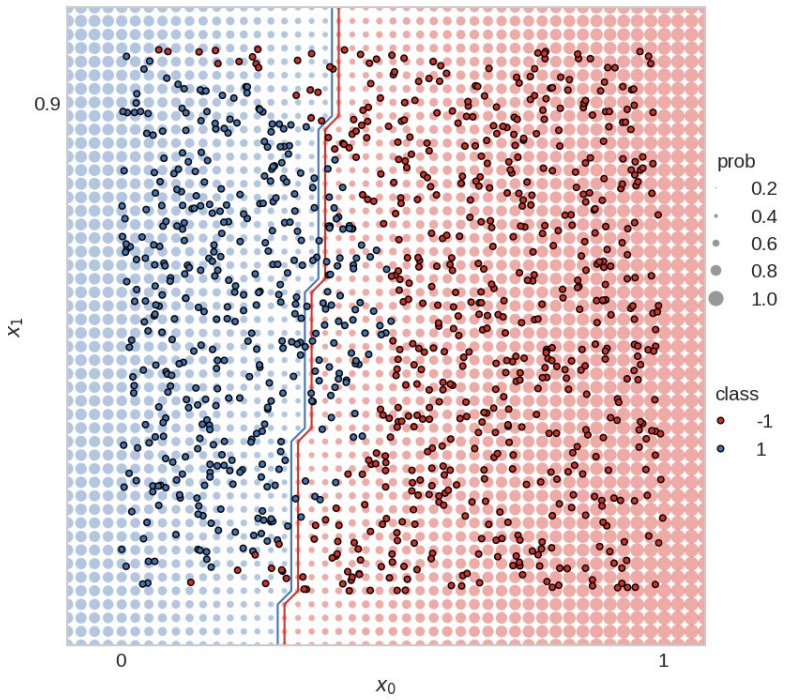
Perceptron is often not enough

- Perceptron is linear, but we often need a non-linear prediction function

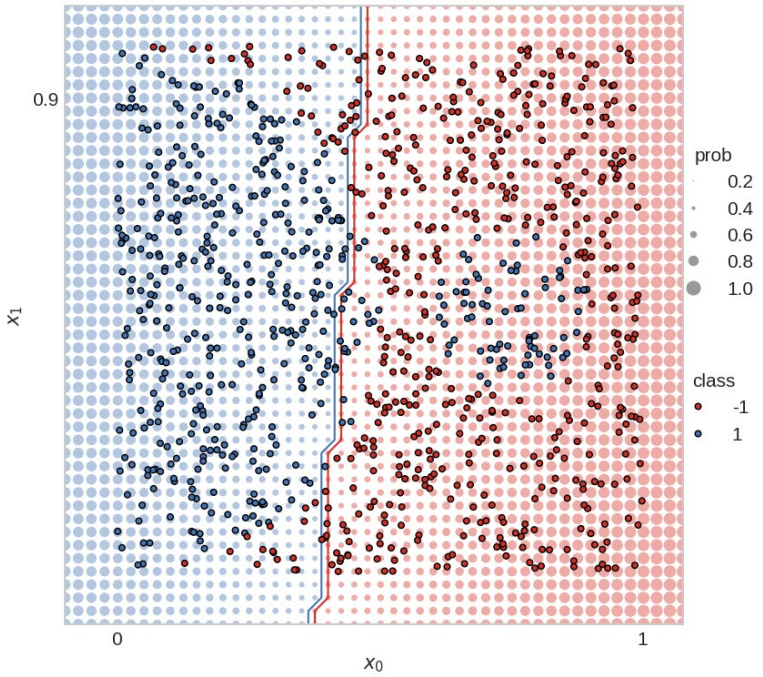
Which of these can a perceptron solve (fit with zero training error)?



