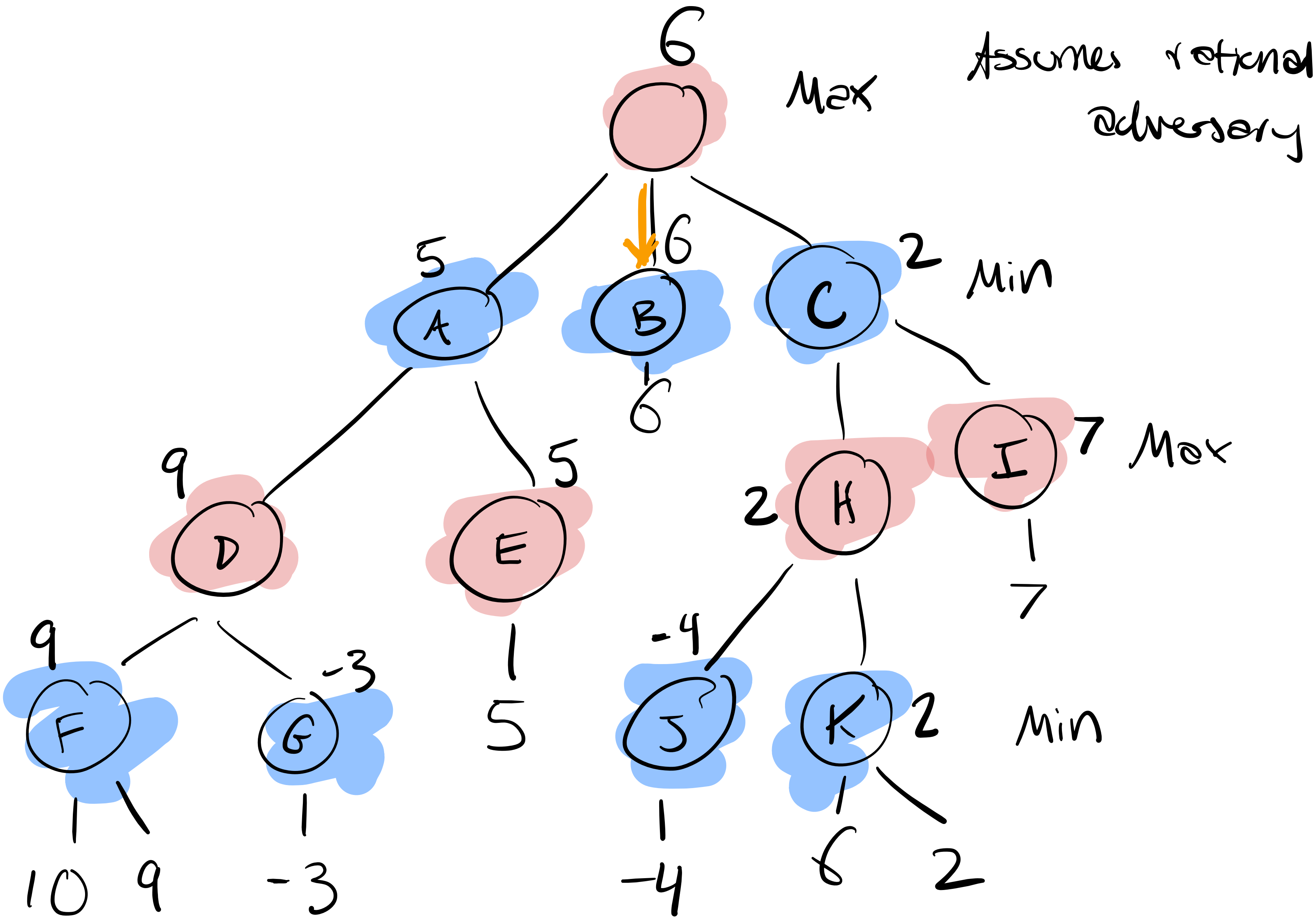
CS 232: Artificial Intelligence

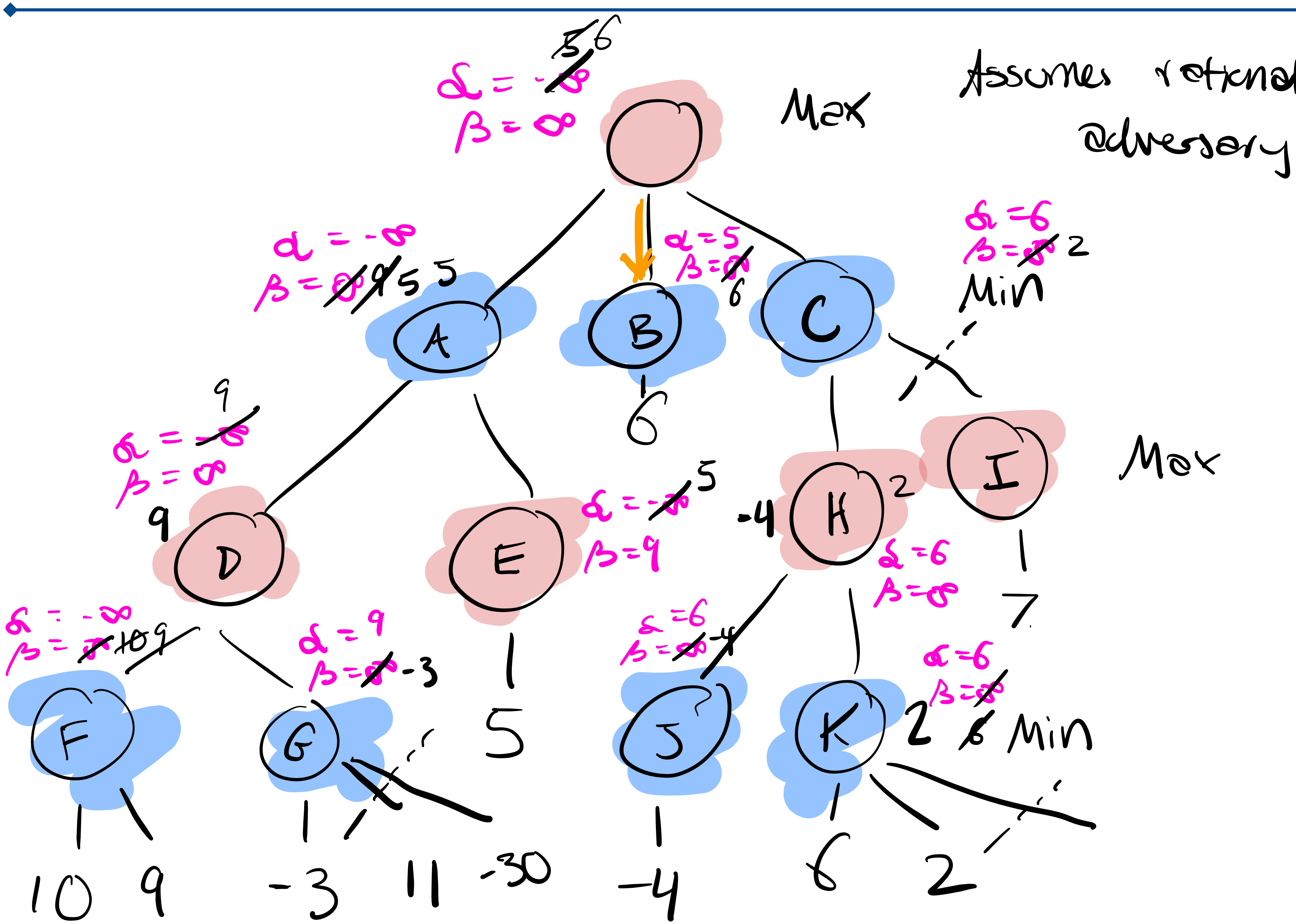
Spring 2024

Prof. Carolyn Anderson
Wellesley College

Midterm Review



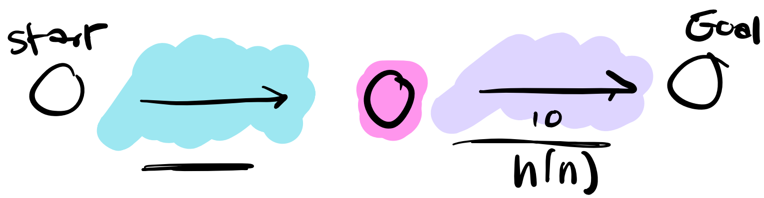
Midterm Review



Midterm Review

Greedy Best First Search : cost = $h(n)$

A* : cost = $g(n) + h(n)$



$h(n)$: guess of how expensive it is from n to goal

$g(n)$: actual cost from start to n
sum of cost of actions taken to reach n

Admissable heuristic is optimistic : never overestimates cost of reaching goal from n

