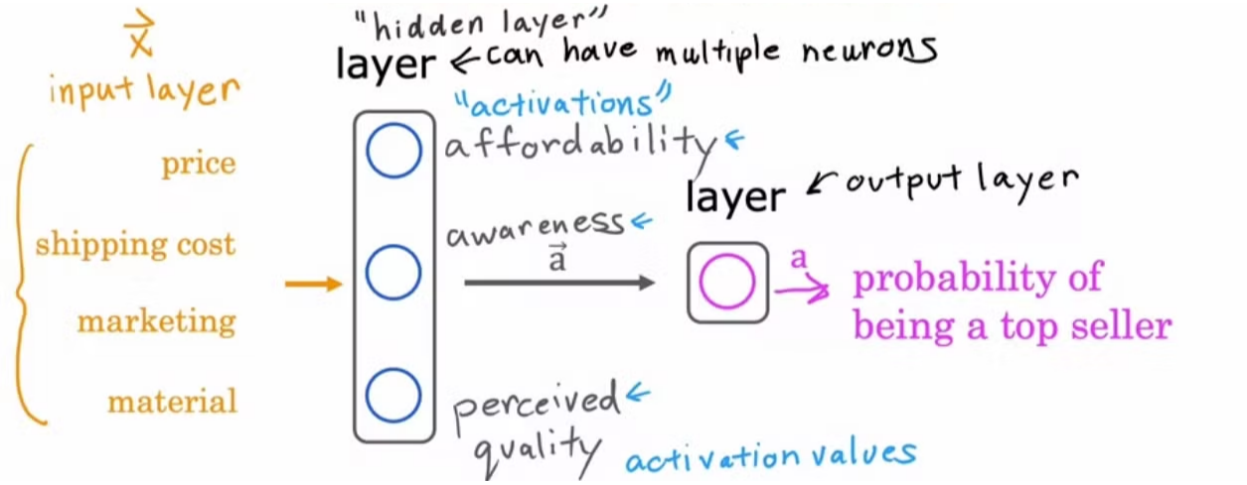
# Advanced Learning Algorithms

Origins: algorithms that try to mimic the brain

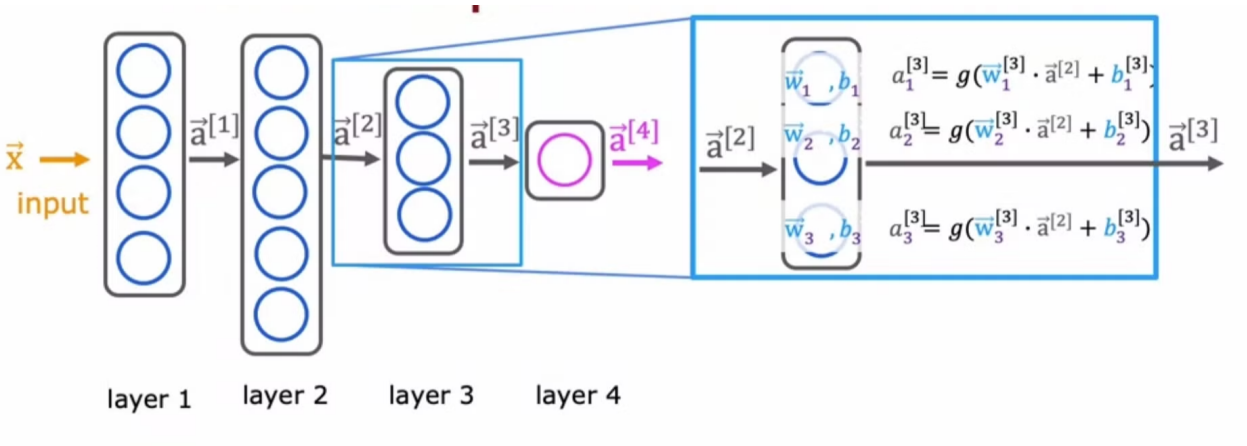
Ressurge from around 2005 under the term *deep learning*, as the amount of data rapidly increased. The rise of GPUs was also a major force.

Applications: speech recognition → computer vision → NLP …

Example: predict if t-shirt will become a top-seller



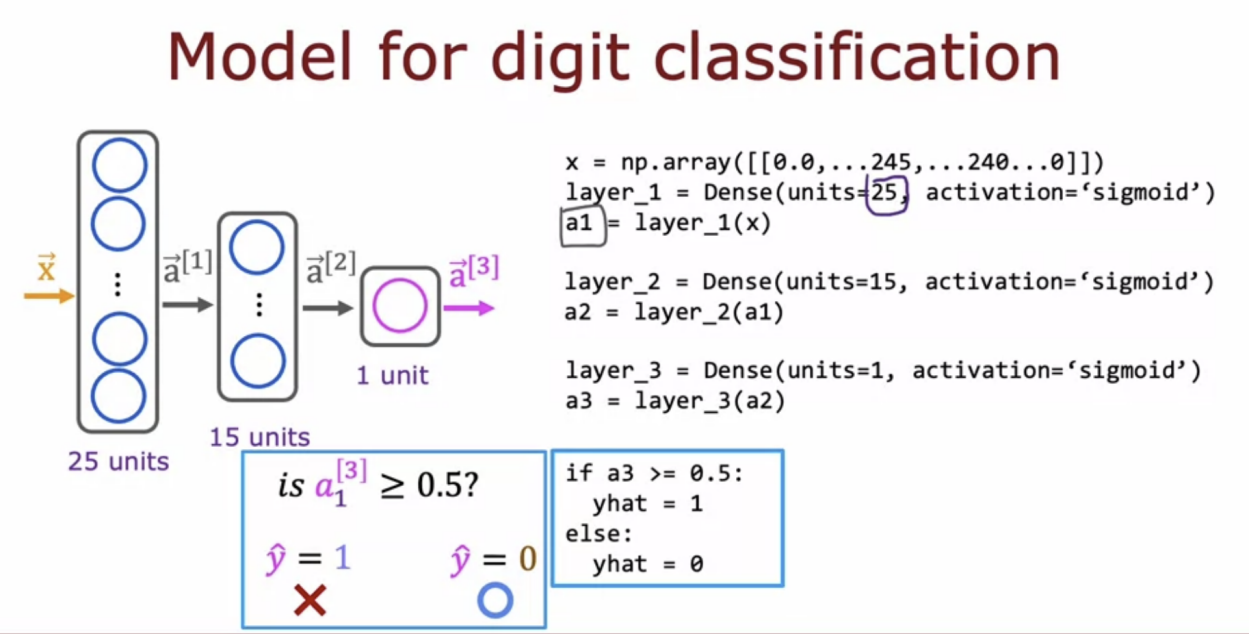

It's like a version of logistic regression that learns its own features layer by layer. That replaces manual feature engineering.

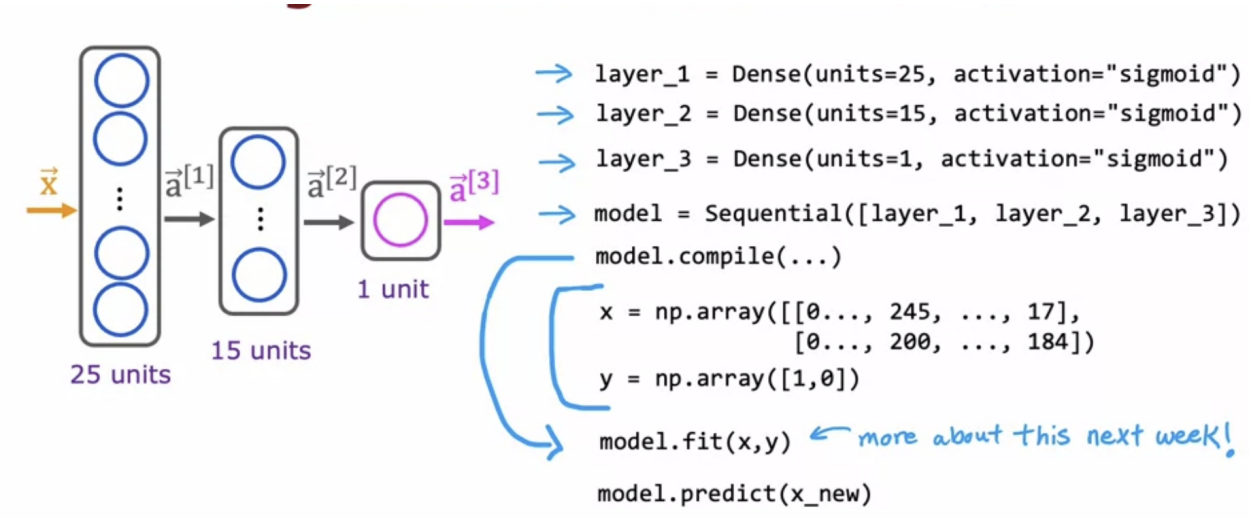


The activation function g could be the sigmoid function or other functions.

**Forward propagation** to make a prediction (inference).

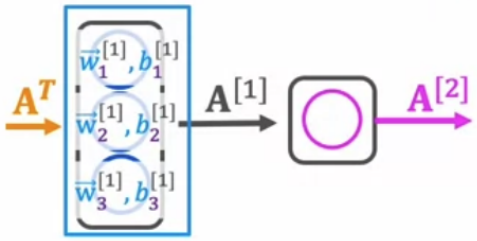
## TensorFlow




Vectorized implementation makes neural networks run much faster!

# Dense layer vectorized




```
A
AT = np.array([[200, 17]])
W = np.array([[1, -3, 5],
              [-2, 4, -6]])
b = np.array([[-1, 1, 2]])
          a_in
def dense(AT,W,b):
  z = np.matmul(AT,W) + b
  a_out = g(z)
  return a_out
```

[[1,0,1]]

$\mathbf{A}^T = [200 \quad 17]$ (1 × 2)

$\mathbf{W} = \begin{bmatrix} 1 & -3 & 5 \\ -2 & 4 & -6 \end{bmatrix}$ (2 × 3)

$\mathbf{B} = [-1 \quad 1 \quad 2]$ (1 × 3)

$\mathbf{Z} = \mathbf{A}^T\mathbf{W} + \mathbf{B}$

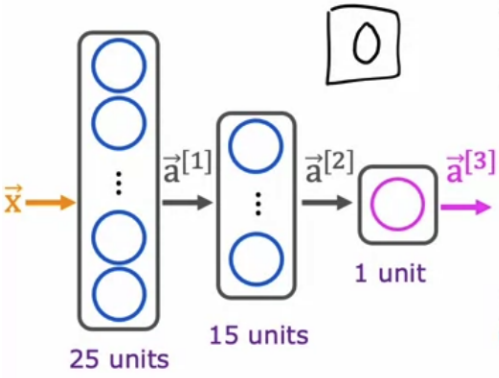
$[165 \quad -531 \quad 900]$

$z_1^{[1]} \quad z_2^{[1]} \quad z_3^{[1]}$

$\mathbf{A} = g(\mathbf{Z})$

$[1 \quad 0 \quad 1]$

# Train a Neural Network in TensorFlow




0

```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
   model = Sequential([
     Dense(units=25, activation='sigmoid'),
     Dense(units=15, activation='sigmoid'),
     Dense(units=1, activation='sigmoid'),
                    ])
from tensorflow.keras.losses import
BinaryCrossentropy
   model.compile(loss=BinaryCrossentropy())
   model.fit(X,Y,epochs=100)
```

① ② ③

epochs: number of steps in gradient descent

Given set of $(x,y)$ examples

How to build and train this in code?

# Model Training Steps

TensorFlow

1. specify how to compute output given input x and parameters w,b (define model)

   $f_{\vec{w},b}(\vec{x})$ =?