Yes

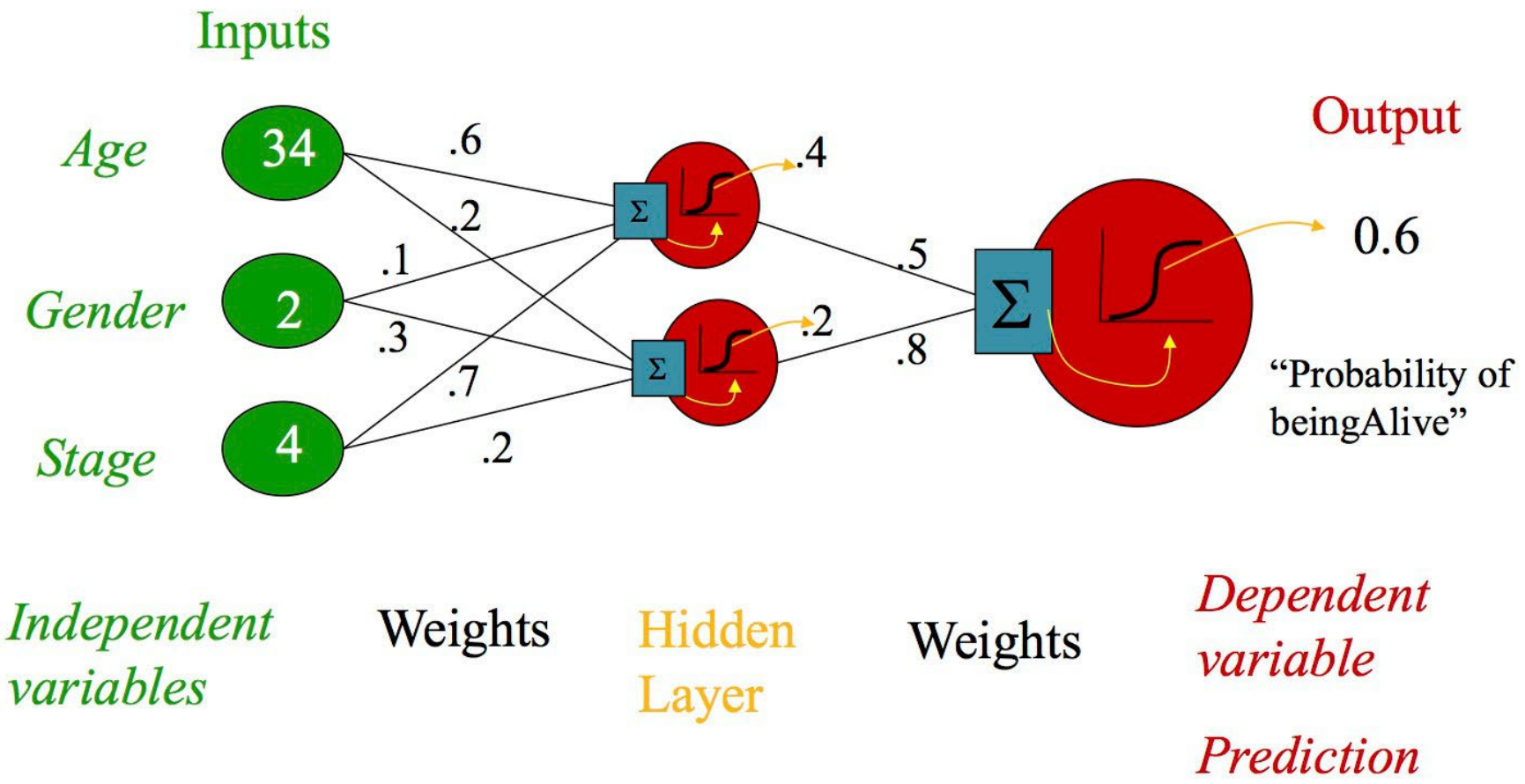


No



Not even close

Multi-Layer Perceptron (MLP)



Nodes in hidden layer(s) encode *latent* relationships

Latent = hidden, not explicitly identified

Fig source: CS 440

Example MLP for MNIST Digits

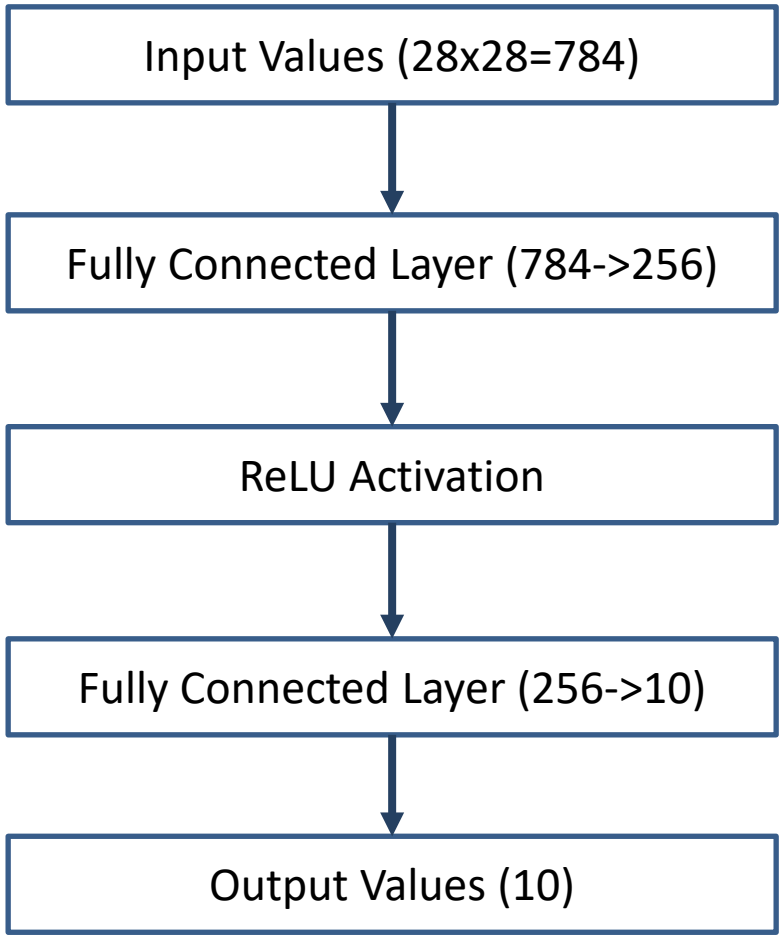
- **Input**: # of features (one per pixel)
- Fully connected (**FC**) layer(s): linear feature transformations
- Non-linear **activation**: enables complex functions to be modeled by multiple FC layers
- **Output**: score per class

$$\boldsymbol{x}_0$$
$$|\boldsymbol{x}_0| = 784$$

$$\boldsymbol{x}_1 = W_{10}\boldsymbol{x}_0$$
$$W_{10}.shape = (256, 784)$$
$$|\boldsymbol{x}_1| = 256$$

