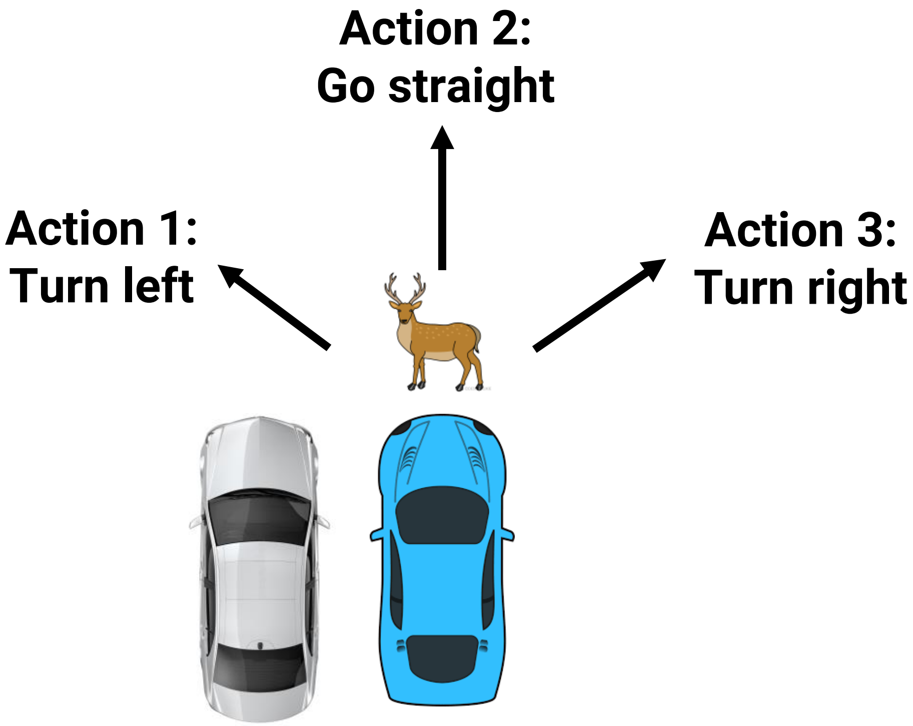
# Introduction to Neural Networks

**Robin Jia**
USC CSCI 467, Spring 2023
February 16, 2023

# Today’s Plan

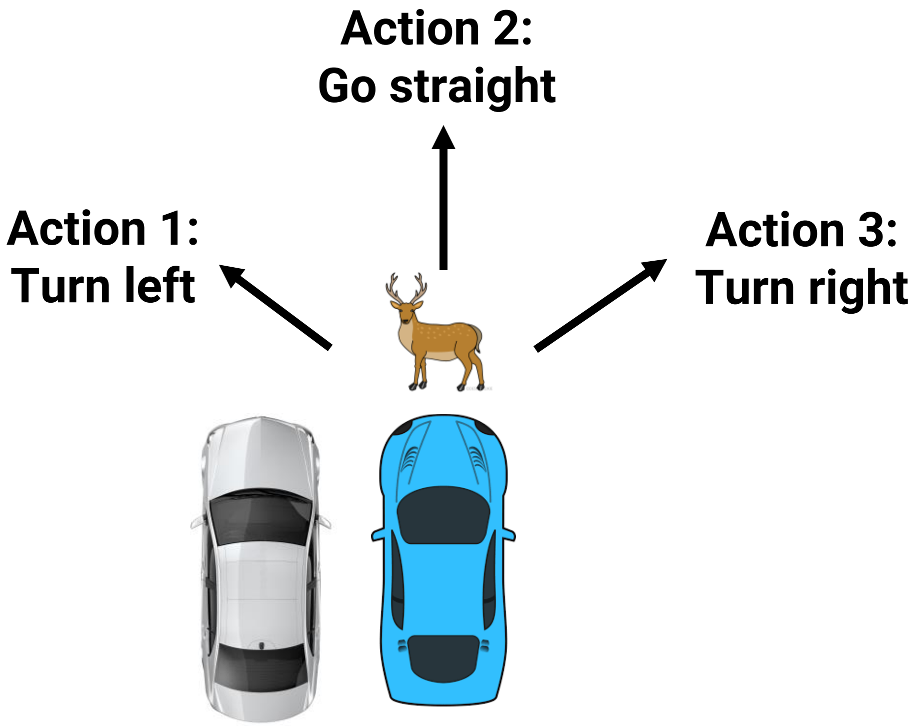
- Neural networks: What and why?
- Training
  - Stochastic gradient descent
  - Random initialization
  - (Next class: How to compute gradients?)
- Regularization
  - Early stopping
  - Dropout

# A (toy) self-driving car example



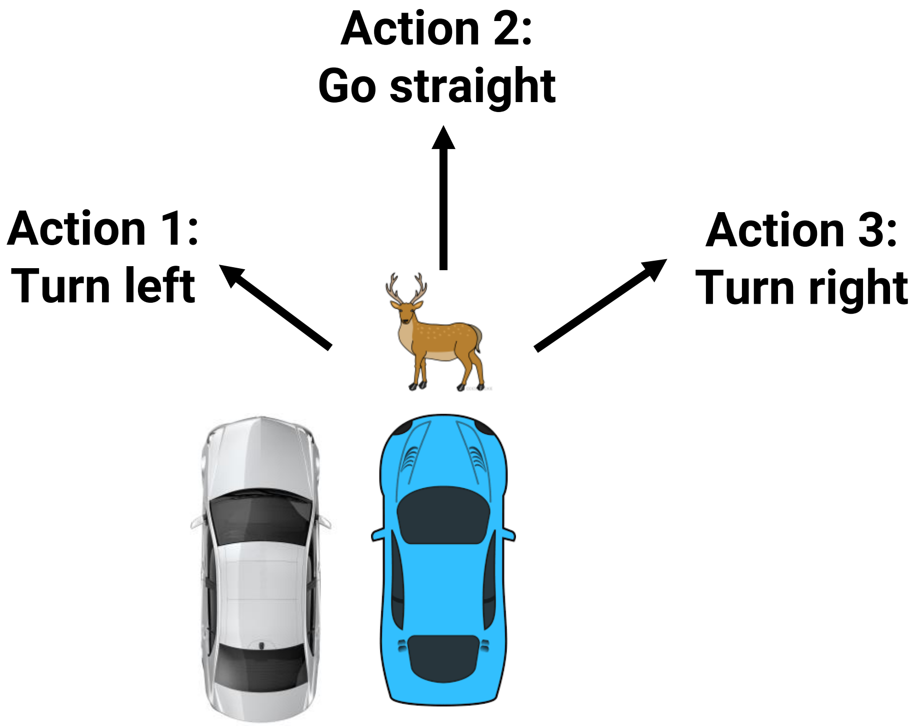
- Three-way classification problem: Go left, straight, or right?
- What features are important here?
  - Is front clear?
  - Is left clear?
  - Is right clear?