Recap:
Logistic Regression Classifiers

How to do classification

For each feature x_i, weight w_i tells us importance of x_i

- (Plus we'll have a bias b)

We'll sum up all the weighted features and the bias

$$z = \left(\sum_{i=1}^{n} w_i x_i\right) + b$$

$$z = w \cdot x + b$$

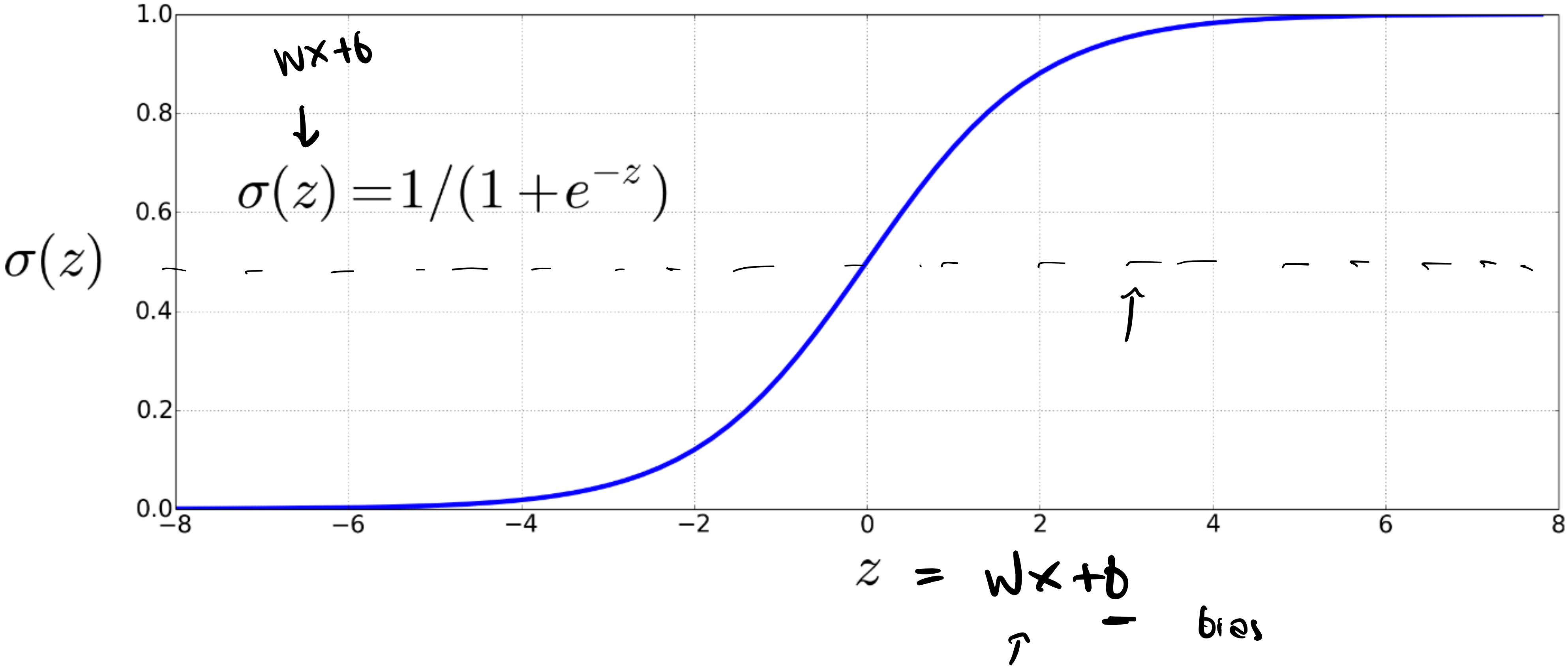
If this sum is high, we say y=1; if low, then y=0

Turning a probability into a classifier

$$\text{decision}(x) = \begin{cases} 1 & \text{if } P(y=1|x) > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