2. specify loss and cost

   $L(f_{\vec{w},b}(\vec{x}), y)$ 1 example

   $J(\vec{w}, b) = \frac{1}{m}\sum_{i=1}^{m} L(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)})$

3. Train on data to minimize $J(\vec{w}, b)$

**logistic regression**

```
z = np.dot(w,x)+ b

f_x = 1/(1+np.exp(-z))
```

**logistic loss**

```
loss = -y * np.log(f_x)
   -(1-y) * np.log(1-f_x)
```

```
w = w - alpha * dj_dw
b = b - alpha * dj_db
```

**neural network**

```
model = Sequential([
          Dense(...)
          Dense(...)
          Dense(...)
                    ])
```

**binary cross entropy**

```
model.compile(
loss=BinaryCrossentropy())
```

```
model.fit(X,y,epochs=100)
```

# 2. Loss and cost functions

handwritten digit classification problem — binary classification

$L(f(\vec{x}), y) = -y\log(f(\vec{x})) - (1-y)\log(1-f(\vec{x}))$

Compare prediction vs. target

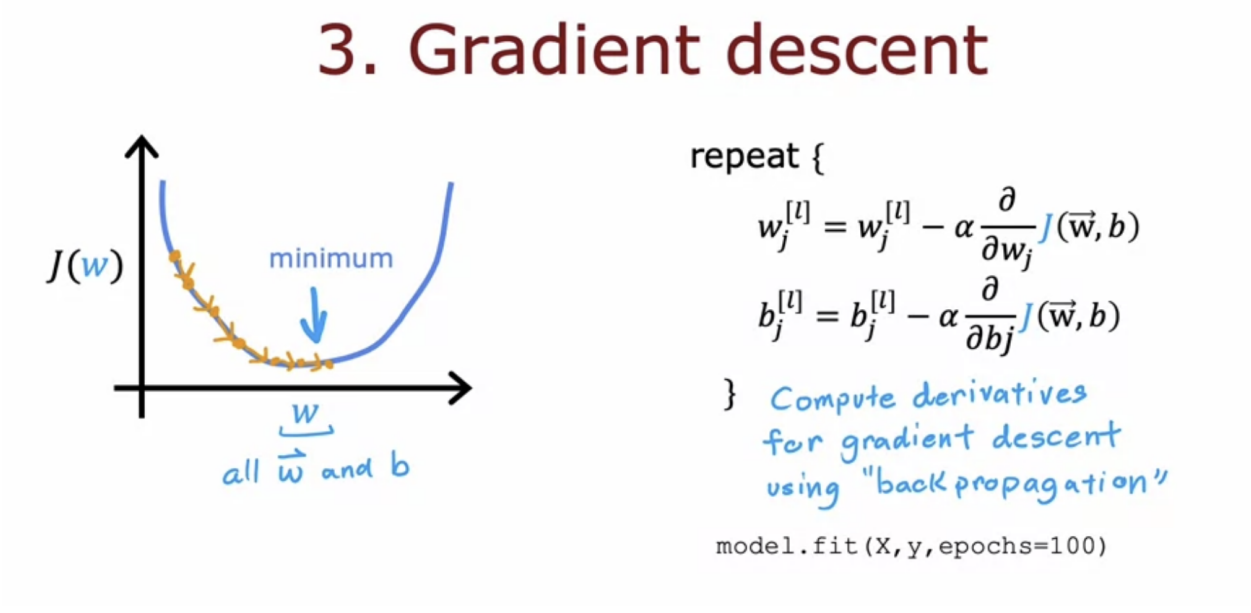
logistic loss

also known as binary cross entropy

```
model.compile(loss= BinaryCrossentropy())
```

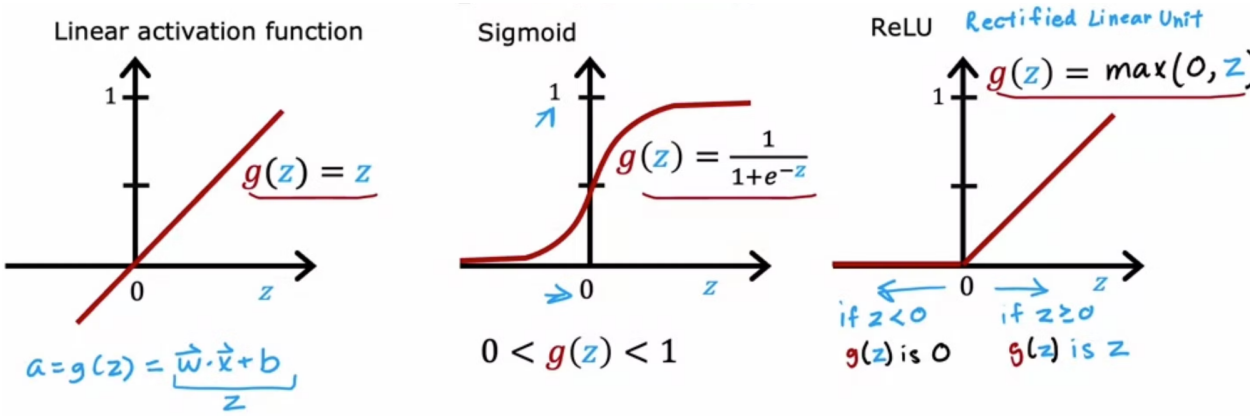
```
from tensorflow.keras.losses import
  BinaryCrossentropy
```

Keras

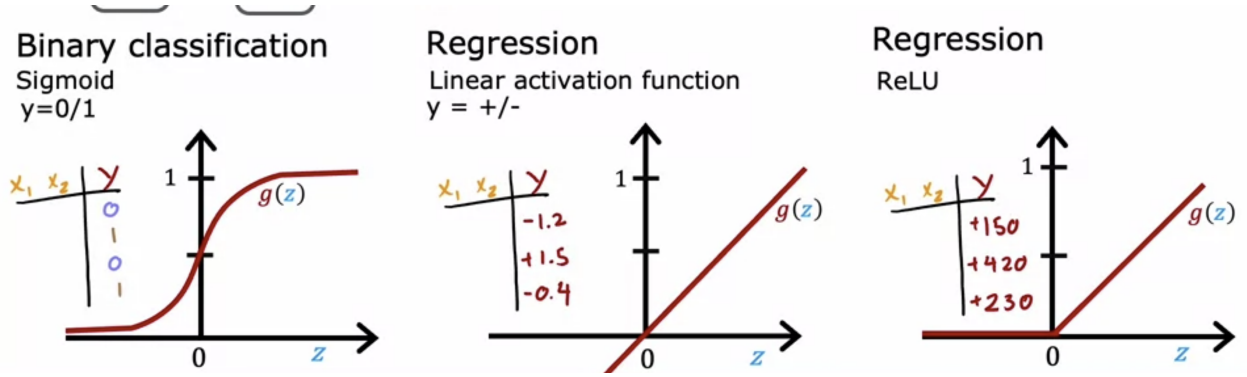



Another popular activation function is **ReLU** (rectified linear unit).

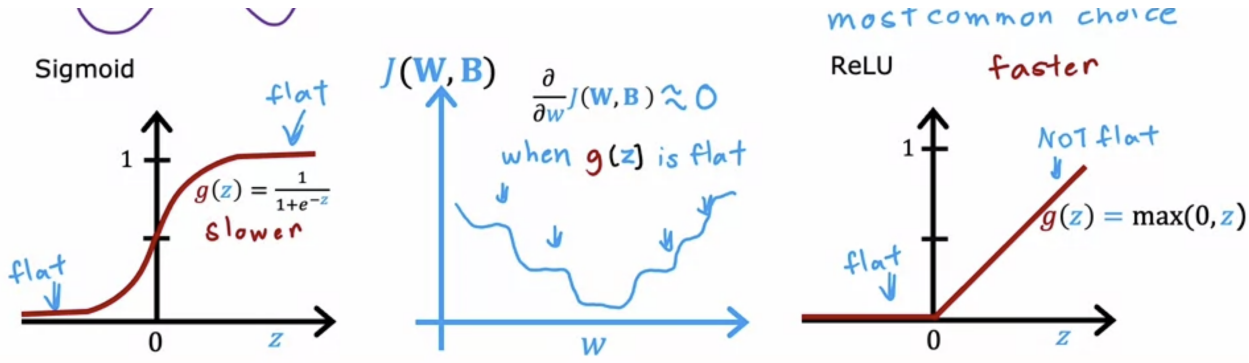
$$g(z) = max(0, z)$$




We choose the activation function for the output layer depending on the nature of the label y we're trying to predict.




For the hidden layers, ReLU is the most commonly used. It's faster to compute and makes gradient descent faster. This is because ReLU is flat to the left only, whereas sigmoid is flat to the left and right.




Why don't we use linear activation function in the entire neural network? Because that would be equivalent to a single linear regression. And if the hidden layers are all linear and the output layer is sigmoid, then the neural network is equivalent to logistic regression.

The "off" or disable feature  of the ReLU activation enables models to stitch together linear segments to model complex non-linear functions.

## Multiclass classification

## Softmax regression (4 possible outputs) $y = 1, 2, 3, 4$

$z_1 = \vec{w}_1 \cdot \vec{x} + b_1$

$$a_1 = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}} = P(y = 1|\vec{x})$$

$z_2 = \vec{w}_2 \cdot \vec{x} + b_2$

$$a_2 = \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}} = P(y = 2|\vec{x})$$

$z_3 = \vec{w}_3 \cdot \vec{x} + b_3$

$$a_3 = \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}} = P(y = 3|\vec{x})$$

$z_4 = \vec{w}_4 \cdot \vec{x} + b_4$

$$a_4 = \frac{e^{z_4}}{e^{z_1} + e^{z_2} + e^{z_3} + e^{z_4}} = P(y = 4|\vec{x})$$

---

## Softmax regression (N possible outputs) $y = 1, 2, 3, \ldots, N$

$$z_j = \vec{w}_j \cdot \vec{x} + b_j \quad j = 1, \ldots, N$$

parameters $w_1, w_2, \ldots, w_N$ $b_1, b_2, \ldots, b_N$

$$a_j = \frac{e^{z_j}}{\sum_{k=1}^{N} e^{z_k}} = P(y = j|\vec{x})$$

# Cost

## Logistic regression

$$z = \vec{w} \cdot \vec{x} + b$$

$$a_1 = g(z) = \frac{1}{1+e^{-z}} = P(y=1|\vec{x})$$

$$a_2 = 1 - a_1 = P(y=0|\vec{x})$$

$$loss = -y \log a_1 - (1-y) \log(1-a_1)$$

if $y=1$ — if $y=0$

$$J(\vec{w}, b) = \text{average loss}$$

## Softmax regression

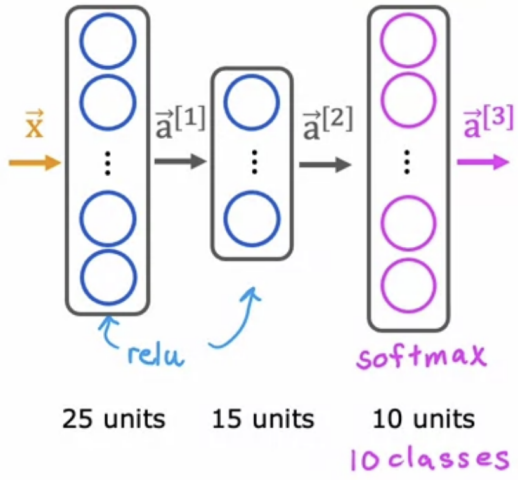
$$a_1 = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + \cdots + e^{z_N}} = P(y=1|\vec{x})$$

$$\vdots$$

$$a_N = \frac{e^{z_N}}{e^{z_1} + e^{z_2} + \cdots + e^{z_N}} = P(y=N|\vec{x})$$

$$loss(a_1, \ldots, a_N, y) = \begin{cases} -\log a_1 & \text{if } y=1 \\ -\log a_2 & \text{if } y=2 \\ \vdots & \\ -\log a_N & \text{if } y=N \end{cases}$$

# Neural Network with Softmax output




$$z_1^{[3]} = \vec{w}_1^{[3]} \cdot \vec{a}^{[2]} + b_1^{[3]} \qquad a_1^{[3]} = \frac{e^{z_1^{[3]}}}{e^{z_1^{[3]}} + \cdots + e^{z_{10}^{[3]}}} = P(y=1|\vec{x})$$

$$\vdots$$

$$z_{10}^{[3]} = \vec{w}_{10}^{[3]} \cdot \vec{a}^{[2]} + b_{10}^{[3]} \qquad a_{10}^{[3]} = \frac{e^{z_{10}^{[3]}}}{e^{z_1^{[3]}} + \cdots + e^{z_{10}^{[3]}}} = P(y=10|\vec{x})$$

# MNIST with softmax

1. specify the model

$f_{\vec{w},b}(\vec{x})$ =?

```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
model = Sequential([
  Dense(units=25, activation='relu'),
  Dense(units=15, activation='relu'),
  Dense(units=10, activation='softmax')
                  ])
```

2. specify loss and cost

$L(f_{\vec{w},b}(\vec{x}), y)$

```
from tensorflow.keras.losses import
  SparseCategoricalCrossentropy
model.compile(loss= SparseCategoricalCrossentropy() )
```

3. Train on data to minimize $J(\vec{w}, b)$

```
model.fit(X,Y,epochs=100)
```

# Numerical Roundoff Errors

More numerically accurate implementation of logistic loss:

Logistic regression:

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

Original loss

$$loss = -y\log(a) - (1-y)\log(1-a)$$

More accurate loss (in code)

$$loss = -y\log\left(\frac{1}{1+e^{-z}}\right) - (1-y)\log\left(1-\frac{1}{1+e^{-z}}\right)$$

```
model = Sequential([
  Dense(units=25, activation='relu'),
  Dense(units=15, activation='relu'),
  Dense(units=1, activation='sigmoid')
                ])
model.compile(loss=BinaryCrossEntropy() )
```

```
model.compile(loss=BinaryCrossEntropy(from_logits=True) )
```

## More numerically accurate implementation of softmax

Softmax regression

$$(a_1, \ldots, a_{10}) = g(z_1, \ldots, z_{10})$$

$$\text{Loss} = L(\vec{a}, y) = \begin{cases} -\log a_1 & \text{if } y = 1 \\ \vdots & \\ -\log a_{10} & \text{if } y = 10 \end{cases}$$

```
model = Sequential([
  Dense(units=25, activation='relu'),
  Dense(units=15, activation='relu'),
  Dense(units=10, activation='softmax')
                  ])
```

'linear'

~~model.compile(loss=SparseCategoricalCrossEntropy() )~~

More Accurate

$$L(\vec{a}, y) = \begin{cases} -\log \dfrac{e^{z_1}}{e^{z_1} + \cdots + e^{z_{10}}} & \text{if } y = 1 \\ \vdots & \\ -\log \dfrac{e^{z_{10}}}{e^{z_1} + \cdots + e^{z_{10}}} & \text{if } y = 10 \end{cases}$$

```
model.compile(loss=SparseCategoricalCrossEntropy(from_logits=True))
```