# A (toy) self-driving car example



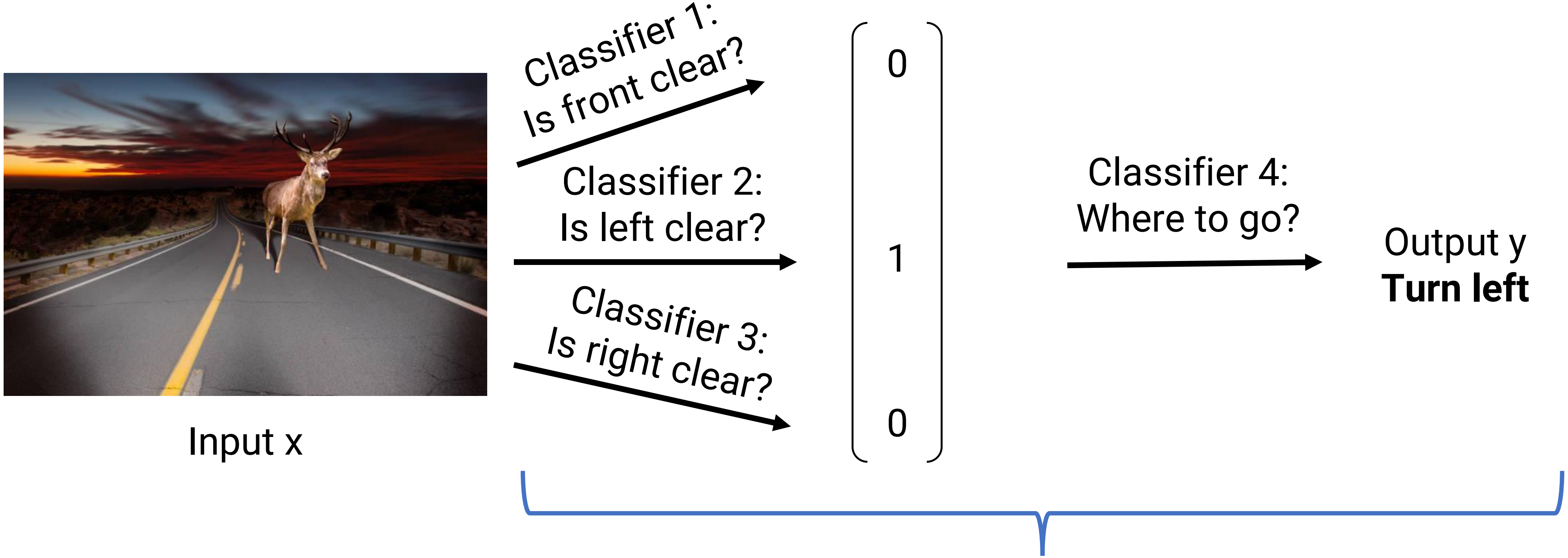
- Suppose we had these features:
  - $z = [z_1, z_2, z_3]$
  - $z_1$ = 1 if front is clear, 0 else
  - $z_2$ = 1 if left is clear, 0 else
  - $z_3$ = 1 if right is clear, 0 else
- With this, we can do softmax regression:
  - Score for “straight”: $20\ z_1 – 10$
  - Score for “left”: $10\ z_2 – 10$
  - Score for “right”: $10z_3 – 10$
- Behavior
  - If everything is clear, go straight
  - If front is blocked, go left or right if those are clear
  - If everything is blocked, all equally bad

# A (toy) self-driving car example



- How can we write the feature "is front clear"?
- Checking if the front is clear **is itself a machine learning problem**
  - Input = camera image/lidar data,
    Output = whether there is an obstacle
  - Obstacle near or far away?
  - Hard obstacle or a plastic bag?
- Can we make our features the outputs of another "classifier"?

# Feature learning



Classifier 1:
Is front clear?

Classifier 2:
Is left clear?

Classifier 3:
Is right clear?

$$\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

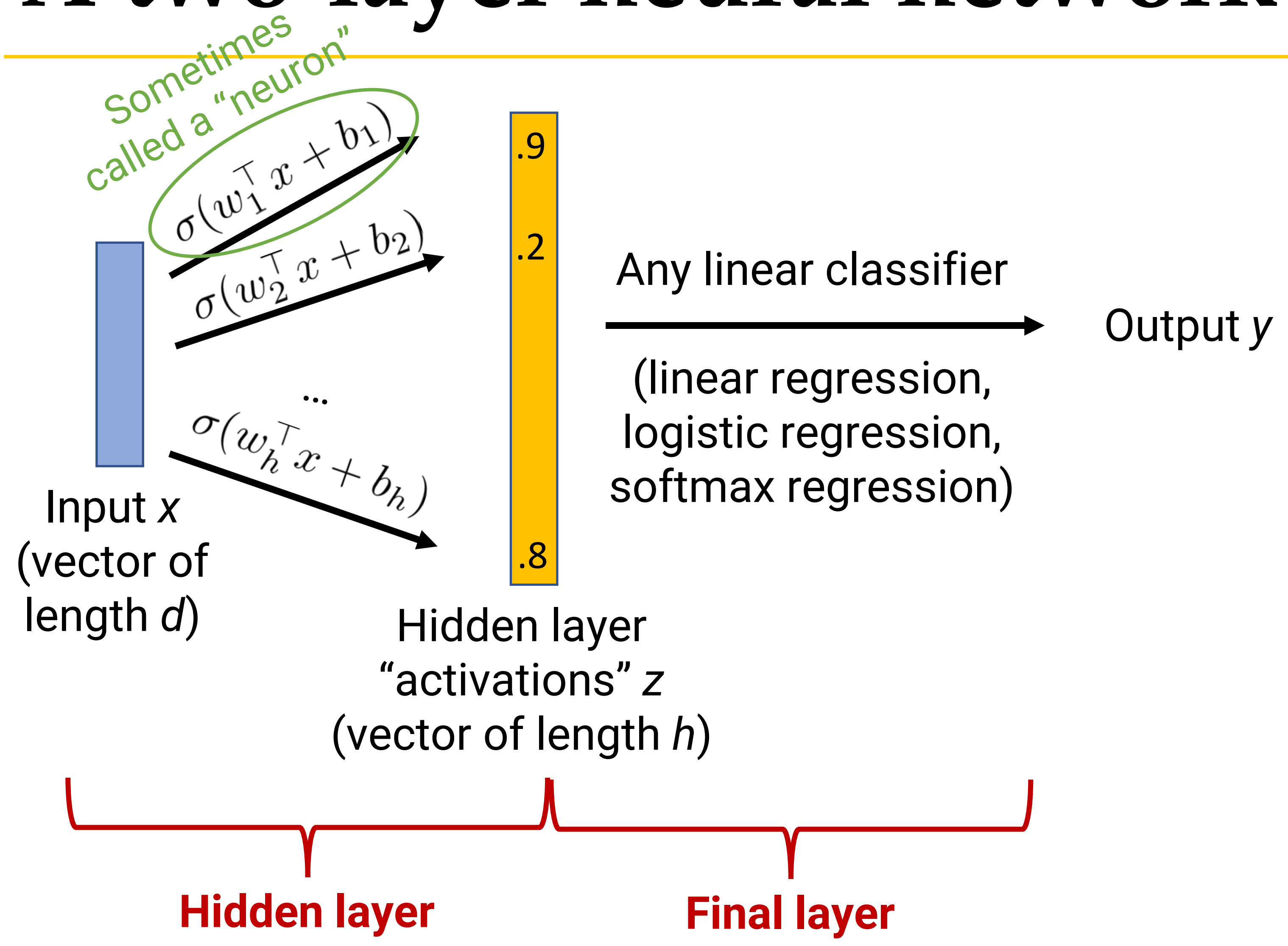
Classifier 4:
Where to go?