0.5 here is called the **decision boundary**

The very useful sigmoid or logistic function



Turning a probability into a classifier

$$\text{decision}(x) = \begin{cases} 1 & \text{if } P(y=1|x) > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

if $w \cdot x + b > 0$

if $w \cdot x + b \leq 0$

Idea of logistic regression

- Compute $w \cdot x + b$
- Pass it through the sigmoid function: $\sigma(w \cdot x + b)$ so that we can treat it as a probability

$$P(y=1) = \sigma(w \cdot x + b)$$

$$P(y=0) = 1 - \sigma(w \cdot x + b)$$

The two phases of logistic regression

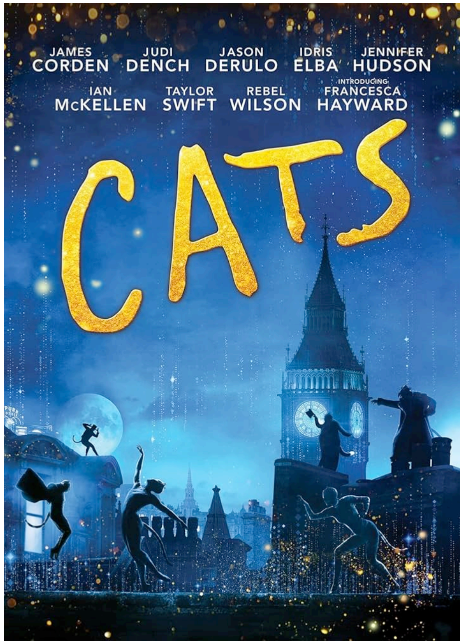
Training: we learn weights *w* and *b* using **stochastic gradient descent** and **cross-entropy loss**.

Test: Given a test example *x* we compute $p(y|x)$ using learned weights *w* and *b*, and return whichever label (y = 1 or y = 0) is higher probability

Logistic Regression Example: Text Classification

Sentiment example: does y=1 or y=0?

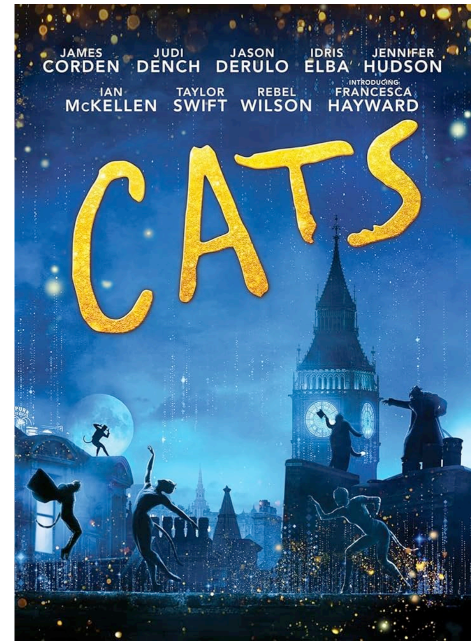
CATS was a marvelous disaster, with witty charm and emotion throughout, cheeky charisma and crying no doubt… I personally went in expecting the worst movie I had ever seen - and it was far more awful and disappointing that I expected.



Sentiment example: does y=1 or y=0?

Verified   Jan 12, 2020

CATS was a marvelous disaster, with witty charm and emotion throughout, cheeky charisma and crying no doubt… I personally went in expecting the worst movie I had ever seen - and it was far more awful and disappointing that I expected.



Features

Var	Definition	Value
x_1	count(positive words)	6
x_2	count(negative words)	4
x_3	$\begin{cases} 1 & \text{if "no"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	1
x_4	$\begin{cases} 1 & \text{if "!"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	0
x_5	log(word count)	3.73

42

$W = [2.5, -5, -1.2, 2, 0.7]$

Weights

$w = [2.5, -5, -1.2, 2, 0.7]$

$b = 0.1$

$x = [6, 4, 1, 0, 3.73]$

$$p(+ \mid x) = P(y=1 \mid x)$$
$$= \sigma(wx + b)$$
$$= \sigma(wx + 0.1)$$
$$= \sigma(2.5 \cdot 6 + -5 \cdot 4 + -1.2 \cdot 1 + 2 \cdot 0 + 0.7 \cdot 3.73 + 0.1)$$
$$= \sigma(-1.78)$$
$$= 0.14$$

$$p(- \mid x) = 1 - 0.14$$
$$= 0.86$$

Classifying sentiment for input x

Classification in (binary) logistic regression: summary

Given:

- a set of classes: (+ sentiment,- sentiment)
- a vector **x** of features `[x1, x2, x3, …, xn]`
  - x1= count("awesome") in document
  - x3 = 1 if "no" in document else 0
- A vector **w** of weights `[w1, w2, …, wn]`
  - w_i for each feature f_i

$$
\begin{aligned}
P(y=1) &= \sigma(w \cdot x + b) \\
&= \frac{1}{1+\exp(-(w \cdot x + b))}
\end{aligned}
$$

Feature Representations

Slides borrowed from Jurafsky & Martin Edition 3

Image Features

For computer vision applications, we need a way of describing images. We represent images as matrices of pixel values.

