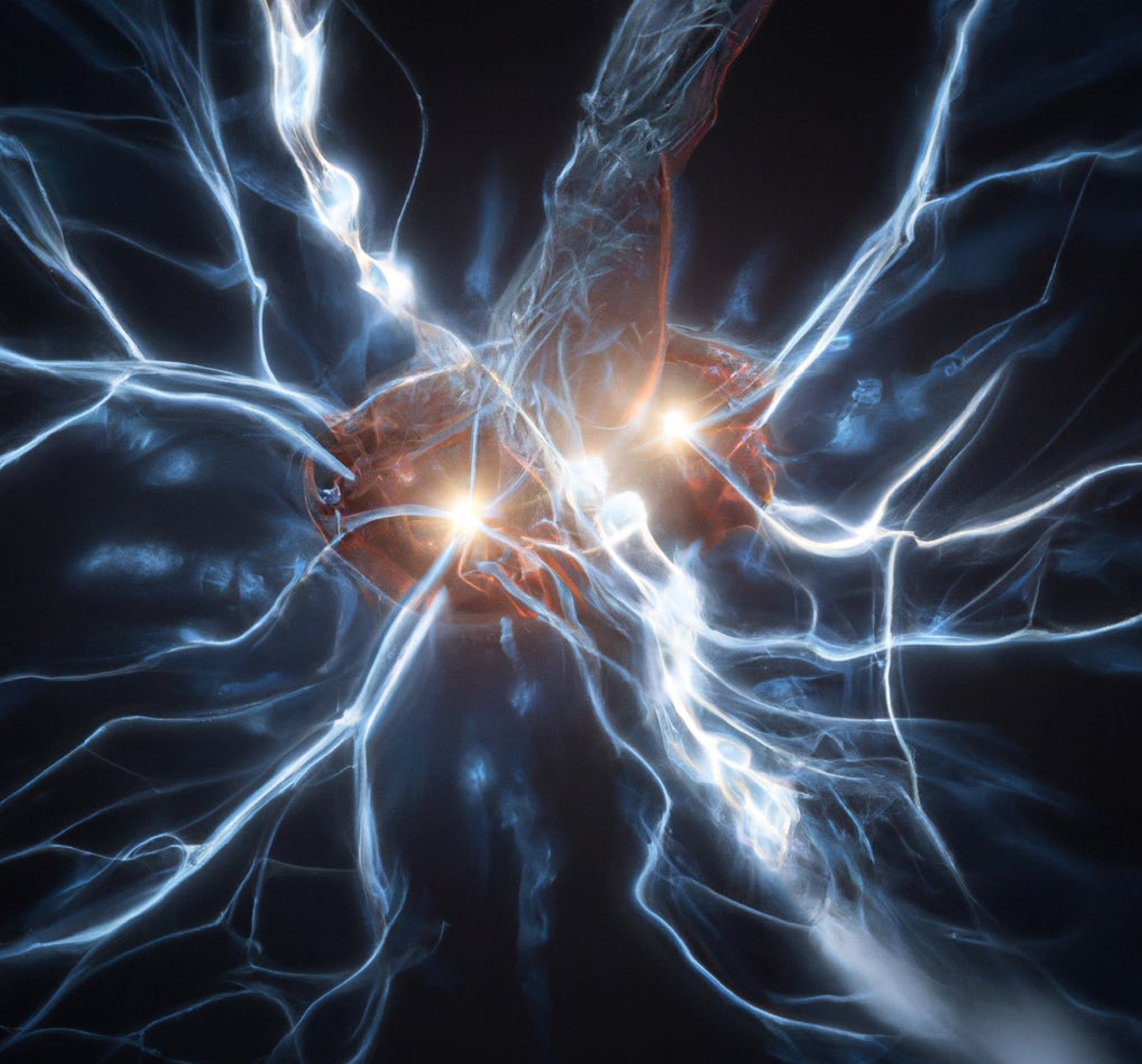


Dall-E

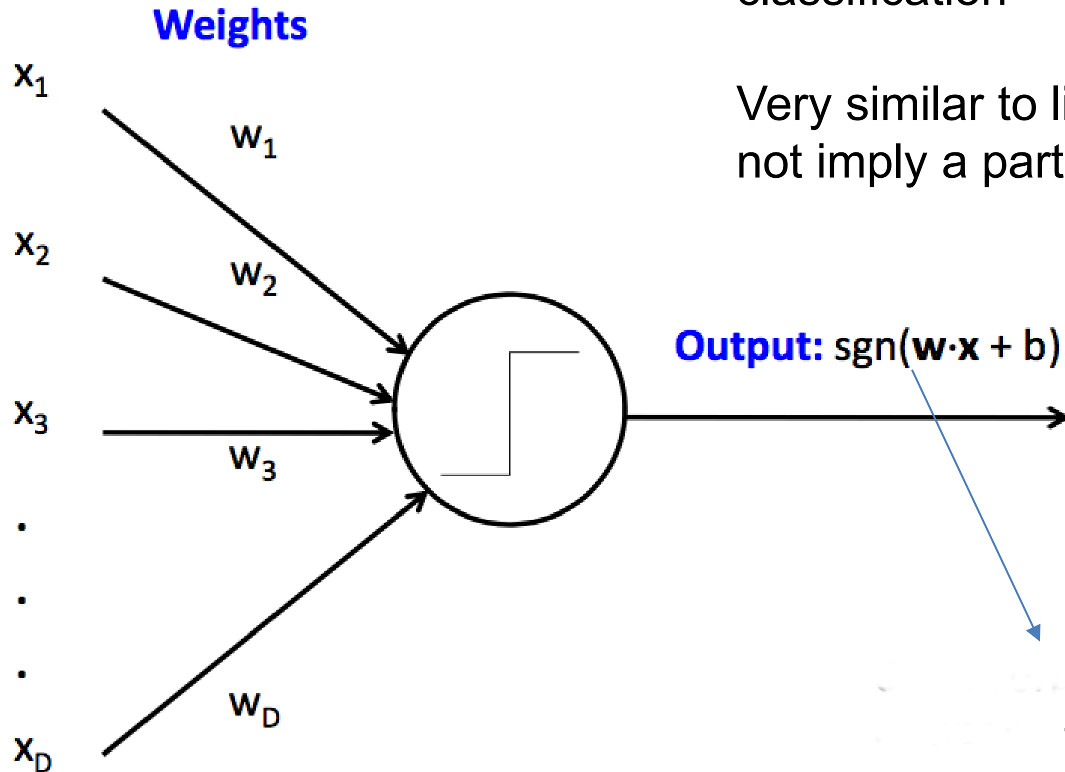
MLPs and Backprop

Applied Machine Learning
Derek Hoiem

Multi-layer Perceptrons (MLPs)