Output y
**Turn left**

Input x

**This is a neural network!**

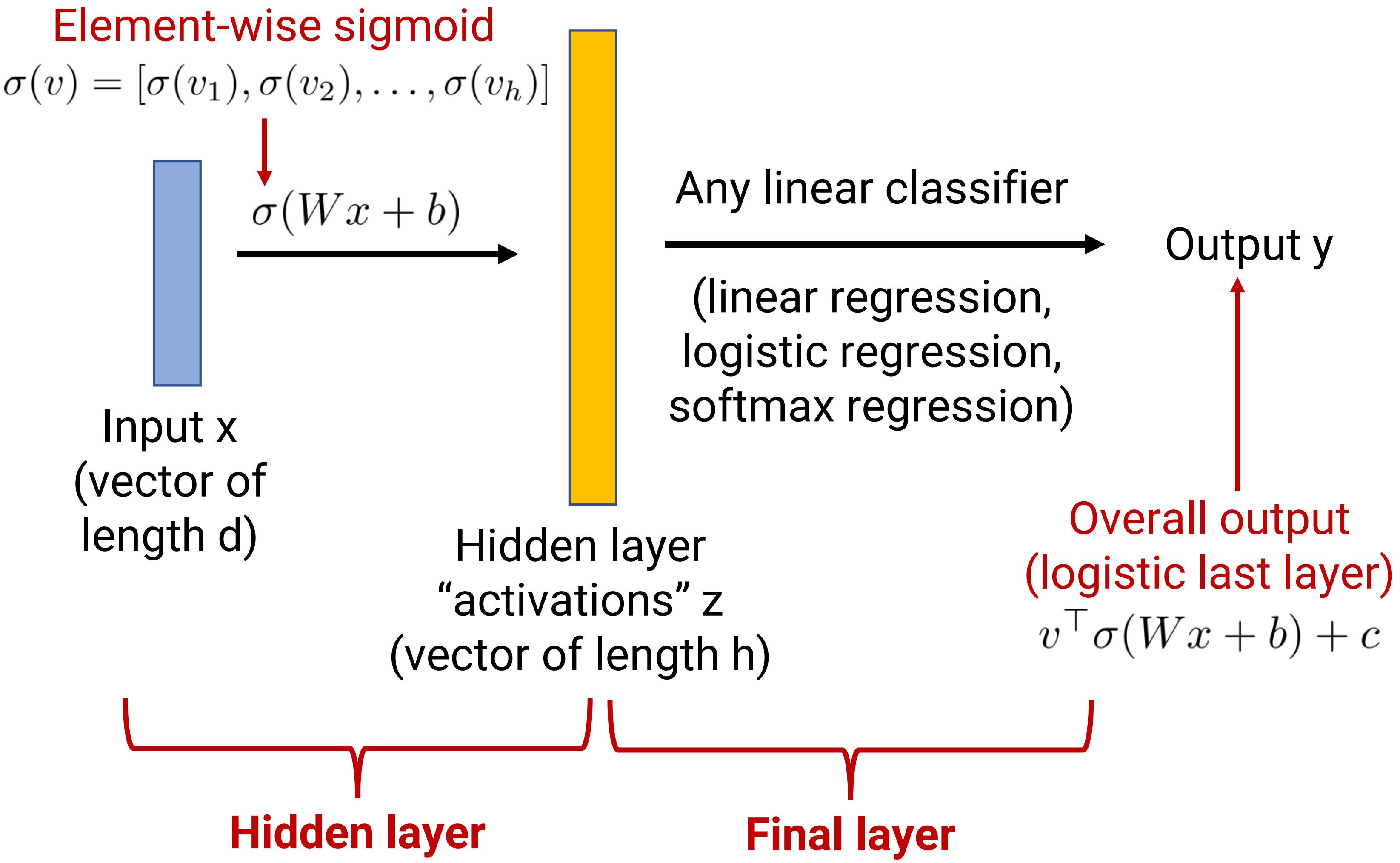
# A two-layer neural network



Sometimes called a "neuron"
$\sigma(w_1^\top x + b_1)$
$\sigma(w_2^\top x + b_2)$
…
$\sigma(w_h^\top x + b_h)$
.9
.2
.8
Input *x* (vector of length *d*)
Hidden layer "activations" *z* (vector of length *h*)
Any linear classifier
(linear regression, logistic regression, softmax regression)
Output *y*
Hidden layer
Final layer

- Hidden layer: A bunch of logistic regression classifiers
  - Parameters: $w_j$ and $b_j$ for each classifier
  - Produces "activations" = learned feature vector
- Final layer: A linear classifier
  - E.g. if logistic regression, has parameter vector *v* and bias *c*

# A two-layer neural network (matrix form)



Element-wise sigmoid

$\sigma(v) = [\sigma(v_1), \sigma(v_2), \ldots, \sigma(v_h)]$

$\sigma(Wx + b)$

Input x (vector of length d)

Hidden layer "activations" z (vector of length h)

Any linear classifier

(linear regression, logistic regression, softmax regression)

Output y

Overall output (logistic last layer)