## MNIST (more numerically accurate)

model

```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
model = Sequential([
  Dense(units=25, activation='relu'),
  Dense(units=15, activation='relu'),
  Dense(units=10, activation='linear') ])
```

loss

```
from tensorflow.keras.losses import
  SparseCategoricalCrossentropy
model.compile(...,loss=SparseCategoricalCrossentropy(from_logits=True) )
```

fit

```
model.fit(X,Y,epochs=100)
```

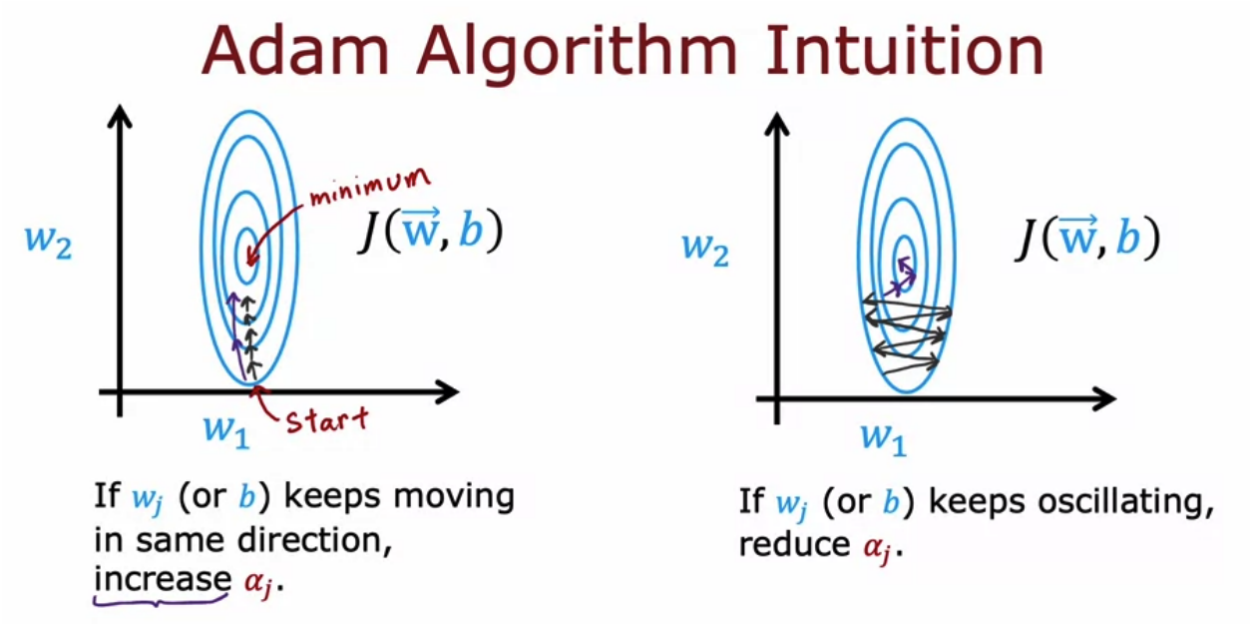
predict

```
logits = model(X)
f_x = tf.nn.softmax(logits)
```

Multi-label classification: single input maps to potentially more than one label. One can build multiple NNs to detect each label or have an output layer with multiple nodes, one for each label, using a sigmoid activation function.

**Adam algorithm** is an advance of gradient descent, increasing the learning rate if the steps are going in the same direction or decreasing the learning rate if the

steps are going all over the place. It uses different learning rates for each parameter of the model. It's become the standard method.




## MNIST Adam

model

```
model = Sequential([
        tf.keras.layers.Dense(units=25, activation='sigmoid'),
        tf.keras.layers.Dense(units=15, activation='sigmoid'),
        tf.keras.layers.Dense(units=10, activation='linear')
])
```

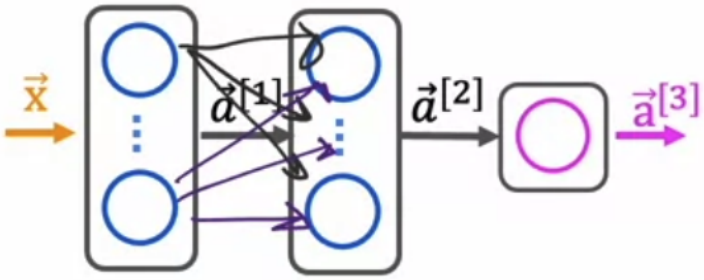
compile

```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True))
```

fit

```
model.fit(X,Y,epochs=100)
```

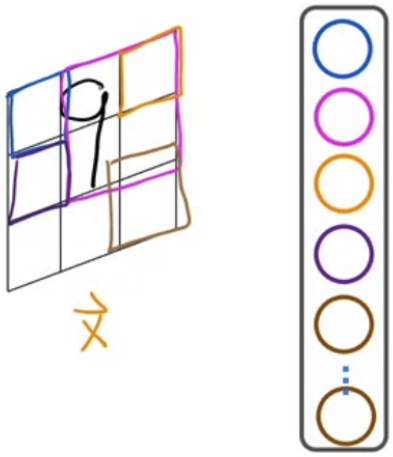
# Dense Layer




Each neuron output is a function of all the activation outputs of the previous layer.

$$\vec{a}_1^{[2]} = g\left(\vec{w}_1^{[2]} \cdot \vec{a}^{[1]} + b_1^{[2]}\right)$$

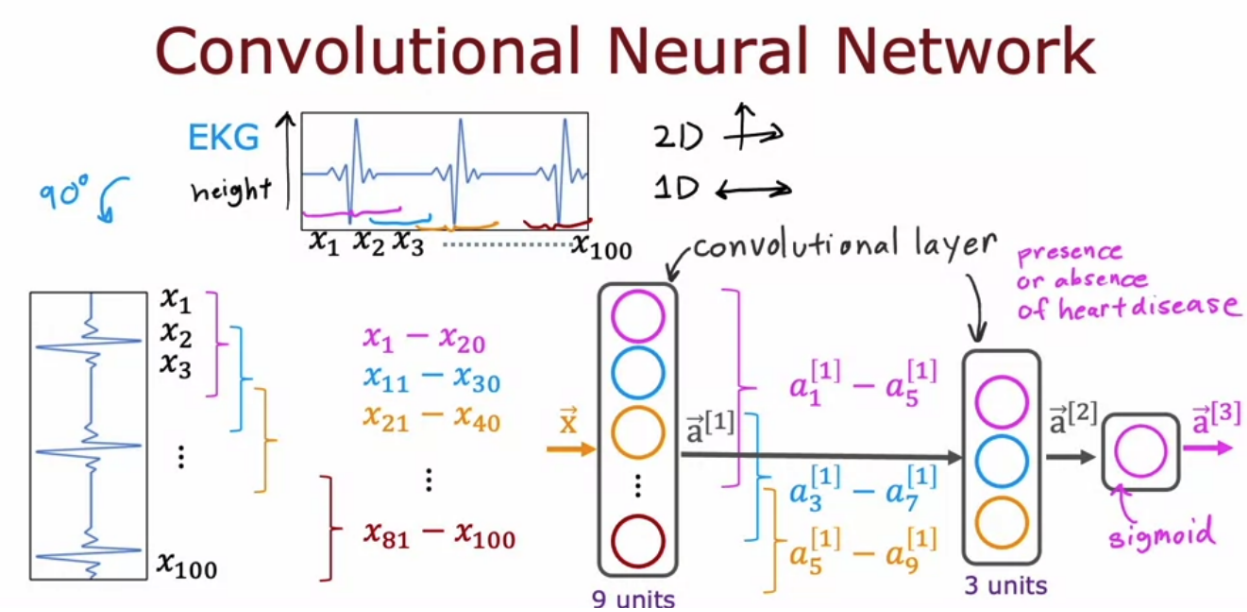
# Convolutional Layer




Each neuron only looks at part of the previous layer's outputs.

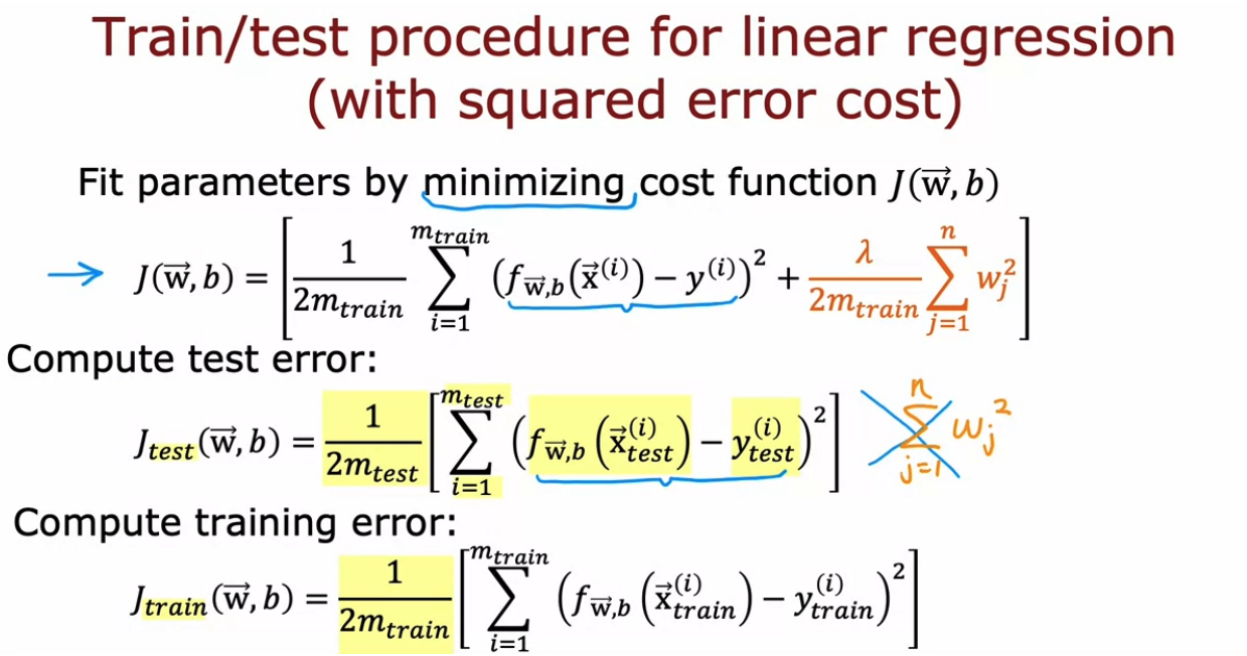
Why?

- Faster computation
- Need less training data (less prone to overfitting)




## Evaluating a Model

Use test set to see how well the model generalizes to unseen data. Training error being small does not guarantee a good model.



# Train/test procedure for classification problem

0/1

Fit parameters by minimizing $J(\vec{w}, b)$ to find $\vec{w}, b$

E.g.,

$$J(\vec{w}, b) = -\frac{1}{m_{train}} \sum_{i=1}^{m_{train}} \left[ y^{(i)} \log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) + (1 - y^{(i)}) \log\left(1 - f_{\vec{w},b}(\vec{x}^{(i)})\right) \right] + \frac{\lambda}{2m_{train}} \sum_{j=1}^{n} w_j^2$$

Compute test error:

$$J_{test}(\vec{w}, b) = -\frac{1}{m_{test}} \sum_{i=1}^{m_{test}} \left[ y_{test}^{(i)} \log\left(f_{\vec{w},b}\left(\vec{x}_{test}^{(i)}\right)\right) + \left(1 - y_{test}^{(i)}\right) \log\left(1 - f_{\vec{w},b}\left(\vec{x}_{test}^{(i)}\right)\right) \right]$$

Compute train error:

$$J_{train}(\vec{w}, b) = -\frac{1}{m_{train}} \sum_{i=1}^{m_{train}} \left[ y_{train}^{(i)} \log\left(f_{\vec{w},b}\left(\vec{x}_{train}^{(i)}\right)\right) + \left(1 - y_{train}^{(i)}\right) \log\left(1 - f_{\vec{w},b}\left(\vec{x}_{train}^{(i)}\right)\right) \right]$$

# Model selection (choosing a model)

$d=1$ 1. $f_{\vec{w},b}(\vec{x}) = w_1 x + b$ → $w^{<1>}, b^{<1>} \rightarrow J_{test}(w^{<1>}, b^{<1>})$

$d=2$ 2. $f_{\vec{w},b}(\vec{x}) = w_1 x + w_2 x^2 + b$ → $w^{<2>}, b^{<2>} \rightarrow J_{test(w^{<2>}, b^{<2>})}$

$d=3$ 3. $f_{\vec{w},b}(\vec{x}) = w_1 x + w_2 x^2 + w_3 x^3 + b$ → $w^{<3>}, b^{<3>} \rightarrow J_{test(w^{<3>}, b^{<3>})}$