- Recap of Perceptrons and SGD
- 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

Output: $\text{sgn}(\mathbf{w} \cdot \mathbf{x} + b)$

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

Fig source: CS 440

Perceptron Update Rule

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

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

prediction target

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

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

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

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

Make error *lower* Learning rate

Chain Rule:

$h(x) = f(g(x))$, then

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

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:

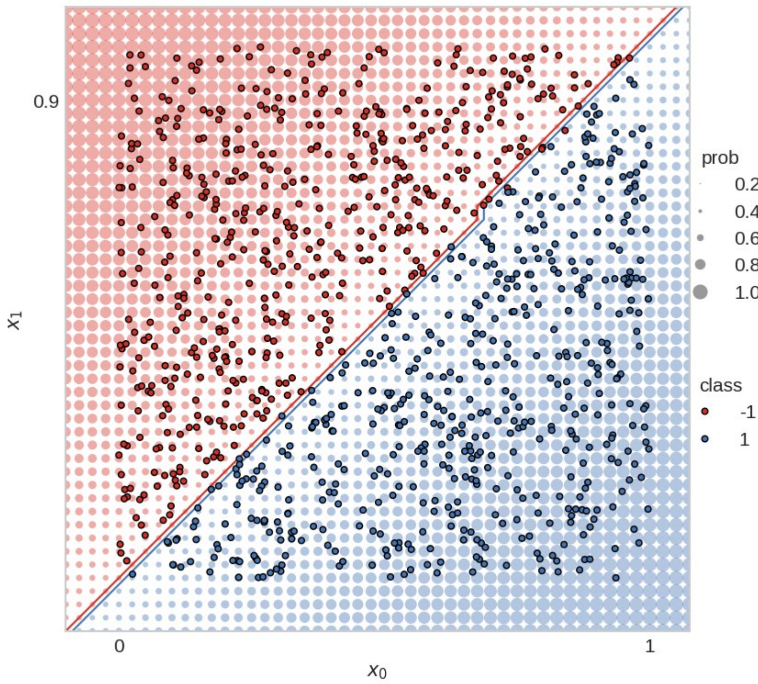
For each weight w_i:

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

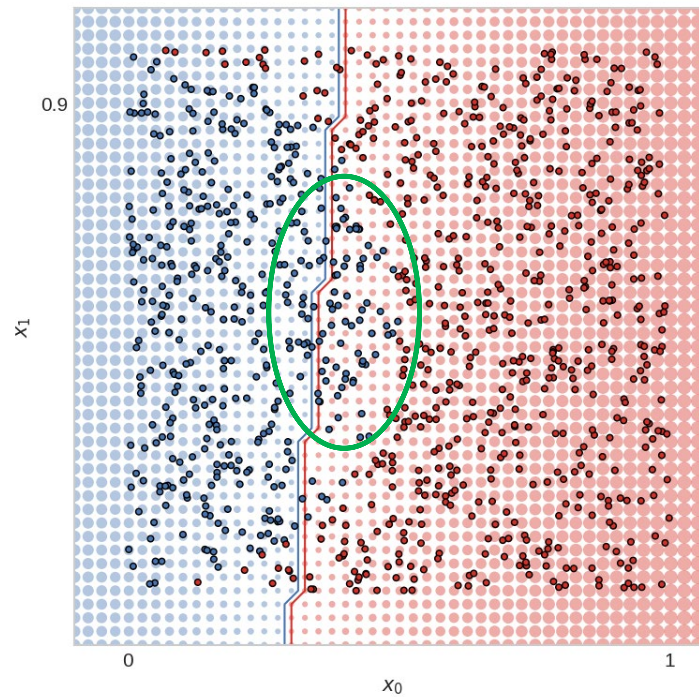
Perceptron is often not enough

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

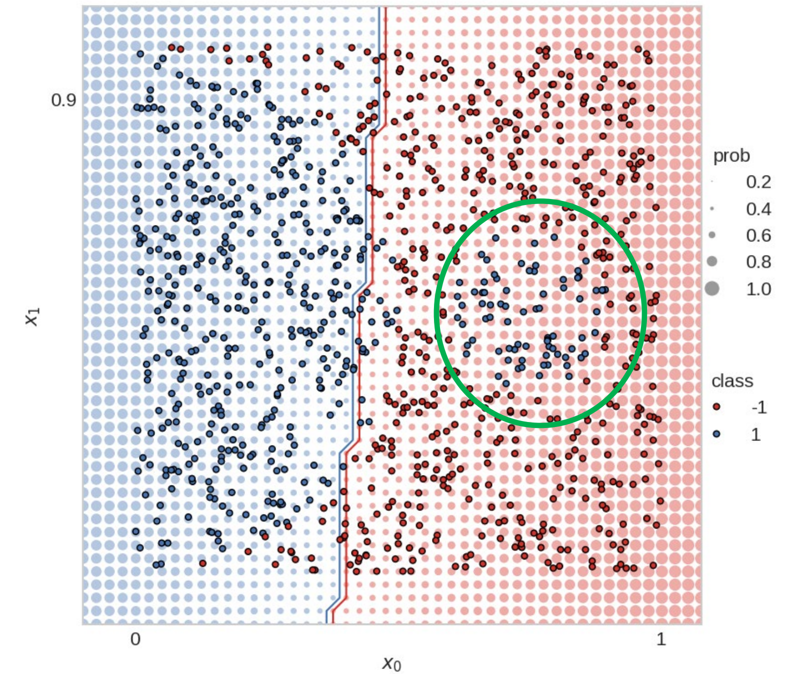
Which of these can a perceptron solve (fit with zero training error)?