$$\boldsymbol{x}_2 = \max(\boldsymbol{x}_1, 0)$$
$$|\boldsymbol{x}_2| = 256$$

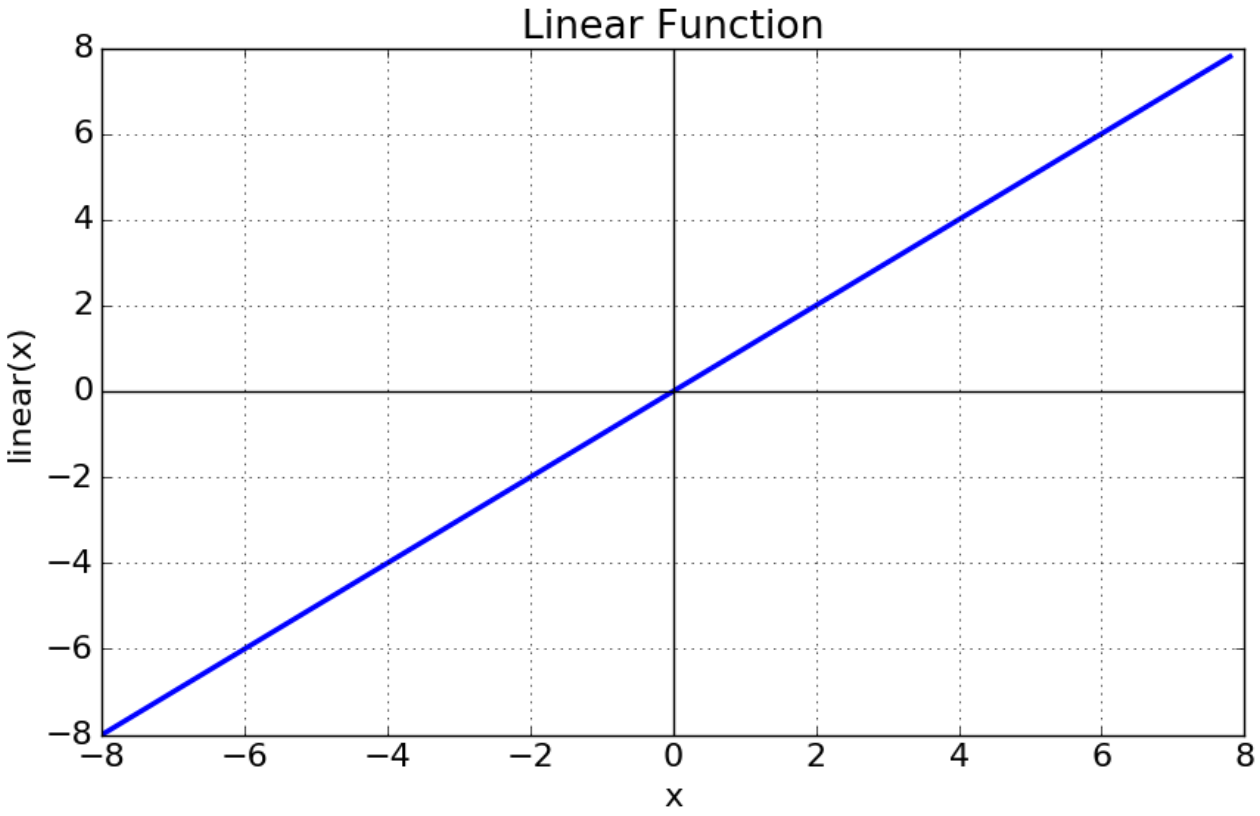
$$\boldsymbol{x}_3 = W_{32}\boldsymbol{x}_2$$
$$W_{32}.shape = (10,256)$$
$$|\boldsymbol{x}_3| = 10$$



Total parameters: (256 x (784+1)) + (10 x (256+1)), +1 is for bias terms

Linear activation

- A no-op activation (i.e. nothing happens)
- Could be used for information compression or data alignment
- Multiple stacked linear layers are equivalent to a single linear layer