Grayscale images can be represented with a single matrix.

Color images need to be represented with a **3D tensor** (3rd dimension encodes color channel).

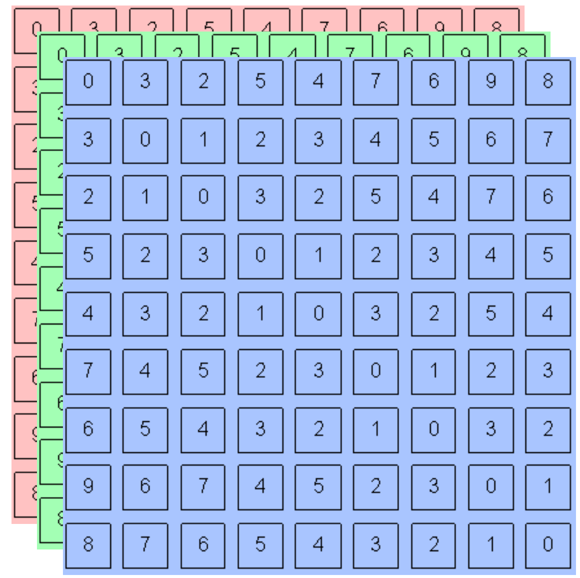
Why matrices for images and vectors for text?
Language is **sequential**, which makes it more useful to concatenate vectors lengthwise rather than stack them.

grayscale images are matrices

0	3	2	5	4	7	6	9	8
3	0	1	2	3	4	5	6	7
2	1	0	3	2	5	4	7	6
5	2	3	0	1	2	3	4	5
4	3	2	1	0	3	2	5	4
7	4	5	2	3	0	1	2	3
6	5	4	3	2	1	0	3	2
9	6	7	4	5	2	3	0	1
8	7	6	5	4	3	2	1	0

what range of values can each pixel take?

color images are tensors



channel x height x width

Channels are usually RGB: Red, Green, and Blue

Other color spaces: HSV, HSL, LUV, XYZ, Lab, CMYK, etc

Logistic Regression Example: Pet Picture Classification

Goal: Classify Pet Pictures

- Dataset: cat + dog pictures
- Goal: classify a picture as either a cat or a dog
- Input: grayscale images

Building a Model

We'll build our model using a machine learning library called **Tensorflow**.

Tensorflow is a Python library, but most functions are implemented in C (so they are fast!).

Tensorflow provides useful abstractions for models:

- **tensor**: n-dimensional container for data
- **layer**: apply functions to an input tensor of n dimensions to produce an output tensor of m dimensions.
- **model**: consist of layers connected together

Example Data

Splitting Our Data

Generate a `Dataset`

In [4]:
```
image_size = (180, 180)
batch_size = 32

train_ds = tf.keras.preprocessing.image_dataset_from_directory(
    "PetImages",
    color_mode='grayscale',
    validation_split=0.2,
    subset="training",
    seed=1337,
    image_size=image_size,
    batch_size=batch_size,
)
val_ds = tf.keras.preprocessing.image_dataset_from_directory(
    "PetImages",
    color_mode='grayscale',
    validation_split=0.2,
    subset="validation",
    seed=1337,
    image_size=image_size,
    batch_size=batch_size,
)
```

```
Found 23410 files belonging to 2 classes.
Using 18728 files for training.
Found 23410 files belonging to 2 classes.
Using 4682 files for validation.
```

Creating Our Model Architecture

```
def make_model(input_shape, num_classes):
    inputs = keras.Input(shape=input_shape)
    x = layers.Flatten()(inputs)
    if num_classes == 2:
        activation = "sigmoid"
        units = 1
    else:
        activation = "softmax"
        units = num_classes
    outputs = layers.Dense(units, activation=activation)(x)
    return keras.Model(inputs, outputs)


model = make_model(input_shape=image_size, num_classes=2)
keras.utils.plot_model(model, show_shapes=True)
```