Yes

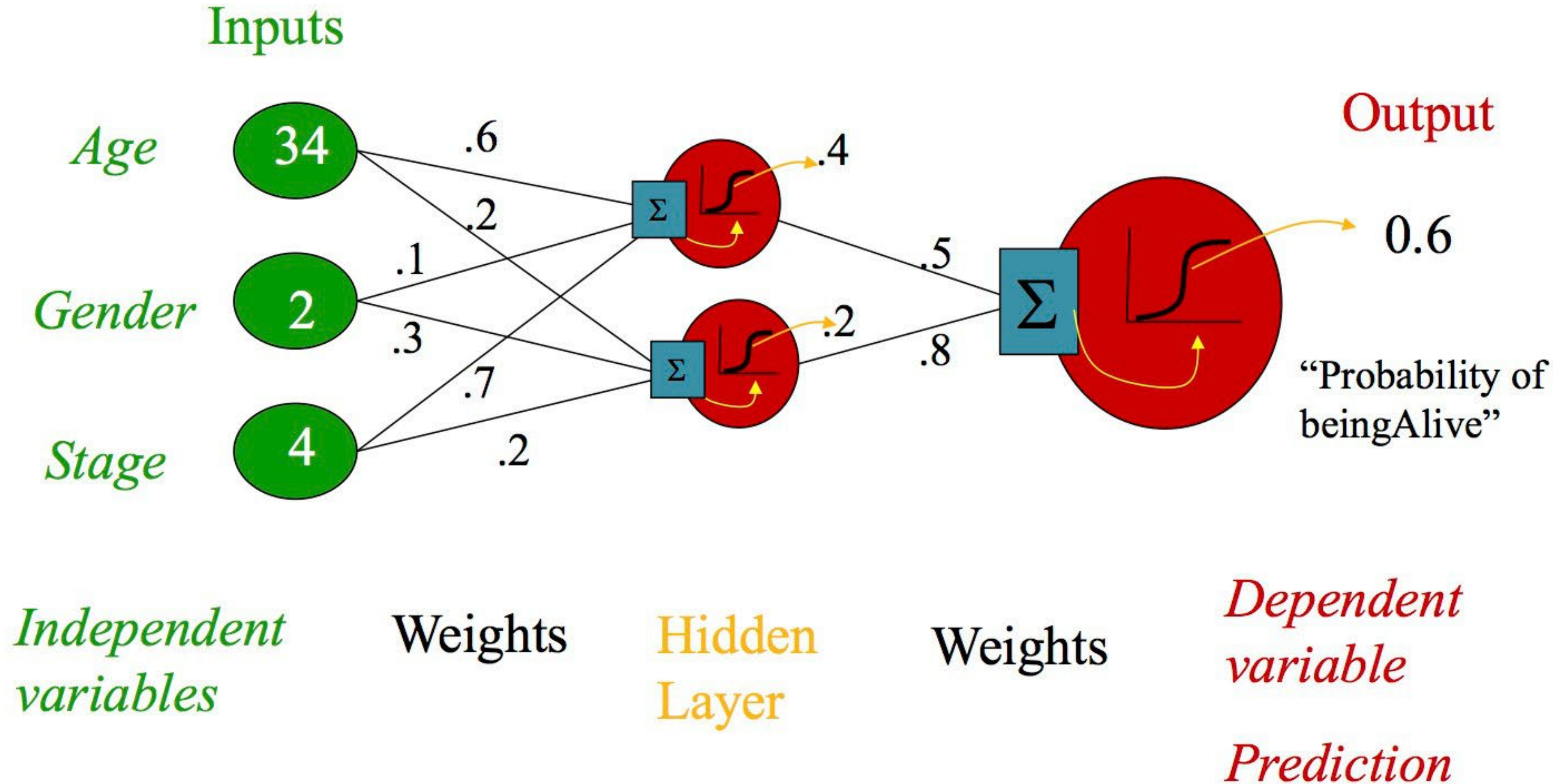


No



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

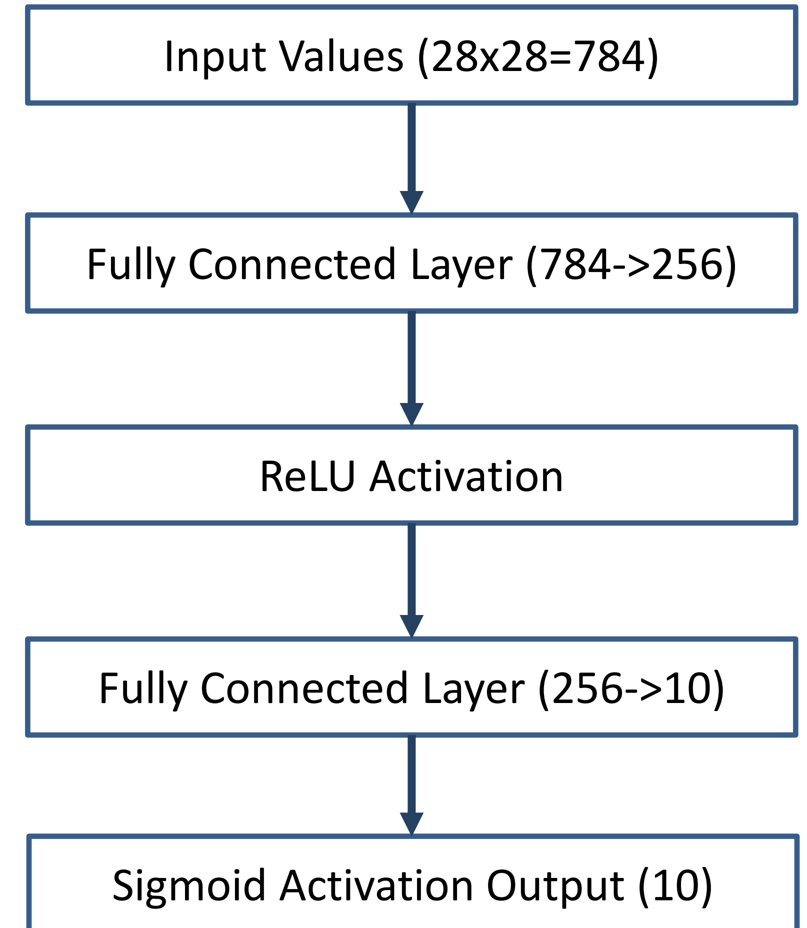
$$\boldsymbol{x}_0$$
$$|\boldsymbol{x}_0| = 784$$