⋮

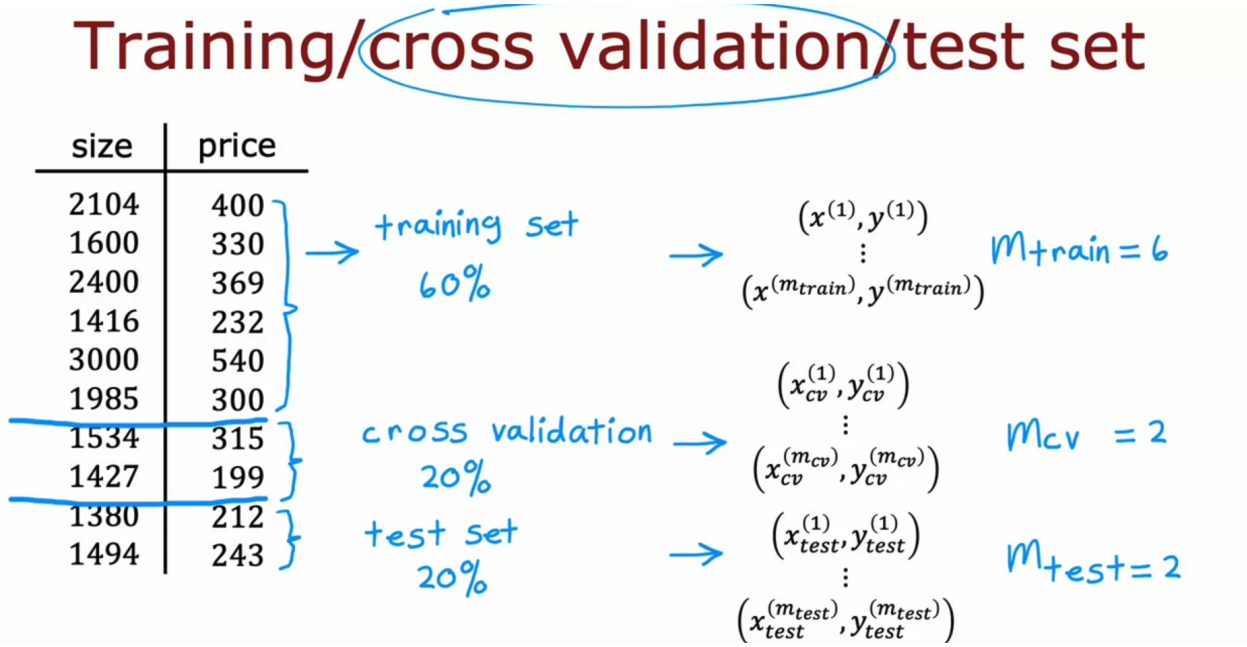
$d=10$ 10. $f_{\vec{w},b}(\vec{x}) = w_1 x + w_2 x^2 + \cdots + w_{10} x^{10} + b$ → $J_{test(w^{<10>}, b^{<10>})}$

Choose $w_1 x + \cdots + w_5 x^5 + b$ $d=5$ $J_{test}(w^{<5>}, b^{<5>})$

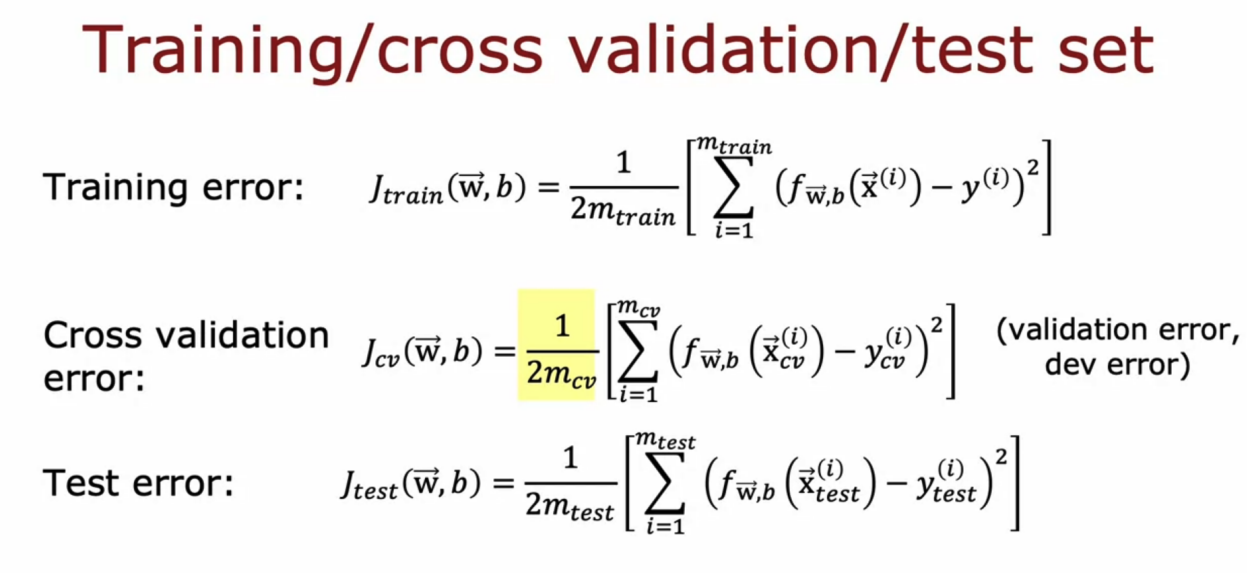
How well does the model perform? Report test set error $J_{test}(w^{<5>}, b^{<5>})$?

The problem: $J_{test}(w^{<5>}, b^{<5>})$ is likely to be an optimistic estimate of generalization error (ie. $J_{test}(w^{<5>}, b^{<5>}) <$ generalization error ). Because an extra parameter d (degree of polynomial) was chosen using the test set.

$w, b$ are overly optimistic estimate of generalization error on training data.

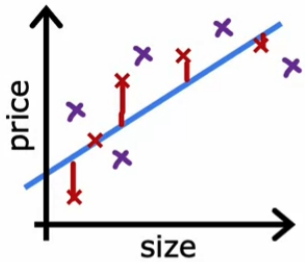



Cross validation is now used to test the validity of the different candidate models.




We pick the model with lowest cross validation error and estimate the generalization error using the test set! This procedure can also be applied to choose between different neural network architectures.

# Bias/variance

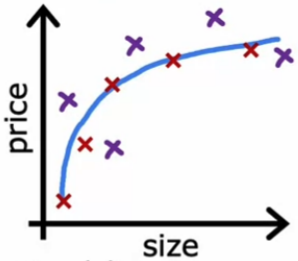



$f_{\vec{w},b}(x) = w_1 x + b$

High bias (underfit)

$J_{train}$ is high

$J_{cv}$ is high

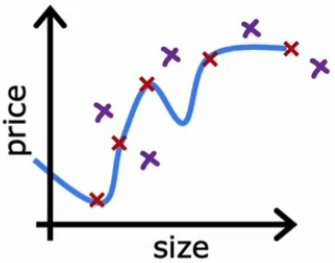



$f_{\vec{w},b}(x) = w_1 x + w_2 x^2 + b$

"Just right"

$J_{train}$ is low

$J_{cv}$ is low




$f_{\vec{w},b}(x) = w_1 x + w_2 x^2 + w_3 x^3 + w_4 x^4 + b$

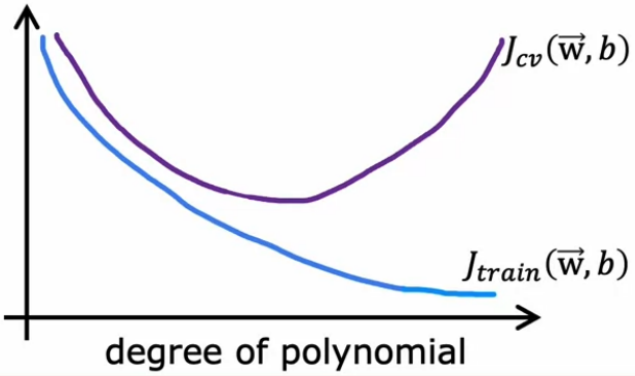
High variance (overfit)

$J_{train}$ is low

$J_{cv}$ is high

# Diagnosing bias and variance

How do you tell if your algorithm has a bias or variance problem?




High bias (underfit)

$J_{train}$ will be high

($J_{train} \approx J_{cv}$)

High variance (overfit)

$J_{cv} \gg J_{train}$

($J_{train}$ may be low)

High bias and high variance

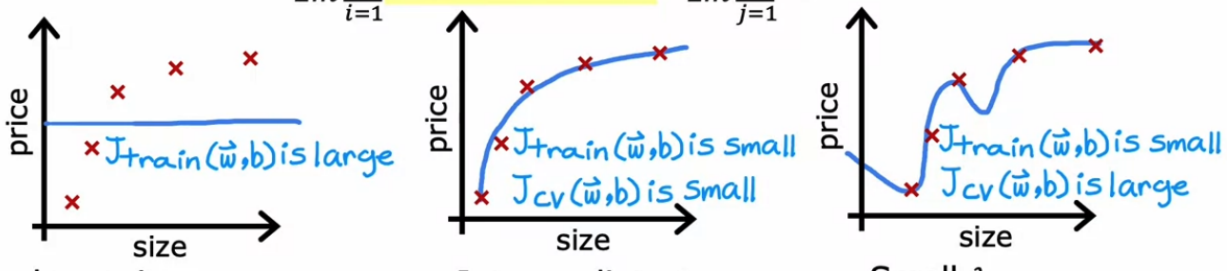
$J_{train}$ will be high

and $J_{cv} \gg J_{train}$

# Linear regression with regularization

Model: $f_{\vec{w},b}(x) = w_1x + w_2x^2 + w_3x^3 + w_4x^4 + b$

$$J(\vec{w}, b) = \frac{1}{2m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)^2 + \frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2$$



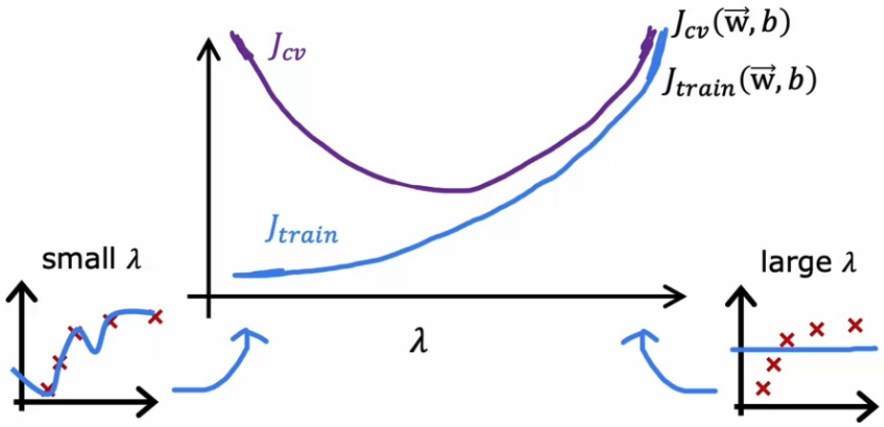

Large $\lambda$
High bias (underfit)
$\lambda = 10{,}000$ $w_1 \approx 0, w_2 \approx 0$
$f_{\vec{w},b}(\vec{x}) \approx b$

Intermediate $\lambda$

Small $\lambda$
High variance (overfit)
$\lambda = 0$

# Bias and variance as a function of regularization parameter $\lambda$

$$J(\vec{w}, b) = \frac{1}{2m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)^2 + \frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2$$



## Establishing a baseline level of performance

What is the level of error you can reasonably hope to get to?

- Human level performance
- Competing algorithms performance
- Guess based on experience

## Bias/variance examples

Baseline performance : 10.6%
Training error ($J_{train}$) : 10.8%
Cross validation error ($J_{cv}$): 14.8%

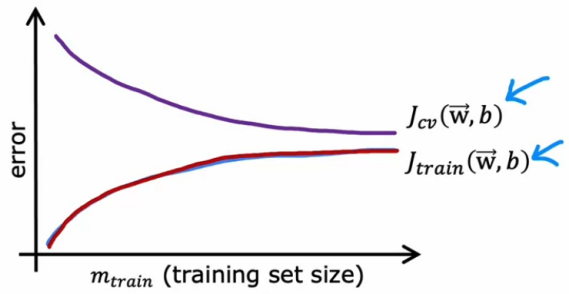
0.2%
4.0%

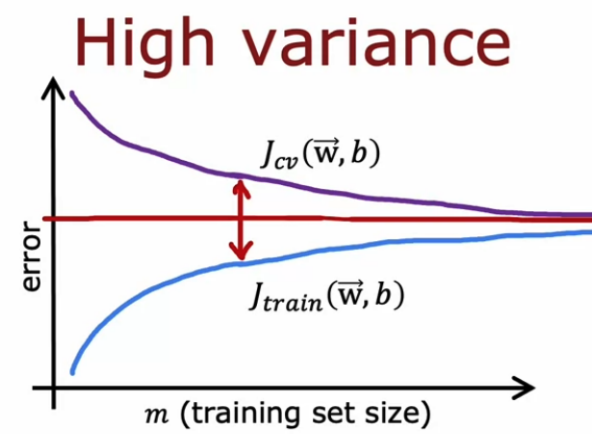
10.6%
4.4%
15.0%
0.5%
15.5%

high variance
high bias

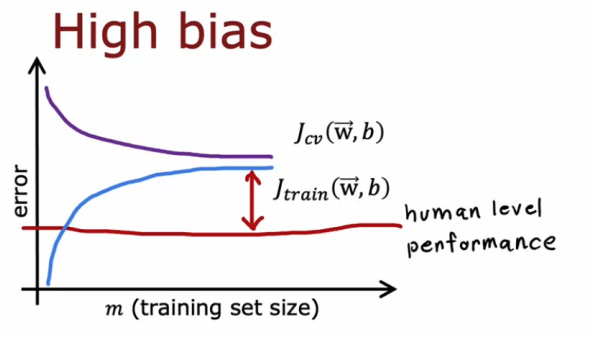
## Learning curves

$J_{train}$= training error
$J_{cv}$= cross validation error






With high variance algorithm, adding more training data will likely help the performance.




With high bias algorithm, adding more training data will not help the performance much.

