Contents

Week 1

Supervised vs Unsupervised Learning

Supervised Learning

- Learn to predict input, output (or X to Y mapping) based on training on “right answers”.
- Types of supervised learning:
  - Regression
    - Predict numbers out of infinitely many possible numbers.
    - Example: predict house prices.
  - Classification:
    - Predict small number of possible outputs or categories/classes.
    - Can have more than 1 input value.
    - Akin to fitting a boundary line or segmenting data points.
    - Example: Breast cancer detection (is tumour malignant 1, malignant 2, or benign?).

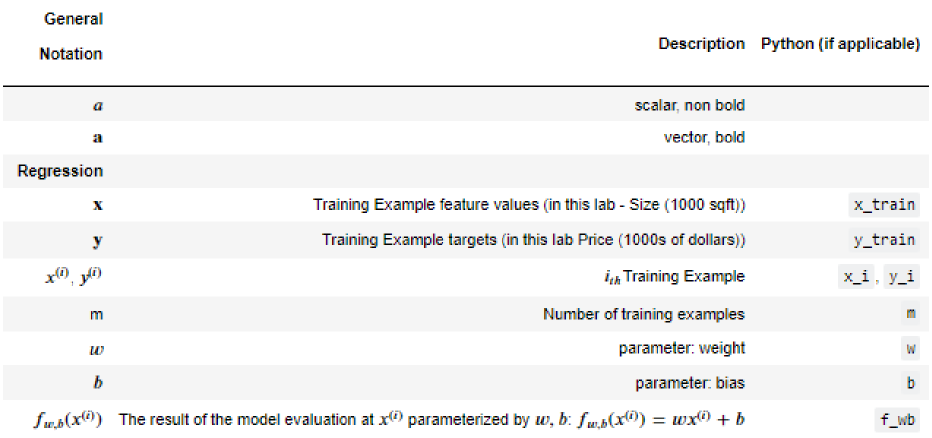
Unsupervised Learning

- Given learning data that is NOT associated with output labels.
- Algorithm’s job is to find some structure or patterns in the data.
- Types of unsupervised learning:
  - Clustering
    - Places unlabelled data into different clusters by similarity.
    - Example: Google News grouping related stories, grouping customers.
  - Anomaly Detection
    - Find unusual data points.
    - Example: Fraud detection in the financial industry.
  - Dimensionality Reduction
    - Compress data using fewer numbers while losing as little data as possible.

Regression Model

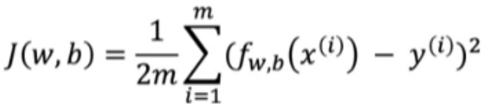
Linear Regression

- Fits a straight line through the data graph to read off input/output.
- Terminology
  - Training Set: data used to train model.
  - x: Notation to represent input.
    - Also called the “feature”.
  - y: Notation to represent output.
    - Also called “target”.
    - This is the actual output in the training set.
  - m: Number of training examples.
    - (x, y) represents a single training example.
    - ($x^{(i)}$, $y^{(i)}$) represents the ith training example, where i is a specific row in the training set table.
  - y-hat: A prediction output by a function that was created using a learning algorithm with a training set, given an input x. y-hat is the function’s estimation of y (target).



General Notation	Description	Python (if applicable)
a	scalar, non bold	
$\mathbf{a}$	vector, bold	
Regression		
$\mathbf{x}$	Training Example feature values (in this lab - Size (1000 sqft))	`x_train`
$\mathbf{y}$	Training Example targets (in this lab Price (1000s of dollars))	`y_train`
$x^{(i)}$, $y^{(i)}$	i_{th} Training Example	`x_i`, `y_i`
m	Number of training examples	`m`
w	parameter: weight	`w`
b	parameter: bias	`b`
$f_{w,b}(x^{(i)})$	The result of the model evaluation at $x^{(i)}$ parameterized by w, b: $f_{w,b}(x^{(i)}) = wx^{(i)} + b$	`f_wb`

- Linear regression with a single variable (feature) is also called univariate linear regression.
- Cost Function
  - Compare y-hat with y, the difference is called the error.
  - Sum over all training examples, the error squared and average it by dividing by m.
  - By convention the sum is also divided by 2 to make further calculations easier (derivative term of gradient descent algorithm).
  - This is called the squared error cost function – this is the most popular for regression models.
  - Denoted by the variable "J" – we wish to minimize this.

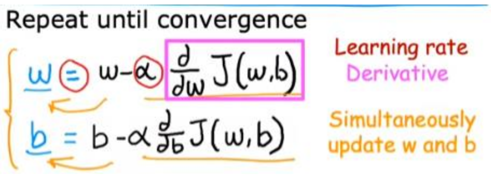


$$J(w,b) = \frac{1}{2m}\sum_{i=1}^{m}(f_{w,b}(x^{(i)}) - y^{(i)})^2$$

  - Visualizing the cost function in a 3D plot with w and b as the other 2 axes, the graph is shaped like a soup bowl.
  - In 2D, can represent J vs w vs b graph as a contour plot with minimum in centre.

Gradient Descent

- Start with any initial values for the parameters w and b.
- Keep changing w and b until J settles at a minimum.

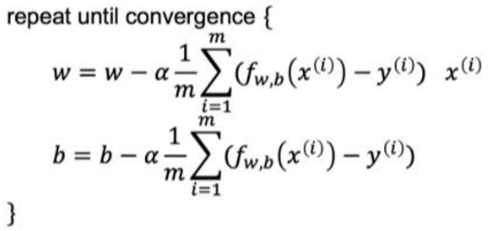


Repeat until convergence

$$w = w - \alpha \boxed{\frac{d}{dw}J(w,b)}$$

$$b = b - \alpha \frac{d}{db}J(w,b)$$

- May have more than 1 minimum in general (but only 1 for linear regression's cost function).
- Can be used on more than just 2 parameters (w, b).
- Expanding out the derivative terms concretely (notice the 2's cancels out for both terms):



repeat until convergence {

$$w = w - \alpha\frac{1}{m}\sum_{i=1}^{m}(f_{w,b}(x^{(i)}) - y^{(i)})\ x^{(i)}$$

$$b = b - \alpha\frac{1}{m}\sum_{i=1}^{m}(f_{w,b}(x^{(i)}) - y^{(i)})$$

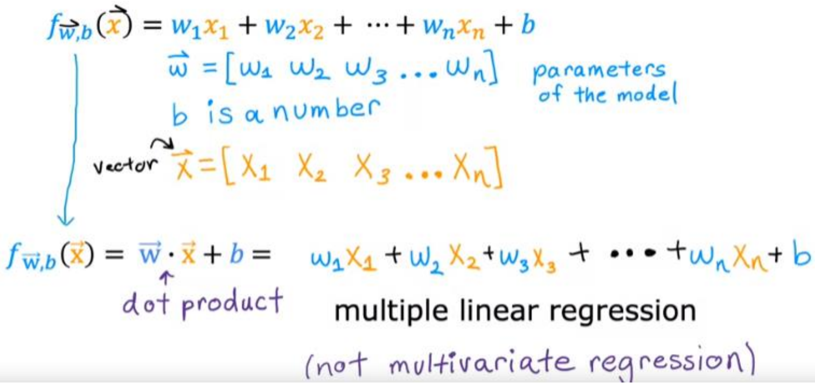
}

- Batch gradient descent: Each step of gradient descent uses all the training examples.

Week 2

Multiple Linear Regression

Multiple Features

- x_j denotes the jth feature in a training set.
- The number of features is n.
- $\mathbf{x}^{(i)}$ now denotes a vector of features (the ith training example).
- $x_3^{(2)}$ is the 3rd feature of the 2nd training example.
- Model can be rewritten in terms of vectors.
- 

$$f_{\vec{w},b}(\vec{x}) = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b$$

$\vec{w} = [w_1\ w_2\ w_3 \ldots w_n]$ parameters of the model

b is a number

vector $\vec{x} = [x_1\ x_2\ x_3 \ldots x_n]$

$$f_{\vec{w},b}(\vec{x}) = \vec{w}\cdot\vec{x} + b = w_1x_1 + w_2x_2 + w_3x_3 + \cdots + w_nx_n + b$$

dot product

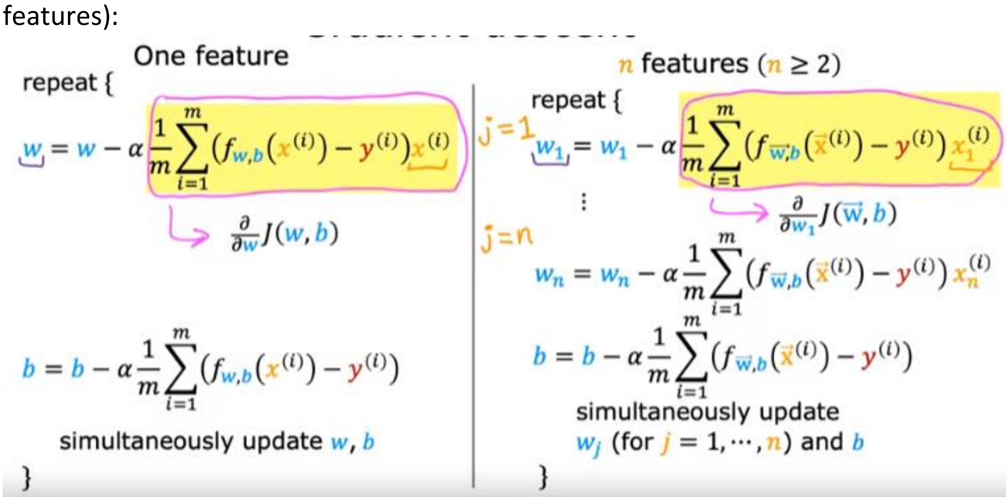
multiple linear regression

(not multivariate regression)

Vectorization

- Speeds up dot product calculations compared to using a for-loop.
- Uses numpy's dot() method which takes two vectors as an argument.
- Faster than for-loop because it uses parallel processing.
- Also applies to other linear algebra operations.

Gradient Descent

- Comparison of single vs multiple feature algorithm (using vectorization for multiple features):
- 

One feature

repeat {

$$w = w - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(f_{w,b}(x^{(i)}) - y^{(i)}\right)x^{(i)}$$

$\frac{\partial}{\partial w}J(w,b)$

$$b = b - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(f_{w,b}(x^{(i)}) - y^{(i)}\right)$$

simultaneously update w, b

}

n features $(n \geq 2)$

repeat {

$j=1$

$$w_1 = w_1 - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)x_1^{(i)}$$

$\frac{\partial}{\partial w_1}J(\vec{w},b)$

$\vdots$

$j=n$

$$w_n = w_n - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)x_n^{(i)}$$

$$b = b - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)$$

simultaneously update w_j (for $j = 1,\cdots,n$) and b

}

- Alternative to using gradient descent
    - Normal equation
        - Only applies to linear regression.
        - Solves for w, b without iterations.
        - Slow when number of features is large.
        - Gradient descent is still the preferred choice for normal practitioners.