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

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

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

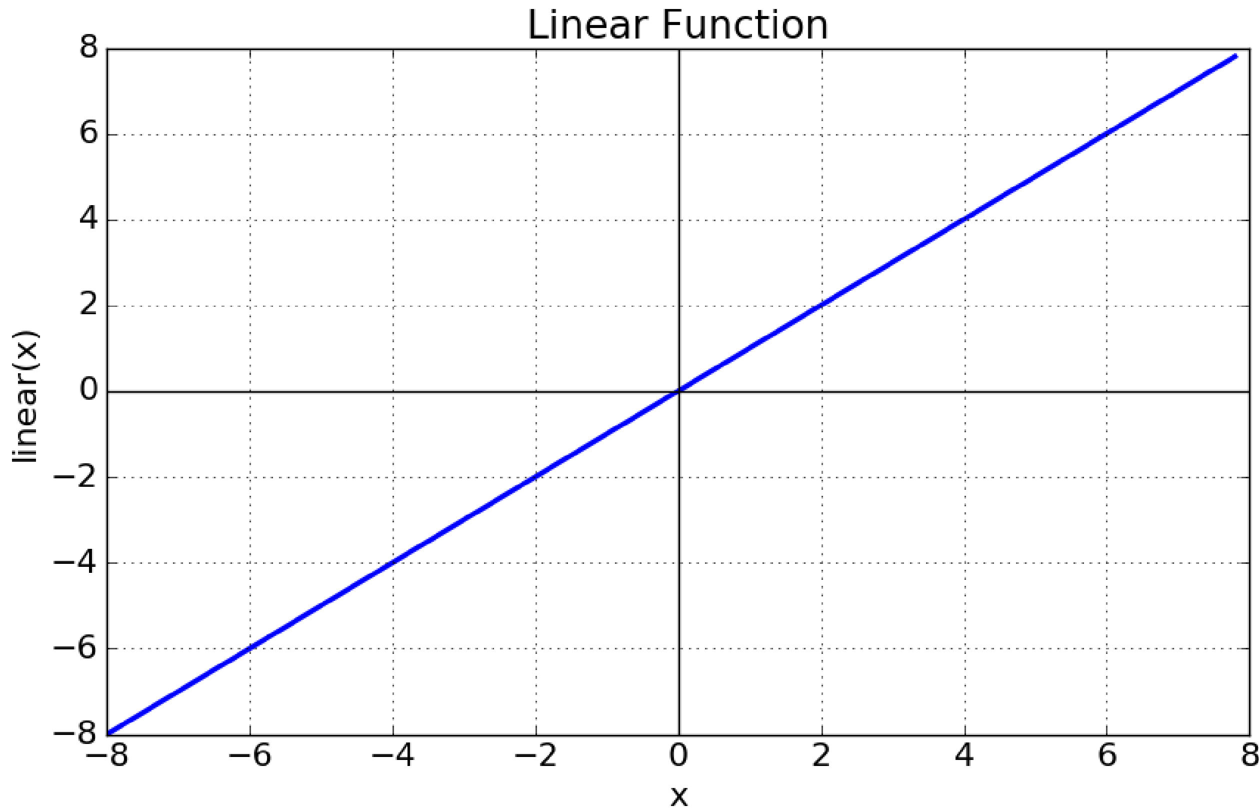
$$\boldsymbol{x}_{out} = 1/(1 + \exp(-\boldsymbol{x}_3))$$
$$|\boldsymbol{x}_{out}| = 10$$



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



$$f(x) = x$$

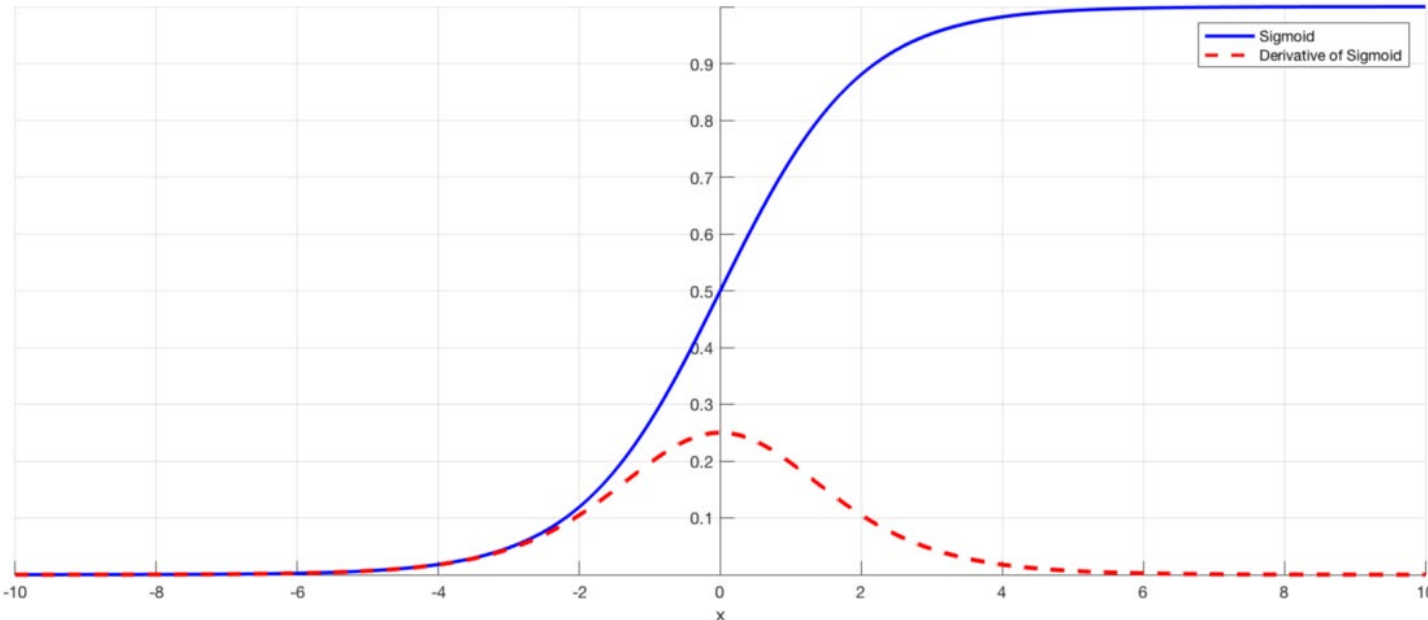
$$f'(x) = 1$$

Sigmoid activation

- Maps any value to 0 to 1 range
- Traditionally, a common choice for internal layers
- Common choice for output layer to map to a probability

  If $\mathrm{f}(x) = \log \frac{P(y=1|x)}{P(y=-1|x)}$: then $P(y=1|x) = sigmoid(f(x))$

- But weak gradients at extremum make it difficult to optimize if there are many layers (“vanishing gradient problem”)