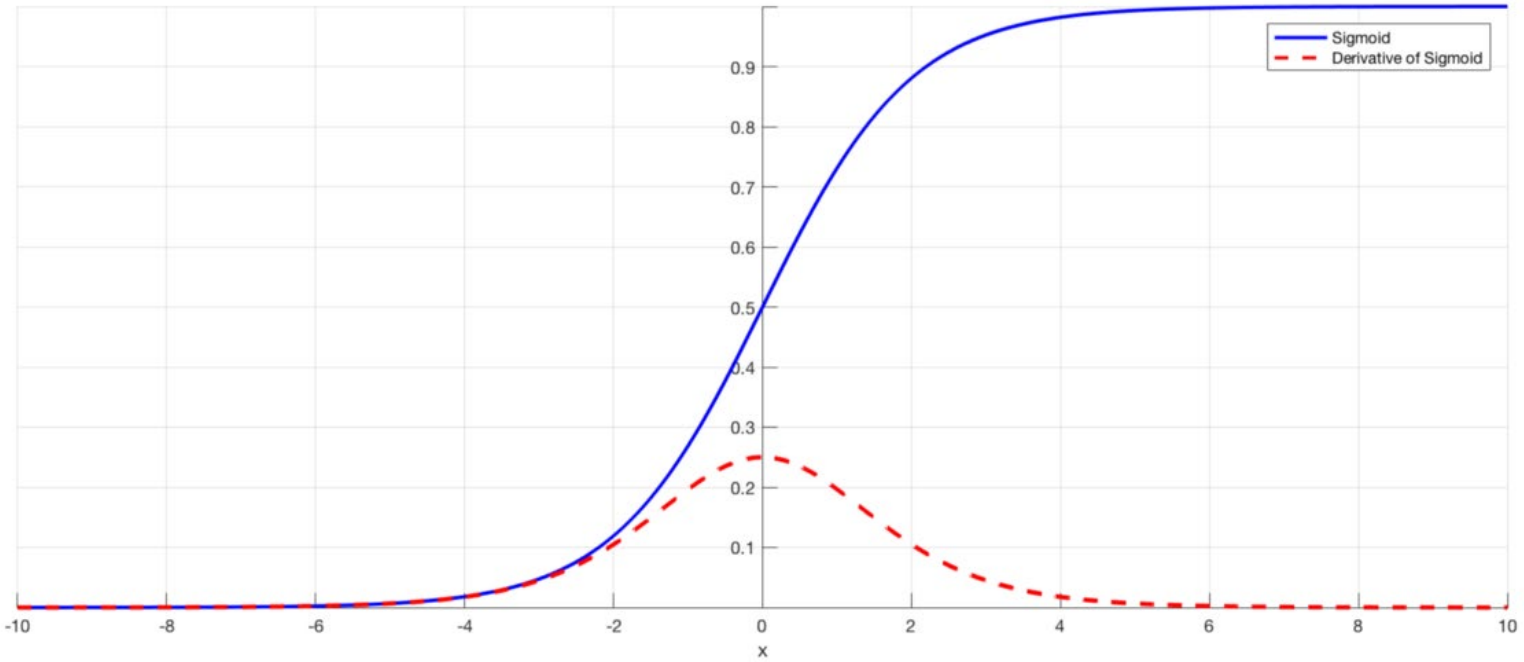


$$f(x) = x$$

$$f'(x) = 1$$

Sigmoid activation

- Maps any value to 0 to 1 range
- Traditionally, a common choice for internal layers
- But weak gradients at extremum make it difficult to optimize if there are many layers ("vanishing gradient problem")
- Common choice for output layer to map to a probability



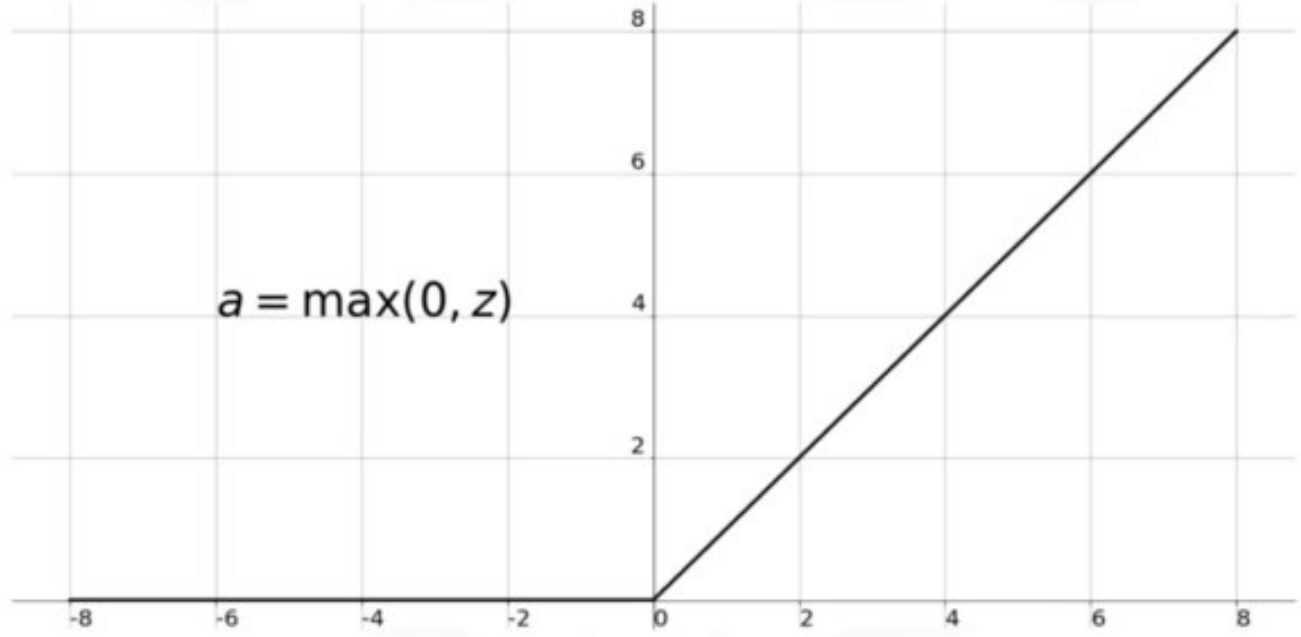
$$f(x) = \frac{1}{1 + \exp(-x)}$$

$$f'(x) = f(x)(1 - f(x))$$

Fig source

ReLU (Rectified Linear Unit) activation

- Maps negative values to zero; others pass through
- Most choice for internal layers in current deep networks
- Results in sparse network activations, and all positive values have gradient of 1