Practical Advice

- Speed up gradient descent
    - Feature scaling
        - Used when feature magnitudes differ greatly.
        - A large variation in feature magnitudes can cause gradient descent to “bounce around” and perform inefficiently.
        - Implementation:
            - Divide each feature by its maximum value.
            - Mean normalization: subtract from each feature value it’s mean and divide the difference by the feature’s range.
            - z-score normalization: Like mean normalization but divide by the feature’s standard deviation instead.
        - Aim for around $-1 <= x_j <= 1$ for each feature, or within 1-digit numbers.
        - When in doubt, no harm to just apply it.
- Checking for gradient descent convergence
    - Plot cost function, J at each iteration of gradient descent.
        - The result is called the “learning curve”.
        - The vertical axis, J should be decreasing after every iteration.
    - Alternatively, use an automatic convergence test in code to check if J decreases by an amount less than a specific, predetermined epsilon value.
- Choice of learning rage
    - If J sometimes increases with each iteration, the learning rate may be too large.
    - Trade-off is that if learning rate is too small, gradient descent will take a long time to converge.
    - Values of alpha to try: …, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, …
    - Then pick the largest possible learning rate which always causes J to decrease at every iteration.
- Feature engineering
    - Using intuition to design new features by transforming or combining original features.

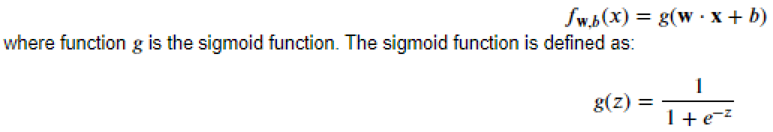
Polynomial Regression

- Uses the ideas of multiple linear regression and feature engineering to fit curves to data.
- Feature scaling becomes important if say you raise a current feature to a power i.e., width -> area or volume.

Week 3

Classification with Logistic Regression

- The output is a small set of finite values called “classes” or “categories”.
- The model is represented by the following equation:

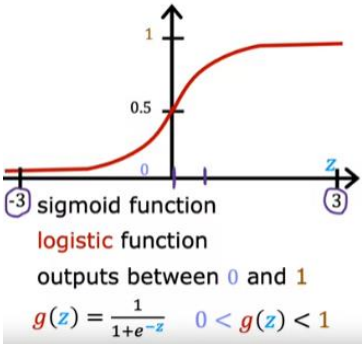


$$f_{\mathbf{w},b}(x) = g(\mathbf{w} \cdot \mathbf{x} + b)$$

where function g is the sigmoid function. The sigmoid function is defined as:

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

- Uses a sigmoid function to model the data – g(z) is equivalent to f_wb_i.



sigmoid function

logistic function

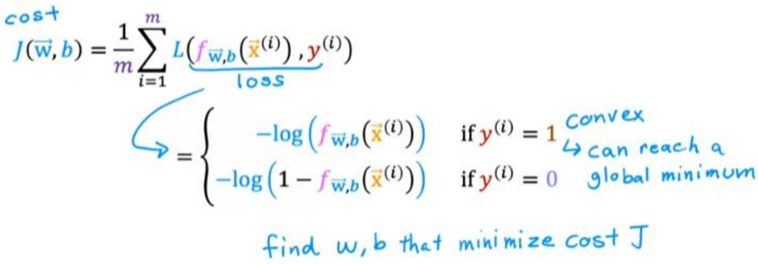
outputs between 0 and 1

$g(z) = \frac{1}{1+e^{-z}} \quad 0 < g(z) < 1$

- g(z) defines the sigmoid function, where the input z (a scalar) is a prediction using an input and the model’s parameters.
- Logistic regression’s output can be interpreted as the probability that a class can be classified under a certain category.

Cost Function for Logistic Regression

- The Loss function is an input to the Cost function and determines whether the graph of J is a convex function.



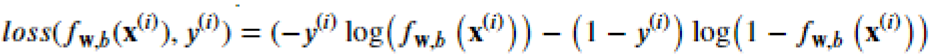
$$\overset{\text{cost}}{J(\vec{w},b)} = \frac{1}{m}\sum_{i=1}^{m} \underbrace{L\left(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}\right)}_{\text{loss}}$$

$$= \begin{cases} -\log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) & \text{if } y^{(i)} = 1 \\ -\log\left(1 - f_{\vec{w},b}(\vec{x}^{(i)})\right) & \text{if } y^{(i)} = 0 \end{cases}$$

convex
↳ can reach a global minimum

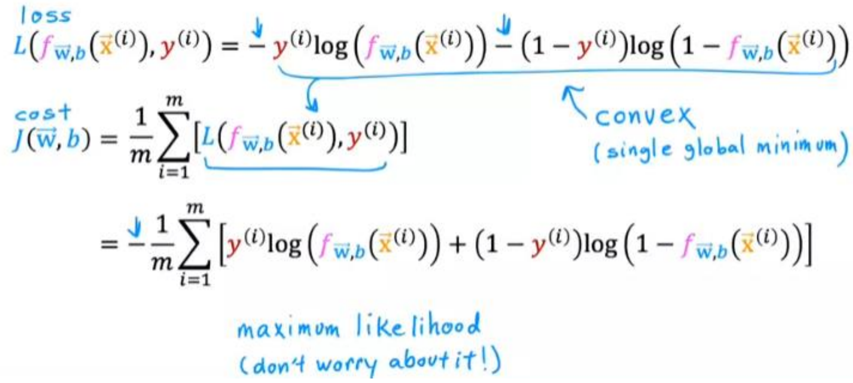
find w, b that minimize cost J

- Alternatively, in a single line, the Loss function can be written:



$$loss(f_{\mathbf{w},b}(\mathbf{x}^{(i)}), y^{(i)}) = (-y^{(i)} \log\left(f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right)\right) - \left(1 - y^{(i)}\right) \log\left(1 - f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right)\right)$$

- And the Cost function can be simplified to:



$$\overset{\text{loss}}{L(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)})} = \overset{\downarrow}{-}\,\underbrace{y^{(i)}\log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) \overset{\downarrow}{-} (1-y^{(i)})\log\left(1-f_{\vec{w},b}(\vec{x}^{(i)})\right)}$$

$$\overset{\text{cost}}{J(\vec{w},b)} = \frac{1}{m}\sum_{i=1}^{m}\underbrace{\left[L\left(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}\right)\right]}$$

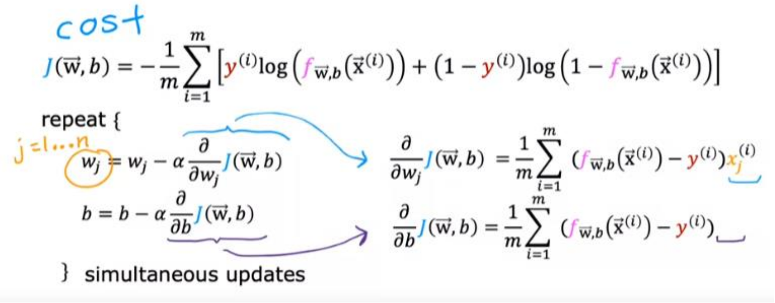
convex
(single global minimum)

$$= \overset{\downarrow}{-}\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) + (1-y^{(i)})\log\left(1-f_{\vec{w},b}(\vec{x}^{(i)})\right)\right]$$

maximum likelihood
(don't worry about it!)

- The Loss function is calculated for each training example.
- The Cost function is obtained by summing the Loss function over all training examples and dividing by the number of training examples.
- The Loss function penalises models (by increasing greatly) if it gets the output wrong.

Gradient Descent for Logistic Regression

- Like gradient descent for linear regression, but with a different Loss function and partial derivative.
- 

cost

$$J(\vec{w},b) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) + (1-y^{(i)})\log\left(1-f_{\vec{w},b}(\vec{x}^{(i)})\right)\right]$$

repeat {

$j=1\ldots n$

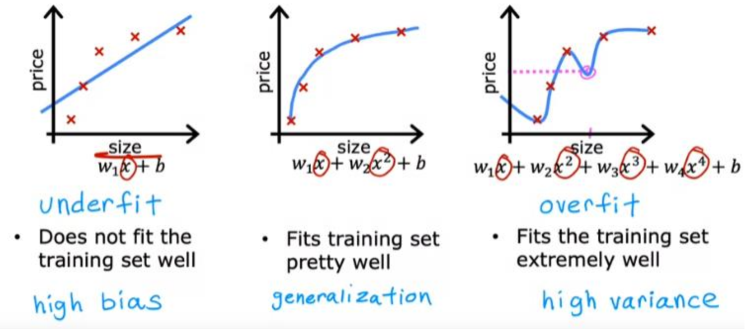
$$w_j = w_j - \alpha\frac{\partial}{\partial w_j}J(\vec{w},b) \qquad \frac{\partial}{\partial w_j}J(\vec{w},b) = \frac{1}{m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)x_j^{(i)}$$

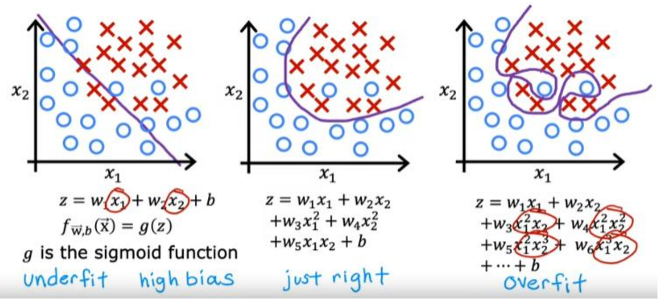
$$b = b - \alpha\frac{\partial}{\partial b}J(\vec{w},b) \qquad \frac{\partial}{\partial b}J(\vec{w},b) = \frac{1}{m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)$$

} simultaneous updates

- Same concepts for:
  - Monitoring gradient descent (learning curve) for convergence.
  - Vectorized implementation.
  - Feature scaling.