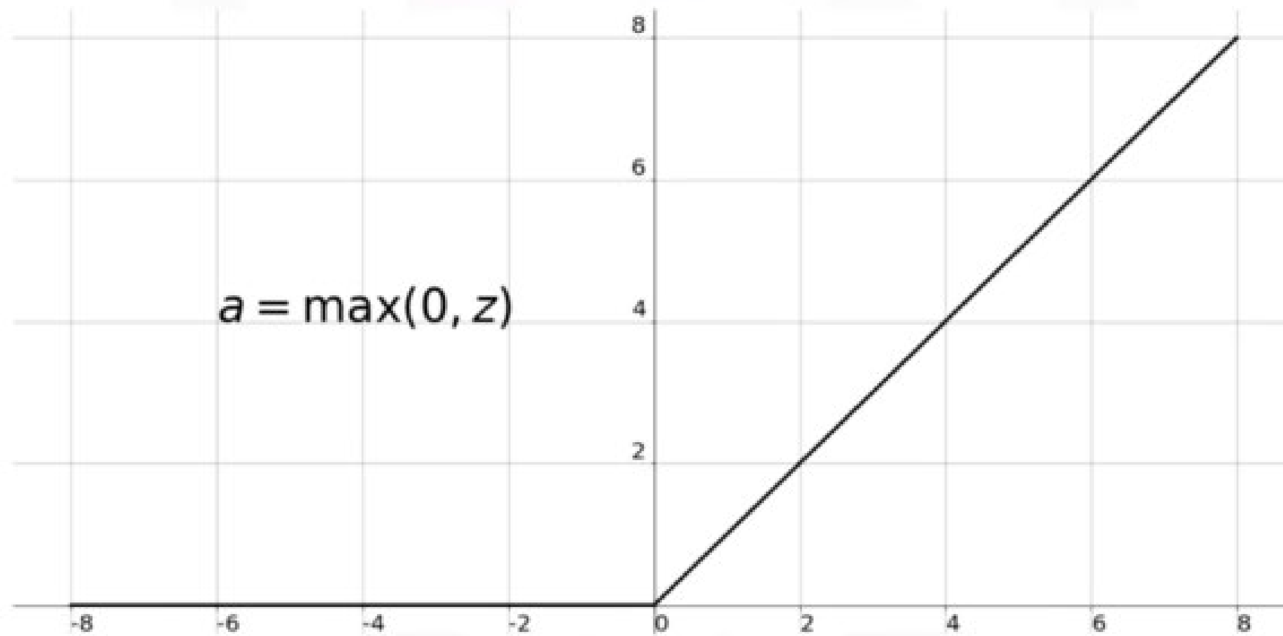
$$f(x) = \frac{1}{1 + \exp(-x)}$$

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

Fig source

ReLU (Rectified Linear Unit) activation

- Maps negative values to zero; others pass through
- Typical 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

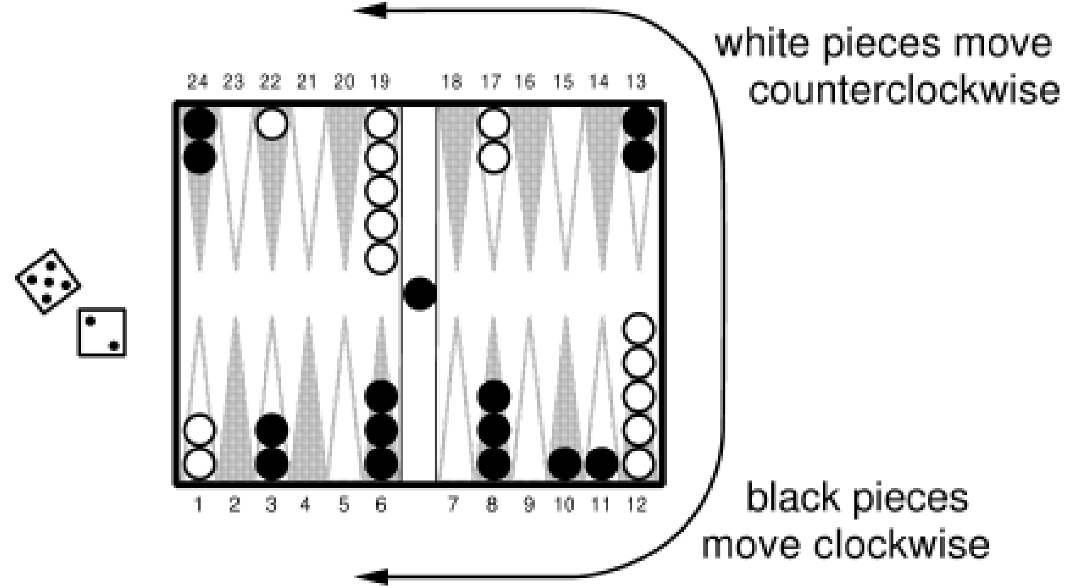
- Number of internal ("hidden") layers
  - Without hidden layers, neural networks (a.k.a. perceptron or linear logistic regressor) can fit linear decision boundaries
  - With enough nodes in one hidden layer, any Boolean function can be fit but the number of nodes required grows exponentially in the 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



After each turn, the learning algorithm updates each weight in the neural net according to the following rule:

$$w_{t+1} - w_t = \alpha(Y_{t+1} - Y_t) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w Y_k$$

where:

- $w_{t+1} - w_t$ is the amount to change the weight from its value on the previous turn.
- $Y_{t+1} - Y_t$ is the difference between the current and previous turn's board evaluations.
- α is a "learning rate" parameter.
- λ is a parameter that affects how much the present difference in board evaluations should feed back to previous estimates. $\lambda = 0$ makes the program correct only the previous turn's estimate; $\lambda = 1$ makes the program attempt to correct the estimates on all previous turns; and values of λ between 0 and 1 specify different rates at which the importance of older estimates should "decay" with time.
- $\nabla_w Y_k$ is the gradient of neural-network output with respect to weights: that is, how much changing the weight affects the output.[2]

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

Training of multi-layer networks

- Find network weights to minimize the *training error* between true and estimated labels of training examples, e.g.:

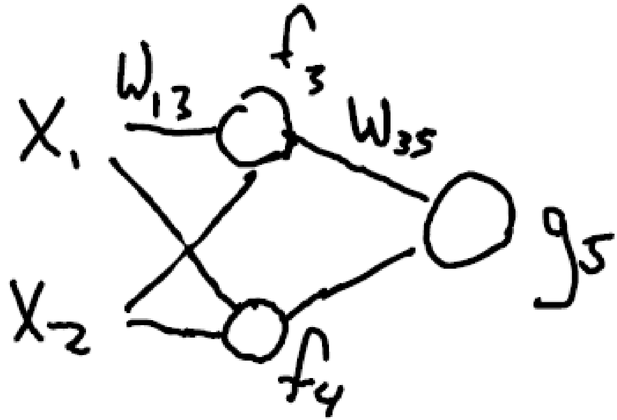
$$E(\mathbf{w}) = \sum_{i=1}^{N} \left(y_i - f_{\mathbf{w}}(\mathbf{x}_i) \right)^2$$

- Update weights by **gradient descent:** $\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial E}{\partial \mathbf{w}}$

- **Back-propagation:** gradients are computed in the direction from output to input layers and combined using chain rule
- **Stochastic gradient descent:** compute the weight update w.r.t. a small batch of examples at a time, cycle through training examples in random order in multiple epochs

Slide: Lazebnik

Back-propagation: network example



Consider this simple network
- Two inputs
- Two nodes in hidden layer
- One output
- For now, linear activation