$$f(x) = \max(0, x)$$

$$f'(x) = \delta(x > 0)$$

Fig source

MLP Architectures: Hidden Layers and Nodes

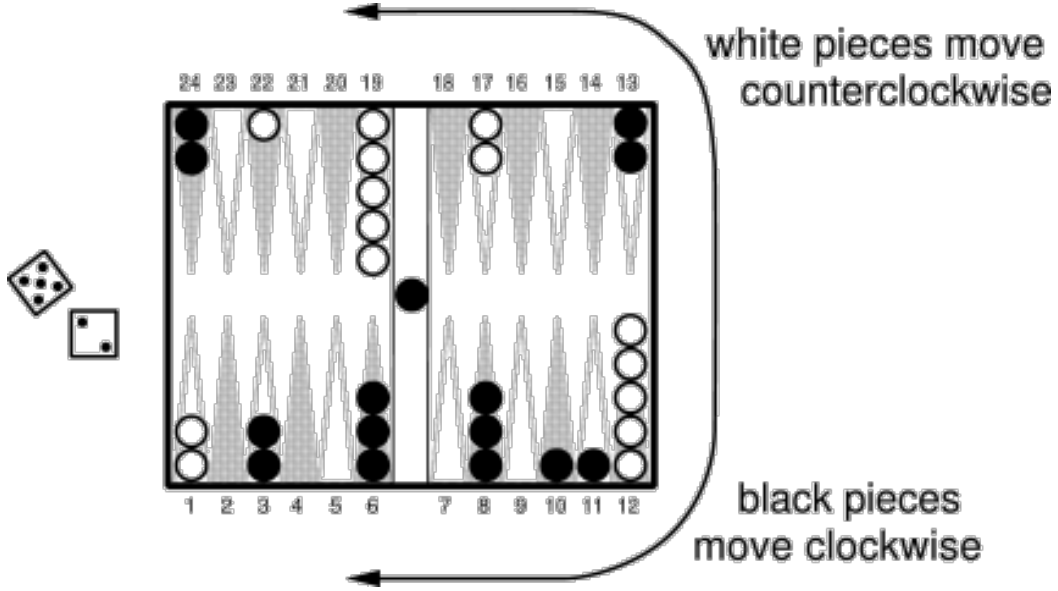
- Number of internal ("hidden") layers
  - Without hidden layers, neural networks (a.k.a. perceptron or linear logistic regressor) can fit linear decision boundaries
  - With enough nodes in one hidden layer, any Boolean function can be fit but the number of nodes required grows exponentially in the worst case (because the nodes can enumerate all joint combinations)
  - Every bounded continuous function can be approximated with one hidden sigmoid layer and one linear output layer
  - Any function can be approximated to arbitrary accuracy by a network with two hidden layers with sigmoid activation (Cybenko 1988)
  - Does it ever make sense to have more than two internal layers?

- Number of nodes per hidden layer (often called the "width")
  - More nodes means more representational power and more parameters

- Each layer has an activation function

Application Example: Backgammon (1992)

- 198 inputs: how many pieces on each space
  - Later versions had expert-defined features
- 1 internal FC layer with sigmoid activation
- Reinforcement learning: reward is evaluation of game position or result
- Network competed well with world experts, demonstrating power of ML

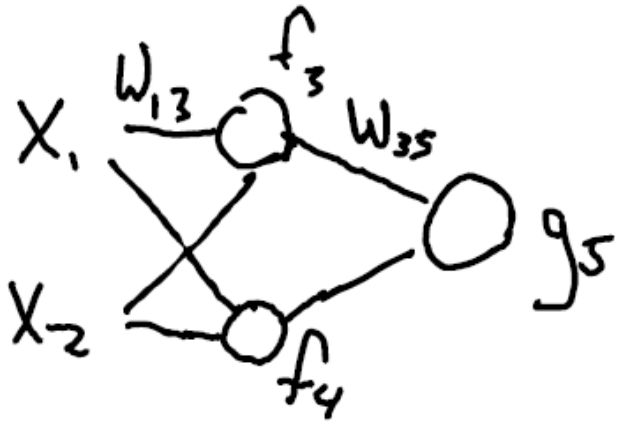


Initialize $\vec{\theta}$ arbitrarily
Repeat (for each episode):
  $\vec{e} = 0$
  $s \leftarrow$ initial state of episode
  Repeat (for each step of episode):
    $a \leftarrow$ action given by π for s
    Take action a, observe reward, r, and next state, s'
    $\delta \leftarrow r + \gamma V(s') - V(s)$
    $\vec{e} \leftarrow \gamma\lambda\vec{e} + \nabla_{\vec{\theta}} V(s)$
    $\vec{\theta} \leftarrow \vec{\theta} + \alpha\delta\vec{e}$
    $s \leftarrow s'$
  until s is terminal

Program	Hidden units	Training games	Opponents	Results
TD-Gammon 0.0	40	200,000	other programs	Tied for best
TD-Gammon 1.0	80	300,000	Robertie, Magriel, Davis	-13 pts / 51 games
TD-Gammon 2.0	40	800,000	various Grandmasters	-7 pts / 38 games
TD-Gammon 2.1	80	1,500,000	Robertie	-1 pts / 40 games
TD-Gammon 3.0	80	1,500,000	Kazaros	+6 pts / 20 games

https://en.wikipedia.org/wiki/TD-Gammon Fig source

Back-propagation: network example



Consider this simple network

- Two inputs
- Two nodes in hidden layer
- One output
- For now, linear activation

$$g_5(\underline{x}) = g_5(f_3(x), f_4(x))$$
$$= W_{35} \cdot f_3(x) + W_{45} \cdot f_4(x)$$