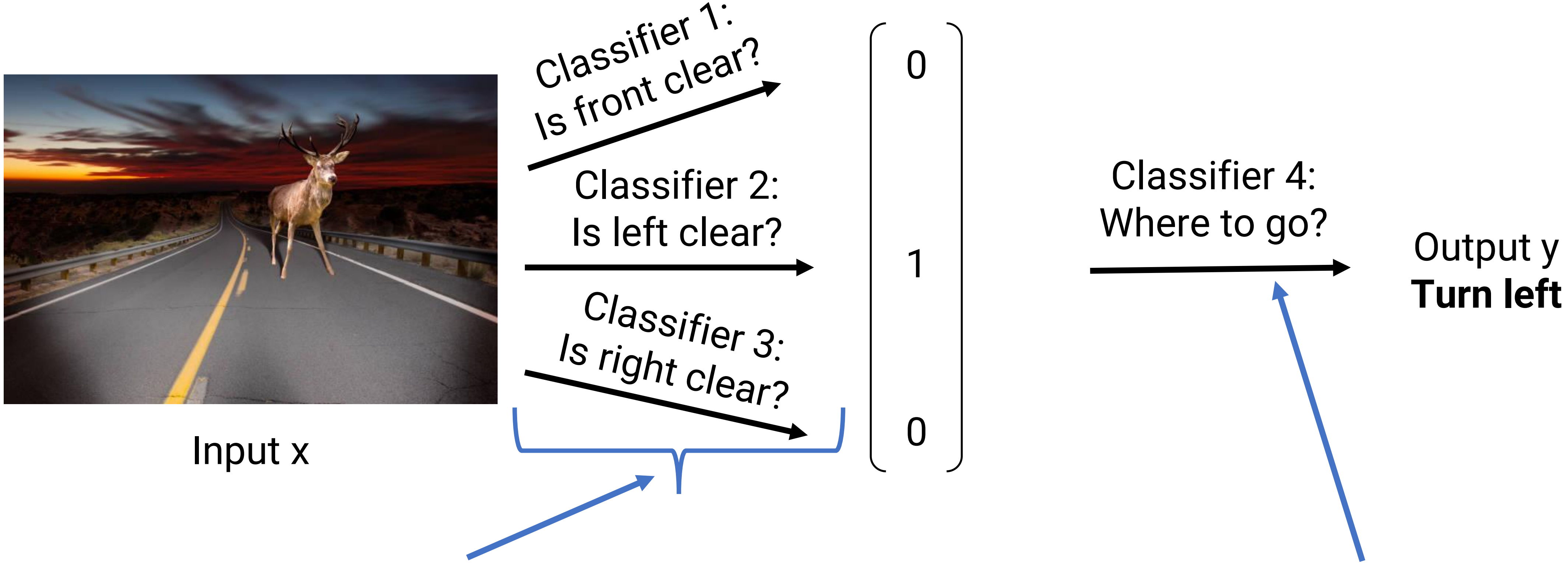
$v^\top \sigma(Wx + b) + c$

**Hidden layer**

**Final layer**

- Hidden layer: A bunch of logistic regression classifiers
  - Parameters: $w_j$ and $b_j$ for each classifier
  - **Equivalently: matrix W (h x d) and vector b (length h)**
  - Produces "activations" = learned feature vector
- Final layer: A linear classifier
  - E.g. if logistic regression, has parameter vector *v* and bias *c*
- **Parameters of model are *θ = (W, b, v, c)***

# Neural networks as feature learners



Classifier 1:
Is front clear?

Classifier 2:
Is left clear?

Classifier 3:
Is right clear?

$$\begin{pmatrix}0\\1\\0\end{pmatrix}$$

Input x

Classifier 4:
Where to go?

Output y
**Turn left**

**Learn a classifier whose output is a good feature**
We don’t tell the model what classifier to learn
Model must learn that “is front clear” is a useful concept

Learn to classify based on features
(same as linear model)

# Do we need “non-linearities”?

With sigmoid, overall output (with logistic last layer) is:

$$v^\top \sigma(Wx + b) + c$$

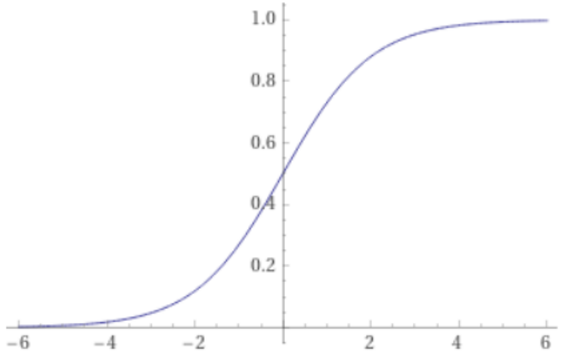
**Without** sigmoid, overall output (with logistic last layer)is:

$$v^\top (Wx + b) + c$$
$$= (v \top W)x + (v^\top b + c)$$

- Suppose we deleted the sigmoids…
- Result: Just another way to write a linear function!
  - New “weight” is $v^T W$
  - New “bias” is $v^T b + c$
- **To learn a non-linear function, need a “nonlinearity” between the two layers**

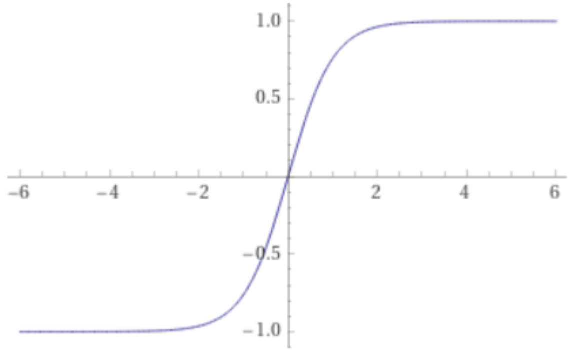
# Options for non-linearities

Sigmoid



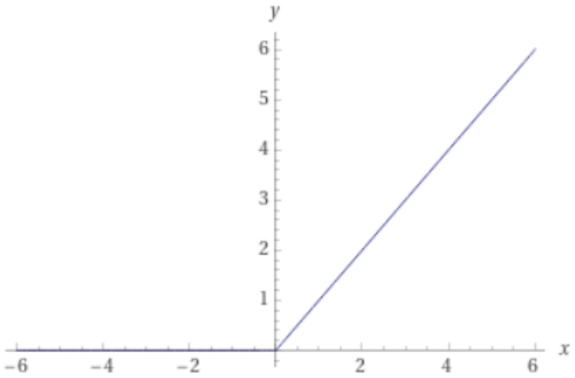
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Tanh



$$\tanh(z) = \frac{e^{2z} - 1}{e^{2z} + 1}$$

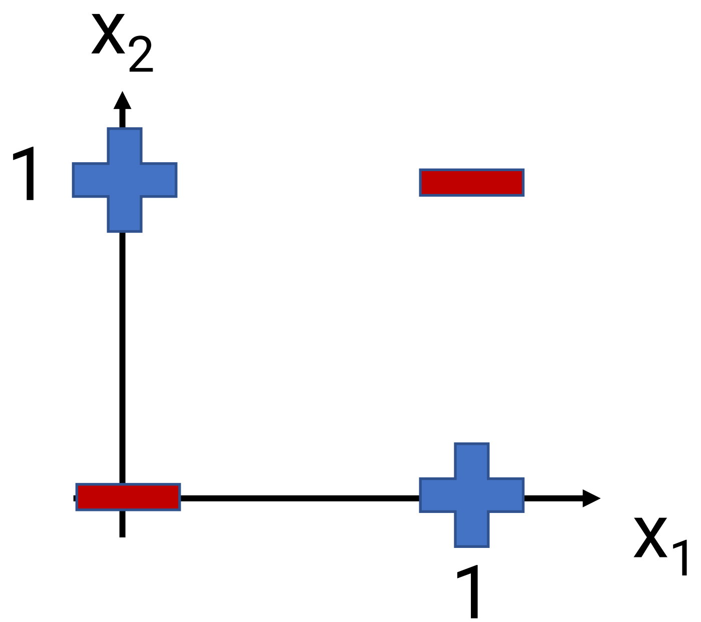
ReLU



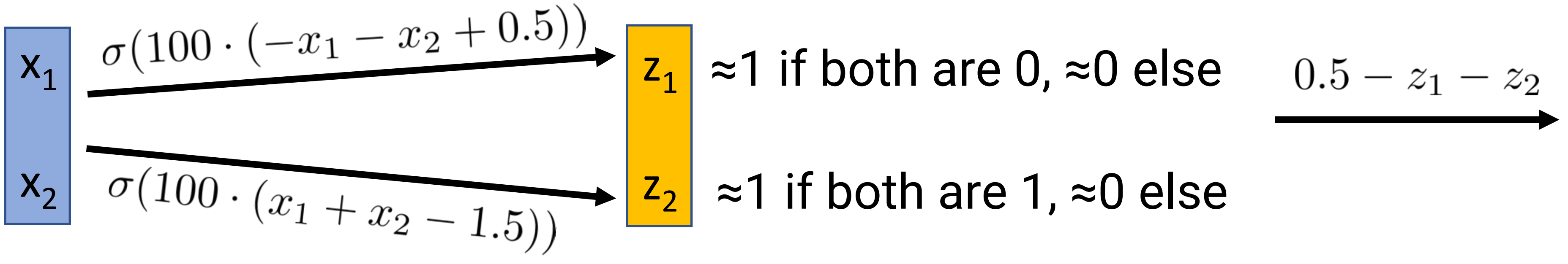
$$\mathrm{ReLU}(z) = \max(z, 0)$$

- Many options work, just must be differentiable
- In practice: tanh and ReLU often preferred
  - Tanh: Better than sigmoid because outputs centered around zero
  - ReLU: Very fast to compute

# Solving XOR



- What functions can we represent with neural networks?
- XOR: Classic binary classification problem that can’t be solved by linear classifier
- A 2-layer neural network can solve it!



