

# Introduction to Neural Networks

---

**Robin Jia**  
USC CSCI 467, Spring 2024  
February 8, 2024

# Review: Linear Models

---

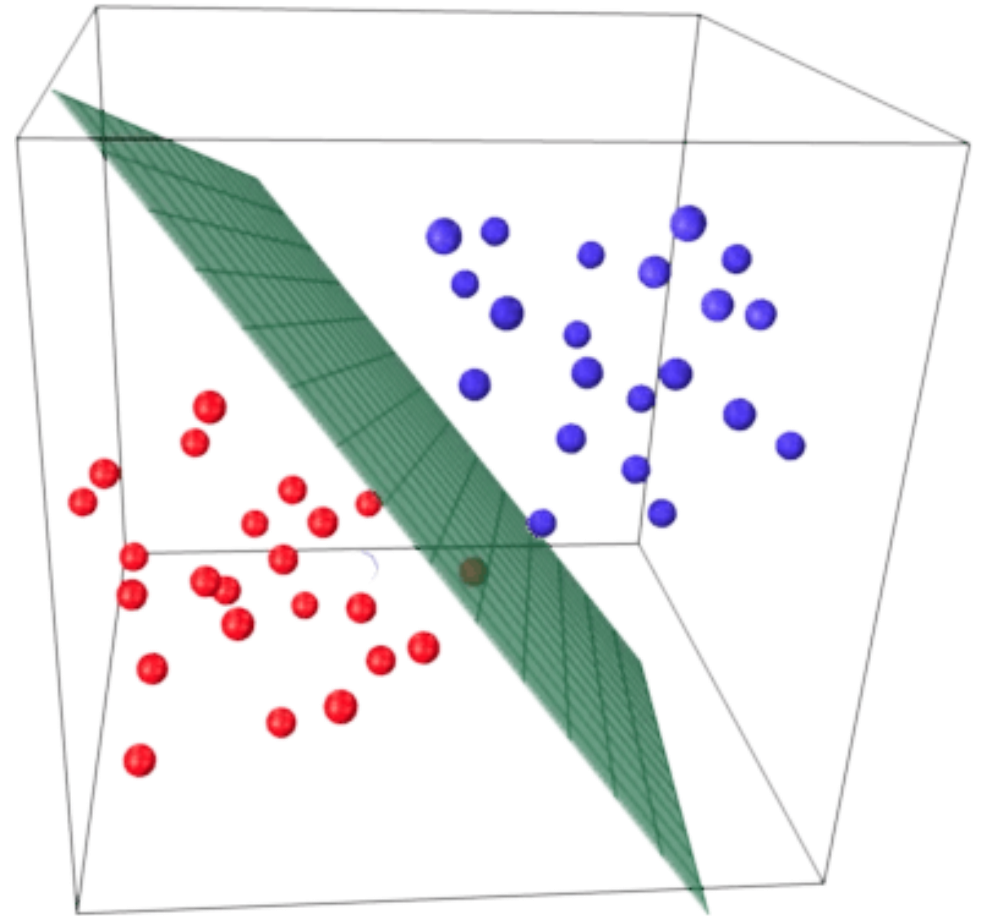
- Examples: Linear regression, logistic regression, softmax regression
- Test time: Make a prediction based on learned parameters
  - E.g., linear regression:  $y_{\text{pred}} = w^{\top} x + b$ , after learning  $w$  and  $b$
- Training time: Learn model parameters
  - Use gradient descent to minimize average loss over training dataset
  - E.g., linear regression:

$$\nabla_w L(w, b) = \frac{1}{n} \sum_{i=1}^n 2(w^{\top} x^{(i)} + b - y^{(i)}) \cdot x^{(i)}$$

# Review: Linear Models

---

- Pro: Easier to understand what model is doing
- Pro: Optimizes convex loss function, gradient descent guaranteed to work
- Con: Can **only learn linear function of input features**
  - Workaround #1: Add more features—but this **requires manual tweaking**
  - Workaround #2: Use kernels, but **only adds more features in pre-defined ways**





Method for learning features from data



Powerful family of non-linear functions



Set of building blocks to create complex models



Method for  
learning features  
from data



Powerful family  
of non-linear  
functions



Set of building  
blocks to create  
complex models

# Deep Learning

Machine Learning using Neural Network models

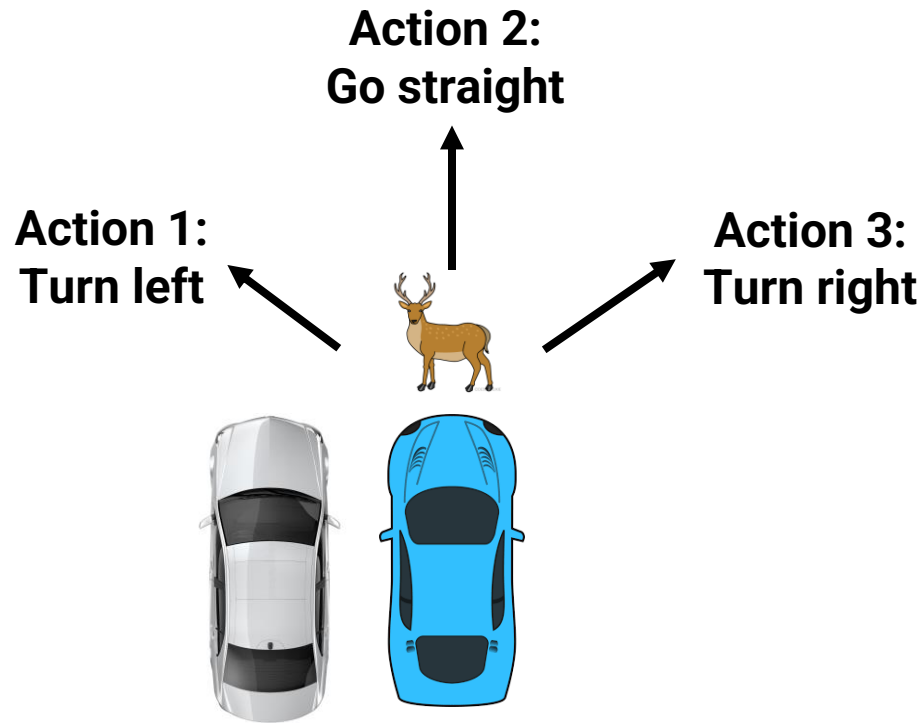




Method for learning features from data

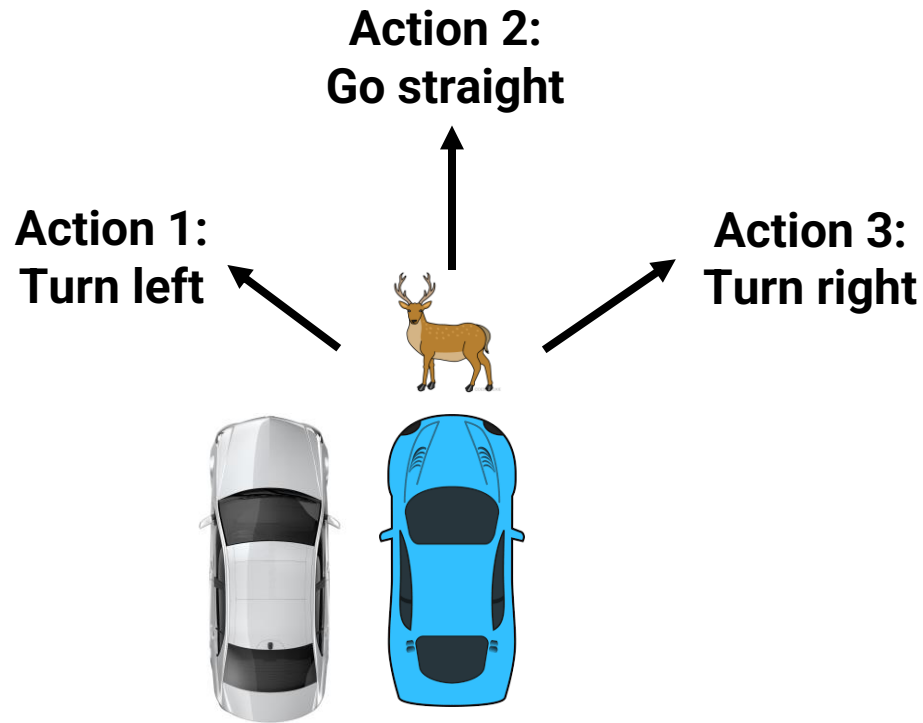
# A (toy) self-driving car example

---



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

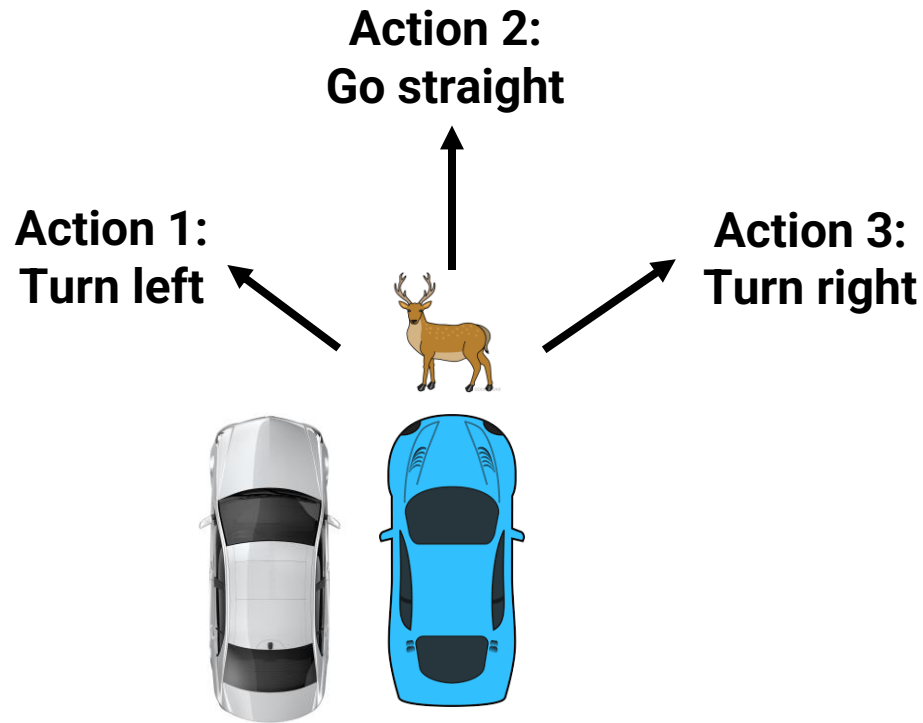
# A (toy) self-driving car example



- 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”:  $10 z_3 - 10$
- 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



Input x

Classifier 1:  
Is front clear?