# Debugging a learning algorithm

You've implemented regularized linear regression on housing prices

$$J(\vec{w}, b) = \frac{1}{2m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)^2 + \frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2$$

But it makes unacceptably large errors in predictions. What do you try next?

- → Get more training examples — fixes high variance
- → Try smaller sets of features $x, x^2, \ldots$ — fixes high variance
- → Try getting additional features — fixes high bias
- → Try adding polynomial features $(x_1^2, x_2^2, x_1x_2, etc)$ — fixes high bias
- → Try decreasing $\lambda$ — fixes high bias
- → Try increasing $\lambda$ — fixes high variance

# The bias variance tradeoff

$f_{\vec{w},b}(x) = w_1 x + b$

Simple model

High bias

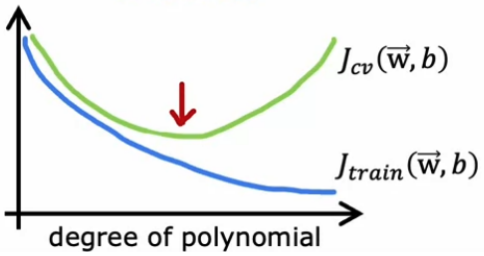
$f_{\vec{w},b}(x) = w_1 x + w_2 x^2 + b$

$f_{\vec{w},b}(x) = w_1 x + w_2 x^2 + w_3 x^3 + w_4 x^4 + b$

Complex model

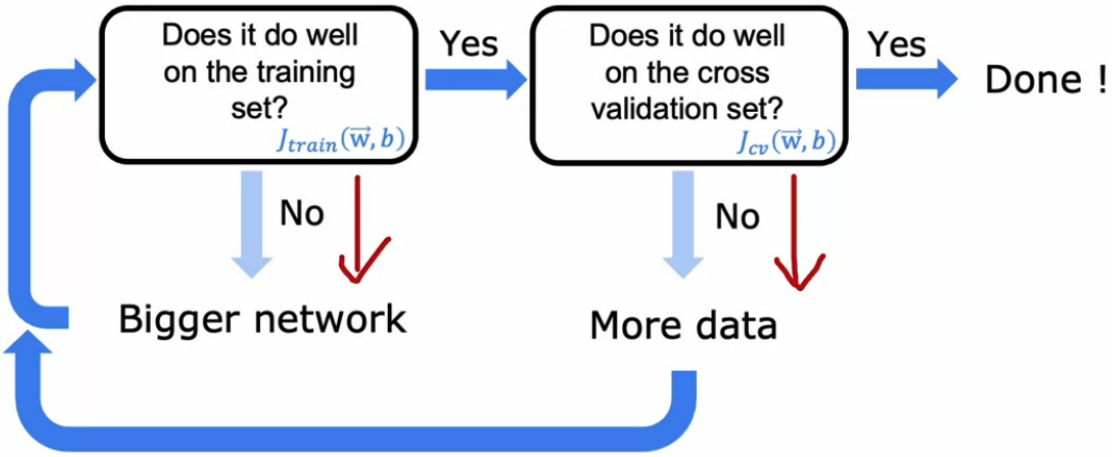
High variance

tradeoff

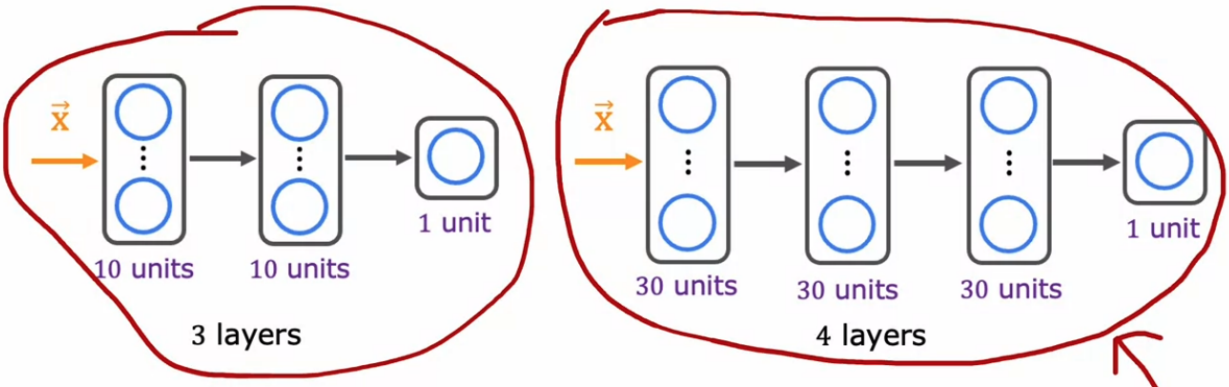



# Neural networks and bias variance

Large neural networks are low bias machines



# Neural networks and regularization




A large neural network will usually do as well or better than a smaller one so long as regularization is chosen appropriately.

# Neural network regularization

$$J(\mathbf{W},\mathbf{B}) = \frac{1}{m}\sum_{i=1}^{m} L\left(f\left(\vec{\mathbf{x}}^{(i)}\right), y^{(i)}\right) + \frac{\lambda}{2m}\sum_{\text{all weights } \mathbf{W}} \left(w^2\right)$$

b

**Unregularized MNIST model**

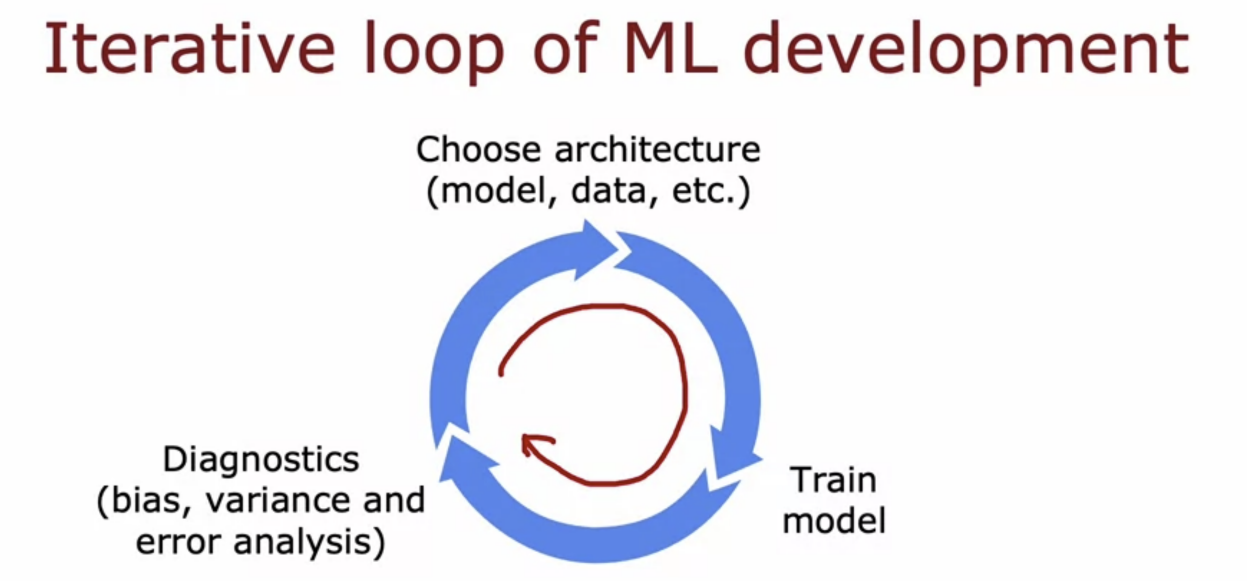
```
layer_1 = Dense(units=25, activation="relu")
layer_2 = Dense(units=15, activation="relu")
layer_3 = Dense(units=1, activation="sigmoid")
model = Sequential([layer_1, layer_2, layer_3])
```

λ

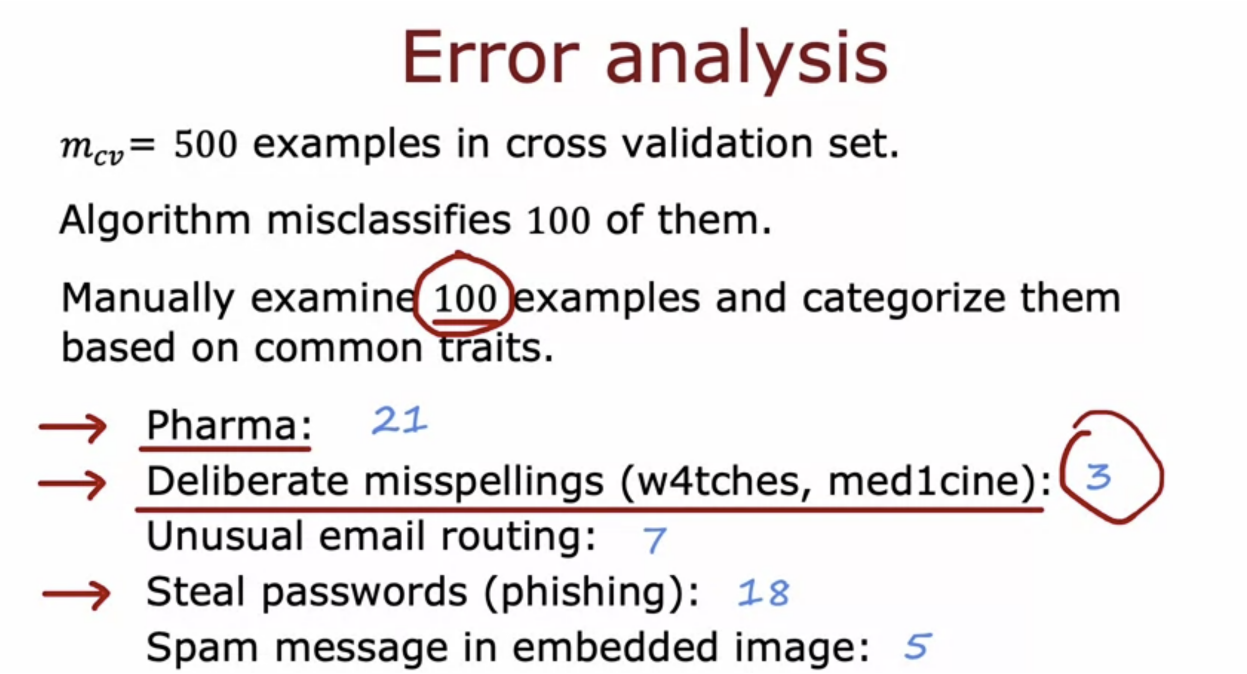
**Regularized MNIST model**

```
layer_1 = Dense(units=25, activation="relu", kernel_regularizer=L2(0.01))
layer_2 = Dense(units=15, activation="relu", kernel_regularizer=L2(0.01))
layer_3 = Dense(units=1, activation="sigmoid", kernel_regularizer=L2(0.01))
model = Sequential([layer_1, layer_2, layer_3])
```

It hardly ever hurts to have a larger NN, so long as you regularize it properly.