$x_1$

$x_2$

$\sigma(100 \cdot (-x_1 - x_2 + 0.5))$

$\sigma(100 \cdot (x_1 + x_2 - 1.5))$

$z_1$ ≈1 if both are 0, ≈0 else

$z_2$ ≈1 if both are 1, ≈0 else

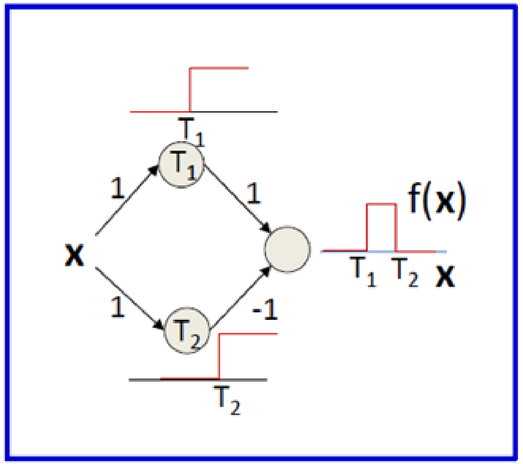
$\xrightarrow{0.5 - z_1 - z_2}$

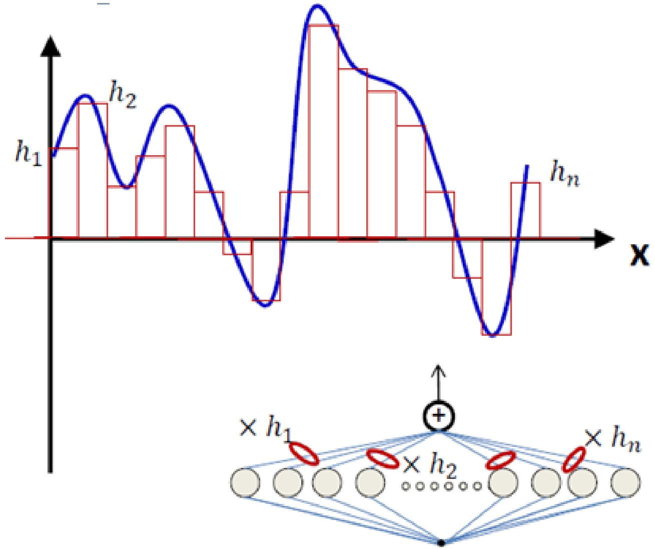
< 0 if XOR($x_1$, $x_2$) = 0

> 0 if XOR($x_1$, $x_2$) = 1

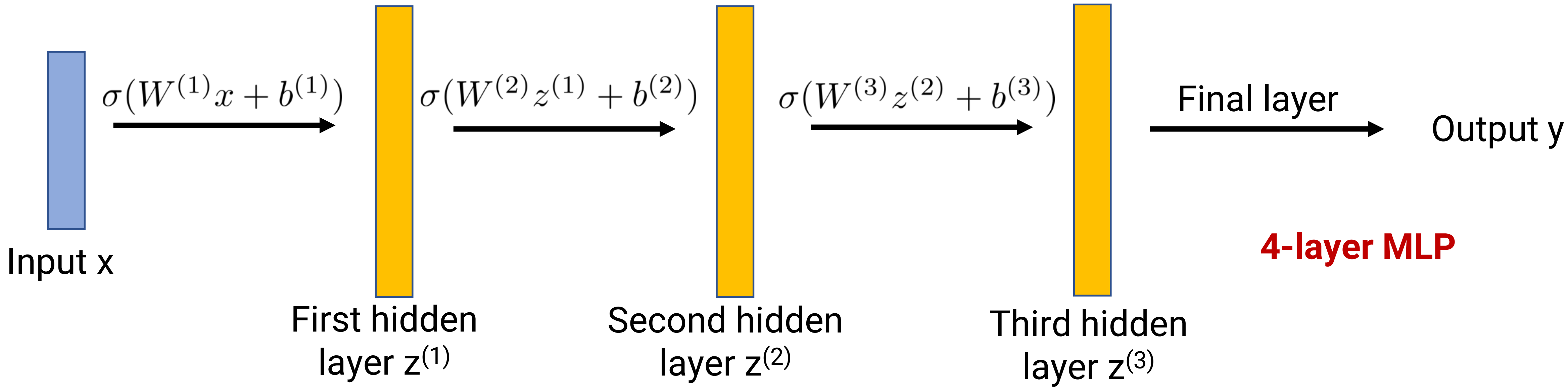
# Universal Approximation

- Fact: **Any function** can be approximated by a 2-layer neural network with enough hidden units
- 2-layer neural networks are thus “**universal approximators**”
  - Note: Also true for k-NN, SVM with RBF kernel…
- Proof sketch
  - First layer learns a bunch of step functions, which divide the domain into “buckets”
  - Second layer assigns correct value to each bucket
  - With enough hidden units, width of buckets can become arbitrarily small





# Multi-Layer Perceptrons


$\sigma(W^{(1)}x + b^{(1)})$
$\sigma(W^{(2)}z^{(1)} + b^{(2)})$
$\sigma(W^{(3)}z^{(2)} + b^{(3)})$
Final layer
Output y
Input x
First hidden layer $z^{(1)}$
Second hidden layer $z^{(2)}$
Third hidden layer $z^{(3)}$

**4-layer MLP**

- What we saw so far is called a “2-layer perceptron”
- But we can add more layers!
  - Corresponds to more complex feature extractor
  - In practice, making networks “deeper” (more layers) often helps more than making them “wider” (more hidden units in each layer)
  - Layers are “fully connected” as each neuron depends on every neuron in previous layer

# Announcements

- HW1 grades out, solutions discussed in tomorrow’s section
- Project proposals due today at 11:59pm
- HW2
  - Problems 1-3 released soon
  - Problem 4 (neural network coding problem) released a bit later
  - Due Thursday, March 2
- Reminders about plagiarism
  - Looking at another student’s solutions or sharing your solutions is **strictly prohibited**
    - This includes posting your code publicly on GitHub
  - While you may talk to classmates at a high level, your entire write-up and code must be written by yourself

# Today's Plan

- Neural networks: What and why?
- Training
  - Stochastic gradient descent
  - Random initialization
  - (Next class: How to compute gradients?)
- Regularization
  - Early stopping
  - Dropout

# Training objectives

## Logistic Regression

- Model's output is

$$g(x) = w^\top x + b$$

- (Unregularized) loss function is

$$\frac{1}{n}\sum_{i=1}^{n} -\log \sigma\left(y^{(i)} \cdot g(x^{(i)})\right)$$

## Binary Classification w/ Neural Networks

- Model's output is

$$g(x) = v^\top \sigma(Wx + b) + c$$

- Loss function is same, in terms of g!

$$\frac{1}{n}\sum_{i=1}^{n} -\log \sigma\left(y^{(i)} \cdot g(x^{(i)})\right)$$

**More generally, write as**

$$\frac{1}{n}\sum_{i=1}^{n} \ell\left(y^{(i)}, g(x^{(i)})\right), \text{ where } \ell(y, u) = -\log \sigma(y \cdot u)$$

Also applies for linear regression, softmax regression, etc.