Overfitting

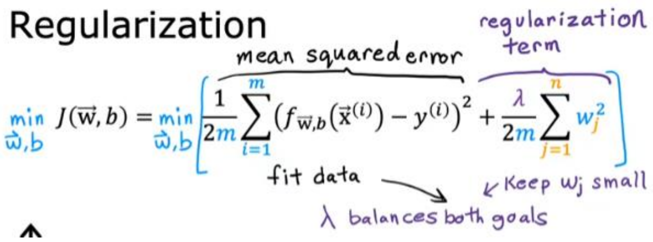
- If a model does not fit the training set well, we say that it is underfitting or have a high bias.
- Bias – the algorithm has underfit the data.
- The strong preconception that a model will fit the data despite data showing the contrary.
- If a model fits a training set well and performs well on data outside the training set, we call this generalization.
- If a model fits the training set extremely well, there is a danger that the model has overfit the data.
- Overfitting does not generalize well, and we can also say the model has high variance.
- Example for Linear Regression:
  - 

- Example for Classification:
  - 

- Ways to address overfitting:
    - Collect more training examples.
    - Select features to include/exclude.
        - Many features with insufficient data can cause overfitting.
        - Feature selection: Use your intuition to pick the most impactful features.
        - The disadvantage of feature selection is that you lose some information.
    - Regularization.
        - Reduce the size of all parameters w_j.
        - Allows you to keep all your features but prevents any feature from having too large an impact.

Regularization

- We often have many features and may not know which feature to penalize (or reduce the size of).
- To solve this, simply add every single w_j to the cost function.

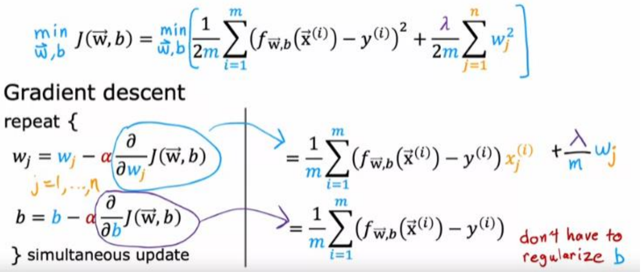


Regularization

$$\min_{\vec{w},b} J(\vec{w},b) = \min_{\vec{w},b}\left[\underbrace{\overbrace{\frac{1}{2m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)})-y^{(i)}\right)^2}^{\text{mean squared error}}}_{\text{fit data}} + \underbrace{\overbrace{\frac{\lambda}{2m}\sum_{j=1}^{n}w_j^2}^{\text{regularization term}}}_{\text{Keep } w_j \text{ small}}\right]$$

λ balances both goals

- This incentivizes the model to pick small values for w_j.
- Choose a value of Lambda that minimizes both the contribution of the mean squared error and the regularization term.
- Regularized linear regression.
    - The difference is the Regularization term in the update for w_j.



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

Gradient descent

repeat {

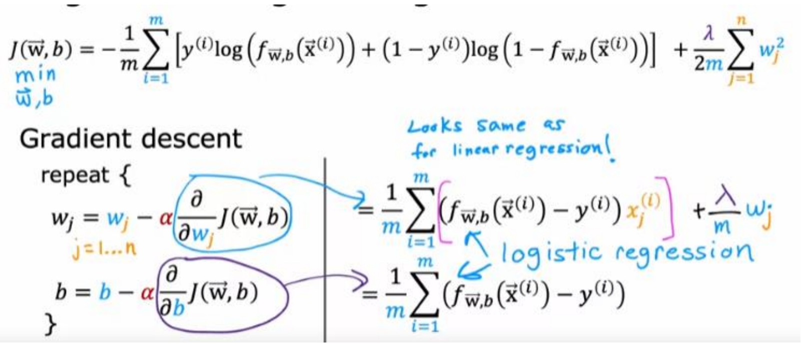
$w_j = w_j - \alpha\frac{\partial}{\partial w_j}J(\vec{w},b)$ $\quad = \frac{1}{m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)})-y^{(i)}\right)x_j^{(i)} + \frac{\lambda}{m}w_j$

$j=1,\ldots,n$

$b = b - \alpha\frac{\partial}{\partial b}J(\vec{w},b)$ $\quad = \frac{1}{m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)})-y^{(i)}\right)$ don't have to regularize b

} simultaneous update

- Regularized logistic regression.
    - Like the cost function for normal logistic regression except with an additional Regularization parameter at the end.



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

Gradient descent

repeat {

$w_j = w_j - \alpha\frac{\partial}{\partial w_j}J(\vec{w},b)$ $\quad = \frac{1}{m}\sum_{i=1}^{m}\left[\left(f_{\vec{w},b}(\vec{x}^{(i)})-y^{(i)}\right)x_j^{(i)}\right] + \frac{\lambda}{m}w_j$

$j=1\ldots n$

$b = b - \alpha\frac{\partial}{\partial b}J(\vec{w},b)$ $\quad = \frac{1}{m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)})-y^{(i)}\right)$

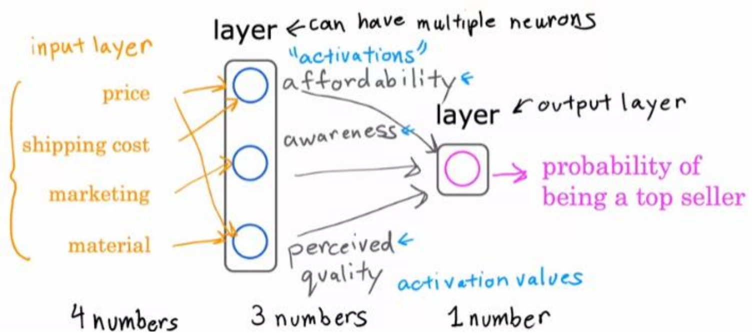
}

Looks same as for linear regression!

logistic regression

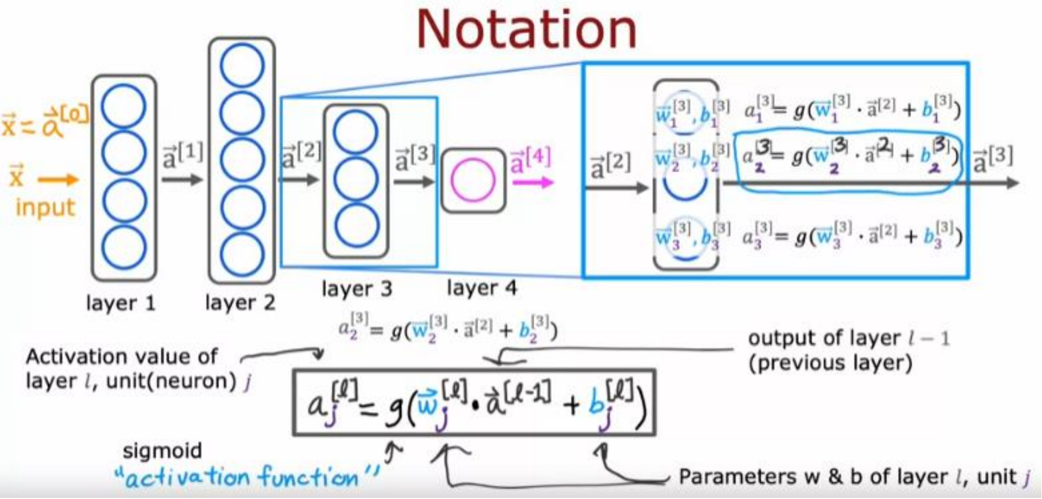
Week 4

Neural Networks

- Terminology used:
- 

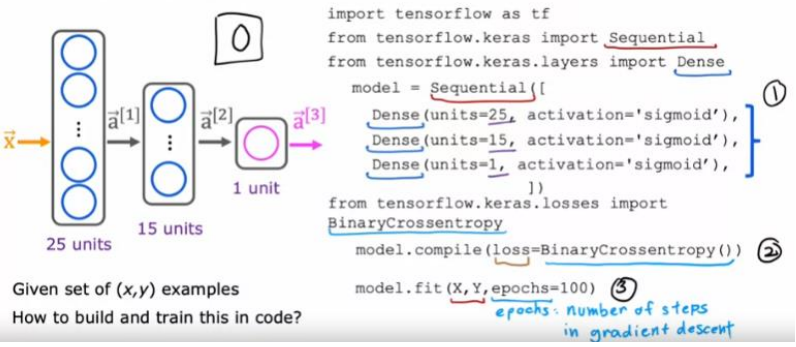
- The outputs from each neuron are called "activations".
- Every neuron has access to all inputs and will learn to ignore irrelevant features.
- The intermediate layers are called the "hidden layers".
- The input vector (Layer 0) is denoted by **x** and the intermediate output vector is denoted by **a**.
- Can think of this as logistic regression with the ability to feature engineer by itself.
- Can have multiple hidden layers as well.

Neural Networks Model

- We say that $\mathbf{w}_1^{[1]}$, $b_1^{[1]}$ are the parameters of neuron 1 in layer [1], and $\mathbf{a}^{[1]}$ are the activation values output from layer [1].
- Each neuron is represented by a logistic regression model with parameters **w** and b.
- The input into a neuron is a vector.
- If predicting for a classification problem, an if-else can be applied to the final n^th^ layer's activation values $a^{[n]}$, say if $a^{[n]}$ >= 0.5 then output 1, else 0.
- When we say a neural network has 4 layers, we do not count the input layer, therefore there are 3 hidden layers and 1 output layer in this example.
- Terminology summary (with a zoom into Layer 3):
- 
Notation
$\vec{x} = \vec{a}^{[0]}$
$\vec{x}$ → input
$\vec{a}^{[1]}$
$\vec{a}^{[2]}$
$\vec{a}^{[3]}$
$\vec{a}^{[4]}$
layer 1
layer 2
layer 3
layer 4
$\vec{a}^{[2]}$
$\vec{w}_1^{[3]}, b_1^{[3]}$ $a_1^{[3]} = g(\vec{w}_1^{[3]} \cdot \vec{a}^{[2]} + b_1^{[3]})$
$\vec{w}_2^{[3]}, b_2^{[3]}$ $a_2^{[3]} = g(\vec{w}_2^{[3]} \cdot \vec{a}^{[2]} + b_2^{[3]})$
$\vec{w}_3^{[3]}, b_3^{[3]}$ $a_3^{[3]} = g(\vec{w}_3^{[3]} \cdot \vec{a}^{[2]} + b_3^{[3]})$
$\vec{a}^{[3]}$
$a_2^{[3]} = g(\vec{w}_2^{[3]} \cdot \vec{a}^{[2]} + b_2^{[3]})$
Activation value of layer l, unit(neuron) j
output of layer $l-1$ (previous layer)
$a_j^{[l]} = g(\vec{w}_j^{[l]} \cdot \vec{a}^{[l-1]} + b_j^{[l]})$
sigmoid
"activation function"
Parameters w & b of layer l, unit j
- When the calculations proceed from left to right of the page, we call this forward propagation.
- Back propagation is an algorithm that goes the opposite direction, used for learning.

Tensorflow

- Overview of how to train a neural network in Tensorflow.



Given set of (x,y) examples

How to build and train this in code?