Classifier 2:  
Is left clear?

Classifier 3:  
Is right clear?

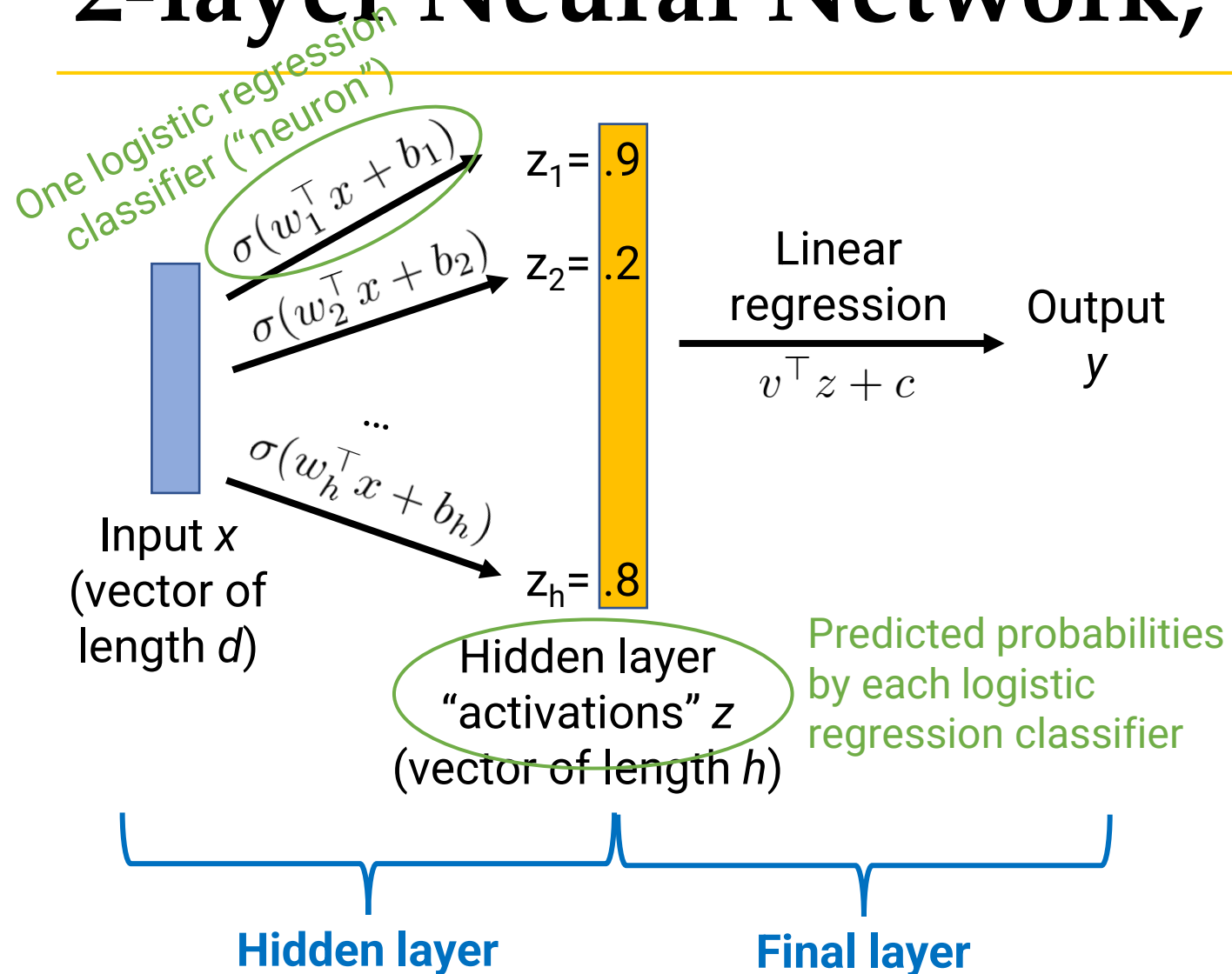
$$\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

Classifier 4:  
Where to go?

Output y  
**Turn left**

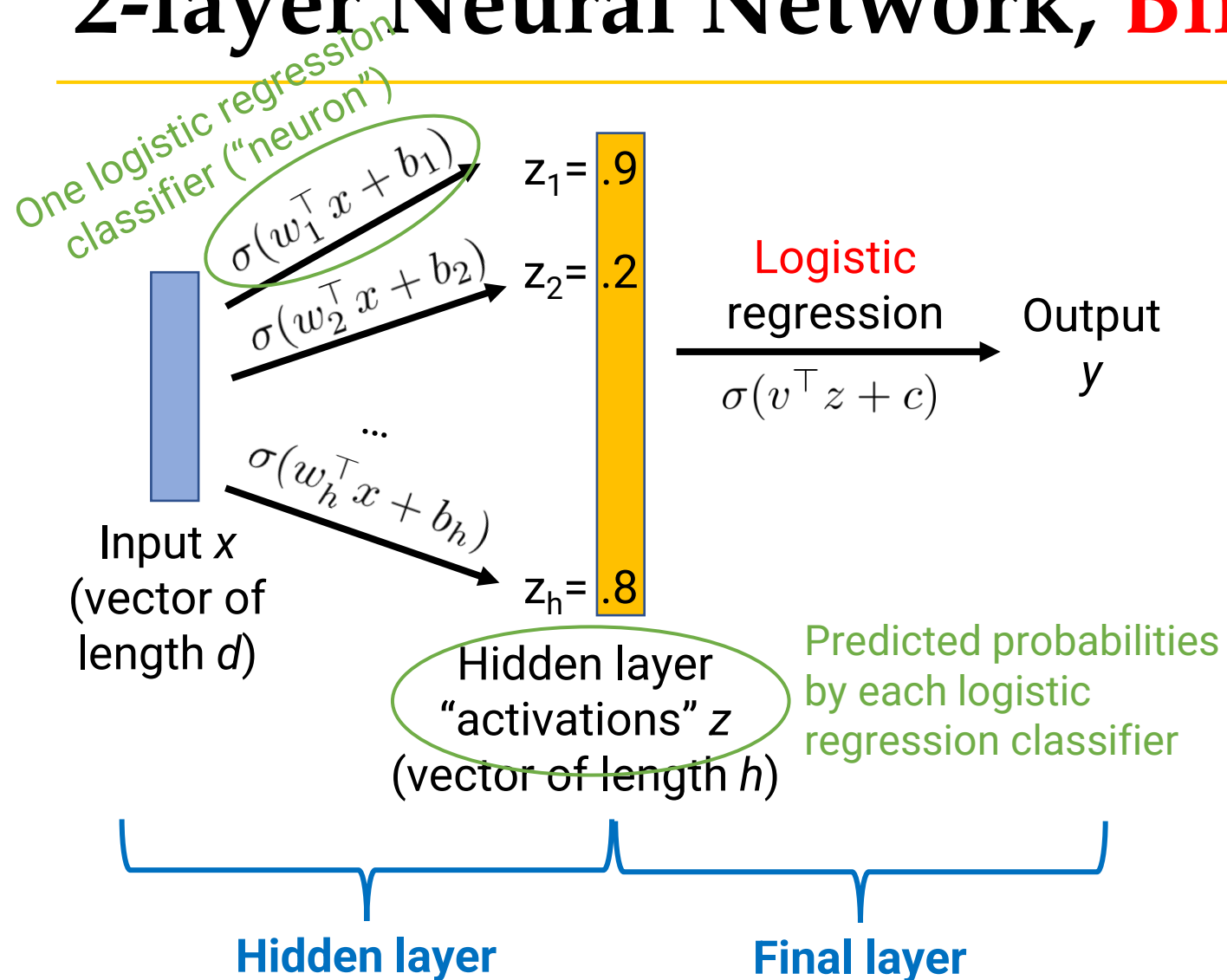
This is a neural network!