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

Output is a weighted sum of middle nodes

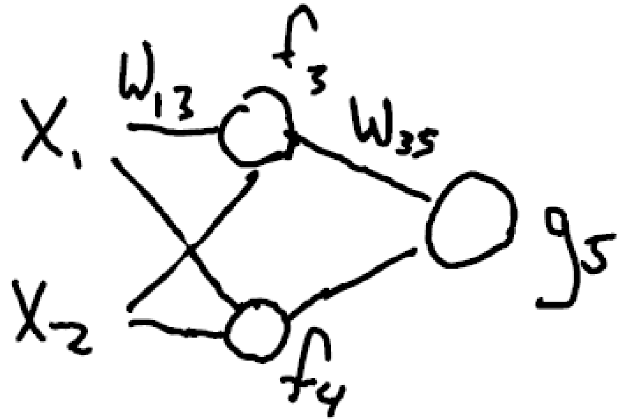
$$f_3(x) = w_{13} x_1 + w_{23} x_2$$

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



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

$$f_3(x) = w_{13}x_1 + w_{23}x_2$$

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

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}

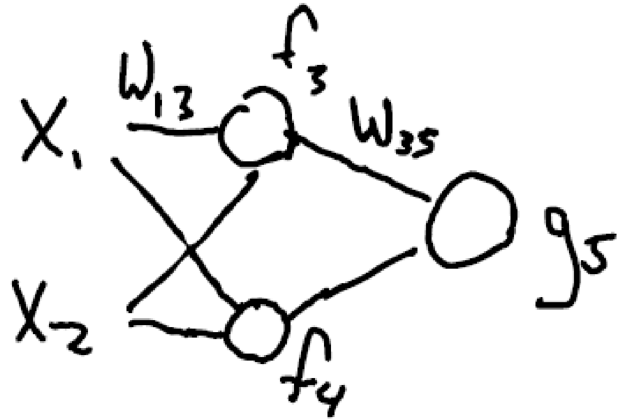
$$w_{35} = w_{35} - \eta \left[2 \cdot (g_5(x) - y) f_3(x) \right]$$

Take step in negative gradient direction

How much increasing the prediction value increases the error

How much increasing the weight increases the prediction value

Back-propagation: internal weights



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

$$f_3(x) = w_{13} x_1 + w_{23} x_2$$

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

Chain rule:

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

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

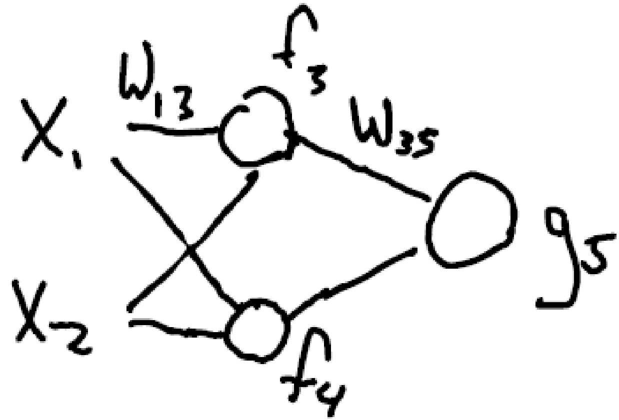
Chain rule is applied recursively, since w_{13} affects $f_3(x)$

How much increasing the prediction value increases the error

How much increasing this output increases prediction value

How much increasing the weight increases the output

What if f3 had ReLU activation?



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

$$f_3(x) = w_{13} x_1 + w_{23} x_2$$

(before, with linear activation)

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

Chain rule:
$h(x) = f(g(x))$
$h'(x) = f'(g(x)) g'(x)$

$$f_3(x) = \max(w_{13} x_1 + w_{23} x_2, 0)$$

Gradient is zero if $f_3(x)$ <=0; otherwise, same as for linear activation

$$\partial E / \partial w_{13} = 2(g_5(x) - y) w_{35} \delta(f_3(x) > 0) x_1$$

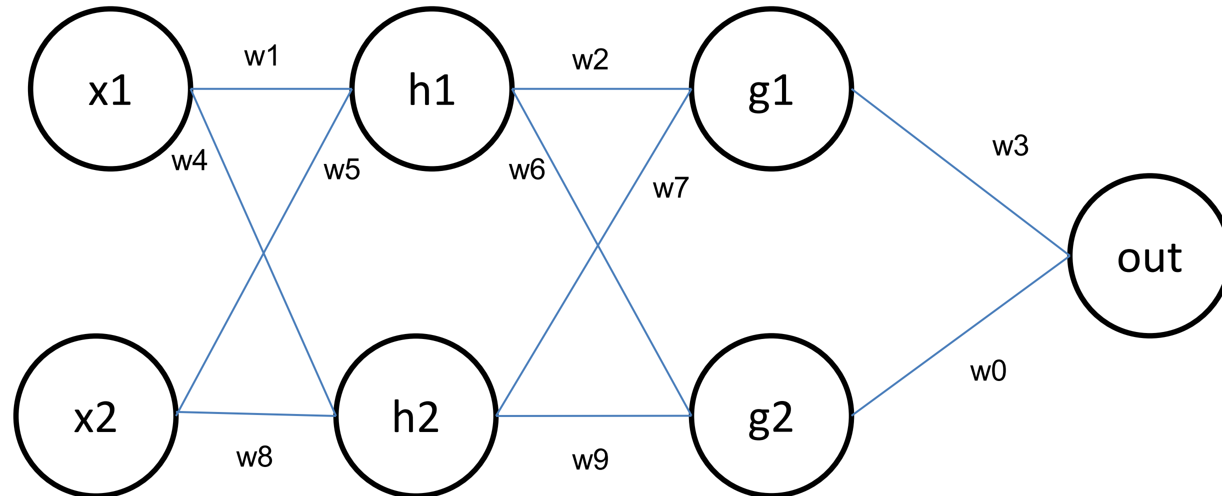
Backpropagation: General Concept

Each weight's gradient is a product of the gradients on the path from the weight's input to the prediction

The gradients can be computed using a kind of dynamic program, i.e., recursively propagating the gradient from the prediction error to the input

Another backpropagation example

Assume all linear layers (and linear activation, for simplicity), fill in the terms for the squared error gradients