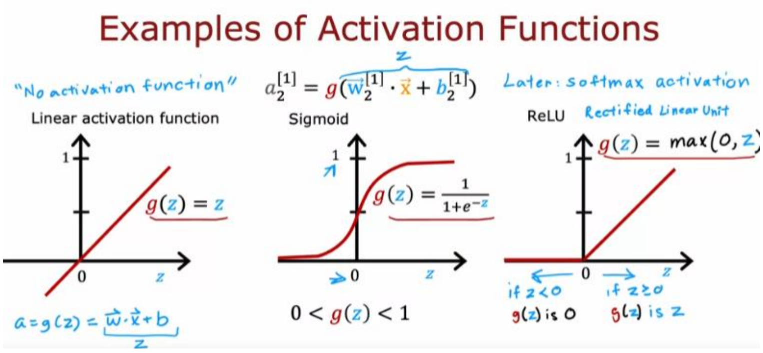
```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
   model = Sequential([                                    ①
    Dense(units=25, activation='sigmoid’),
    Dense(units=15, activation='sigmoid’),
    Dense(units=1, activation='sigmoid’),
                    ])
from tensorflow.keras.losses import
BinaryCrossentropy
   model.compile(loss=BinaryCrossentropy())   ②

   model.fit(X,Y,epochs=100)  ③
```

epochs: number of steps in gradient descent

- Step 1 creates the model.
- Step 2 defines the loss and cost functions (Binary Cross Entropy is the same thing as the logistic loss function – for binary classification tasks, 2 classes).
- For a regression problem using mean squared error, you can use the argument loss=MeanSquaredError() instead (from tensorflow.keras.losses import MeanSquaredError).
- Step 3 computes derivatives for gradient descent using back propagation (epochs are the number of iterations).
- Using model.fit() updates the network parameters to reduce cost.

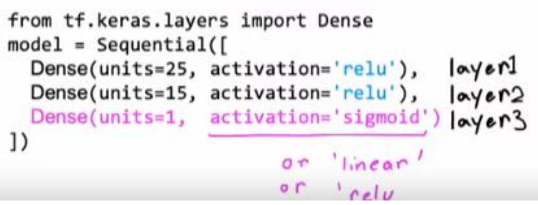
Week 5

Activation Functions

- Three of the most common activation functions:
- 

Examples of Activation Functions

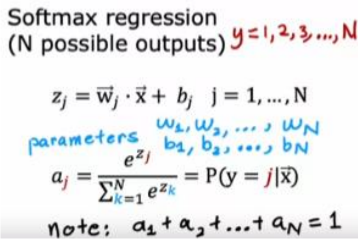
$a_2^{[1]} = g(\overbrace{\vec{w}_2^{[1]} \cdot \vec{x} + b_2^{[1]}}^{z})$

- ReLU can be good for when you want activation values that are natural numbers i.e., 0, 1, …
- How to choose an activation function.
    - Activation values of target/ground truth label, y.
        - Binary classification (0/1) – sigmoid (i.e. true or false)
        - Regression (can be positive or negative) – linear (i.e. stock prices can go up or down)
        - Nonnegative values – ReLU (i.e. house prices cannot be negative)
    - Hidden layers.
        - ReLU is the most common choice.
        - Faster to compute since it only needs to calculate the max() function.
        - Flat on only one part of the graph, so gradient descent is more efficient (faster learning).
    - 

```
from tf.keras.layers import Dense
model = Sequential([
  Dense(units=25, activation='relu'),    layer1
  Dense(units=15, activation='relu'),    layer2
  Dense(units=1,  activation='sigmoid') layer3
])
                        or  'linear'
                        or  'relu
```

- Why do we need activation functions?
    - If we just use linear activation function, the entire network will essentially become linear regression, defeating the purpose of even using neural networks in the first place.
    - If all hidden layers use the linear activation function and the output is the sigmoid, we end up with logistic regression.
    - The takeaway: don't use linear activation function in the hidden layer, just use ReLU.

Multiclass Classification

- The Softmax algorithm is a generalization of logistic regression.
- Model for softmax regression:
- 

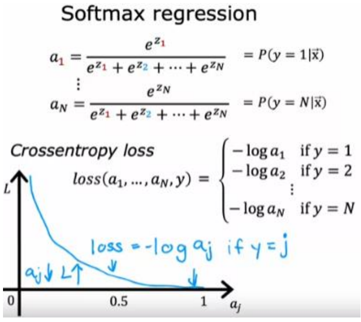
Softmax regression
(N possible outputs) $y = 1, 2, 3, \ldots, N$

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

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

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

note: $a_1 + a_2 + \ldots + a_N = 1$

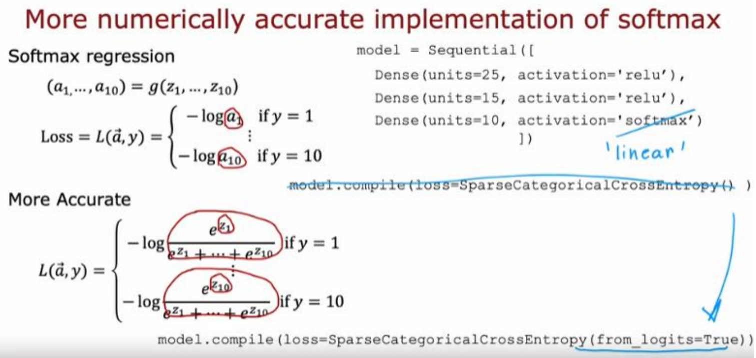
- Each a_i value is dependent on z_1 to z_N – this is different to other activation functions which only depend on their own model.
- When n=2, it ends up reducing to logistic regression.
- Cost of softmax regression:
- 

Softmax regression

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

Crossentropy loss

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

- Softmax is used in the output layer – if 10 categories, then use 10 units in this layer.
- Implementing MNIST with softmax (use the code highlighted in blue instead):
- 

More numerically accurate implementation of softmax

Softmax regression

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

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

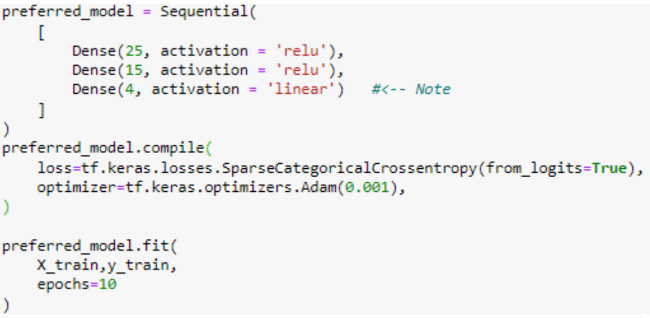
More Accurate

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

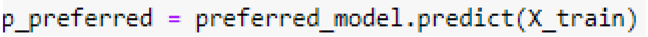
```
model = Sequential([
  Dense(units=25, activation='relu'),
  Dense(units=15, activation='relu'),
  Dense(units=10, activation='~~softmax~~')
                  ])
                            'linear'
```

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

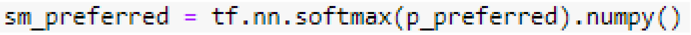
```
model.compile(loss=SparseCategoricalCrossEntropy(from_logits=True))
```

- With the more accurate version, we will need to add a couple of extra steps since the output is z rather than a.
- The code in full:
  - 

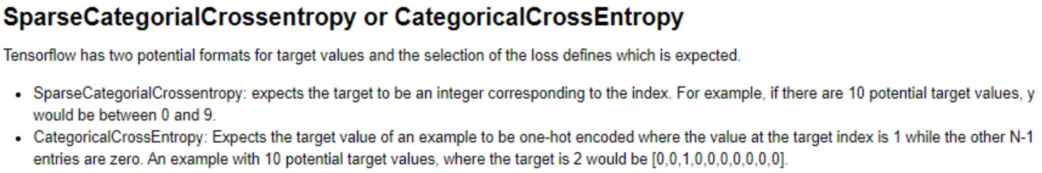
```
preferred_model = Sequential(
    [
        Dense(25, activation = 'relu'),
        Dense(15, activation = 'relu'),
        Dense(4, activation = 'linear')    #<-- Note
    ]
)
preferred_model.compile(
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    optimizer=tf.keras.optimizers.Adam(0.001),
)

preferred_model.fit(
    X_train,y_train,
    epochs=10
)
```

  - 

```
p_preferred = preferred_model.predict(X_train)
```

  - 

```
sm_preferred = tf.nn.softmax(p_preferred).numpy()
```

- Note the following difference in Loss function.
- 

SparseCategorialCrossentropy or CategoricalCrossEntropy

Tensorflow has two potential formats for target values and the selection of the loss defines which is expected.

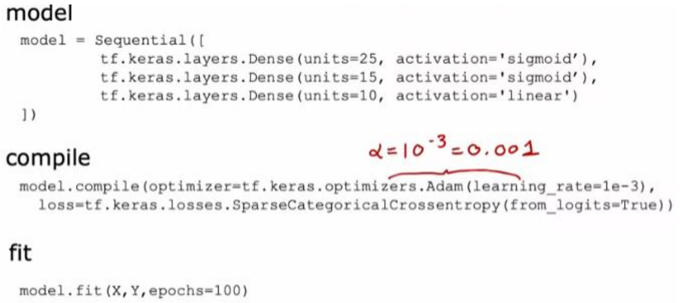
- SparseCategorialCrossentropy: expects the target to be an integer corresponding to the index. For example, if there are 10 potential target values, y would be between 0 and 9.
- CategoricalCrossEntropy: Expects the target value of an example to be one-hot encoded where the value at the target index is 1 while the other N-1 entries are zero. An example with 10 potential target values, where the target is 2 would be [0,0,1,0,0,0,0,0,0,0].

Multi-label Classification

- Associated with a single input x, there are three different labels (i.e., in self-driving cars, does an image contain a car? Does it contain a bus? Does it contain a pedestrian?).
- Neural network can have more than one output (stored as a vector of outputs).

- It is also reasonable to make a few different multiclass classification models, each of single output as an alternative.