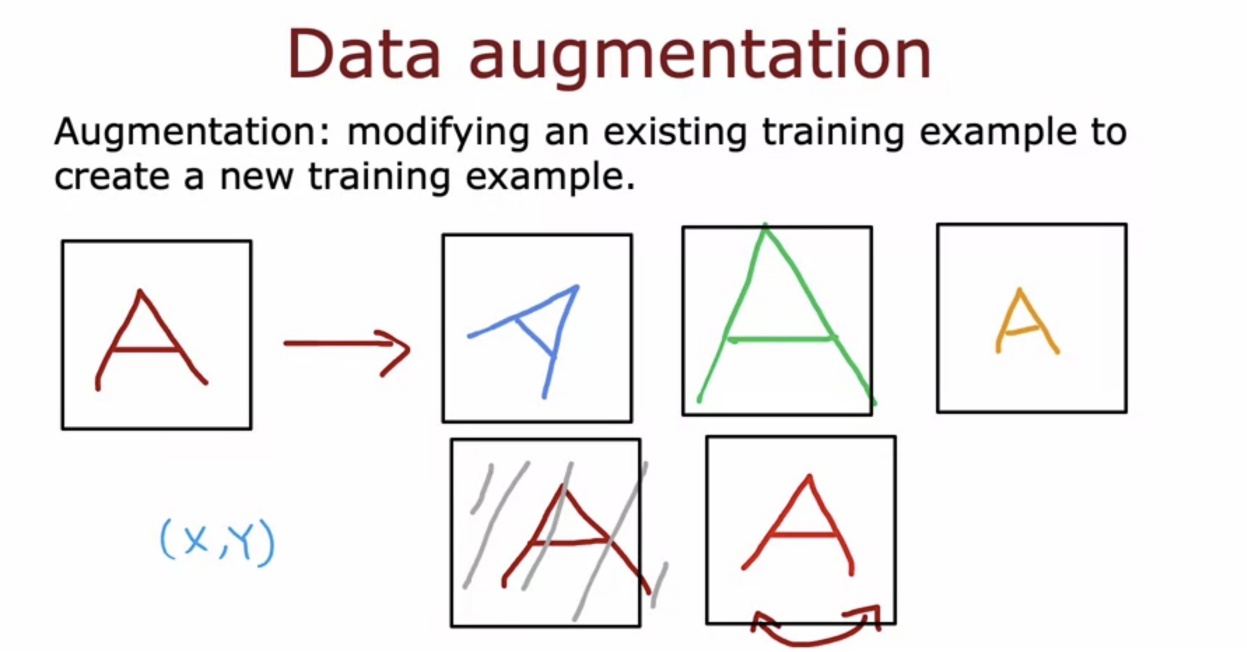


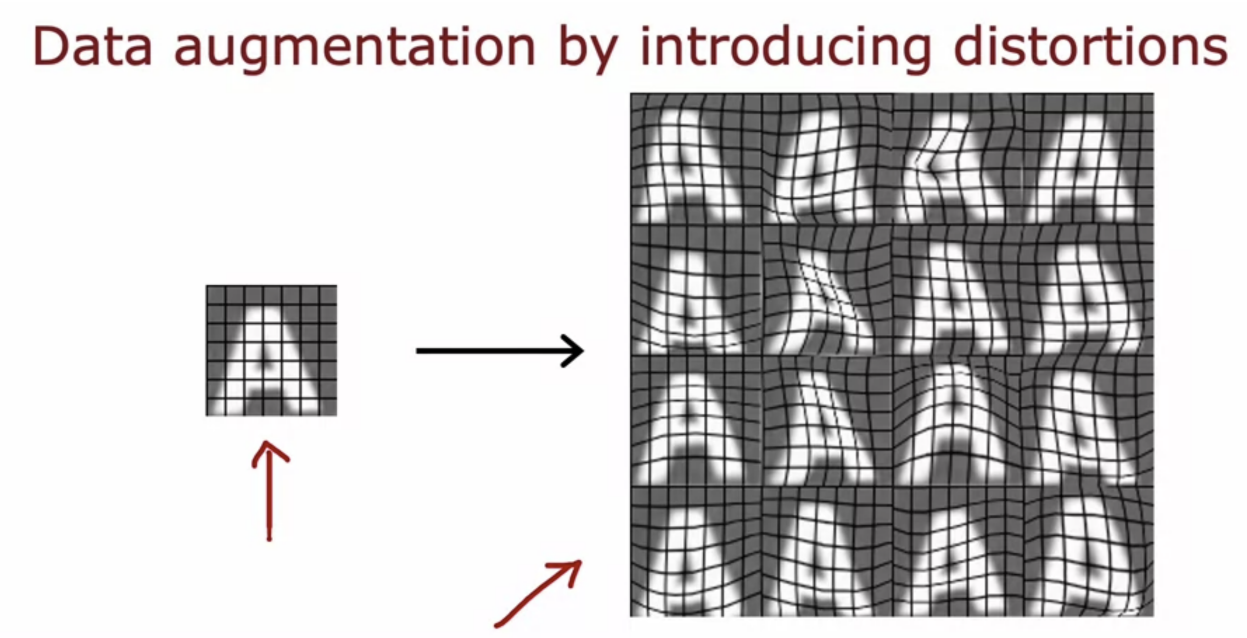

Avoid spending time in issues that are not very common by manually inspecting counts of categories of issues. We might then spend time collecting more data or building features related to a specific type of issue rather than another.




We can also artificially increase the size of the training data, without getting brand new examples, using data augmentation. These distortions should still be

representative of what you'd get in the test set, or they wouldn't aid in performance.

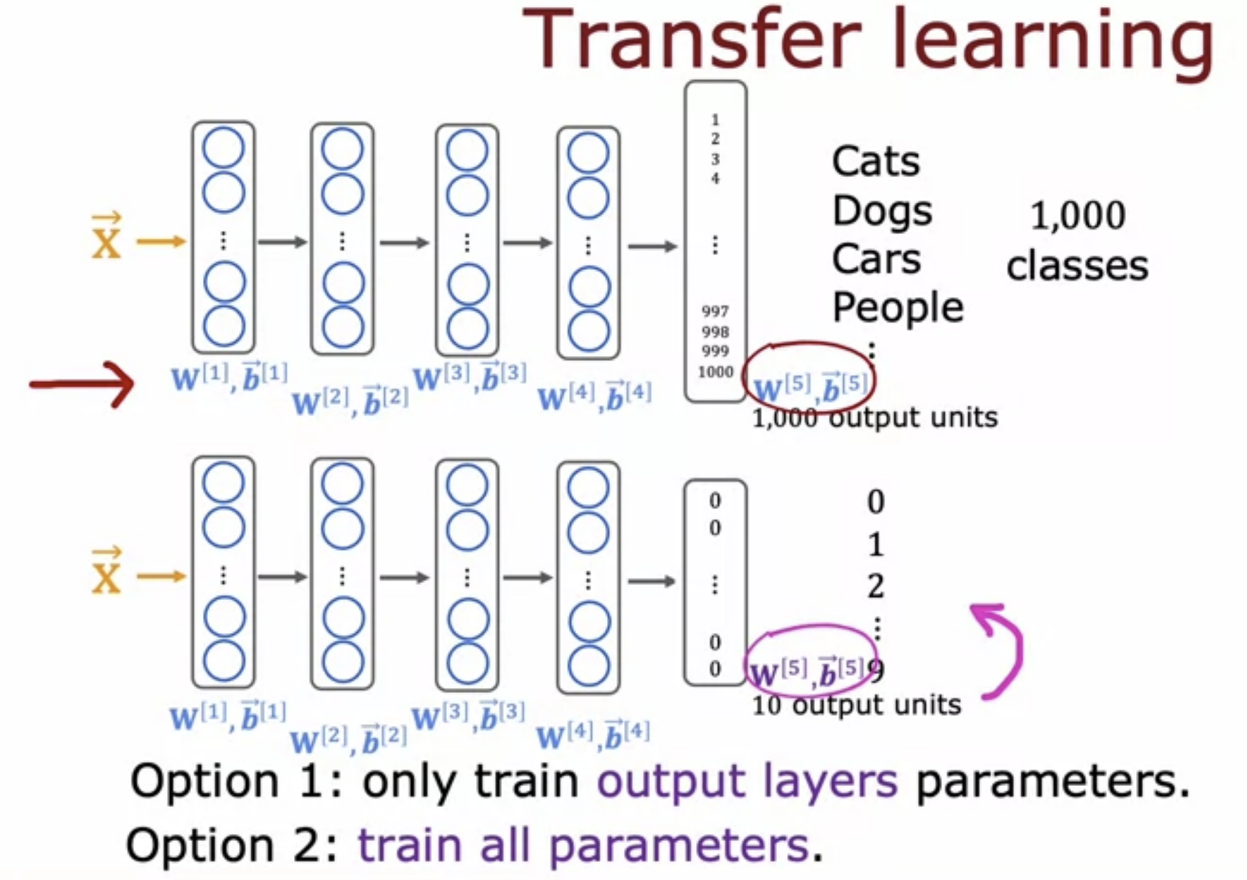



For speech data, we can add background noise to the original audio, for example.

We can also use data synthesis, for example, for computer vision tasks.

When we don’t have much data for a particular problem, we can apply **transfer learning**. The idea is to use data from another similar problem (same input type), train a NN and then re-use the model, adjusting its output layer. We can then use the weights from the first NN and train the new NN to get the weights of the output layer. Or we can retrain the whole NN, using the original NN parameters as initial parameters. e.g. from classifying digits 0-9 to classifying animals. We call these steps: **supervised pre-training and fine-tuning**. We can also use NNs that are available online and fine-tune for our problem!




It’s found that the fine-tuning requires a lot less examples if the pre-training was done with a large dataset.

# Transfer learning summary

1. Download neural network parameters pretrained on a large dataset with same input type (e.g., images, audio, text) as your application (or train your own).

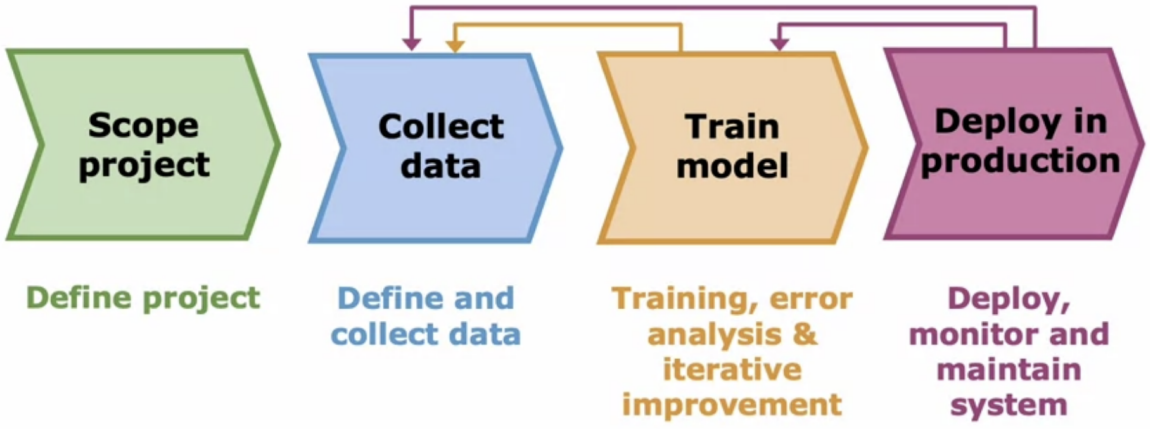
1 million images

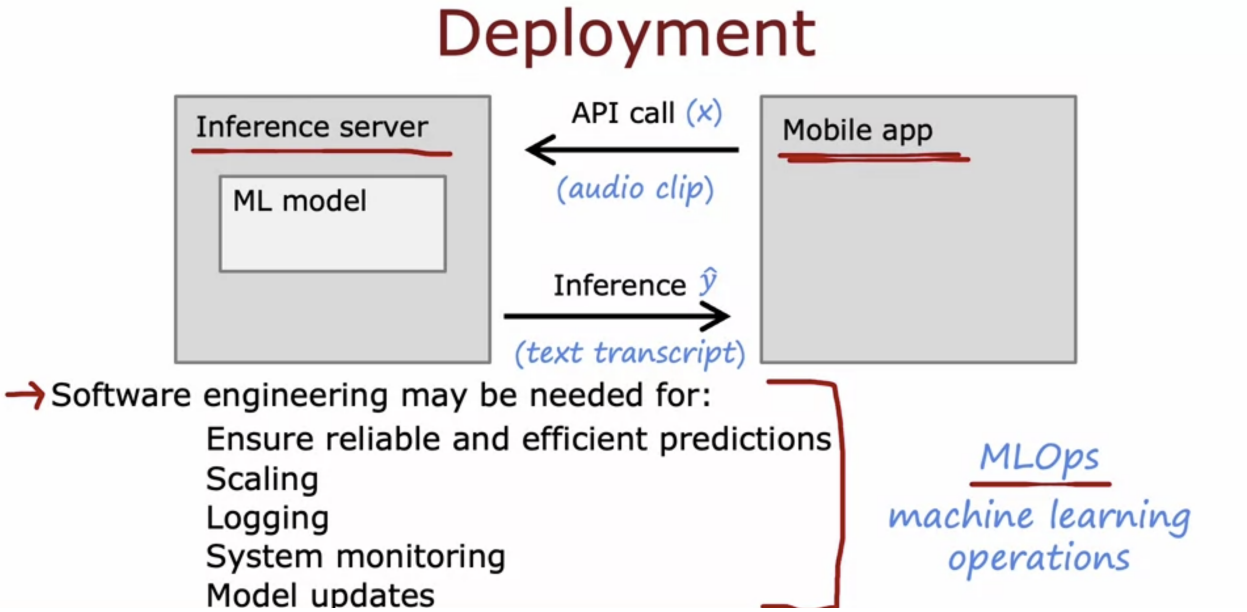
2. Further train (fine tune) the network on your own data.