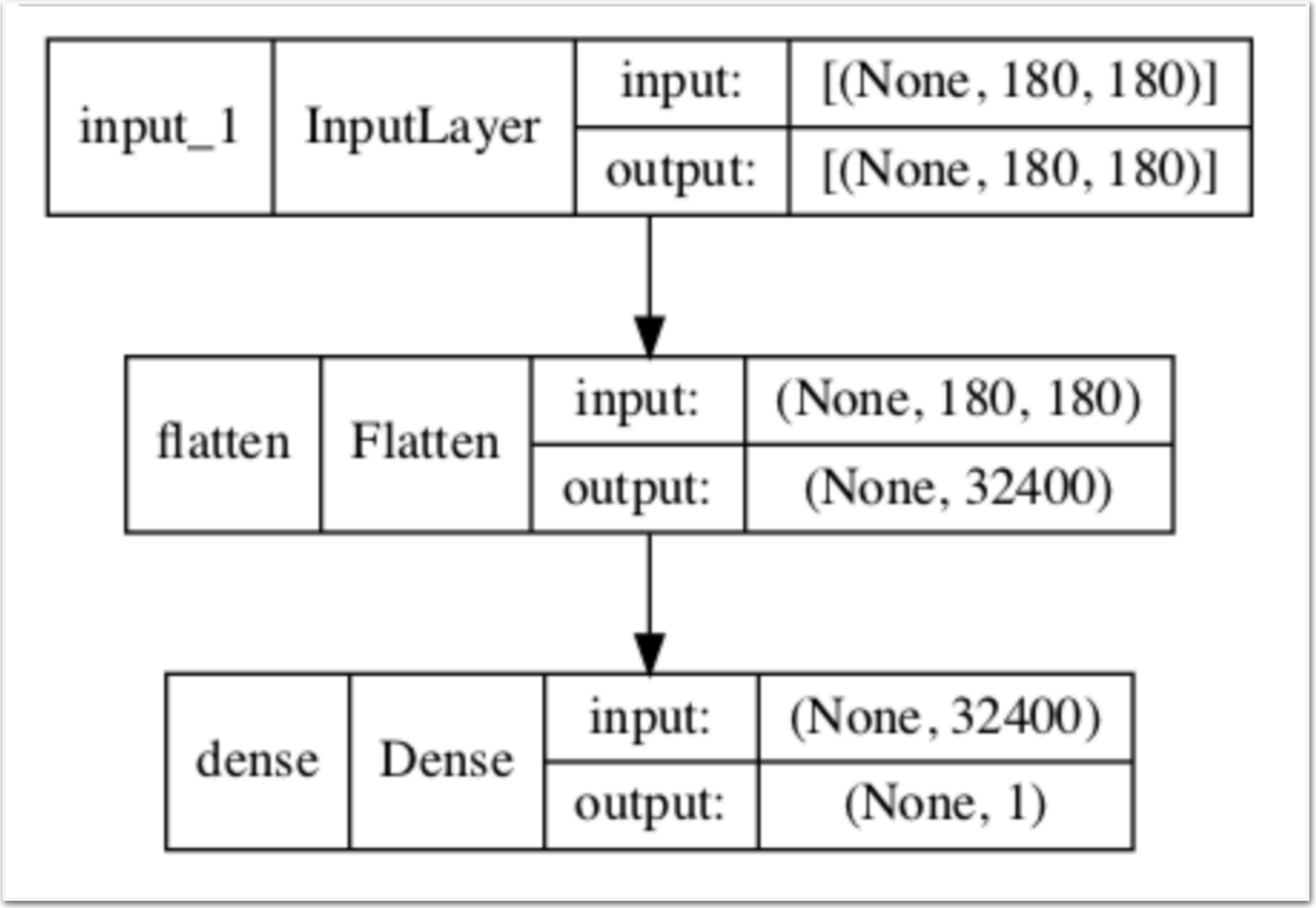
input layer

flatten to a single dimension

select sigmoid or softmax based on number of classes

weights + bias layer - this is the regression bit!

Creating Our Model Architecture



Training

Train the model

```
epochs = 50

callbacks = [
    keras.callbacks.ModelCheckpoint("save_at_{epoch}.h5"),
]
model.compile(
    optimizer=keras.optimizers.Adam(1e-3),
    loss="binary_crossentropy",
    metrics=["accuracy"],
)
model.fit(
    train_ds, epochs=epochs, callbacks=callbacks, validation_data=val_ds,
)
```

Computing with Probabilities

Numerical Underflow

So far we've been working with relatively small sample spaces. This means our probabilities have been decently large.