Additional Concepts

- Alternative optimizations
    - Adam algorithm as an improvement to gradient descent.
        - Go faster – increase alpha (learning rate) if we are taking many little steps in the same direction.
        - Go slower – decrease alpha if we are bouncing around too much.
        - Adam: Adaptive Moment estimation.
        - Has as many alphas as parameters (w_1, w_2, …, b).
        - 

          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

          $\alpha=10^{-3}=0.001$

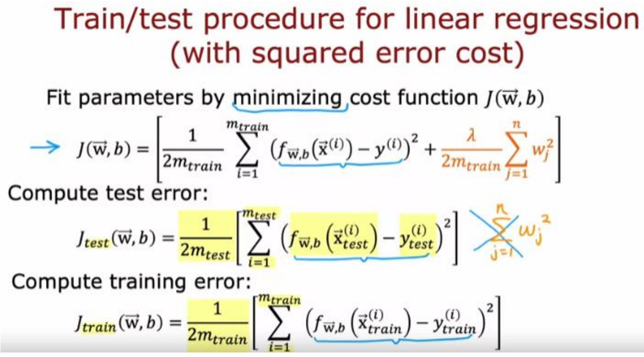
          ```
          model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
            loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True))
          ```

          fit

          ```
          model.fit(X,Y,epochs=100)
          ```
- Additional layer types
    - Dense layer: each neuron output is a function of ALL the activation outputs of the previous layer.
    - Convolutional layer: each neuron only looks at part of the previous layer's inputs.
        - Why?
            - Speeds up computation.
            - Need less training data (less prone to overfitting).
            - Multiple convolutional layers in a neural network are called a convolutional neural network.

Week 6

Advice for applying machine learning

- Diagnostic: A test that you run to gain insight into what is/isn't working with a learning algorithm.
- If you have a lot of errors, try:
    - Fixes High variance:
        - Get more examples.
        - Try smaller sets of features.
        - Try increasing lambda (regularization).
    - Fixes High bias:
        - Try getting additional features.
        - Try adding polynomial features.
        - Try decreasing lambda (regularization).
- Evaluating a model.
    - Split data into training set and test set (i.e., 70%, 30%).
    - Terminology:
        - m_{train}: number of training examples.
        - Denote the training examples by: $(x^{(1)}, y^{(1)})$, $(x^{(2)}, y^{(2)})$, …, $(x^{(m)}, y^{(m)})$.
        - m_{test}: number of test examples.
        - Denote the test examples by: $(x_{test}^{(1)}, y_{test}^{(1)})$.
    - Train data (find parameters) using the training set and calculate the error using the test set.
    - 

      Train/test procedure for linear regression (with squared error cost)

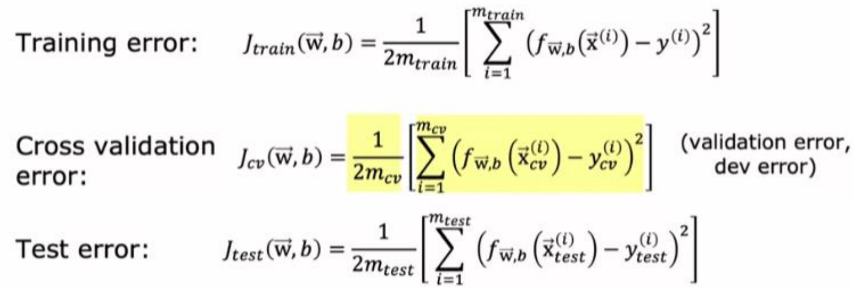
      Fit parameters by minimizing cost function $J(\vec{w},b)$

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

      Compute test error:

      $$J_{test}(\vec{w},b) = \frac{1}{2m_{test}}\left[\sum_{i=1}^{m_{test}}\left(f_{\vec{w},b}\left(\vec{x}_{test}^{(i)}\right) - y_{test}^{(i)}\right)^2\right] \quad \cancel{\sum_{j=1}^{n}} w_j^2$$

      Compute training error:

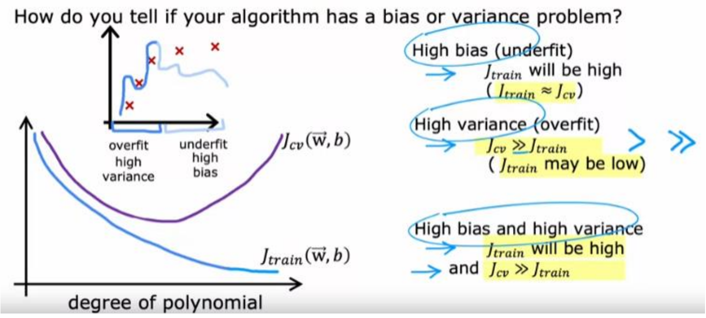
      $$J_{train}(\vec{w},b) = \frac{1}{2m_{train}}\left[\sum_{i=1}^{m_{train}}\left(f_{\vec{w},b}\left(\vec{x}_{train}^{(i)}\right) - y_{train}^{(i)}\right)^2\right]$$
    - If J_{test} is high, that is an indication that the model is not performing well.
- Model selection
    - Split data into three sets: Training (60%), Cross validation/Dev (20%), and Test (20%) sets.
    - 

      Training error: $J_{train}(\vec{w},b) = \frac{1}{2m_{train}}\left[\sum_{i=1}^{m_{train}}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)^2\right]$

      Cross validation error: $J_{cv}(\vec{w},b) = \frac{1}{2m_{cv}}\left[\sum_{i=1}^{m_{cv}}\left(f_{\vec{w},b}\left(\vec{x}_{cv}^{(i)}\right) - y_{cv}^{(i)}\right)^2\right]$ (validation error, dev error)

      Test error: $J_{test}(\vec{w},b) = \frac{1}{2m_{test}}\left[\sum_{i=1}^{m_{test}}\left(f_{\vec{w},b}\left(\vec{x}_{test}^{(i)}\right) - y_{test}^{(i)}\right)^2\right]$
    - Pick model with the lowest cross validation error.
    - Report generalization error using the test set.
    - If you don't use a cross validation set, your reported generalization error will be biased since we chose parameters which fit the test set.

Bias and Variance

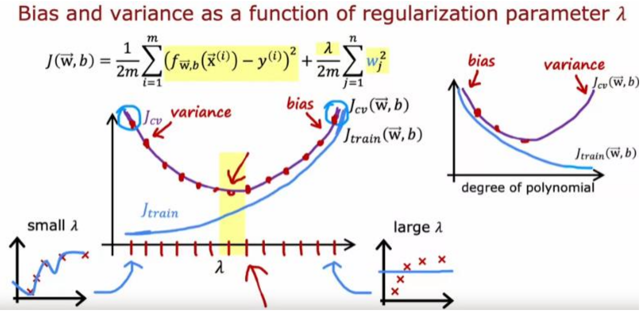
- Recall:
    - High bias: underfitting
    - High variance: overfitting
- When J_{train} and J_{cv} is high, this is usually an indicator of high bias.
- When J_{train} is low and J_{cv} is high, this is usually an indicator of high variance.
- If J_{train} is low and J_{cv} is also low, this usually indicates that the model is just right.
- Finding the right degree of polynomial is a balancing act:
- 

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

High bias (underfit)
→ J_{train} will be high
($J_{train} \approx J_{cv}$)

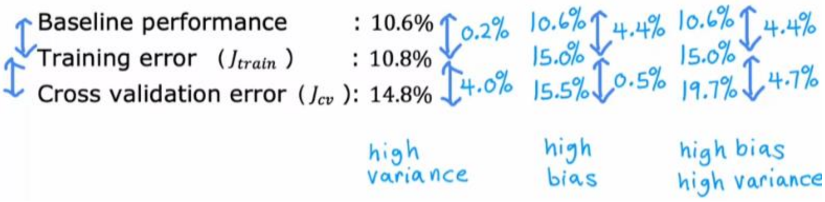
High variance (overfit)
→ $J_{cv} \gg J_{train}$ > ≫
(J_{train} may be low)

High bias and high variance
→ J_{train} will be high
→ and $J_{cv} \gg J_{train}$

- Regularization
    - For linear regression:
        - If Lambda is large, w parameters are small and f(x) = b (a straight line).
        - This means J_{train} will be large.
        - If Lambda is small, no regularization so just fitting a high degree polynomial.
        - This means J_{train} is small and J_{cv} is large (high variance, overfitting).
        - An intermediate value for lambda gives a small J_{train} and J_{cv}.
    - To choose a Lambda value, use trial and error.
    - 

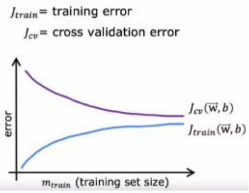
Bias and variance as a function of regularization parameter λ

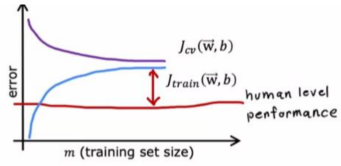
$$J(\vec{w}, b) = \frac{1}{2m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)^2 + \frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2$$

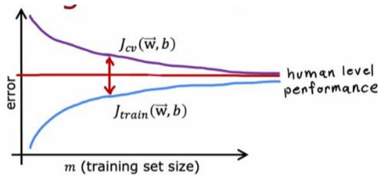
- Establishing a baseline level of performance – you can judge your algorithm according to the following:
    - Human level performance.
    - Competing algorithms performance.
    - Guess based on experience.
- 

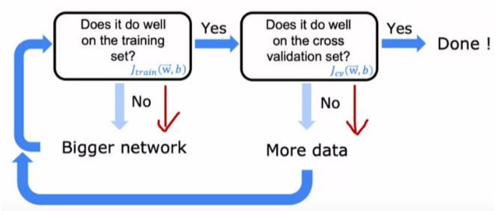
Baseline performance	: 10.6%	0.2%	10.6%	4.4%	10.6%	4.4%
Training error (J_{train})	: 10.8%		15.0%		15.0%	
Cross validation error (J_{cv})	: 14.8%	4.0%	15.5%	0.5%	19.7%	4.7%
	high variance		high bias		high bias high variance	

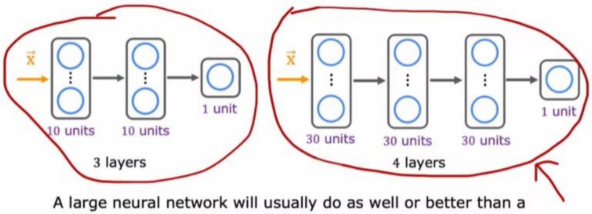
- Learning curves – plot your errors on y-axis, m_{train} on x-axis.