Output is a weighted sum of middle nodes

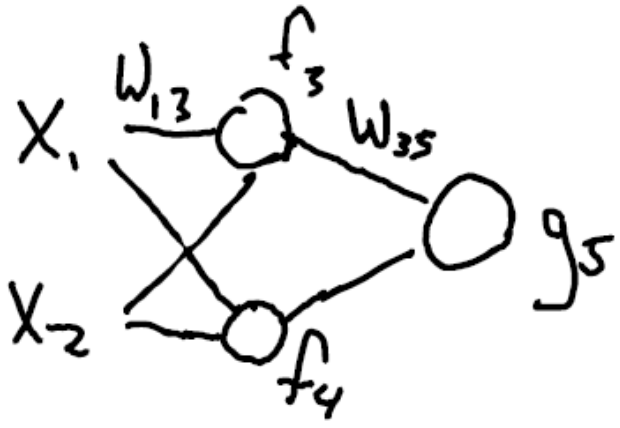
$$f_3(\underline{x}) = W_{13} \cdot X_1 + W_{23} \cdot X_2$$

Each middle node is a weighted sum of inputs

$$E(g_5(x), y) = (g_5(x) - y)^2$$

Error function is squared error

Back-propagation: output weights



$$g_5(\underline{x}) = g_5(f_3(x), f_4(x))$$
$$= w_{35} \cdot f_3(x) + w_{45} \cdot f_4(x)$$
$$f_3(\underline{x}) = w_{13} \cdot x_1 + w_{23} \cdot x_2$$
$$E(g_5(x), y) = (g_5(x) - y)^2$$

Chain rule:

$$h(x) = f(g(x))$$
$$h'(x) = f'(g(x))\, g'(x)$$

$$\partial E / \partial w_{35} = 2 \cdot (g_5(x) - y)\ \partial g_5(x) / \partial w_{35}$$
$$= 2 \cdot (g_5(x) - y)\, f_3(x)$$

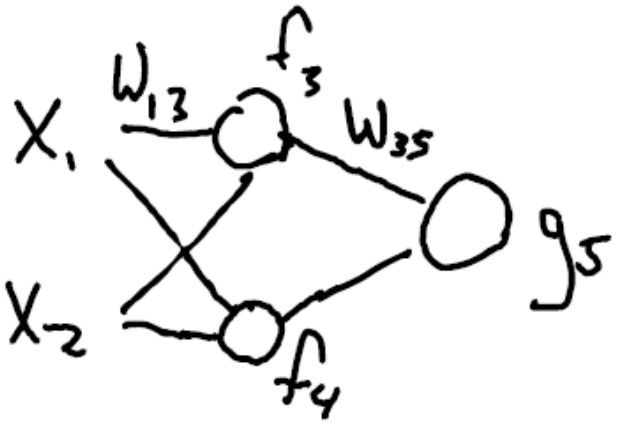
Apply chain rule to solve for error gradient wrt w_{35}

$$w_{35} = w_{35} - \eta \left[2 \cdot (g_5(x) - y)\, f_3(x) \right]$$

Take step in negative gradient direction

Error gradient Input to weight

Back-propagation: internal weights



$$g_5(\underline{x}) = g_5(f_3(x), f_4(x))$$
$$= w_{35} \cdot f_3(x) + w_{45} \cdot f_4(x)$$
$$f_3(\underline{x}) = w_{13} \cdot x_1 + w_{23} \cdot x_2$$
$$E(g_5(x), y) = (g_5(x) - y)^2$$

Chain rule:

$$h(x) = f(g(x))$$
$$h'(x) = f'(g(x))\, g'(x)$$

$$\partial E / \partial w_{13} = 2 \cdot (g_5(x) - y)\,(\partial g_5(x) / \partial w_{13})$$
$$(\partial (w_{35} f_3(x) + w_{45} f_4(x)) / \partial w_{13}$$
$$(\partial (w_{35} (w_{13} x_1 + w_{23} x_2)) / \partial w_{13})$$
$$(w_{35} x_1)$$
$$= 2 \cdot (g_5(x) - y)(w_{35} x_1)$$

Gradient of this output

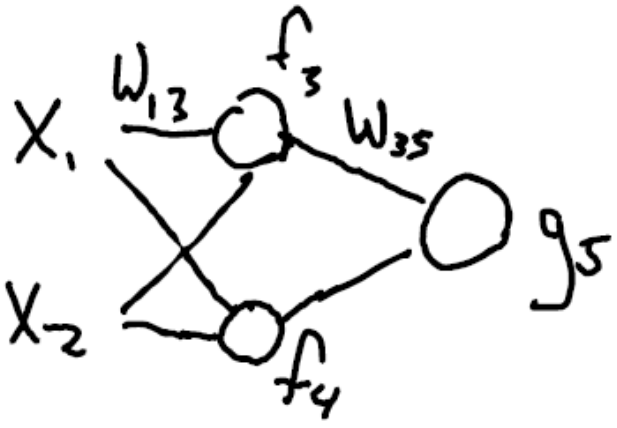
Error gradient

Contribution of this output to final output

Chain rule is applied recursively, since w_{13} affects $f_3(x)$

Gradient update is product of gradient to output, gradient of this output to final output, and error gradient of final output

What if f3 had ReLU activation?



$$g_5(\underline{x}) = g_5(f_3(x), f_4(x))$$
$$= w_{35} \cdot f_3(x) + w_{45} \cdot f_4(x)$$

$$f_3(\underline{x}) = w_{13} \cdot x_1 + w_{23} \cdot x_2$$ (before, with linear activation)

$$E(g_5(x), y) = (g_5(x) - y)^2$$

Chain rule:

$$h(x) = f(g(x))$$
$$h'(x) = f'(g(x))g'(x)$$

What if $f_3(x)$ is ReLU $\quad f_3(x) = w_{13} \max(x_1, 0) + w_{23} \cdot \max(x_2, 0)$