# 2-layer Neural Network, Regression



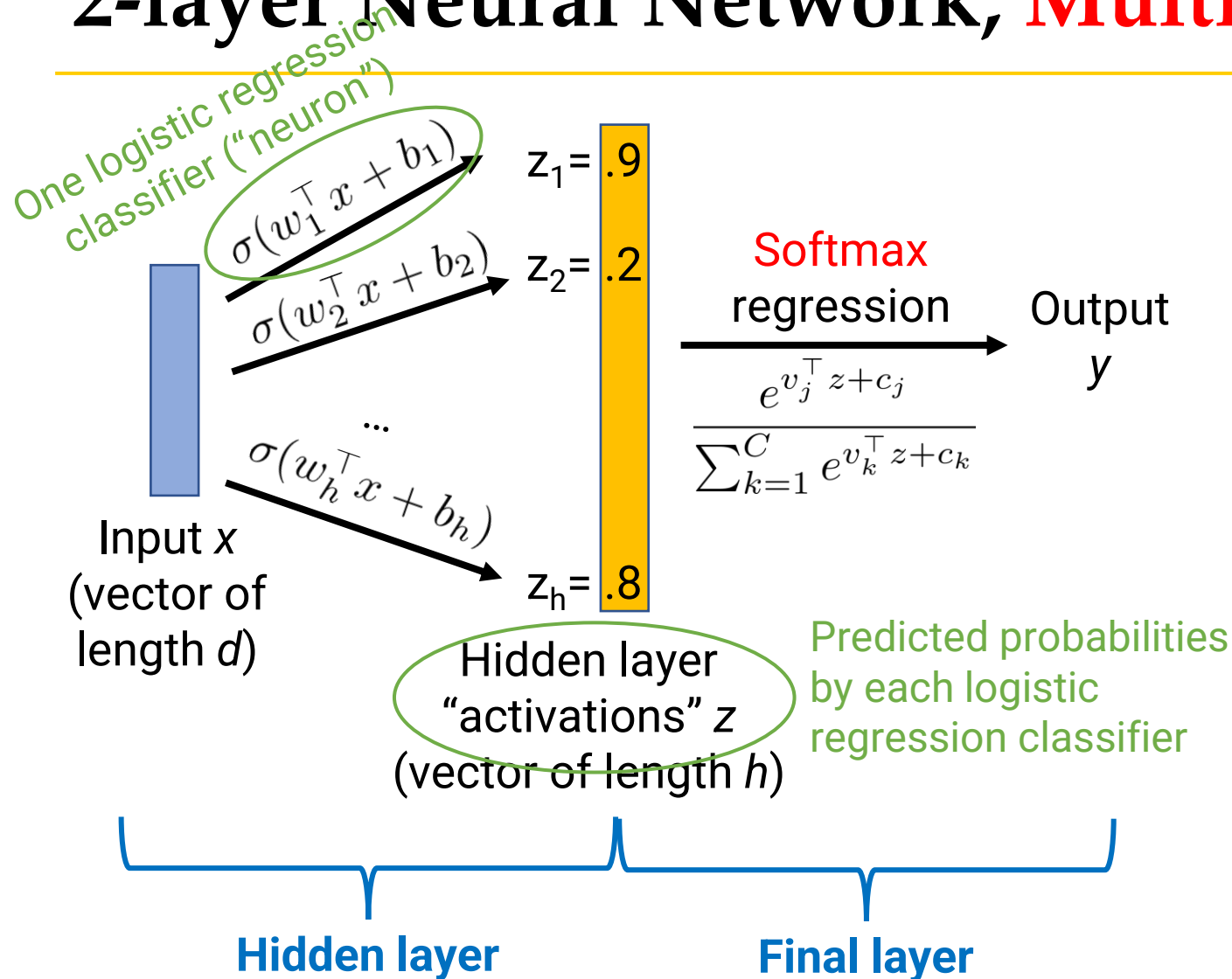
- Hidden layer = A bunch of logistic regression classifiers
  - Parameters:  $w_j$  and  $b_j$  for each classifier, for each  $j=1, \dots, h$
  - $h$  = number of neurons in hidden layer ("hidden nodes")
  - Produces "activations" = learned feature vector
- Final layer = linear model
  - For regression: linear model with weight vector  $v$  and bias  $c$

# 2-layer Neural Network, Binary Classification



- Hidden layer = A bunch of logistic regression classifiers
  - Parameters:  $w_j$  and  $b_j$  for each classifier, for each  $j=1, \dots, h$
  - $h$  = number of neurons in hidden layer ("hidden nodes")
  - Produces "activations" = learned feature vector
- Final layer = linear model
  - For **binary classification**: linear model with weight vector  $v$  and bias  $c$
  - **Only final layer changes when changing to a different task**

# 2-layer Neural Network, Multi-Class Classification



- Hidden layer = A bunch of logistic regression classifiers
  - Parameters:  $w_j$  and  $b_j$  for each classifier, for each  $j=1, \dots, h$
  - $h$  = number of neurons in hidden layer ("hidden nodes")
  - Produces "activations" = learned feature vector
- Final layer = linear model
  - For **multi-class classification**: linear model with weight vector  $v_k$  and bias  $c_k$  for each class  $k$
  - **Only final layer changes when changing to a different task**



# Feature Learning with Neural Networks



Input x

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?

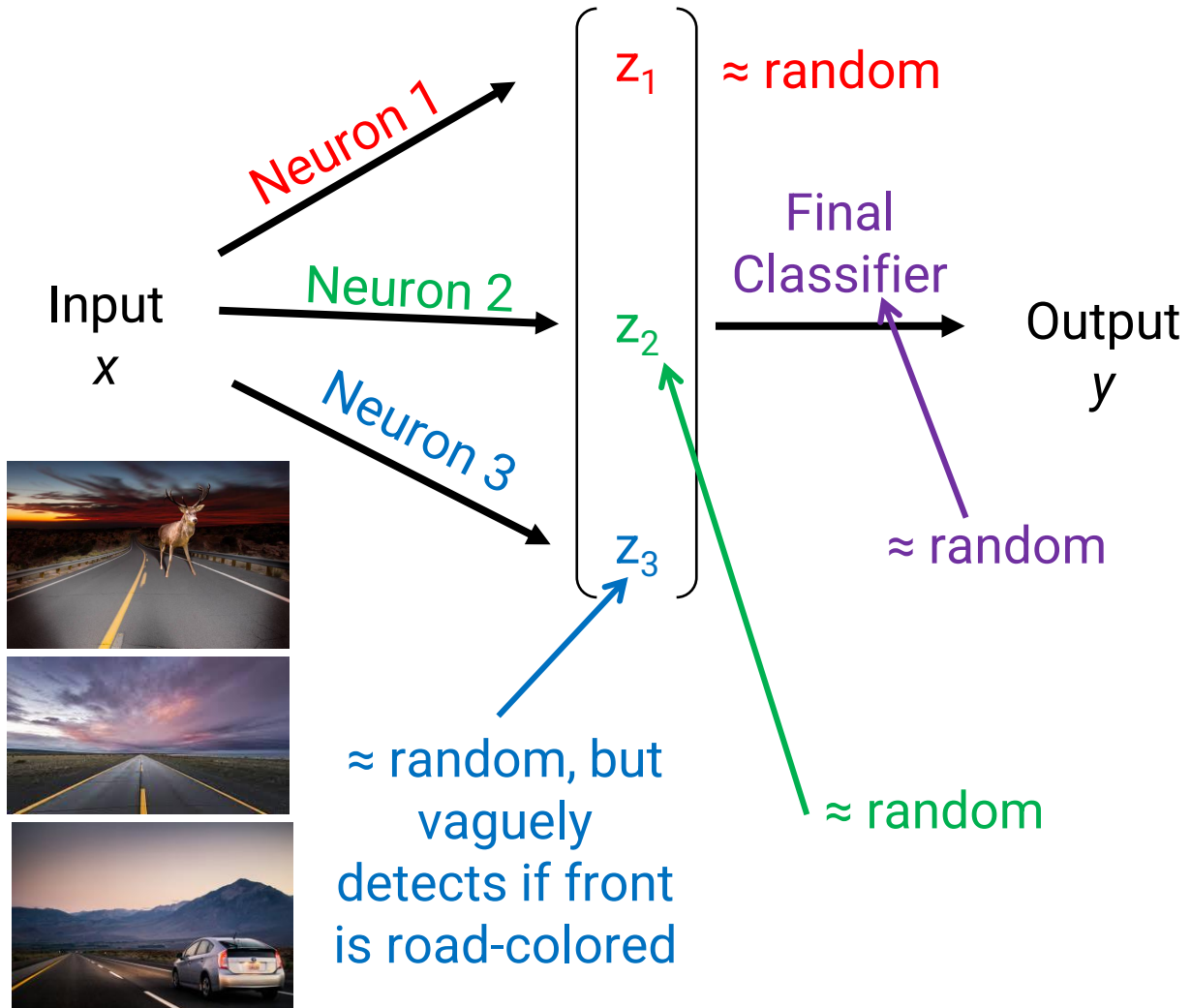
Output y  
**Turn left**

Learn a classifier whose output is a good feature

We don't tell the model what classifier to learn  
Learn from data that "is front clear" is a useful concept

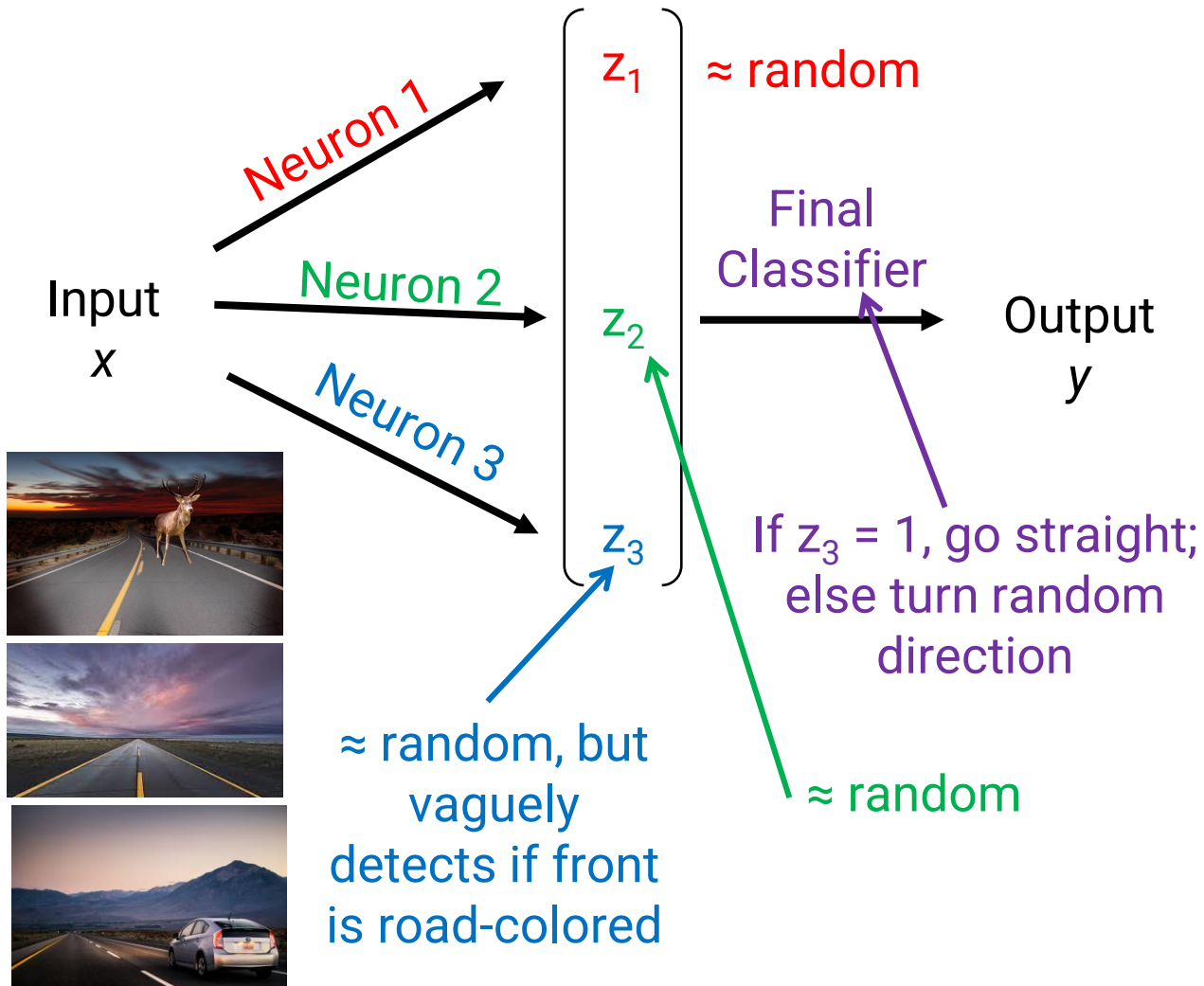
Learn to classify based on features  
(same as linear model)