# Stochastic gradient descent

General loss function: $\frac{1}{n}\sum_{i=1}^{n}\ell\left(y^{(i)}, g(x^{(i)})\right)$

Model’s output, depends on parameters θ

## Gradient Descent

$$\theta \leftarrow \theta - \eta \cdot \underbrace{\frac{1}{n}\sum_{i=1}^{n}\nabla_\theta \ell\left(y^{(i)}, g(x^{(i)})\right)}_{\text{Average of per-example gradients}}$$

- Disadvantage: 1 update is O(n) time
  - What if dataset is very large?
- Idea: Approximate with sample mean

## Stochastic Gradient Descent

1. Sample a *batch* B of examples from the training dataset
2. Do the update

$$\theta \leftarrow \theta - \eta \cdot \underbrace{\frac{1}{|B|}\sum_{(x,y)\in B}\nabla_\theta \ell\left(y, g(x)\right)}_{\text{Sample mean within batch}}$$

# Stochastic gradient descent

In practice, a slightly different version is used to ensure equal usage of all training examples:

```
for t = 1, …, T:
    Randomly partition training examples into batches $B_1$, …, $B_k$
    for i = 1, …, k:
        $\theta \leftarrow \theta - \eta \cdot \frac{1}{|B_i|} \sum_{(x,y) \in B_i} \nabla_\theta \ell\left(y, g(x)\right)$
```

Each t (i.e., each pass over the dataset) is called one “epoch”

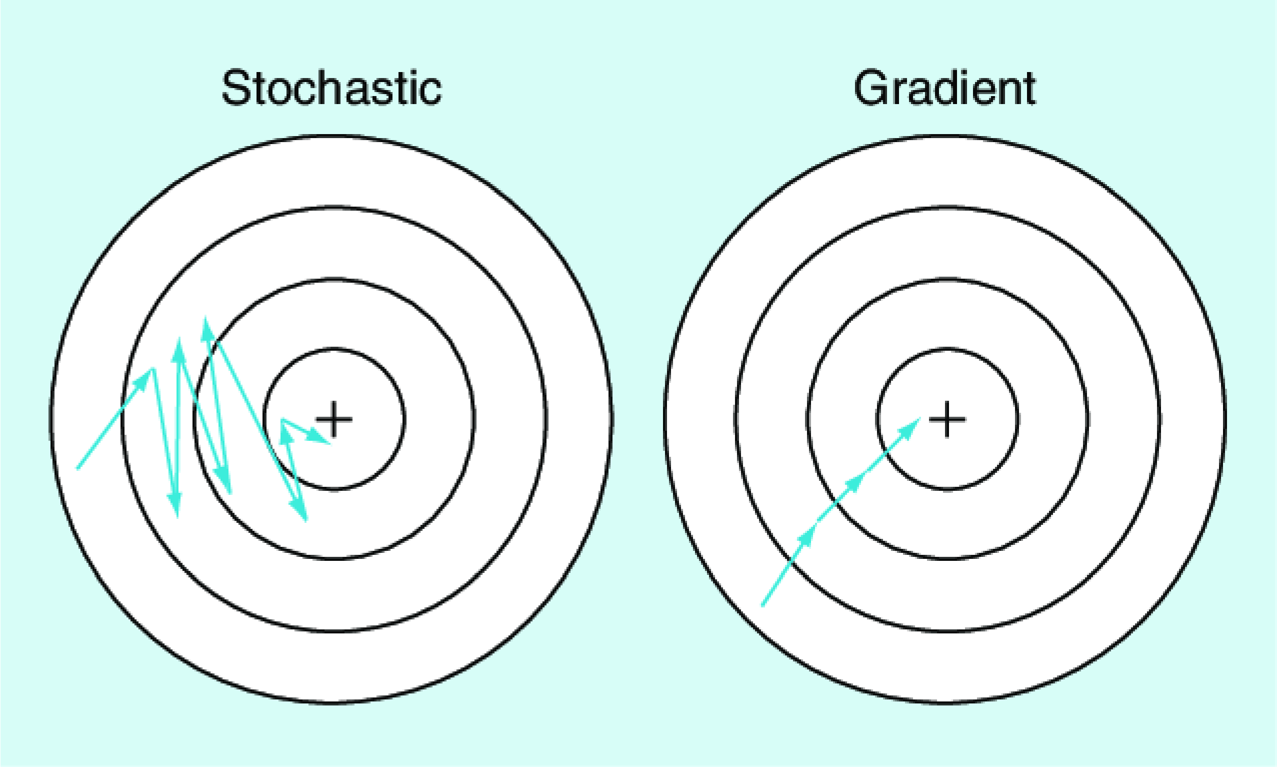
Update based on sample mean within current batch

How many batches? Desired “batch size” (# examples/batch) is another hyperparameter to tune

- Larger batch size = more accurate gradient, but slower
- Smaller batch size = faster, but may wander in suboptimal directions

- Again, can be used with any model, but especially common with neural networks

# Stochastic gradient descent



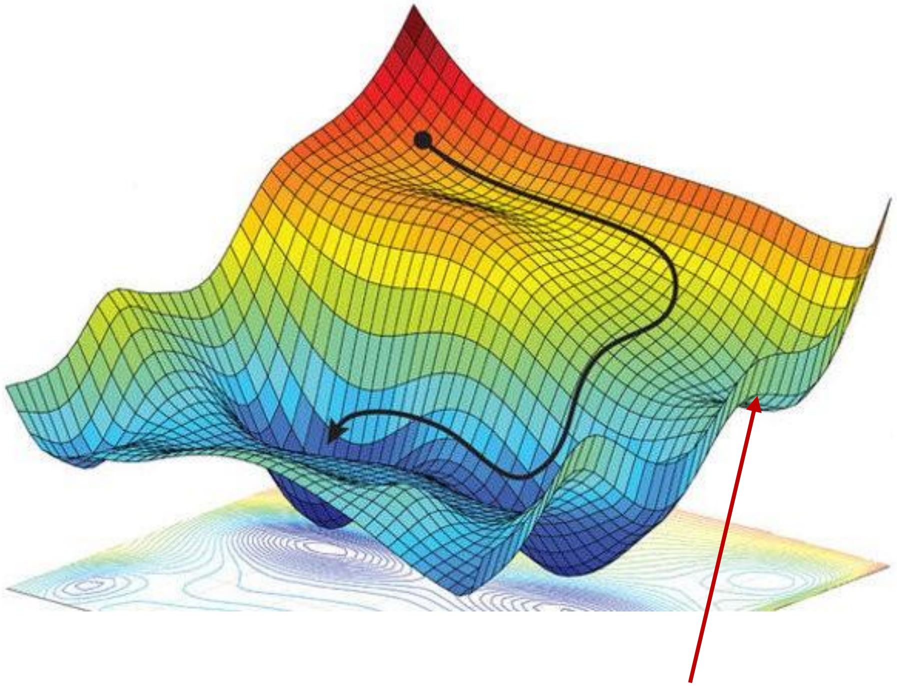
- SGD: Each parameter update is only “approximately” going towards the minimum
- But given enough time, you’ll end up in (almost) the same place
  - Plus each step is much faster!