Then $\partial E / \partial w_{13} = 2 \cdot (g_5(x) - y)(w_{35} \max(x_1, 0))$

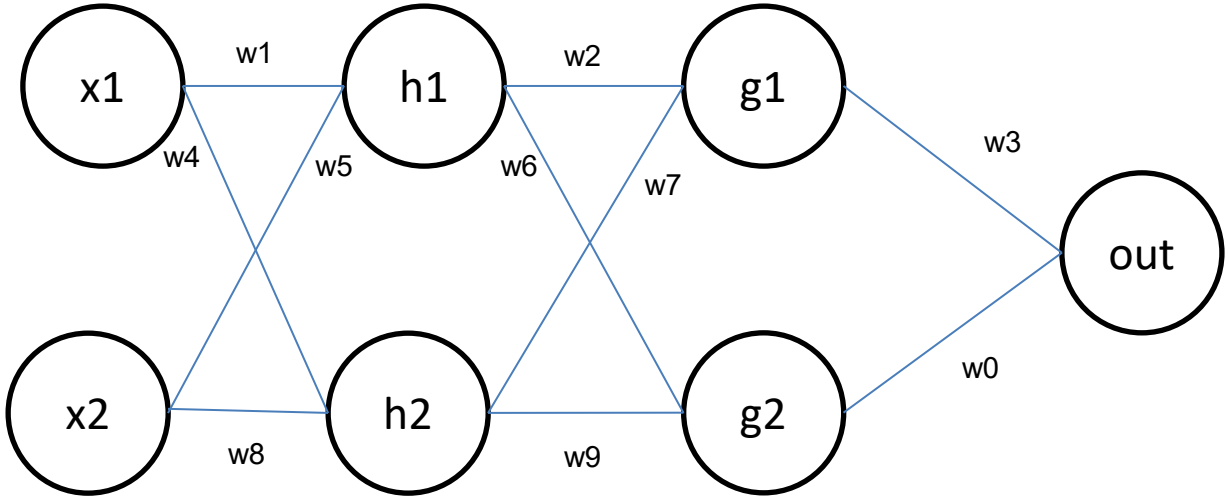
Gradient is zero if x1<=0; otherwise, same as for linear activation

Backpropagation: General Concept

- Each weight's gradient based on one training sample is a product of:
  - Gradient of loss function (should final output have been higher or lower)
  - Gradient of prediction wrt activation (how this node's activation is affecting the final prediction)
  - Gradient of activation wrt weight (how is the weight changing its node's activation)

2 Min Break Question

Assuming all linear layers (for simplicity), fill in the terms for the error gradients



Error gradient wrt **w8** = 2*(out-y)*(w0 * w9 + w3 * w7) * x2

Error gradient wrt **w2** = 2*(out-y)*(w3) * h1

MLP Optimization by SGD

For each epoch t:

Split data into batches

$\eta = 0.001$ (or some schedule)

For each batch X_b:

1. Compute output
2. Evaluate loss
3. Compute gradients with backpropagation
4. Update the weights

What is the benefit and cost of going from a perceptron to MLP?