# Feature Learning with Neural Networks



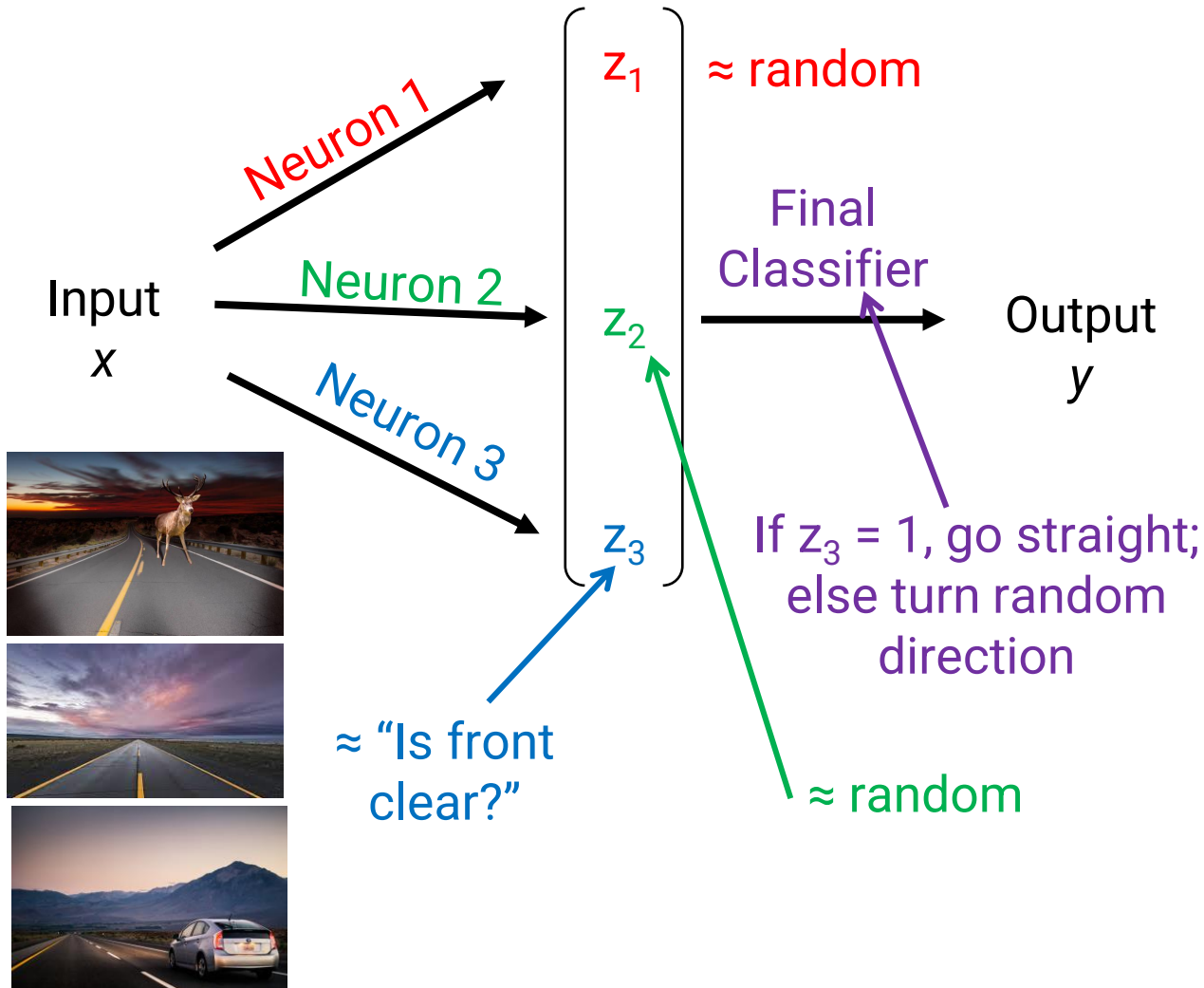
- Plan: Train neural networks to mimic training data via gradient descent
- Initially, each neuron makes random classification “predictions.” Some by chance are mildly useful

# Feature Learning with Neural Networks



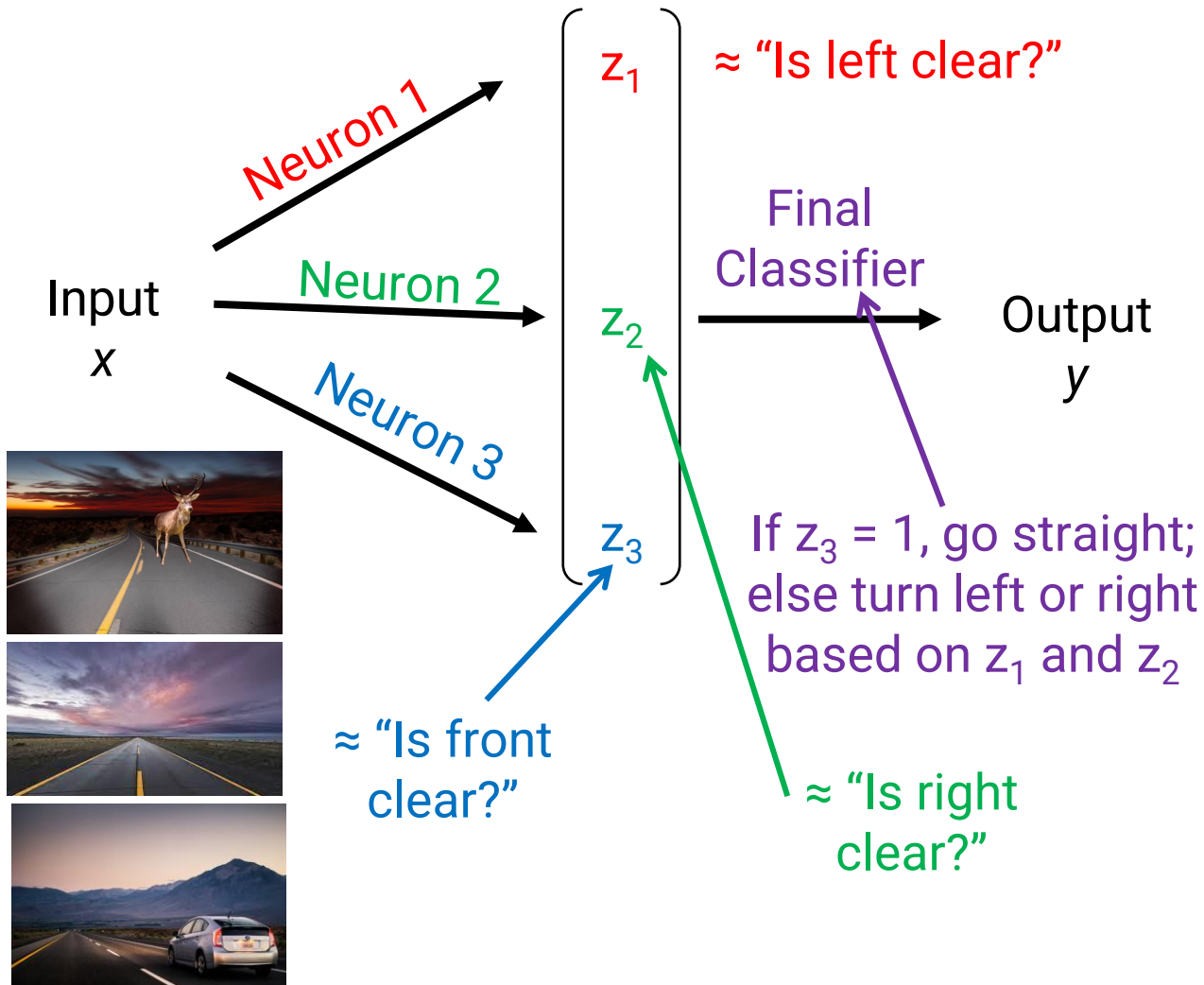
- Plan: Train neural networks to mimic training data via gradient descent
- Initially, each neuron makes random classification “predictions.” Some by chance are mildly useful
- Final layer learns to use the most useful neurons to make final prediction

# Feature Learning with Neural Networks



- Plan: Train neural networks to mimic training data via gradient descent
- Initially, each neuron makes random classification "predictions." Some by chance are mildly useful
- Final layer learns to use the most useful neurons to make final prediction
- The neurons that get used are incentivized to become better, to improve final accuracy

# Feature Learning with Neural Networks



- Plan: Train neural networks to mimic training data via gradient descent
- Initially, each neuron makes random classification "predictions." Some by chance are mildly useful
- Final layer learns to use the most useful neurons to make final prediction
- The neurons that get used are incentivized to become better, to improve final accuracy
- Other neurons slowly change to surface useful information not already captured by other neurons
- Eventually, learned features are very useful and final layer can predict well!