# OK, so how do we compute the gradient?

$$\theta \leftarrow \theta - \eta \cdot \frac{1}{|B|} \sum_{(x,y)\in B} \underbrace{\nabla_\theta \ell\left(y, g(x)\right)}_{\text{How to compute this gradient?}}$$

- Neural networks can get very big & complicated…
- Taking gradients by hand is tedious…
- **Can we write a program to take gradients for us?**
- Yes! **Backpropagation** (focus of next class)

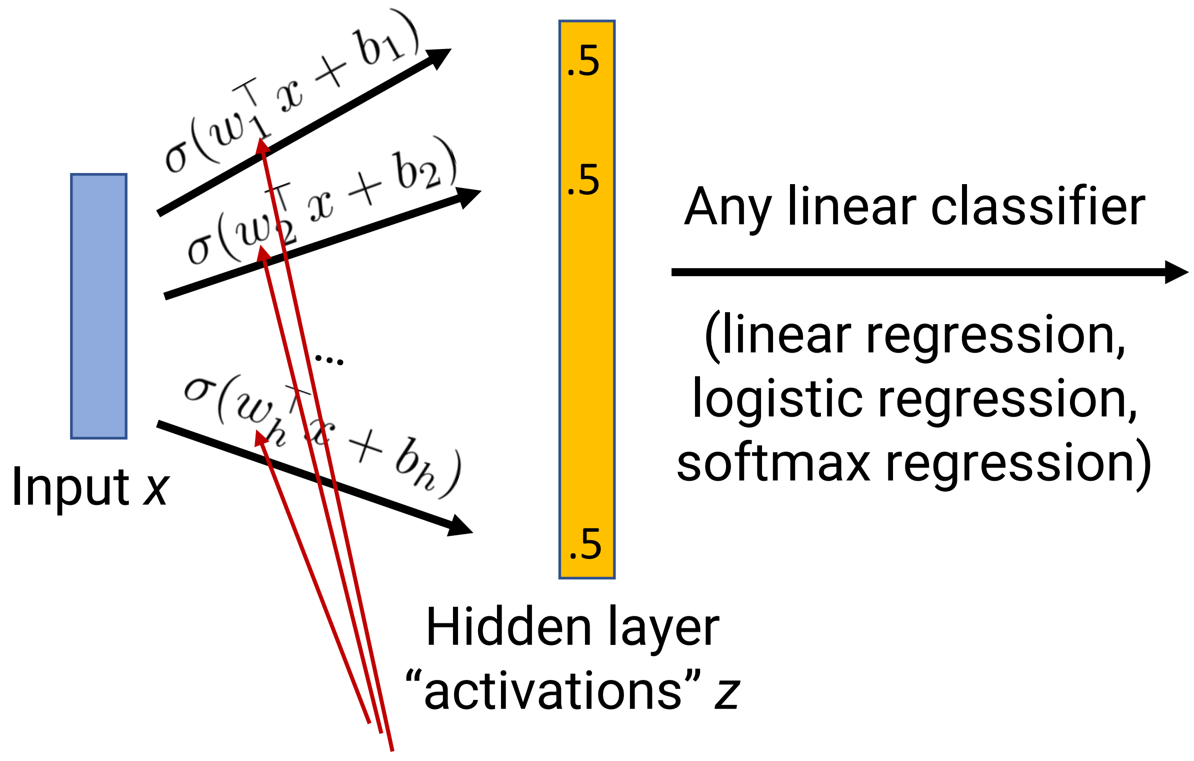
# Initialization



**Local minimum**
Gradient descent can get stuck here!

- For convex problems (e.g. logistic regression), initialization doesn’t matter much for final result
  - We just initialize parameters to all 0’s
- For neural networks, initialization matters a lot!
  - Optimization problem is non-convex
  - Where you start determines what parameters you learn

# The problem with all-0’s initialization

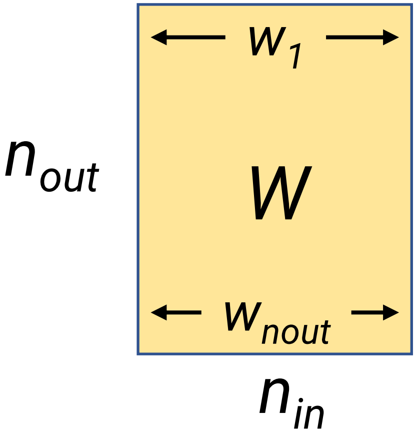

$\sigma(w_1^\top x + b_1)$
$\sigma(w_2^\top x + b_2)$
…
$\sigma(w_h^\top x + b_h)$
.5
.5
.5
Input *x*
Hidden layer
“activations” *z*
Any linear classifier
(linear regression, logistic regression, softmax regression)
Output *y*

If every $w_j$ starts as 0 vector, gradient update to each $w_j$ will be the same

- What if we initialize with all 0’s?
- Problem: Symmetry
  - All hidden units start out the same, so gradients will be the same for each
  - Thus, all hidden units will stay the same!
- **We must initialize in a way that breaks the symmetry**

# How to initialize neural networks

- TL;DR: Initialize every entry in *W* to a small random number
- How small? Many options…
  - Depends on “fan-in” $n_{in}$ (# input features) and “fan-out” $n_{out}$ (# output features)



  - Xavier initialization: $\text{Normal}\left(0, \dfrac{2}{n_{in} + n_{out}}\right)$
  - He initialization: $\text{Normal}\left(0, \dfrac{2}{n_{in}}\right)$
  - Pytorch: $\text{Uniform}\left(-\dfrac{1}{\sqrt{n_{in}}}, \dfrac{1}{\sqrt{n_{in}}}\right)$

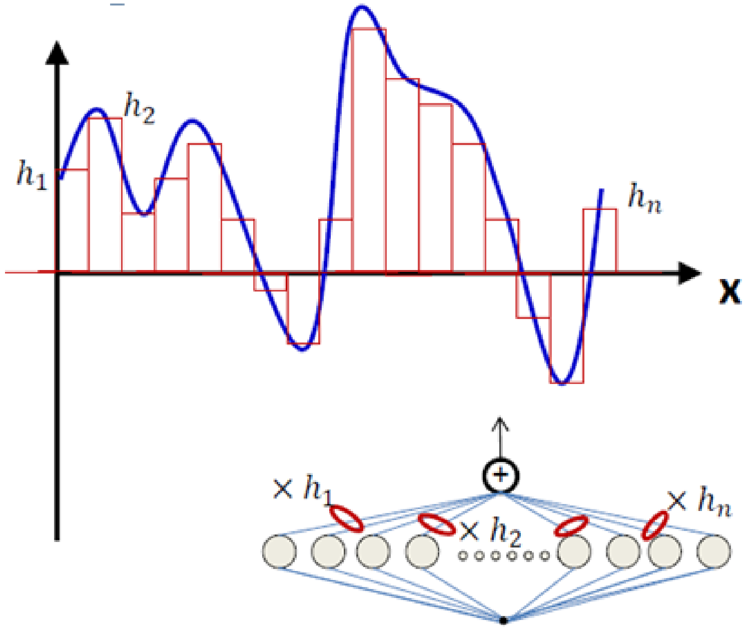
Intuition: If many dimensions, each individual weight can be smaller because dot product will sum many small numbers together

Uniform avoids large outliers

# Today's Plan

- Neural networks: What and why?
- Training
  - Stochastic gradient descent
  - Random initialization
  - (Next class: How to compute gradients?)
- Regularization
  - Early stopping
  - Dropout

# Regularization & Neural Networks



- Recall: Neural networks are universal approximators
- This means they are prone to overfitting!
- How to avoid overfitting too badly?