Benefit

1. Much greater expressivity, can model non-linear functions

Cost

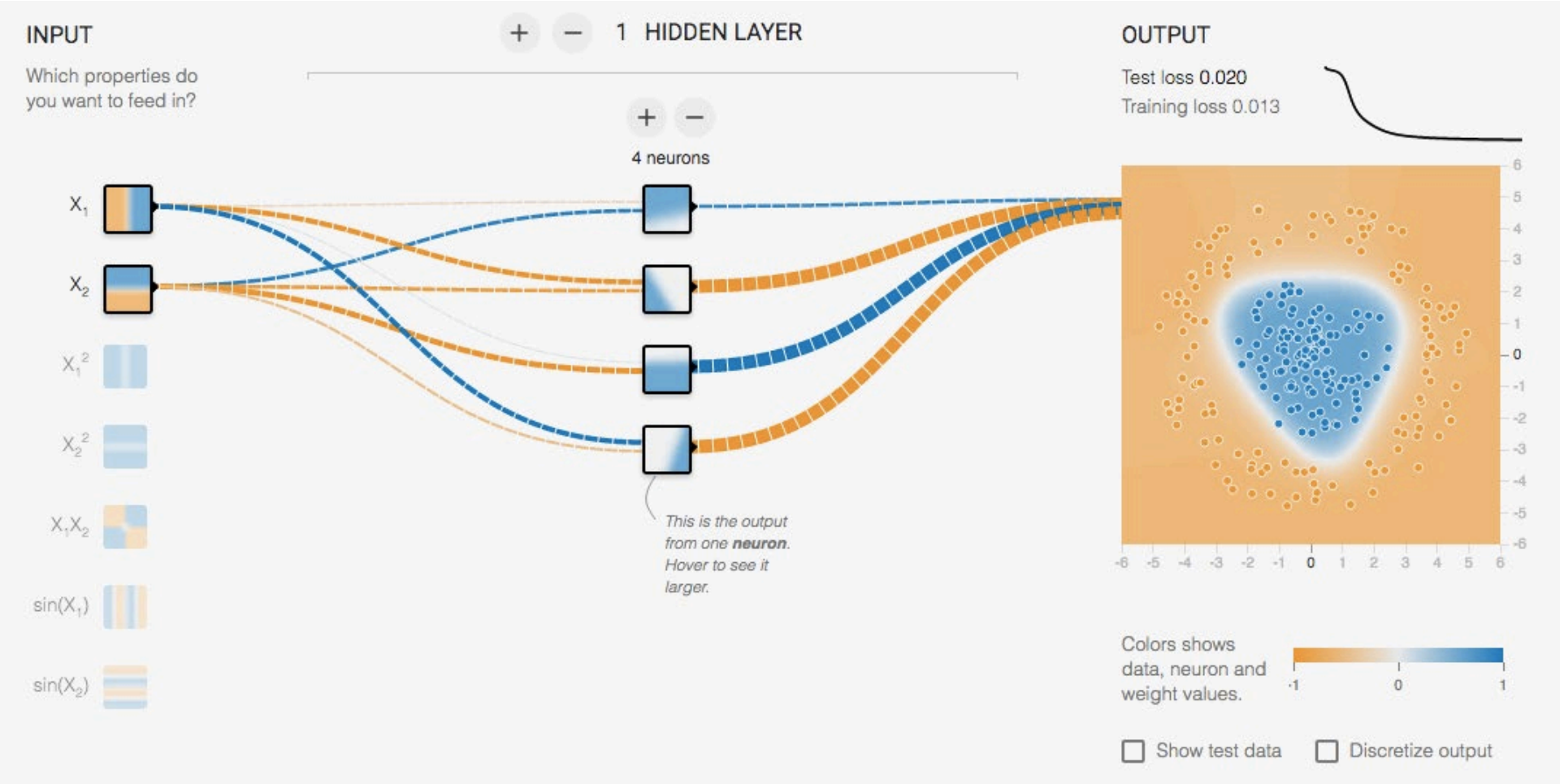
1. Optimization is no longer convex, globally optimum solution no longer guaranteed (or even likely)
2. Larger model = more training and inference time
3. Larger model = more data required to obtain a good fit

In summary: MLP has lower bias and higher variance, and additional error due to optimization challenges

Demo: Part 2

https://colab.research.google.com/drive/1nKNJyolqgzW53Rz59M2BZtyQM8bbrExb?usp=sharing

Multi-Layer Network Demo



http://playground.tensorflow.org/

Try many layers with sigmoid vs relu

Slide: Lazebnik

Another application example: mapping position/rays to color

(a) Coordinate-based MLP

(b) Image regression $(x,y) \rightarrow$ RGB

(c) 3D shape regression $(x,y,z) \rightarrow$ occupancy

(d) MRI reconstruction $(x,y,z) \rightarrow$ density

(e) Inverse rendering $(x,y,z) \rightarrow$ RGB, density

- L2 loss
- ReLU MLP with 4 layers and 256 channels (nodes per layer)
- Sigmoid activation on output
- 256 frequency positional encoding

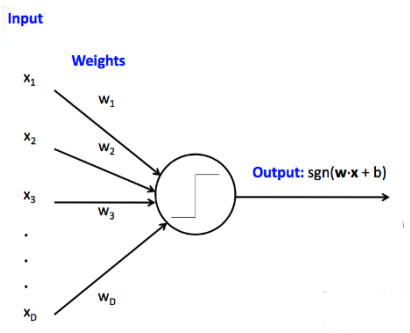
Fourier Features (Tancik et al. 2020) NeRF (Mildenhall et al. 2020)

HW 2

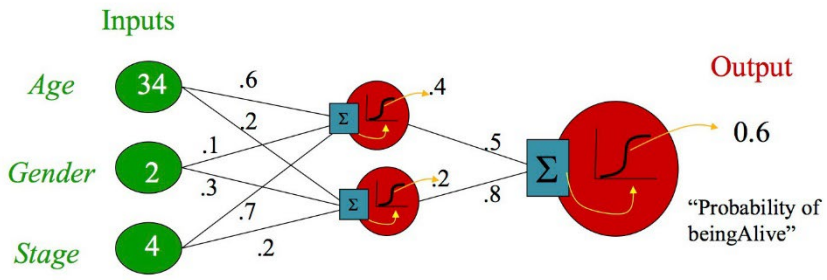
https://docs.google.com/document/d/13vTEGx3fdfc4rtcF86xoUyXm6eHmuvys5ZkMbcEgK4s/edit

What to remember

- Perceptrons are linear prediction models



- MLPs are non-linear prediction models, composed of multiple linear layers with non-linear activations



- MLPs can model more complex functions, but are harder to optimize

- Optimization is by a form of stochastic gradient descent

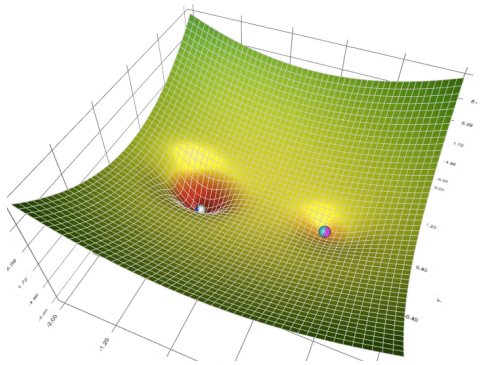


Figure source

Next class

- Deep learning
  - Background
  - AlexNet
  - Adam Optimization
  - Residual Networks