10 00 images

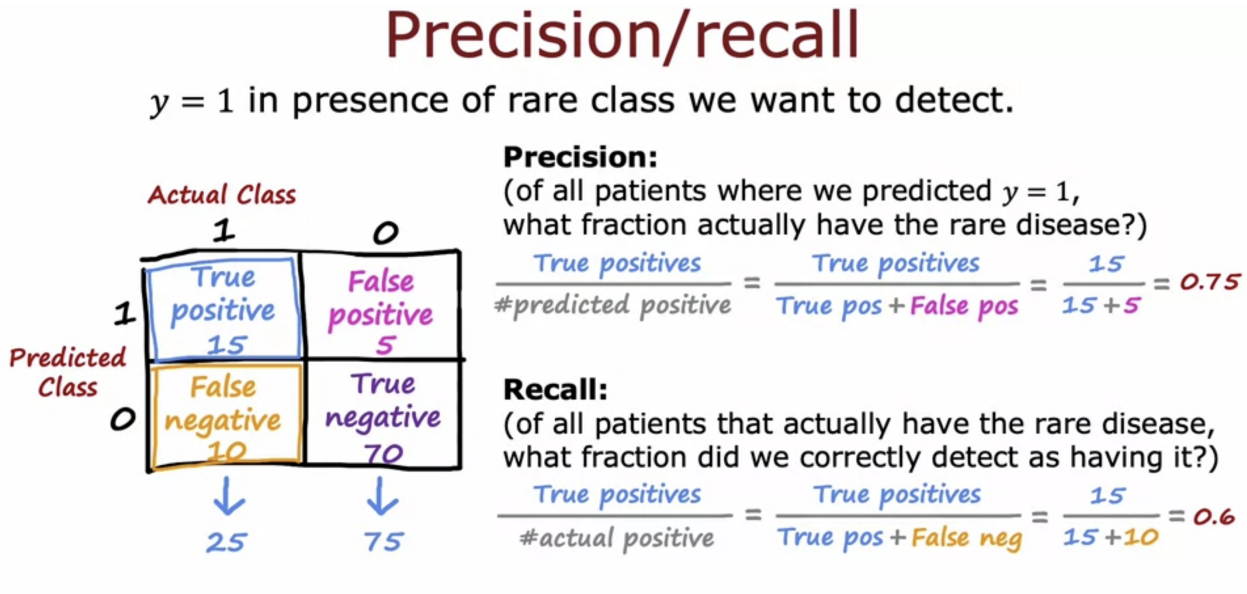
50 images

# Full cycle of a machine learning project

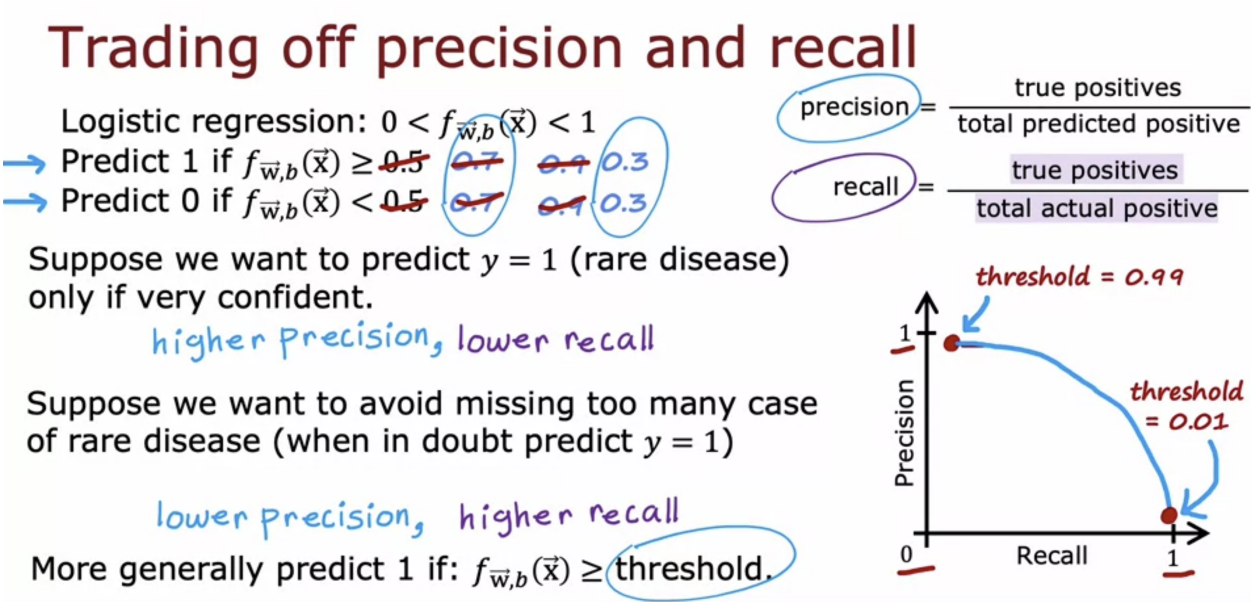


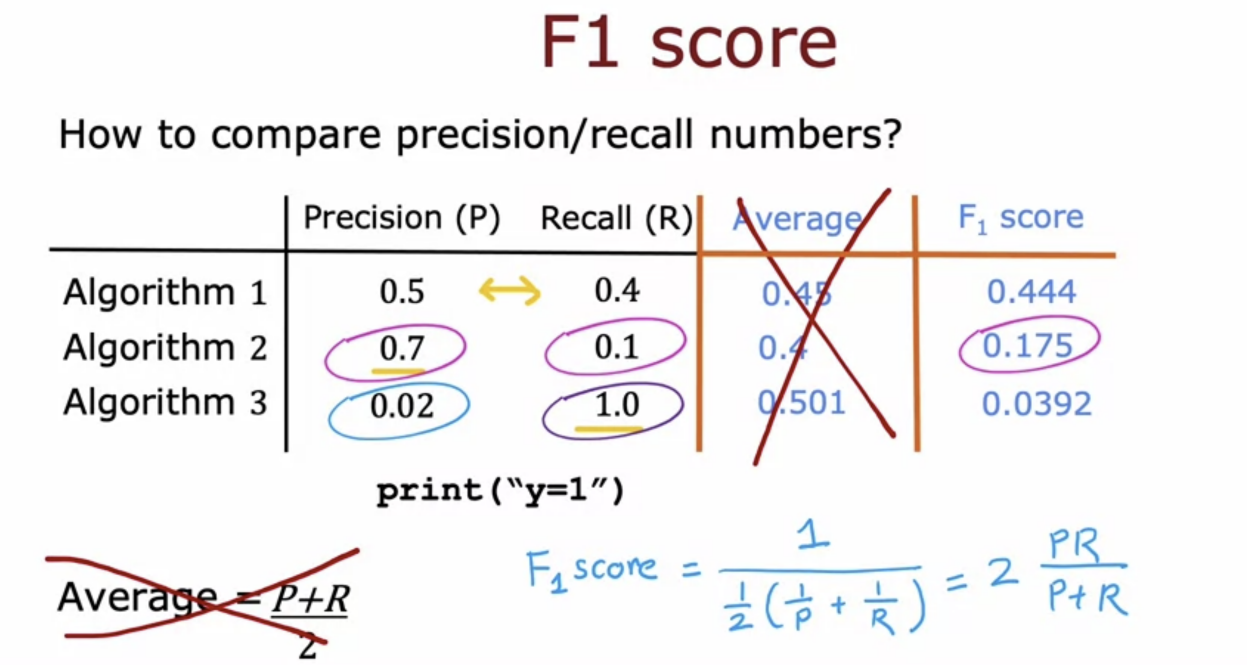



For skewed datasets specially, it's advised to use other metrics, such as precision and recall.




The threshold used for classification is based on a tradeoff between precision and recall, specific to the problem. If we want high precision, we'd increase the threshold. Else, if we want high recall, we'd lower the threshold.

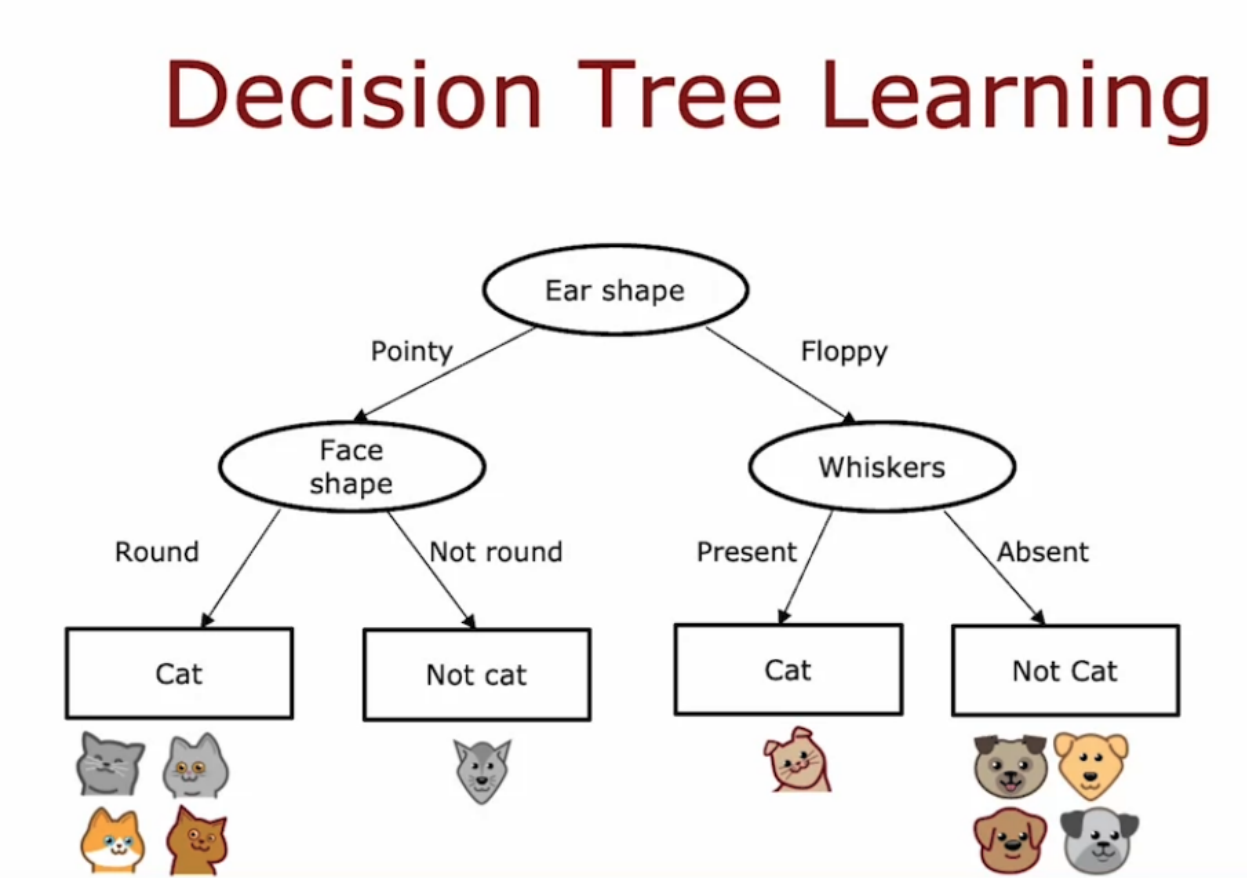



The F1 score is the harmonic mean of precision and recall and it'll give more emphasis on the one that is very small. This is because, if any of the two is too small, then the algorithm is not a very good one.

## Decision Trees

# Decision Tree Learning

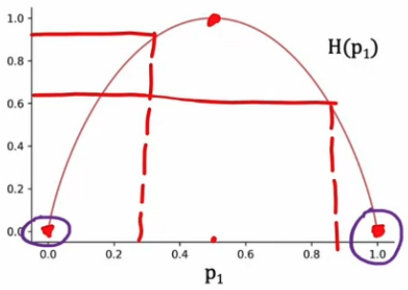



Decisions:

- How to choose what feature to split on at each node? Maximize purity
- When do you stop splitting?
  - When a node is 100% one class
  - When it reached a certain depth
  - When improvements in purity score are below a threshold
  - When number of examples in a node is below a threshold