# Summary: Neural Networks as Feature Learners

---

- Neural networks **learn new features from data**
- Each learned feature is the output of a classifier using the original input features
- These classifiers can be “trained” to produce features that help the final layer make good predictions

# Announcements

---

- Project proposals due next Tuesday @ 11:59pm
  - Submit as one group on gradescope (one submission per group)
- Section tomorrow: Review of linear methods
  - Fill out the poll on piazza!
- Homework 2 released soon, Due February 29



Powerful family of non-linear functions

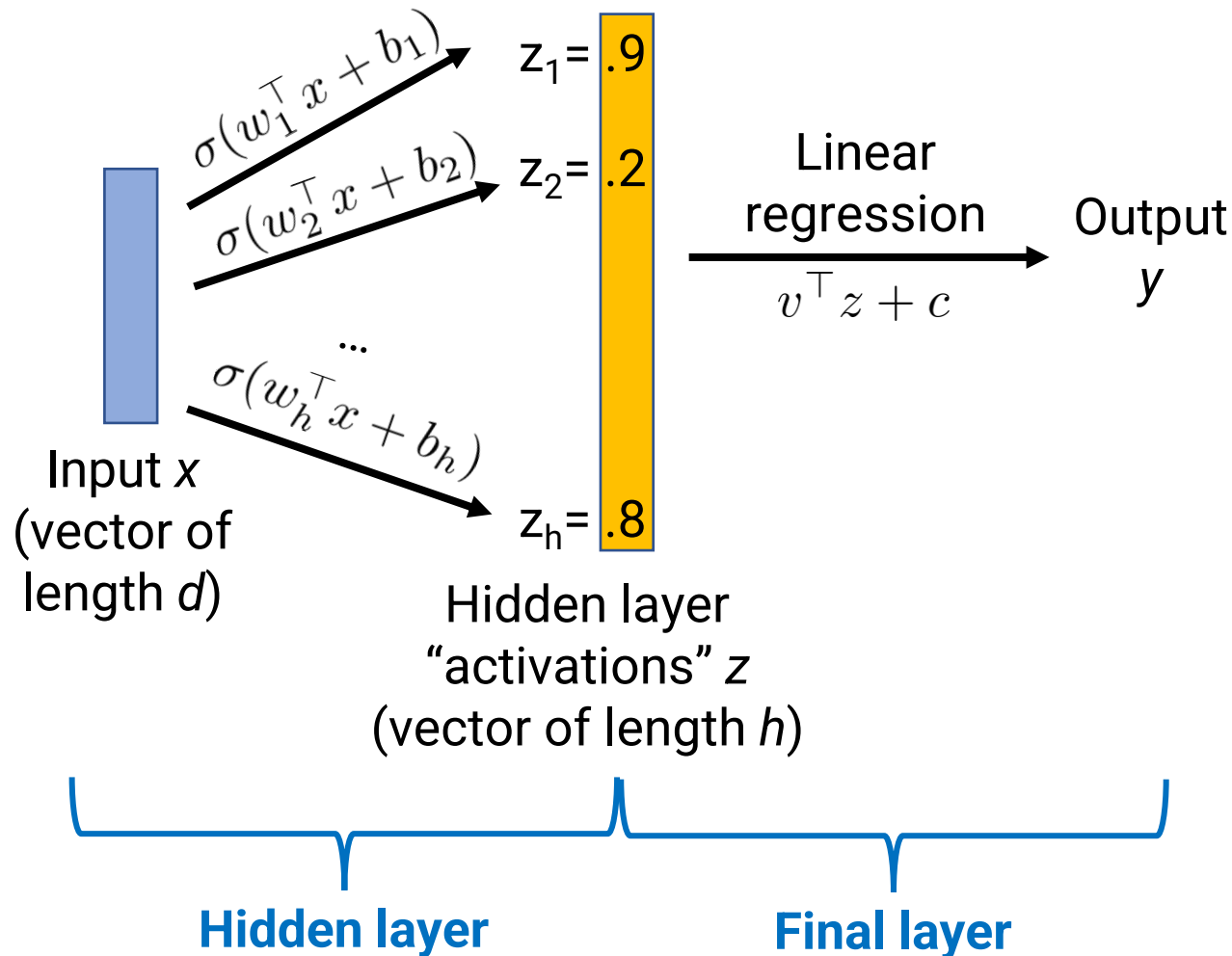


# Neural Networks are Non-linear Functions

---

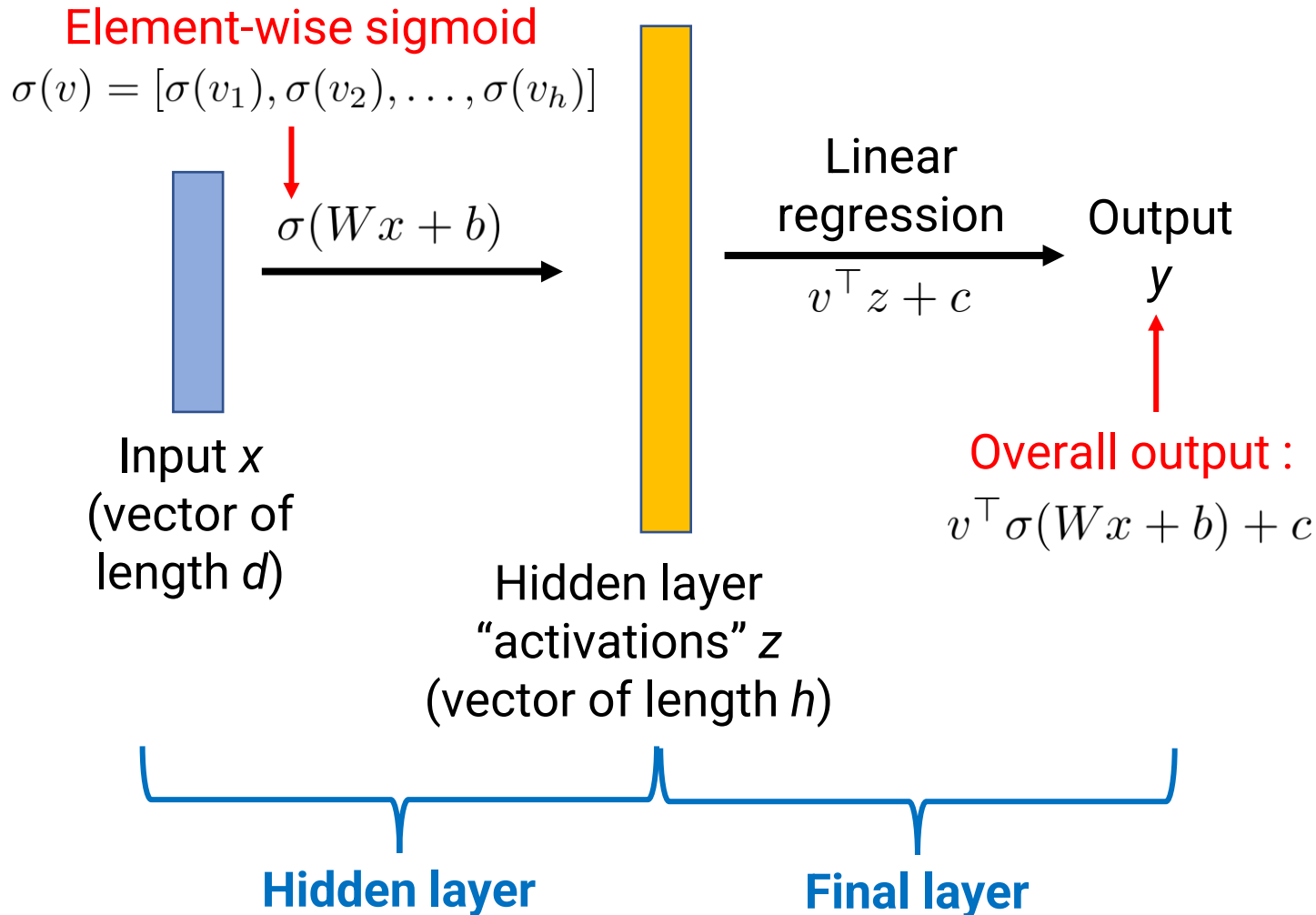
- Second view: Neural networks compute a **non-linear** function of input to make predictions
  - As a result, neural networks can learn many functions that linear models cannot
- In most other ways, neural networks are **very similar** to their linear counterparts!
  - E.g., Training is mostly the same

# 2-layer Neural Network, Regression



- Hidden layer = A bunch of logistic regression classifiers
  - Parameters:  $w_j$  and  $b_j$  for each classifier, for each  $j=1, \dots, h$
  - $h$  = number of neurons in hidden layer ("hidden nodes")
  - Produces "activations" = learned feature vector
- Final layer = linear model
  - For regression: linear model with weight vector  $v$  and bias  $c$

# 2-layer Neural Network in Matrix Form



- Hidden layer = A bunch of logistic regression classifiers
  - Parameters:  $w_j$  and  $b_j$  for each classifier, for each  $j=1, \dots, h$
  - Equivalently: matrix  $W$  ( $h \times d$ ) and vector  $b$  (length  $h$ )
  - $h$  = number of neurons in hidden layer ("hidden nodes")
  - Produces "activations" = learned feature vector
- Final layer = linear model
  - For regression: linear model with weight vector  $v$  and bias  $c$
- Parameters of model are  $\theta = (W, b, v, c)$

# Training Neural Networks

---

## Linear Regression

- Model's output is

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

- (Unregularized) loss function is

$$\frac{1}{n} \sum_{i=1}^n (g(x^{(i)}) - y^{(i)})^2$$

## Regression w/ Neural Networks

- Model's output is

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

- **Use same loss function**, in terms of  $g$ !

$$\frac{1}{n} \sum_{i=1}^n (g(x^{(i)}) - y^{(i)})^2$$

### Training objective for both types of models:

$$\frac{1}{n} \sum_{i=1}^n \ell(y^{(i)}, g(x^{(i)})), \text{ where } \ell(y, u) = (y - u)^2$$

Also applies for  
logistic regression,  
softmax regression, etc.

# Training Neural Networks

---

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

Model's output, depends on all model parameters  $\theta$  (includes all layers)

- How to minimize? Gradient Descent!