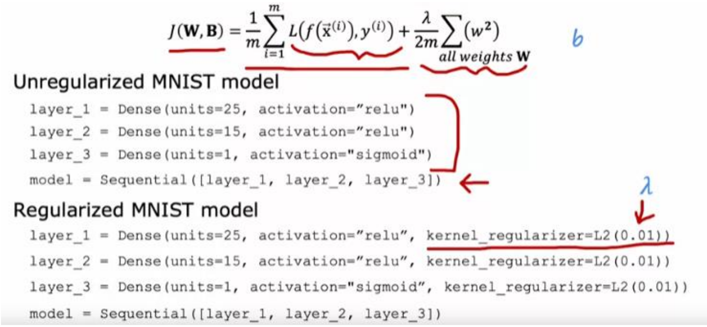
- 

- For high bias:
  - 

  - If a learning algorithm suffers from high bias, getting more training data will not help much.
- For high variance:
  - 

  - If a learning algorithm suffers from high variance, getting more training data is likely to help.
- Downside of plotting learning curves is that it is computationally expensive to train so many different models.
- Large enough neural networks always have low bias.
- Strategy for Neural networks:
  - 

- Comparison between small and large neural networks:
  - Intuitively, you would think that larger NNs are more prone to overfitting, but that is not the case.
  - 

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

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

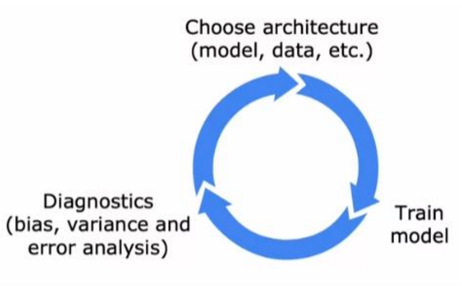
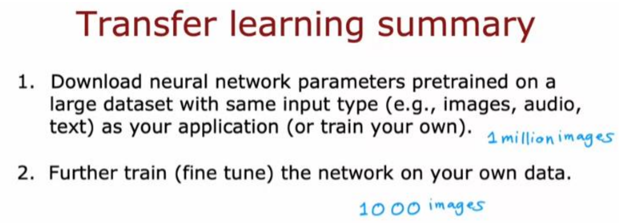
Unregularized MNIST model

```
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])
```

Regularized MNIST model

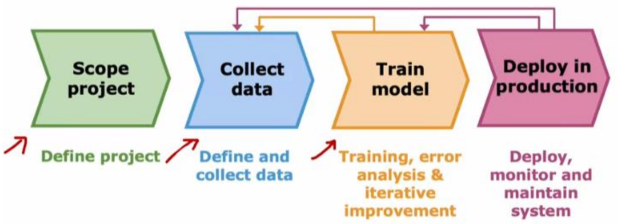
```
layer_1 = Dense(units=25, activation="relu", kernel_regularizer=L2(0.01))
layer_2 = Dense(units=15, activation="relu", kernel_regularizer=L2(0.01))
layer_3 = Dense(units=1, activation="sigmoid", kernel_regularizer=L2(0.01))
model = Sequential([layer_1, layer_2, layer_3])
```

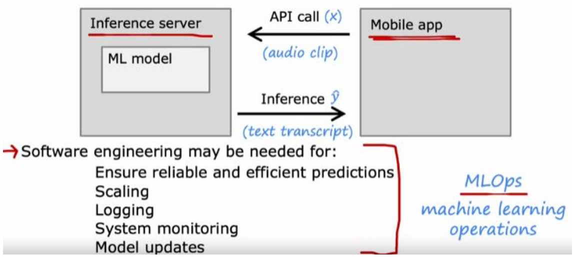
Development Process

- Steps:
    - 
Choose architecture (model, data, etc.)
Train model
Diagnostics (bias, variance and error analysis)
- Error analysis:
    - Manually examine the misclassified examples (or a subset if there are too many) and categorize them based on common traits.
    - The limitation is that it is only easy to do for tasks that humans are good at.
- Adding data:
    - If error analysis has shown that there are certain subsets of the data that the algorithm is doing poorly on, then getting more data of just those types will be sufficient and more efficient way to use your resources.
    - Data augmentation: modifying an existing training example to create a new training example i.e., a new example by distorting an image for handwriting classification.
    - Distortion introduced should be representation of the type of the noise/distortions in the test set.
    - Usually does not help to add purely random/meaningless noise to your data.
    - Data synthesis: using artificial data inputs to create a new training example (usually used in computer vision).
- Transfer learning – using data from a different task.
    - Make a copy of an existing trained NN (with many outputs) and replace the output layer (with less outputs) with your desired outputs.
        - Option 1: only train output layers parameters.
        - Option 2: train all parameters.
    - Training the larger model first is called "supervised pretraining".
    - The second step of replacing the output layer and retraining is called "fine tuning".
    - The nice thing about transfer learning is that you might not need to do the pretraining step – you can find existing models on the internet/get someone else to do it first.
    - 

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. 1000 images

- Full cycle of a ML project:
    - 

- Deployment:
    - 

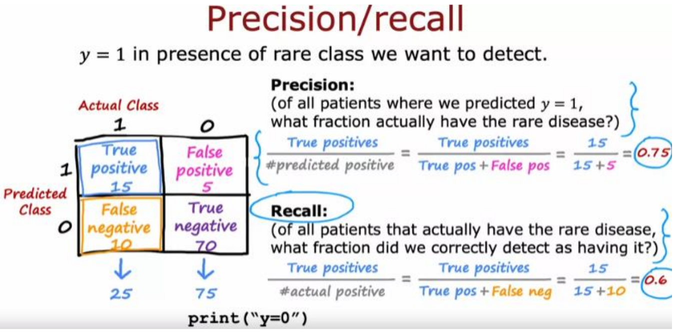
→Software engineering may be needed for:
Ensure reliable and efficient predictions
Scaling
Logging
System monitoring
Model updates

MLOps
machine learning operations

Fairness, Bias, and Ethics

- Guidelines:
    - Get a diverse team to brainstorm things that might go wrong, with emphasis on possible harm to vulnerable groups.
    - Carry out literature search on standards/guidelines for your industry.
    - Audit systems against possible harm prior to deployment.
    - Develop mitigation plan (if applicable), and after deployment, monitor for possible harm.

Skewed Datasets

- If your test error is larger than the true positive rate, you can’t really tell if your algorithm is useful or not.
- 

Precision/recall

$y = 1$ in presence of rare class we want to detect.

Predicted Class	Actual Class 1	Actual Class 0
1	True positive 15	False positive 5
0	False negative 10	True negative 70
	25	75

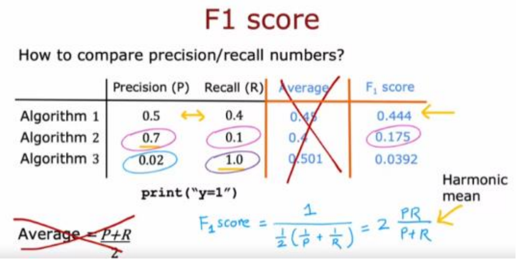
print(“y=0”)

Precision:
(of all patients where we predicted $y = 1$, what fraction actually have the rare disease?)

$$\frac{\text{True positives}}{\#\text{predicted positive}} = \frac{\text{True positives}}{\text{True pos} + \text{False pos}} = \frac{15}{15+5} = 0.75$$

Recall:
(of all patients that actually have the rare disease, what fraction did we correctly detect as having it?)

$$\frac{\text{True positives}}{\#\text{actual positive}} = \frac{\text{True positives}}{\text{True pos} + \text{False neg}} = \frac{15}{15+10} = 0.6$$

- Using precision/recall will help you tell if your algorithm is useful.
- For a useless algorithm like print(“y=0”), this performs poorly on Recall.
- Values close to 1.0 for both precision/recall is good.
- There is a trade-off between precision and recall.
    - Suppose in Logistic regression, we raise the threshold from 0.5 to 0.7 (predict positive only if very confident):
        - This results in higher precision, lower recall.
    - Suppose we lower the threshold from 0.5 to 0.3 (avoid missing too many cases):
        - This results in lower precision, higher recall.
    - How to compare algorithms with different precision/recall values?
    - 

F1 score

How to compare precision/recall numbers?

	Precision (P)	Recall (R)	~~Average~~	F_1 score
Algorithm 1	0.5	0.4	~~0.45~~	0.444
Algorithm 2	0.7	0.1	~~0.4~~	0.175
Algorithm 3	0.02	1.0	~~0.501~~	0.0392

print(“y=1”)

~~Average = $\frac{P+R}{2}$~~

$$F_1 \text{ score} = \frac{1}{\frac{1}{2}\left(\frac{1}{P} + \frac{1}{R}\right)} = 2\frac{PR}{P+R}$$

Harmonic mean

    - This metric emphasizes the lower of the two scores.

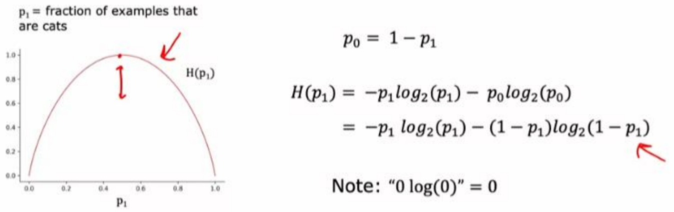
Week 7

Decision Trees

- Make a series of decisions to classify/predict an input based on how the tree is structured.
- A decision tree consists of root node, decision nodes, and leaf nodes.
- The goal is to pick a decision tree that performs well on training data and generalizes well.
- The learning algorithm tries to iteratively split the examples into their correct groups/classes until all the examples in a group are the same – then put them all in the same leaf node.
- How to choose what feature to split on at each node?
    - Maximize purity (or minimize impurity).
- When do you stop splitting?
    - When a node is 100% one class.
    - When splitting a node will result in the tree exceeding a maximum depth.
    - Improvements in a purity score are below a threshold.
    - Number of examples in a node are below a threshold.

Decision Tree Learning

- Entropy is a measure of impurity – a function of the fraction of examples that are a certain class i.e., the fraction of examples that are classified as cats.