# Entropy as a measure of impurity

$p_1$ = fraction of examples that are cats





$p_1 = 0 \quad H(p_1) = 0$

$p_1 = 2/6 \quad H(p_1) = 0.92$

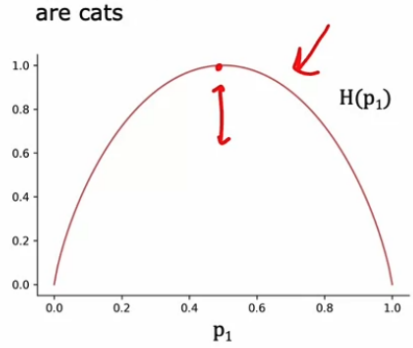
$p_1 = 3/6 \quad H(p_1) = 1$

$p_1 = 5/6 \quad H(p_1) = 0.65$

$p_1 = 6/6 \quad H(p_1) = 0$

# Entropy as a measure of impurity

$p_1$ = fraction of examples that are cats

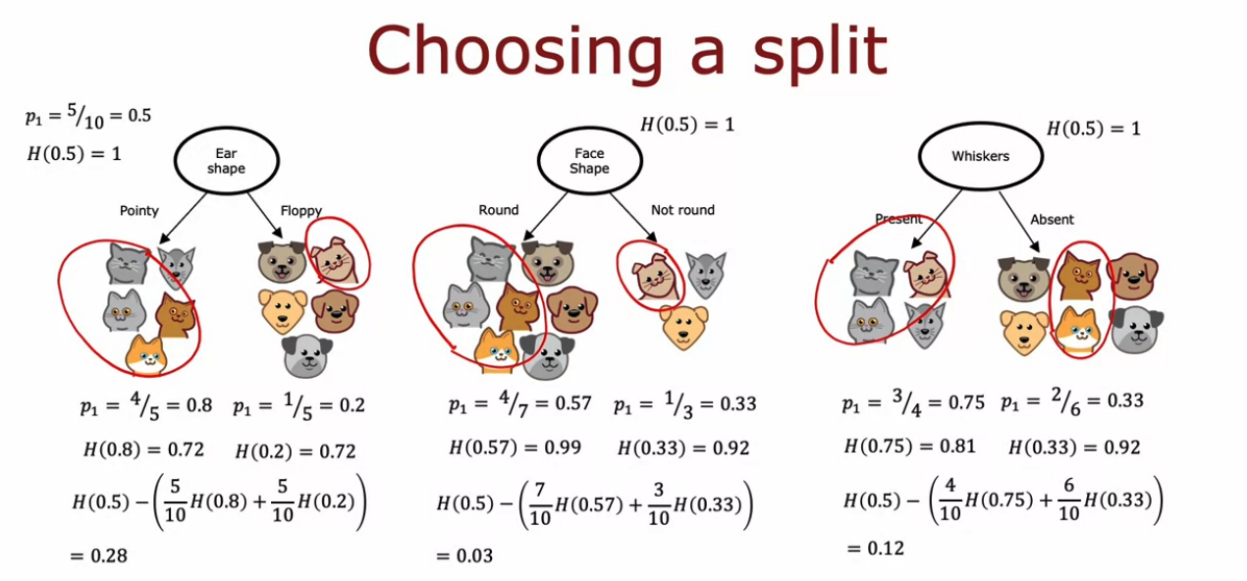



$$p_0 = 1 - p_1$$

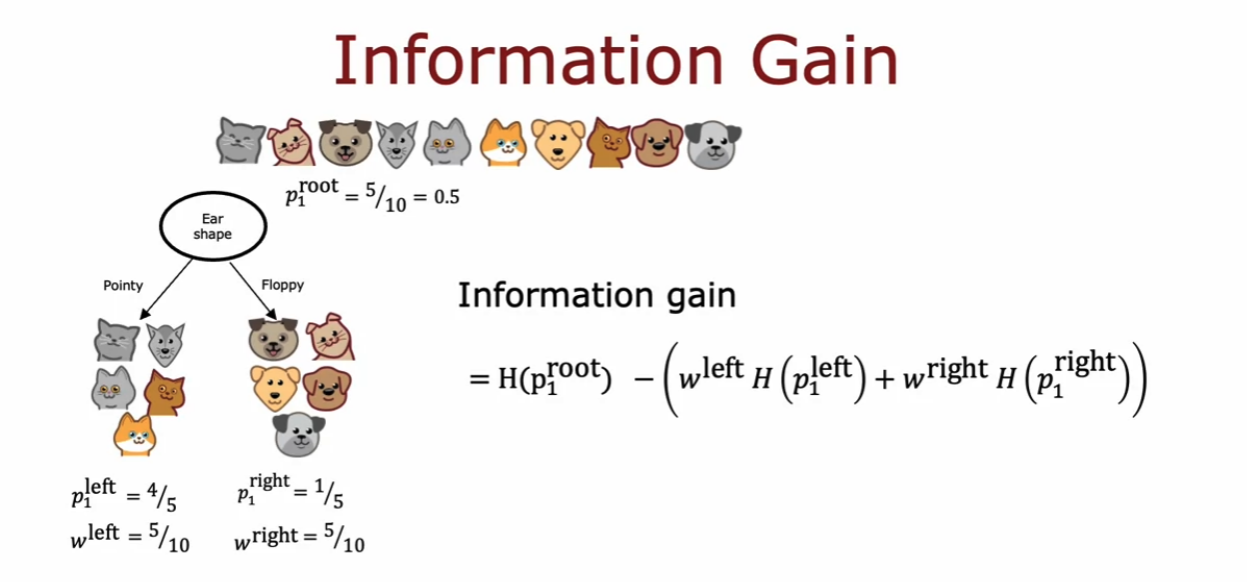
$$H(p_1) = -p_1 log_2(p_1) - p_0 log_2(p_0)$$
$$= -p_1 log_2(p_1) - (1 - p_1) log_2(1 - p_1)$$

Note: "0 log(0)" = 0

**Information Grain = Reduction in Entropy due to Split**




When information gain is too small, we're just increasing the size of the tree and not gaining much prediction capacity, and risking overfitting.



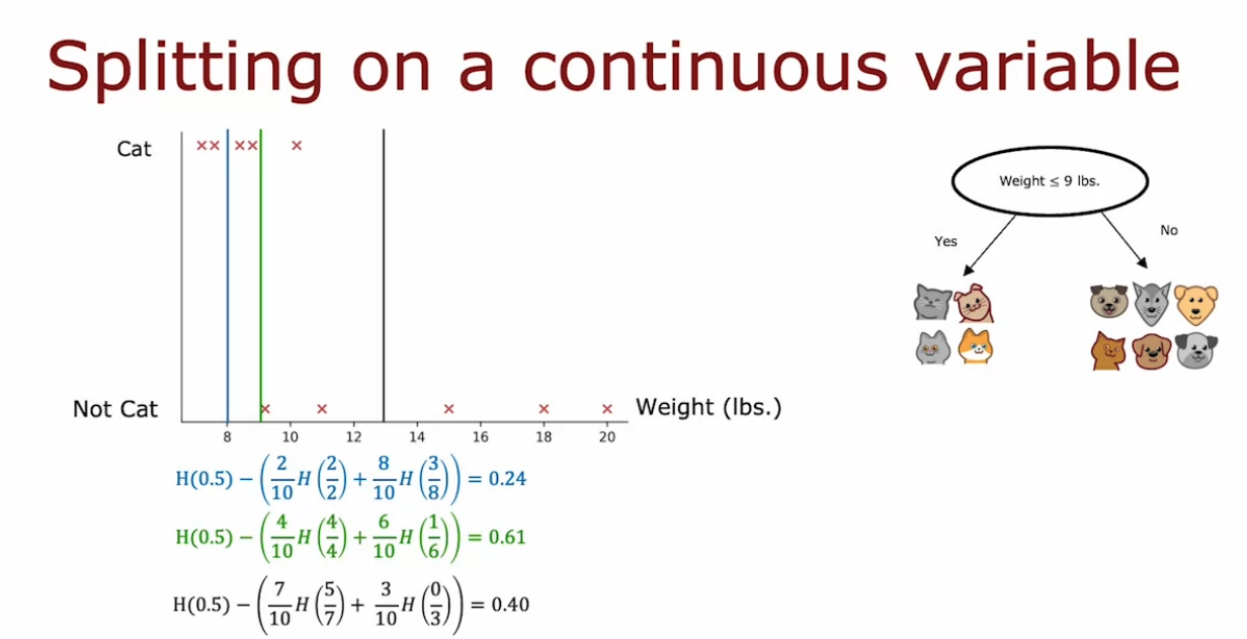
# Decision Tree Learning

- Start with all examples at the root node
- Calculate information gain for all possible features, and pick the one with the highest information gain
- Split dataset according to selected feature, and create left and right branches of the tree
- Keep repeating splitting process until stopping criteria is met:
  - When a node is 100% one class
  - When splitting a node will result in the tree exceeding a maximum depth
  - Information gain from additional splits is less than threshold
  - When number of examples in a node is below a threshold

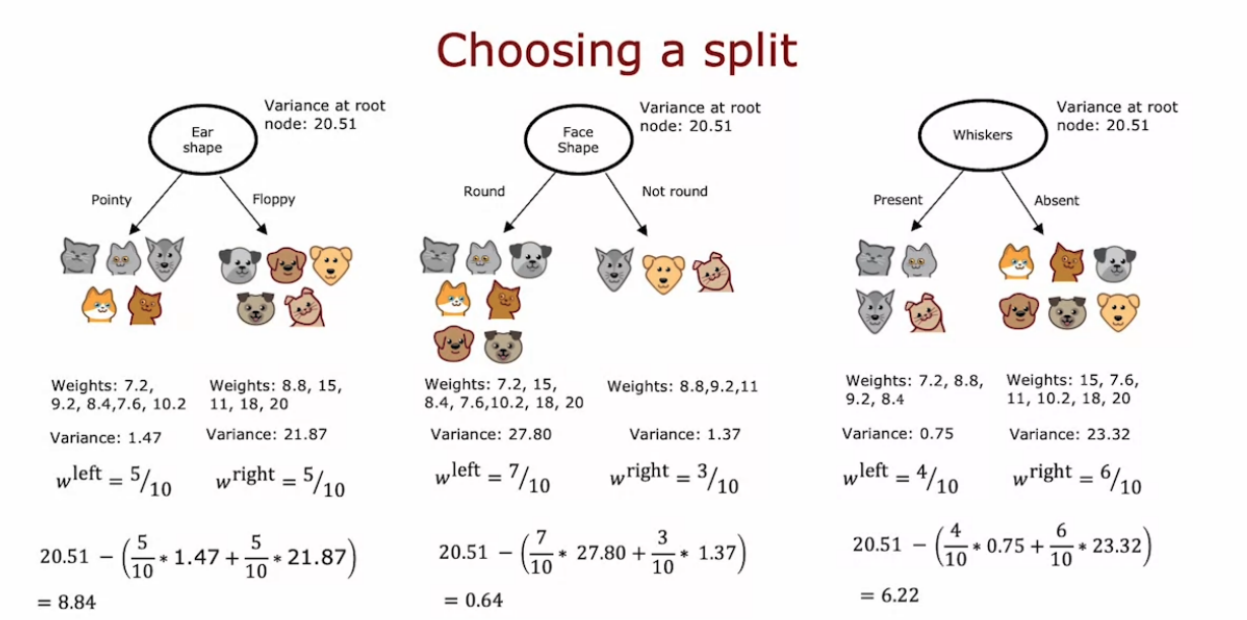
# One hot encoding

| ~~Ear shape~~ | Pointy ears | Floppy ears | Oval ears | Face shape | Whiskers | Cat |
|---|---|---|---|---|---|---|
| ~~Pointy~~ | 1 | 0 | 0 | Round | Present | 1 |
| ~~Oval~~ | 0 | 0 | 1 | Not round | Present | 1 |
| ~~Oval~~ | 0 | 0 | 1 | Round | Absent | 0 |
| ~~Pointy~~ | 1 | 0 | 0 | Not round | Present | 0 |
| ~~Oval~~ | 0 | 0 | 1 | Round | Present | 1 |
| ~~Pointy~~ | 1 | 0 | 0 | Round | Absent | 1 |
| ~~Floppy~~ | 0 | 1 | 0 | Not round | Absent | 0 |
| ~~Oval~~ | 0 | 0 | 1 | Round | Absent | 1 |
| ~~Floppy~~ | 0 | 1 | 0 | Round | Absent | 0 |
| ~~Floppy~~ | 0 | 1 | 0 | Round | Absent | 0 |

If a categorical feature can take on k values, then create k binary features (0 or 1 valued).




For regression trees, instead of minimizing impurity, we want to minimize variance on the leaf nodes. So we choose features that maximizes the reduction in variance.




One decision tree might be highly sensitive to small changes in the dataset. Then we can build a **tree ensemble**! The majority vote of all trees would be the final prediction.

We can construct different datasets that are different than the original dataset using sampling with replacement. The final number of training examples is the kept but we can have repeated examples in the new dataset.

## Sampling with replacement

| | Ear shape | Face shape | Whiskers | Cat |
|---|---|---|---|---|
| | Pointy | Round | Present | 1 |
| | Floppy | Not round | Absent | 0 |
| | Pointy | Round | Absent | 1 |
| | Pointy | Not round | Present | 0 |
| | Floppy | Not round | Absent | 0 |
| | Pointy | Round | Absent | 1 |
| | Pointy | Round | Present | 1 |
| | Floppy | Not round | Present | 1 |
| | Floppy | Round | Absent | 0 |
| | Pointy | Round | Absent | 1 |

## Generating a tree sample

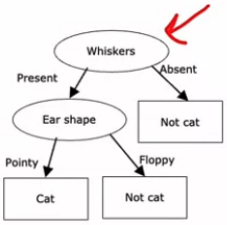
Given training set of size $m$

For $b = 1$ to $B$:

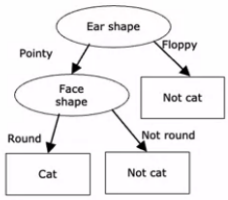
Use sampling with replacement to create a new training set of size $m$

Train a decision tree on the new dataset

| Ear shape | Face shape | Whiskers | Cat |
|---|---|---|---|
| Pointy | Round | Present | Yes |
| Floppy | Round | Absent | No |
| Floppy | Round | Absent | No |
| Pointy | Round | Present | Yes |
| Pointy | Not Round | Present | Yes |
| Floppy | Round | Absent | No |
| Floppy | Round | Present | Yes |
| Pointy | Not Round | Absent | No |
| Pointy | Not Round | Absent | No |
| Pointy | Not Round | Present | Yes |




| Ear shape | Face shape | Whiskers | Cat |
|---|---|---|---|
| Pointy | Round | Present | Yes |
| Pointy | Round | Absent | Yes |
| Floppy | Not Round | Absent | No |
| Floppy | Not Round | Absent | No |
| Pointy | Round | Absent | Yes |
| Floppy | Round | Absent | No |
| Floppy | Round | Absent | No |
| Floppy | Round | Absent | No |
| Pointy | Not Round | Absent | No |
| Pointy | Round | Present | Yes |




...

Bagged decision tree

## Randomizing the feature choice

At each node, when choosing a feature to use to split, if $n$ features are available, pick a random subset of $k < n$ features and allow the algorithm to only choose from that subset of features.

## Boosted Trees

We focus more attention on the training examples we're not doing well yet to build decision trees at later iterations... We look at misclassified examples from all previously generated decision trees.

## XGBoost (eXtreme Gradient Boosting)

- Open source implementation of boosted trees
- Fast efficient implementation
- Good choice of default splitting criteria and criteria for when to stop splitting
- Built in regularization to prevent overfitting
- Highly competitive algorithm for machine learning competitions (eg: Kaggle competitions)

XGBoost assigns weights to different training examples, instead of doing sampling with replacement.

# Using XGBoost

**Classification**

```
from xgboost import XGBClassifier

model = XGBClassifier()

model.fit(X_train, y_train)
y_pred = model.predict(X_test)
```

**Regression**

```
from xgboost import XGBRegressor

model = XGBRegressor()

model.fit(X_train, y_train)
y_pred = model.predict(X_test)
```

# Decision Trees vs Neural Networks

**Decision Trees and Tree ensembles**

- Works well on tabular (structured) data
- Not recommended for unstructured data (images, audio, text)
- Fast
- Small decision trees may be human interpretable

**Neural Networks**

- Works well on all types of data, including tabular (structured) and unstructured data
- May be slower than a decision tree
- Works with transfer learning
- When building a system of multiple models working together, it might be easier to string together multiple neural networks