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

Average of per-example gradients

- In practice, use a variant of traditional gradient descent
  - (Will discuss in 2 classes)

# Importance of “Non-linearities”

---

With sigmoid,  
overall output is:

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

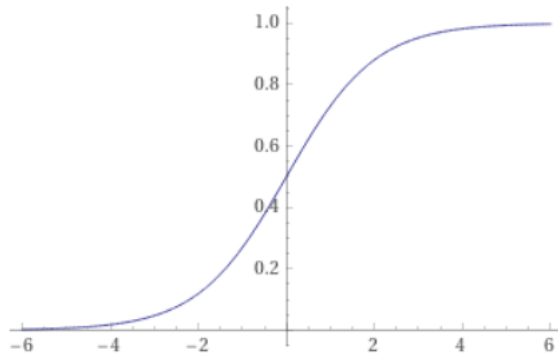
**Without** sigmoid,  
overall output is:

$$\begin{aligned} v^T (Wx + b) + c \\ = (v^T W)x + (v^T b + c) \end{aligned}$$

- Having the sigmoid is very important!
- What if we skipped the sigmoid?
- Result: Just another way to write a linear function!
  - New “weight” is  $v^T W$
  - New “bias” is  $v^T b + c$
- Having a **simple non-linear function** (like sigmoid) between the two linear operations enables us to learn a **complex non-linear function!**

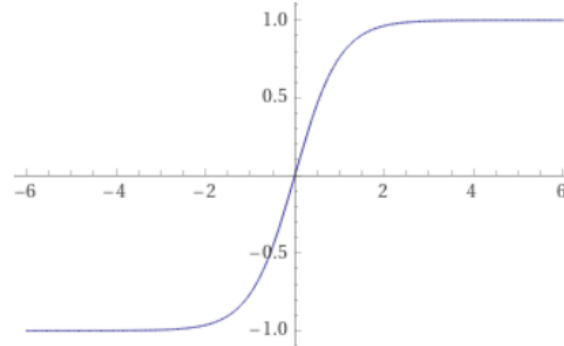
# Options for Non-linearities

## Sigmoid



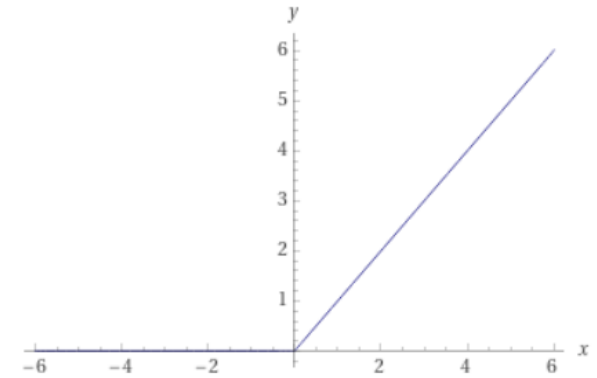
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

## Tanh



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

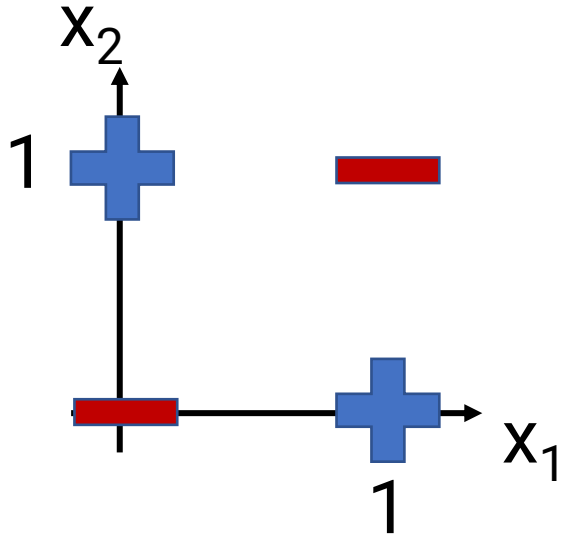
## ReLU



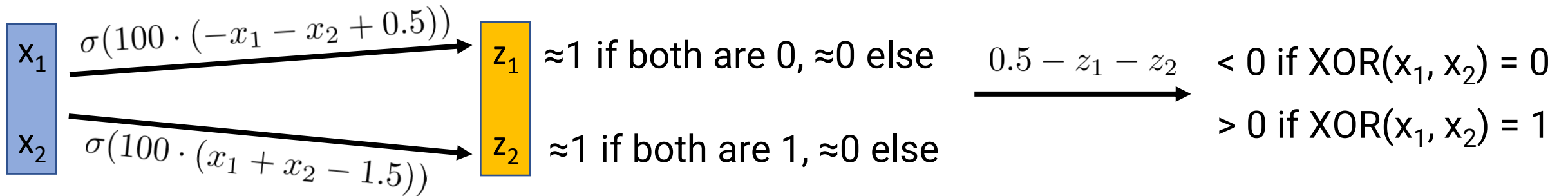
$$\text{ReLU}(z) = \max(z, 0)$$

- Many options work, just must be differentiable (for gradient descent)
  - Sometimes called “activation functions” or just “non-linearities”
- In practice: tanh and ReLU often preferred
  - Tanh: Better than sigmoid because outputs centered around zero
  - ReLU: Very fast to compute

# Non-linearities make NN's more expressive



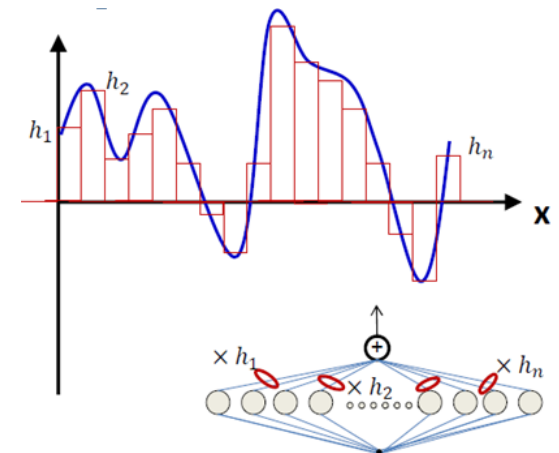
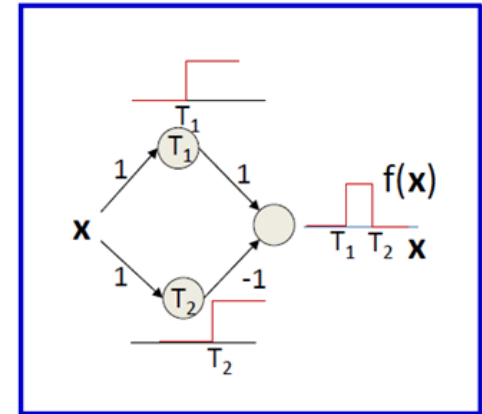
- XOR: Classic binary classification problem that can't be solved by linear classifier
- A 2-layer neural network can solve it!
  - I can choose values for the parameters that lead to perfect classification on this dataset
  - Conclusion: Neural networks are **more expressive** than linear models





# 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 indicator-like features like “is  $x > 1$ ?”, “is  $x > 2$ ?”, etc.
  - This divides space into a bunch of buckets of width 1
  - Second layer assigns correct value to each bucket
  - If you have enough hidden units, you can make buckets really small and approximate a function very well

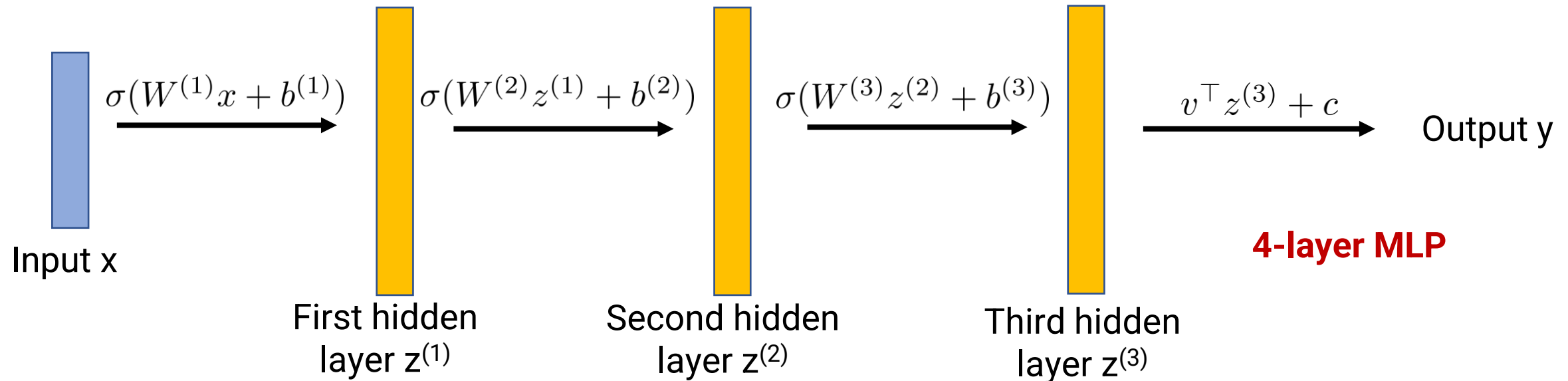


# Summary: Neural Networks as Non-linear Functions

---

- Neural Networks are a type of **learnable non-linear function**
  - Contrast with previous models, which are learnable linear functions
- Constructed from multiple layers, each of which have their own learnable parameters
- Non-linearities between layers make the model much more expressive—can represent **any function** with a big enough network and right choice of parameters
  - Whereas linear models just cannot learn certain functions, like XOR
- Learning process works the same way as for linear models

# Multi-Layer Perceptron (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



Set of building blocks to create complex models

# Deep Learning as a Set of Building Blocks

- Neural Network = Many “layers” stacked on top of each other
  - Each “layer” takes in some input and computes some output
  - Simplest layers are basic building blocks, can build more complex layers from those
  - Arrangement of layers is called an “architecture”
- Deep Learning: **Design suitable neural architectures with various reusable building blocks**
  - This is the view of most deep learning programming libraries like pytorch