$$h1 = w1 * x1 + w5 * x2$$

$$h2 = w4 * x1 + w8 * x2$$

$$g1 = w2 * h1 + w7 * h2$$

$$g2 = w6 * h1 + w9 * h2$$

$$out = w3 * g1 + w0 * g2$$

Error gradient wrt **w2** = 2*(out-y)*(_w3_) * _h1_

Grad loss wrt out

Grad out wrt g1

Grad g1 wrt w2

Error gradient wrt **w8** = 2*(out-y)*(_w0_ * _w9_ + _w3_ * _w7_) * _x2_

Q1-Q3

https://tinyurl.com/441-fa24-L15

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

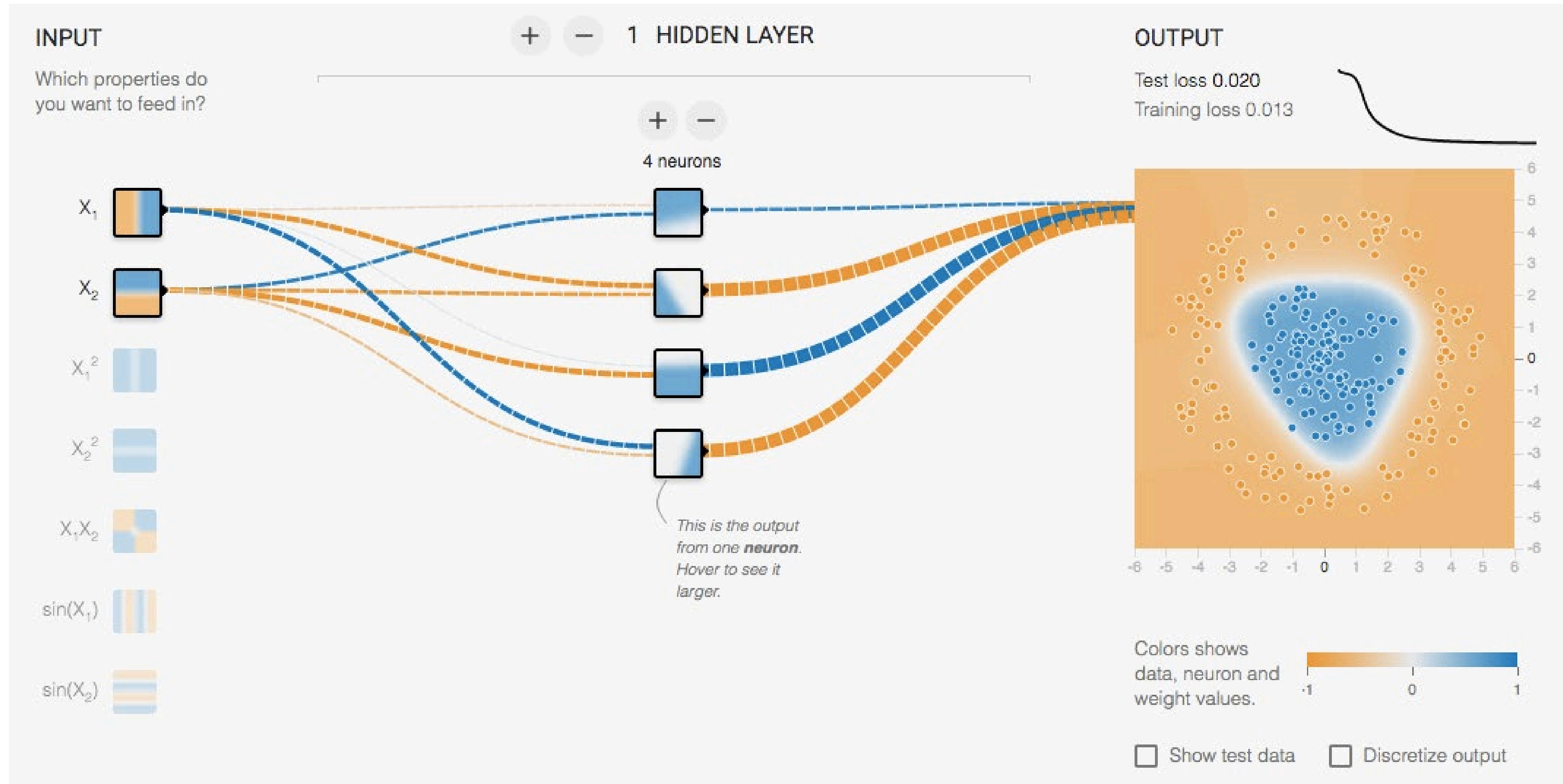
1. Optimization is no longer convex, globally optimum solution no longer guaranteed (or even likely)
2. Larger model = more training and inference time
3. Larger model = more data required to obtain a good fit

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

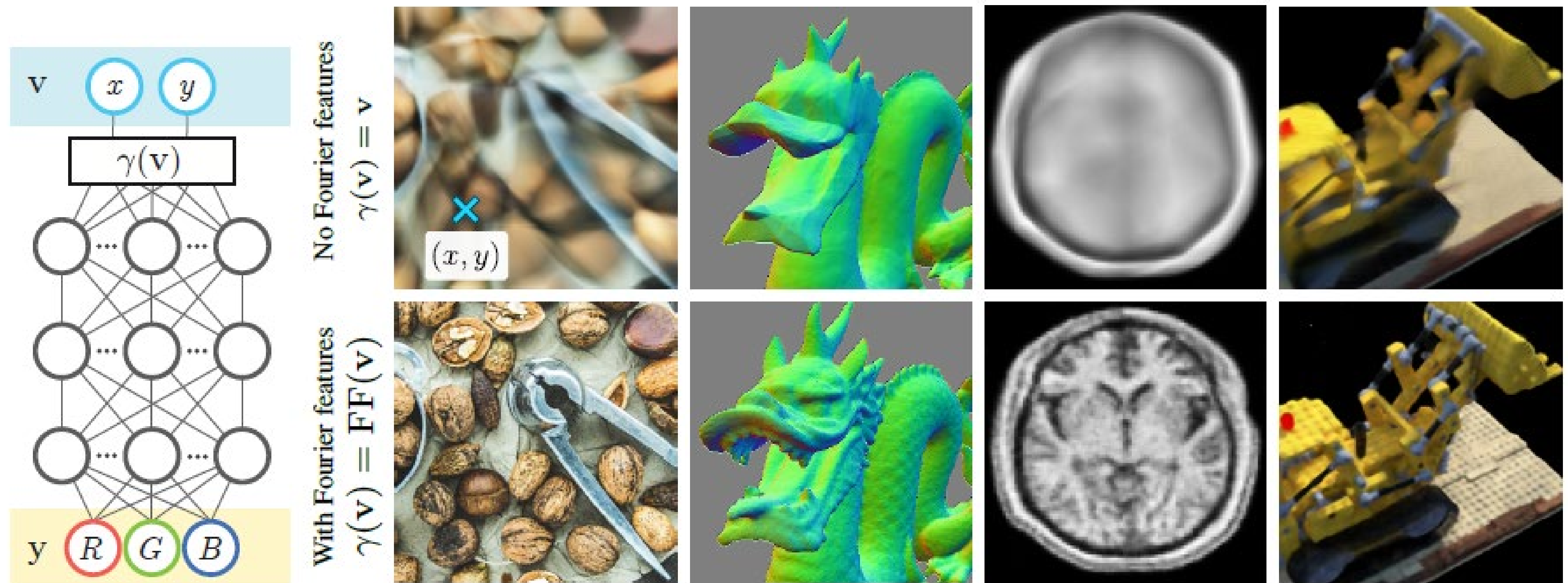


http://playground.tensorflow.org/

Try many layers with sigmoid vs relu

Slide: Lazebnik

Another application example: mapping position/rays to color



(a) Coordinate-based MLP

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

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

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

(e) Inverse rendering $(x,y,z) \to$ 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)