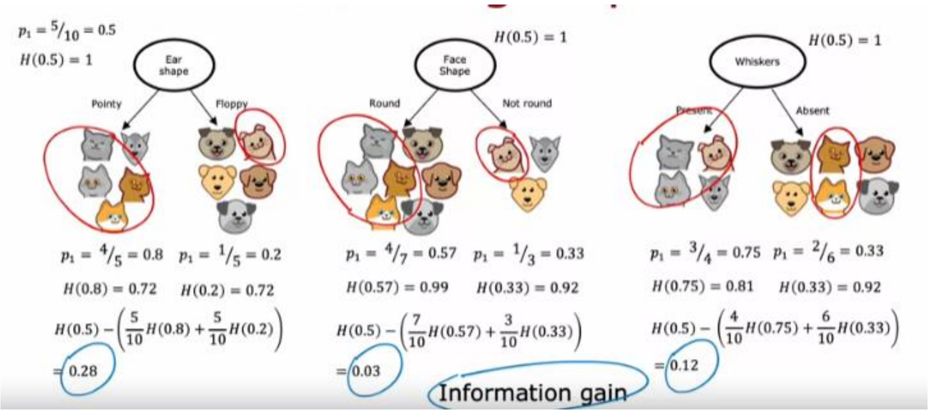


$$p_0 = 1 - p_1$$

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

Note: "0 log(0)" = 0

- Deciding which feature to split on at a node will be based on what choice of feature reduces entropy the most.
- The reduction of entropy is called information gain.
- An example of calculating and choosing the highest information gain split:



$p_1 = {}^5/_{10} = 0.5$

$H(0.5) = 1$

$p_1 = {}^4/_5 = 0.8$ $p_1 = {}^1/_5 = 0.2$

$H(0.8) = 0.72$ $H(0.2) = 0.72$

$H(0.5) - \left(\frac{5}{10}H(0.8) + \frac{5}{10}H(0.2)\right)$

$= 0.28$

$H(0.5) = 1$

$p_1 = {}^4/_7 = 0.57$ $p_1 = {}^1/_3 = 0.33$

$H(0.57) = 0.99$ $H(0.33) = 0.92$

$H(0.5) - \left(\frac{7}{10}H(0.57) + \frac{3}{10}H(0.33)\right)$

$= 0.03$

$H(0.5) = 1$

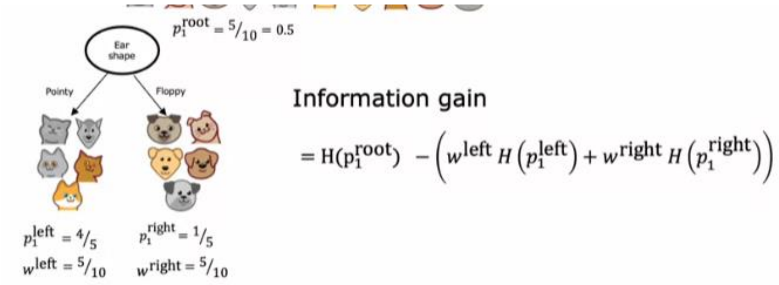
$p_1 = {}^3/_4 = 0.75$ $p_1 = {}^2/_6 = 0.33$

$H(0.75) = 0.81$ $H(0.33) = 0.92$

$H(0.5) - \left(\frac{4}{10}H(0.75) + \frac{6}{10}H(0.33)\right)$

$= 0.12$

- General formula for information gain:



$p_1^{root} = {}^5/_{10} = 0.5$

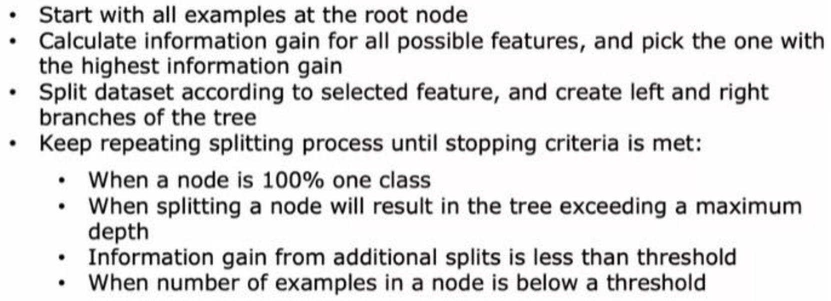
$p_1^{left} = {}^4/_5$ $p_1^{right} = {}^1/_5$

$w^{left} = {}^5/_{10}$ $w^{right} = {}^5/_{10}$

Information gain

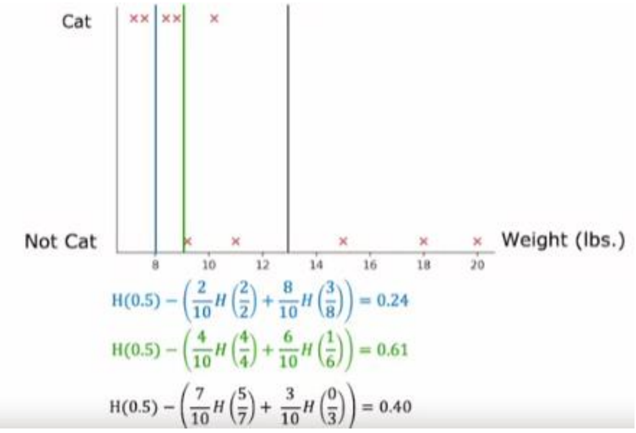
$$= H(p_1^{root}) - \left(w^{left} H\left(p_1^{left}\right) + w^{right} H\left(p_1^{right}\right)\right)$$

- Algorithm pseudocode (uses recursion):



- 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.
    - If you have features with more than 2 categories, you can use one hot encoding to reduce it to binary classification.
    - Definition: if a categorical feature can take on k values, create k binary features (0 or 1 valued).
- Continuous valued features.
    - Try out different thresholds and pick the one with the highest information gain.
    - Example:

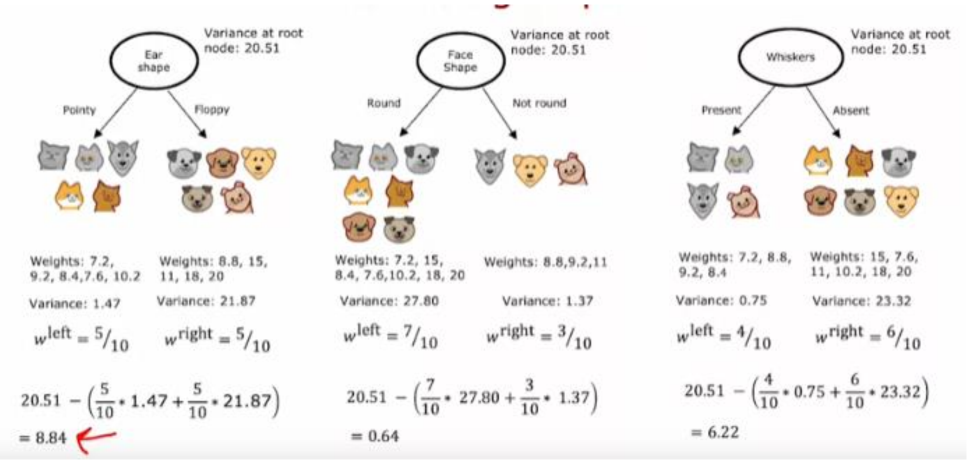


$$H(0.5) - \left(\frac{2}{10}H\left(\frac{2}{2}\right) + \frac{8}{10}H\left(\frac{3}{8}\right)\right) = 0.24$$

$$H(0.5) - \left(\frac{4}{10}H\left(\frac{4}{4}\right) + \frac{6}{10}H\left(\frac{1}{6}\right)\right) = 0.61$$

$$H(0.5) - \left(\frac{7}{10}H\left(\frac{5}{7}\right) + \frac{3}{10}H\left(\frac{0}{3}\right)\right) = 0.40$$

    -
- Regression trees.
    - Predicted output is a continuous value.
    - The output is the average of all target values at the leaf node.
    - Instead of maximizing information gain at each split, when constructing the decision tree, we look to minimize the variance of the continuous target values.
    - Like computing information gain, pick the highest reduction in the weighted variances of the left and right branches (like that done in classification).
    - Example:


Ear shape
Variance at root node: 20.51
Pointy
Floppy
Face Shape
Variance at root node: 20.51
Round
Not round
Whiskers
Variance at root node: 20.51
Present
Absent

Weights: 7.2, 9.2, 8.4,7.6, 10.2 | Weights: 8.8, 15, 11, 18, 20 | Weights: 7.2, 15, 8.4, 7.6,10.2, 18, 20 | Weights: 8.8,9.2,11 | Weights: 7.2, 8.8, 9.2, 8.4 | Weights: 15, 7.6, 11, 10.2, 18, 20

Variance: 1.47 | Variance: 21.87 | Variance: 27.80 | Variance: 1.37 | Variance: 0.75 | Variance: 23.32

$w^{\text{left}} = 5/10$ $w^{\text{right}} = 5/10$ | $w^{\text{left}} = 7/10$ $w^{\text{right}} = 3/10$ | $w^{\text{left}} = 4/10$ $w^{\text{right}} = 6/10$

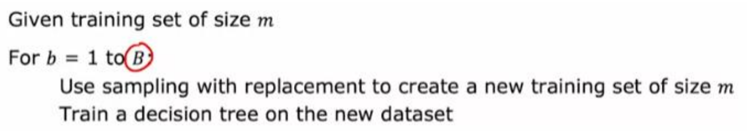
$$20.51 - \left(\frac{5}{10} * 1.47 + \frac{5}{10} * 21.87\right) = 8.84 \leftarrow$$

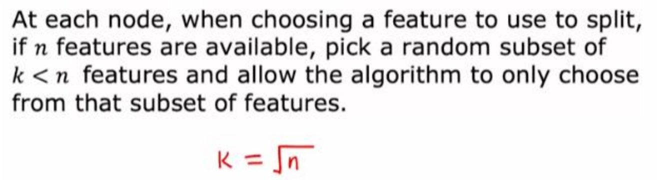
$$20.51 - \left(\frac{7}{10} * 27.80 + \frac{3}{10} * 1.37\right) = 0.64$$

$$20.51 - \left(\frac{4}{10} * 0.75 + \frac{6}{10} * 23.32\right) = 6.22$$

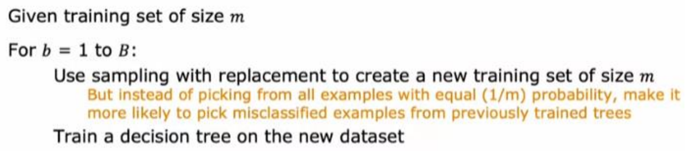
    -

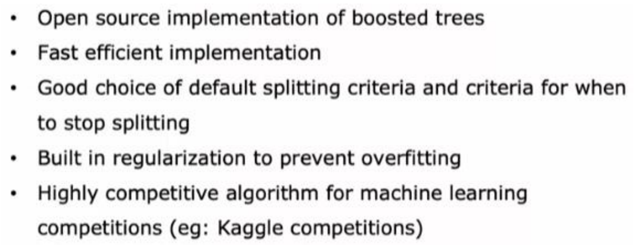
Tree Ensembles

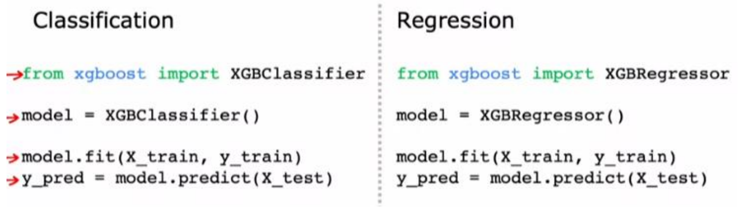
- Single trees are highly sensitive to small changes of data.
- Changing one training example causes the algorithm to come up with a different split at the root and therefore a totally different tree.
- We often get more accurate results if we train a collection of trees.
- Essentially running the examples through many trees and taking a majority vote of the predictions.
- Bagged Decision Tree:
  - 

    ```
    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
    ```
  - Where B is a number can be any number (typically from 64 to 228).
  - Never hurts to have a large B, but up to a certain point, there will be diminishing returns and hurts computation time.
- Random Forest Algorithm:
  - Randomizing the feature choice helps to make the decision trees more different from each other (good thing!).
    - 

      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.

      $K = \sqrt{n}$
- Boosted Decision Trees.
  - Modification to the random forest algorithm.
  - 

    ```
    Given training set of size $m$
    For $b = 1$ to $B$:
        Use sampling with replacement to create a new training set of size $m$
            But instead of picking from all examples with equal (1/m) probability, make it
            more likely to pick misclassified examples from previously trained trees
        Train a decision tree on the new dataset
    ```
  - Note that the previously trained trees will be run on the original training set (not the one generated through sampling).
  - The most popular today is XGBoost.
    - 

      - 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)
    - Example using classification/regression:
    - 

      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)
      ```
- When to use decision trees?
  - Works well on tabular (structured) data i.e., data looks like a giant spreadsheet.
  - Not recommended for unstructured data (images, audio, text).