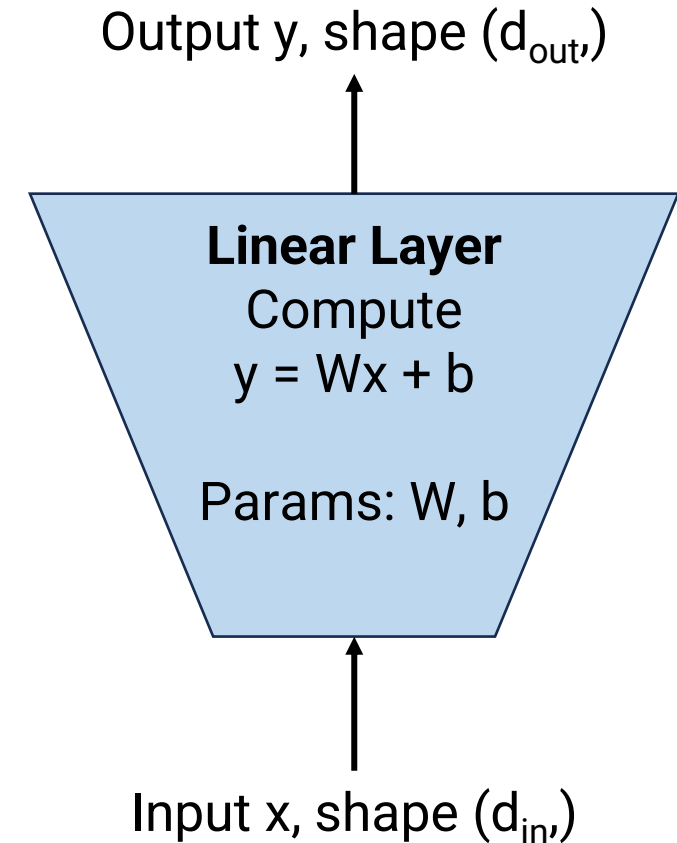


# The Basic “Building Blocks”

---

## (1) Linear Layer

- Input  $x$ : Vector of dimension  $d_{in}$
- Output  $y$ : Vector of dimension  $d_{out}$
- Formula:  $y = Wx + b$
- Parameters
  - $W$ :  $d_{out} \times d_{in}$  matrix
  - $b$ :  $d_{out}$  vector
- In pytorch: `nn.Linear()`

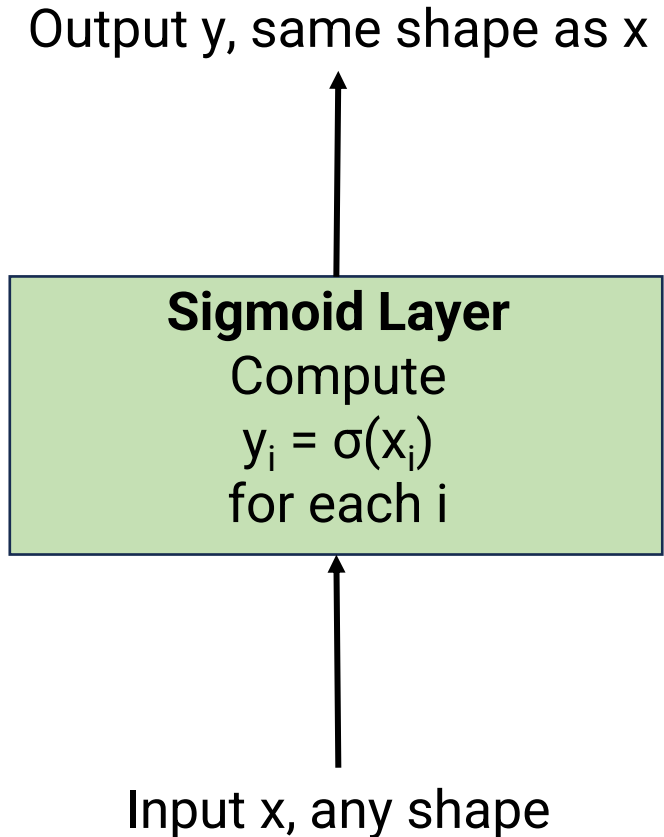


# The Basic “Building Blocks”

---

## (2) Non-linearity Layer

- Input  $x$ : Any number/vector/matrix
- Output  $y$ : Number/vector/matrix of same shape
- Possible formulas:
  - Sigmoid:  $y = \sigma(x)$ , elementwise
  - Tanh:  $y = \tanh(x)$ , elementwise
  - Relu:  $y = \max(x, 0)$ , elementwise
- Parameters: None
- In pytorch: `torch.sigmoid()`, `nn.functional.relu()`, etc.

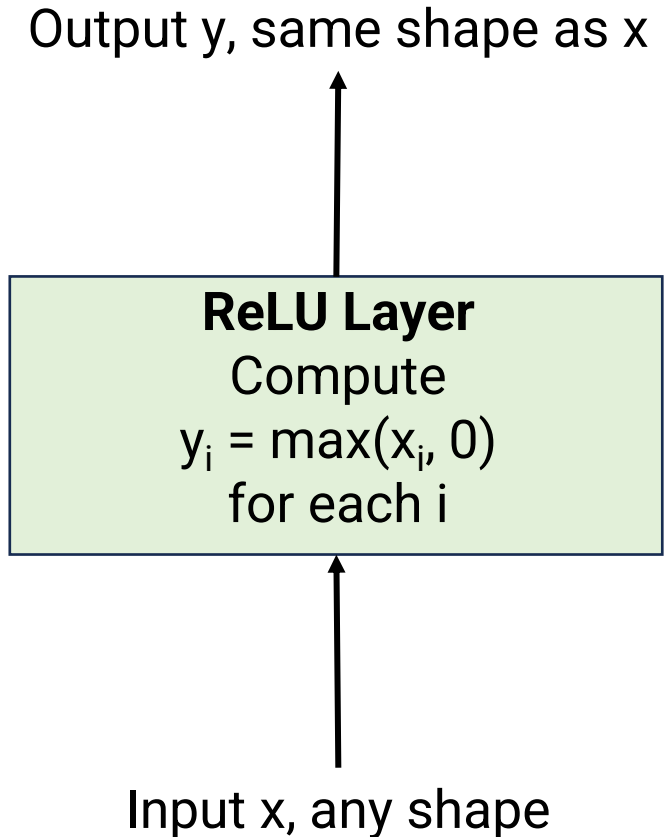


# The Basic “Building Blocks”

---

## (2) Non-linearity Layer

- Input  $x$ : Any number/vector/matrix
- Output  $y$ : Number/vector/matrix of same shape
- Possible formulas:
  - Sigmoid:  $y = \sigma(x)$ , elementwise
  - Tanh:  $y = \tanh(x)$ , elementwise
  - Relu:  $y = \max(x, 0)$ , elementwise
- Parameters: None
- In pytorch: `torch.sigmoid()`, `nn.functional.relu()`, etc.