Generalized insight from “Fourier Features”

- Input matters – it’s best to represent data in a way that makes it linearly predictive, even if you have a non-linear model
- $f(x, y) \rightarrow R, G, B$ requires a complex network to model because $x_i^T x_j$ is a bad similarity function (maximized when x_j is large, instead of similar to x_i)
- Representing x with a Fourier encoding, e.g. $\gamma(x) = [\sin(x), \cos(x), \sin(2x), \cos(2x), \ldots]$ enables a simpler network because $\gamma(x_i)^T \gamma(x_j)$ falls off smoothly as x_j moves away from x_i
  - This means the initial network layer can model similarity to different positions with each hidden unit

HW 4 (due Nov 4)

1. Model Complexity with Tree Regressors [30 pts]

One measure of a tree's complexity is the maximum tree depth. Train tree, random forest, and boosted tree regressors on temperature regression (data), using all default parameters except:

- `max_depth={2,4,8,16,32}`
- `random_state=0`
- For random forest: `max_features=1/3`

For each method, train one model using the training set and measure RMSE on the training and validation sets. Plot the max_depth vs RMSE for all methods on the same plot using the provided `plot_depth_error` function. You should have six lines (train/val for each model type), each with 5 data points (one for each max depth value). *Include the plot and answer the analysis questions in the report.*

b. Find a second rule [10 pts]

Find a second rule to classify Gentoo that also requires only two checks but is different from the other one you reported.

HW 4

2. **MLPs with MNIST** [40 pts]

For this part, you will want to use a GPU to improve runtime. Google Colab provides limited free GPU acceleration to all users. It can be run with CPU, but will be a few times slower. See Tips for detailed guidance on this problem. Note that this problem may require tens of minutes of computation.

First, use **PyTorch** to implement a Multilayer Perceptron network with one hidden layer (size 64) with ReLU activation. Set the network to minimize cross-entropy loss, which is the negative log probability of the training labels given the training features. This objective function takes unnormalized logits as inputs. *Do not use MLP in sklearn for this HW - use Torch.*

a. Using the train/val split provided in the starter code, train your network for 100 epochs with learning rates of 0.01, 0.1, and 1. Use a batch size of 256 and the SGD optimizer. After each epoch, record the mean training and validation loss and compute the validation error of the final model. The mean validation loss should be computed after the epoch is complete. The mean training loss can either be computed after the epoch is complete, or, for efficiency, computed using the losses accumulated during the training of the epoch. Plot the training and validation losses using the `display_error_curves` function.

b. Based on the loss curves, select the learning rate and number of epochs that minimizes the validation loss. Retrain that model (if it's not stored), and *report training loss, validation loss, training error, validation error, and test error*. You should be able to get test error lower than 2.5%.

a. **Improve MNIST Classification Performance using MLPs** [up to 30 pts]

Finally, see if you can improve the model by adjusting the learning rate, the hidden layer size, adding a hidden layer, or trying a different optimizer such as Adam (recommended). Report the train/val/test loss and the train/val/test classification error for the best model. Report your hyperparameters (network layers/size, optimizer type, learning rate, data augmentation, etc.). You can also use an ensemble of networks to achieve lower error for this part. Describe your method and report your val/test error. You must select a model using the validation set and then test your selected model with the test set. Points are awarded as follows: +10 for test error < 2.2%, +10 for test error < 2.0%, +10 for test error < 1.8%.

c. **Positional encoding** [30 pts]

Advanced

Because linear functions are easier to represent in MLPs, it can help to represent features in a way that makes them more useful linearly. An example is the use of positional encoding to represent a pixel position, as described in https://arxiv.org/pdf/2006.10739.pdf.

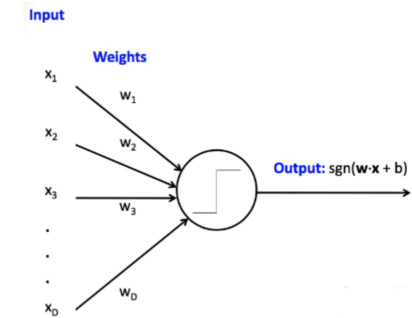
For this problem, use positional encoding to predict RGB values given pixel coordinate of this image. You can resize the image to a smaller size for speed (e.g. 64 pixels on a side). In this problem, the network is acting as a kind of encoder — you train it on the same pixels that you will use for prediction.

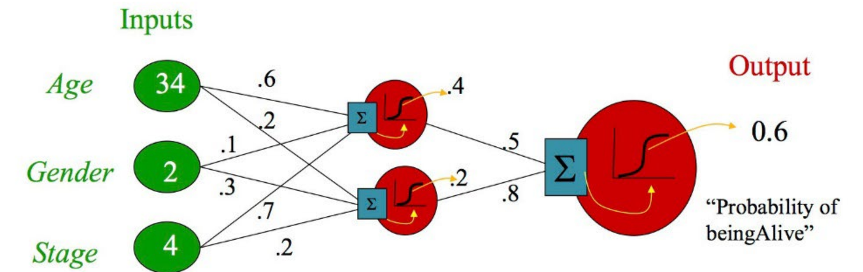
1. Create an MLP that predicts the RGB values of a pixel from its position (x,y). Display the RGB image generated by the network when it receives each pixel position as an input.
2. Write code to extract a sinusoidal positional encoding of (x, y). See this page for details.
3. Create an MLP that predicts a pixel's RGB values from its positional encoding of (x, y). Display the RGB image generated by the network when it receives each pixel position as an input.

The paper uses these MLP design parameters: L2 loss, ReLU MLP with 4 layers and 256 channels (nodes per layer), sigmoid activation on output, and 256 frequencies.

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
- Deeper networks are subject to vanishing gradient problems that can be partially avoided with ReLU





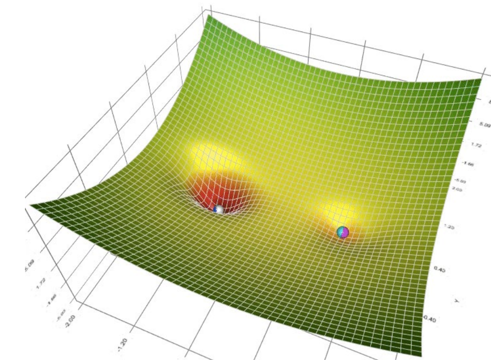


Figure source

Next lectures

- Convolutional networks (CNNs)
- Deep networks
  - (Brief) history of deep networks
  - What made deep networks work?
  - Residual networks
- More about deep network optimization
  - Improvements on SGD
  - Normalization and data augmentation
  - Linear probe and fine-tuning
- Mask-RCNN line of work