As we go on in this class, our sample spaces are going to get much larger. We want to be able to reason about the probabilities of things like:

- All words in English
- All pixels in a photo
- All possible game states for Pacman

Numerical Underflow

Problem: when our probabilities get really really small, programming languages start making mistakes.

There is a **bound on precision** in numerical computing.

This is because of the limitations on space allocation for (floating point) numbers.

Solution: make the numbers bigger

- Intuition: we care about how big probabilities are relative to the other probabilities in our distribution, not the actual value.

Probabilities:
p(heart) = 0.1
p(rainbow) = 0.2
p(letter) = 0.7

Interpretation: a letter is **7 times more likely** than a heart!

Solution: make the numbers bigger

- Intuition: we care about how big probabilities are relative to the other probabilities in our distribution, not the actual value.

<u>Probabilities:</u>
p(heart) = ~~0.1~~ 100
p(rainbow) = ~~0.2~~ 200
p(letter) = ~~0.7~~ 700

What if we just multiply all our probs by 100?

This preserves the ratio.

Solution: make the numbers bigger

- What if we just multiply all our probs by 100? This preserves the ratio.

Probabilities:
p(heart) = ~~0.1~~ 100
p(rainbow) = ~~0.2~~ 200
p(letter) = ~~0.7~~ 700

However, if we want to recover the probabilities later, we'll need to **renormalize** them. This means **remembering that we multiplied by 100**.

Solution: log-transform the numbers

- Instead, we use a log transformation. This changes the range from [0,1] to [-∞, 0].

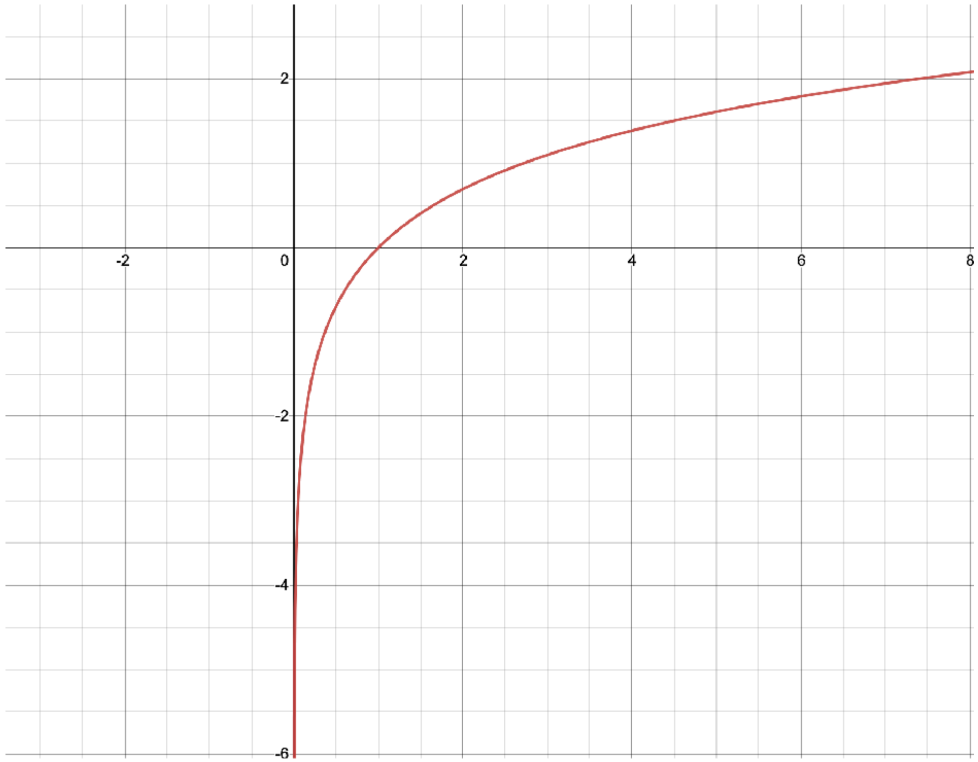
Log base doesn't matter much but we usually use natural log (base e):

<u>Probabilities:</u>

p(heart) = ~~0.1~~ -2.3

p(rainbow) = ~~0.2~~ -1.6

p(letter) = ~~0.7~~ -0.36



www.desmos.com/calculator/aczt76asao