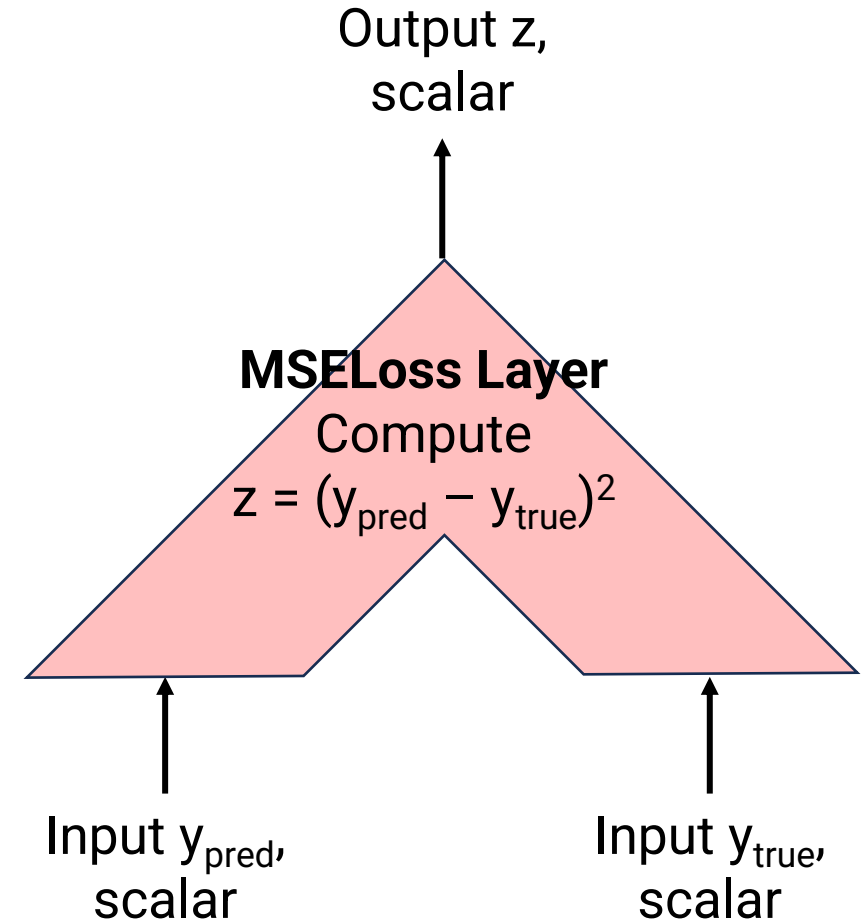




# The Basic “Building Blocks”

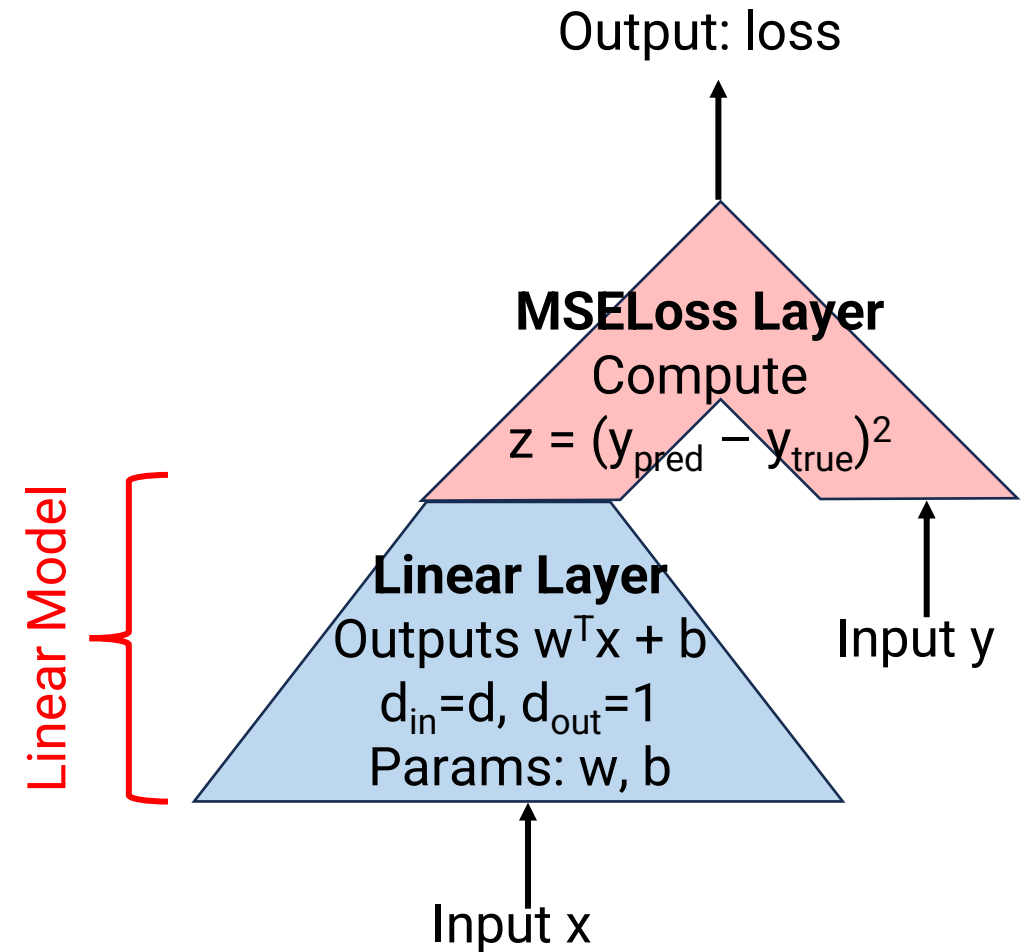
## (3) Loss Layer

- Inputs:
  - $y_{\text{pred}}$ : shape depends on task
  - $y_{\text{true}}$ : scalar (e.g., correct regression value or class index)
- Output  $z$ : scalar
- Possible formulas:
  - Squared loss:  $y_{\text{pred}}$  is scalar,  $z = (y_{\text{pred}} - y_{\text{true}})^2$
  - Softmax regression loss:  $y_{\text{pred}}$  is vector of length  $C$ ,
$$z = - \left( y_{\text{pred}}[y_{\text{true}}] - \log \sum_{i=1}^C \exp(y_{\text{pred}}[i]) \right)$$
- Parameters: None
- In pytorch: `nn.MSELoss()`, `nn.CrossEntropyLoss()`, etc.



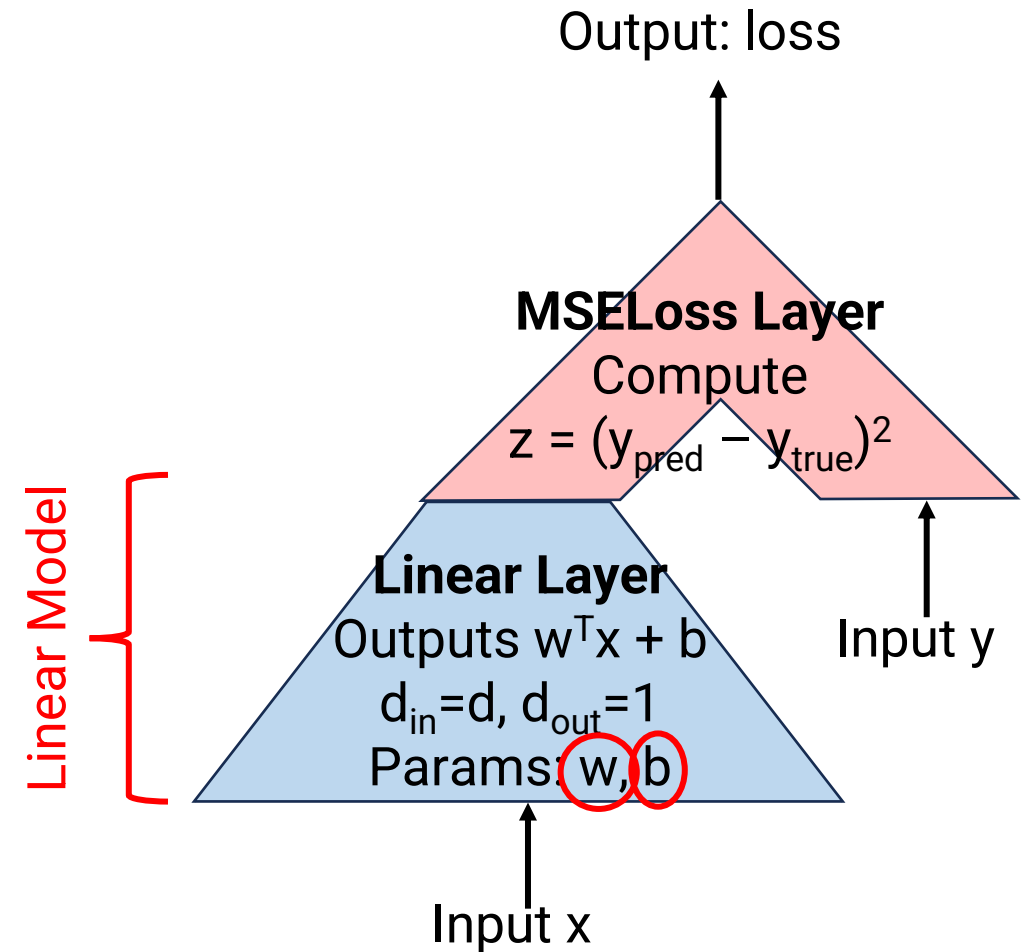
# Building Linear Regression Training Loop

- Step 1: Compute the loss on one example at a time
  - Training example is  $(x, y)$
  - $x$  is vector of length  $d$ ,  $y$  is scalar



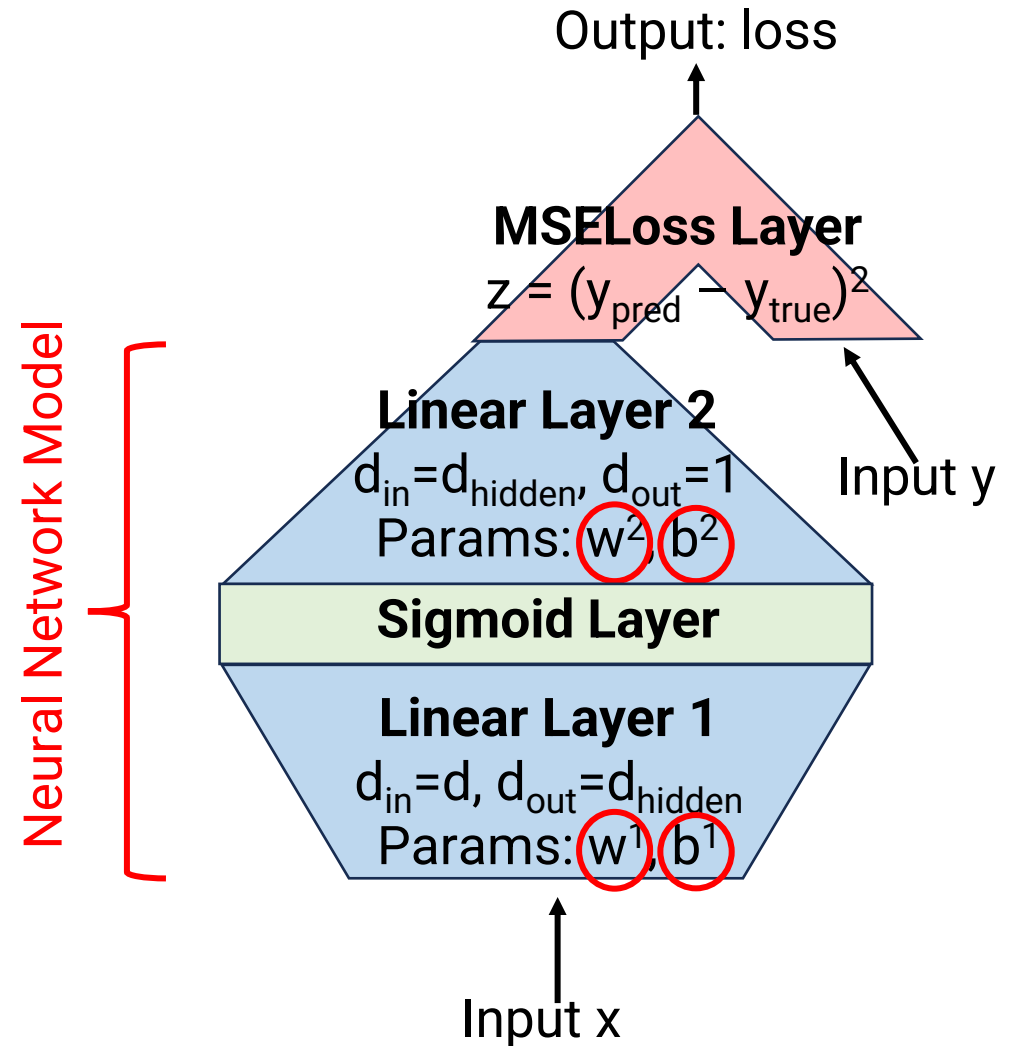
# Building Linear Regression Training Loop

- Step 1: Compute the loss on one example at a time
  - Training example is  $(x, y)$
  - $x$  is vector of length  $d$ ,  $y$  is scalar
- Step 2: Compute gradient of loss with respect to all parameters
- Step 3: Update **all parameters** with gradient descent update rule



# Building a 2-layer MLP for Regression

- **Steps for training are exactly the same!**
- Step 1: Compute the loss on one example at a time
  - Training example is  $(x, y)$
  - $x$  is vector of length  $d$ ,  $y$  is scalar
- Step 2: Compute gradient of loss with respect to all parameters
  - **Next class: Compute gradient automatically with backpropagation. Easy if each building block is differentiable!**
- Step 3: Update **all parameters** with gradient descent update rule



# Summary: Deep Learning as Building Blocks

---

- Power of deep learning: You can **stack building blocks together any way you want**
  - No “right” or “wrong” architecture, just different design decisions
  - Best architecture choice depends on the task and data
  - Endless possibilities for new architectures





Method for  
learning features  
from data



Powerful family  
of non-linear  
functions



Set of building  
blocks to create  
complex models