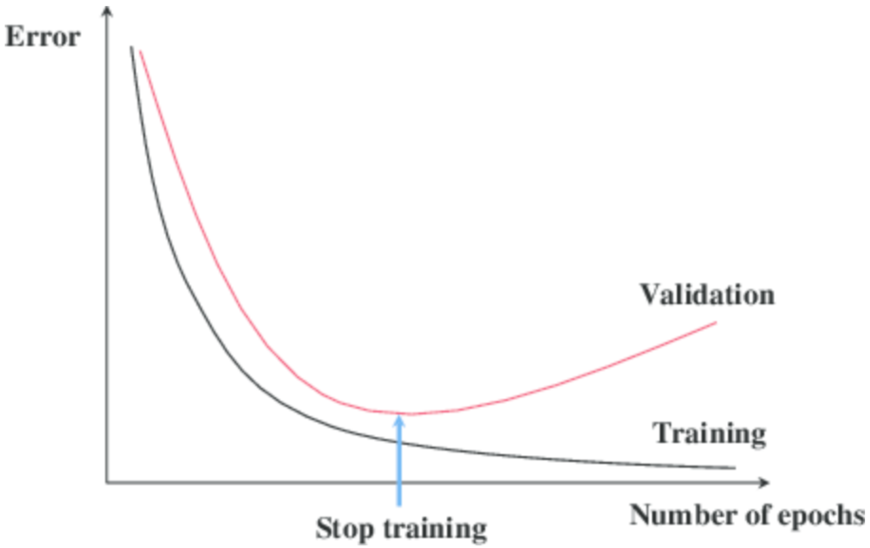
# Weight decay (AKA L2 Regularization)

- L2 regularization is a valid strategy!
- Often called “weight decay” when used with neural networks
  - Because every gradient step, you add the update

$$\theta \leftarrow \theta - \eta \cdot \lambda \cdot \theta$$

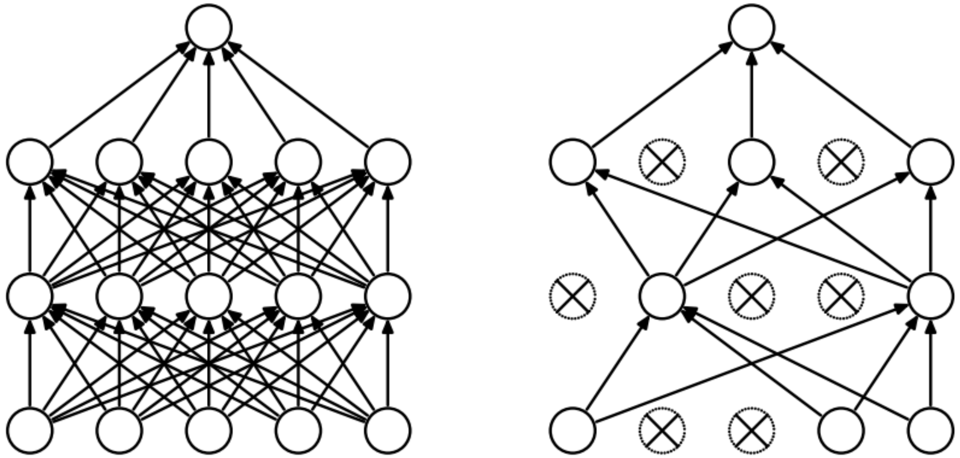
Weights literally “decay” by factor of (1 – ηλ)

# Early stopping



- Idea: Prevent overfitting by stopping training before you overfit too much
- How it works
  - Every so often during training, save “checkpoint” of model parameters and evaluate development loss
  - Remember which checkpoint had best development loss
  - If development loss keeps going up, stop training
- Can be used for any model, but especially common for neural networks
  - For linear models, also common to train all the way to convergence

# Dropout

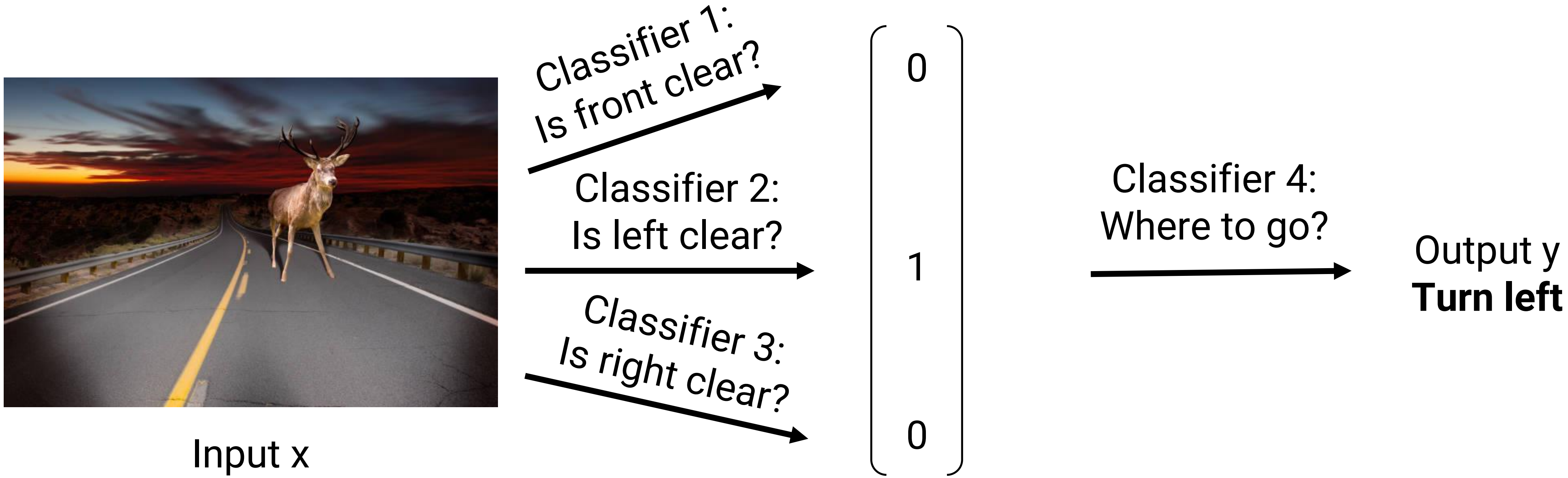


(a) Standard Neural Net

(b) After applying dropout.

- Idea: During training you randomly "drop out" some neurons by setting their value to 0
  - Drop each out with probability p
  - To compensate, scale the other neurons up by 1/p
  - During testing, don't do dropout
- Why?
  - Standard intuition is about "co-adaptation" of neurons
  - My personal intuition: Making the problem harder during training is good practice

# Conclusion



Classifier 1: Is front clear?

Classifier 2: Is left clear?

Classifier 3: Is right clear?

$$\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

Classifier 4: Where to go?

Input x

Output y
**Turn left**

- Neural networks let us learn useful features & approximate any function
- Use multiple layers, with non-linearity between each layer
- Train with stochastic gradient descent, need random initialization to break symmetry
- Regularization is important (especially early stopping)