- Tree ensembles are fast to train.
- Small decision trees may be human interpretable.
- When to use Neural Networks instead?
    - Works well on all types of data, including tabular (structured) and unstructured data.
    - May be slower than a decision tree.
    - Works with transfer learning (good when you only have a small data set).
    - When building a system of multiple models working together, easier to splice together multiple neural networks.

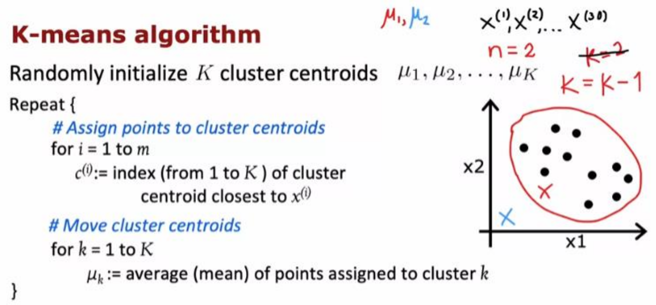
Week 8

Clustering

- Looks at several data points and automatically finds data points that are related or like each other.
- Not given target labels, y – not able to tell algorithm the right answer.
- Rather, we ask the algorithm to find an interesting structure in the data.

k-means Algorithm

- Step 1 (assign each point to its closest centroid):
  - Take a random guess of k points, which are the centre of k clusters.
  - Cluster centroid: The centre of the cluster.
  - Iterate over each data point and determine the distance to each cluster centroid.
  - Algorithm assigns each data point to each cluster centroid group.
- Step 2 (recompute the centroids):
  - Take the average (centre) of all points in each cluster and move the guess there.
  - Repeat step 1 (from the iterating over each point sub-step).
- When no further changes, the solution has converged, and the algorithm terminates.



K-means algorithm

Randomly initialize K cluster centroids $\mu_1, \mu_2, \ldots, \mu_K$

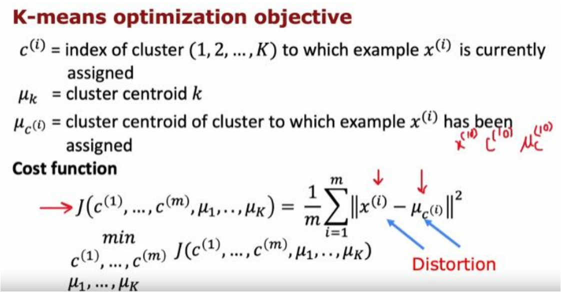
```
Repeat {
    # Assign points to cluster centroids
    for $i$ = 1 to $m$
        $c^{(i)}$ := index (from 1 to $K$) of cluster
                 centroid closest to $x^{(i)}$
    # Move cluster centroids
    for $k$ = 1 to $K$
        $\mu_k$ := average (mean) of points assigned to cluster $k$
}
```

μ_1, μ_2 $x^{(1)}, x^{(2)}, \ldots x^{(30)}$

$n = 2$ ~~$K=2$~~

$K = K - 1$

-
- Edge case: if a cluster is assigned 0 points, it is common to just eliminate that cluster and set k = k – 1.
- Alternatively, you can randomly re-initialize that cluster centroid.
- Optimization objective.
  - Not gradient descent.



K-means optimization objective

$c^{(i)}$ = index of cluster $(1, 2, \ldots, K)$ to which example $x^{(i)}$ is currently assigned

μ_k = cluster centroid k

$\mu_{c^{(i)}}$ = cluster centroid of cluster to which example $x^{(i)}$ has been assigned

$x^{(10)}$ $c^{(10)}$ $\mu_c^{(10)}$

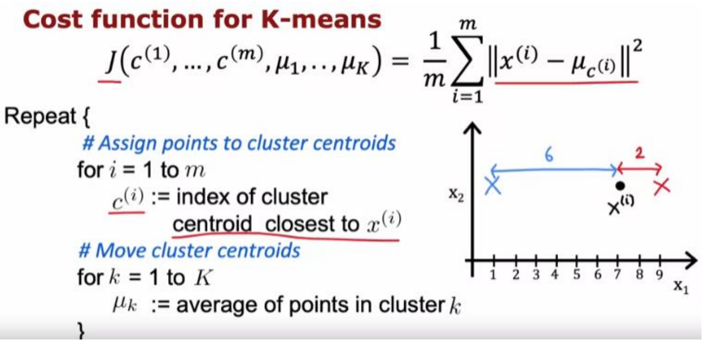
Cost function

$$\rightarrow J(c^{(1)}, \ldots, c^{(m)}, \mu_1, \ldots, \mu_K) = \frac{1}{m}\sum_{i=1}^{m} \| \overset{\downarrow}{x^{(i)}} - \overset{\downarrow}{\mu_{c^{(i)}}} \|^2$$

$$\min_{\substack{c^{(1)}, \ldots, c^{(m)} \\ \mu_1, \ldots, \mu_K}} J(c^{(1)}, \ldots, c^{(m)}, \mu_1, \ldots, \mu_K)$$

Distortion

  -
  - The goal of step 1 is to minimize x$^{(i)}$ while holding mu$_c$$^{(i)}$ constant.
  - The goal of step 2 is to minimize mu$_c$$^{(i)}$ while keeping x$^{(i)}$ constant.

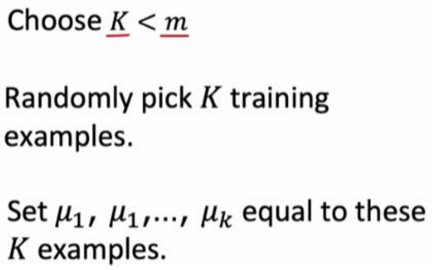
- 

  Cost function for K-means

  $$J(c^{(1)},...,c^{(m)},\mu_1,..,\mu_K) = \frac{1}{m}\sum_{i=1}^{m}\left\|x^{(i)}-\mu_{c^{(i)}}\right\|^2$$

  ```
  Repeat {
      # Assign points to cluster centroids
      for $i$ = 1 to $m$
          $c^{(i)}$ := index of cluster
              centroid closest to $x^{(i)}$
      # Move cluster centroids
      for $k$ = 1 to $K$
          $\mu_k$ := average of points in cluster $k$
  }
  ```

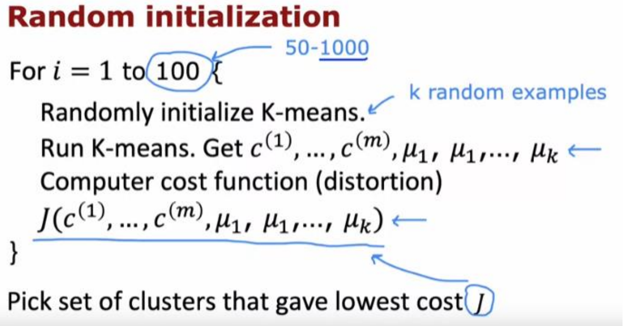
- On every iteration, the cost function should decrease or stay the same.
- If it ever goes up, there is a bug in the code because it should decrease at every step.

- Random initialization.
  - How to choose good starting points for cluster centroids?
  - 

    Choose $K < m$

    Randomly pick K training examples.

    Set $\mu_1, \mu_1,..., \mu_k$ equal to these K examples.

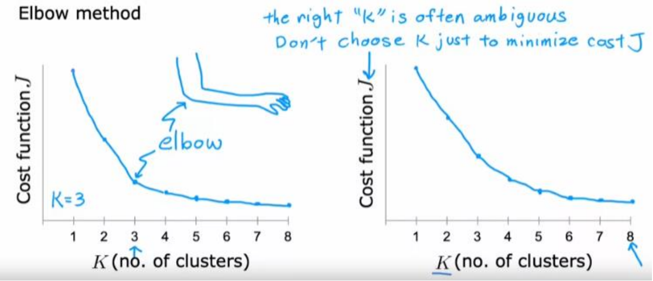
  - You don't always get the optimal placement with a single random choice, and this can affect the result cluster locations.
  - You can run k-means multiple times with different random starting points and compute the cost J for each – you pick the cost J which is the smallest.
  - 

    Random initialization

    ```
    For $i = 1$ to 100 {
        Randomly initialize K-means.
        Run K-means. Get $c^{(1)},...,c^{(m)},\mu_1, \mu_1,..., \mu_k$
        Computer cost function (distortion)
        $J(c^{(1)},...,c^{(m)},\mu_1, \mu_1,..., \mu_k)$
    }
    Pick set of clusters that gave lowest cost $J$
    ```

    50-1000

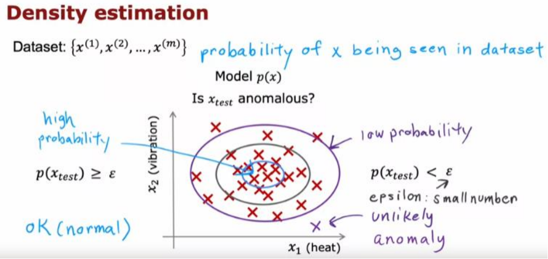
    k random examples

- Choosing the number of clusters, k.
  - Elbow method
    - 

    - Sometimes difficult because there is often no clear "elbow" as in the plot on the right.
  - Evaluate k-means based on how well it performs on the later downstream method.
    - If you need 3 clusters for the downstream application, then choose k=3.

Anomaly Detection

- Density Estimation.
  - Build a model of the probability that the event you are checking for exists.
  - Compare the new example (x_{test}) to the model.



Density estimation

Dataset: $\{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}$

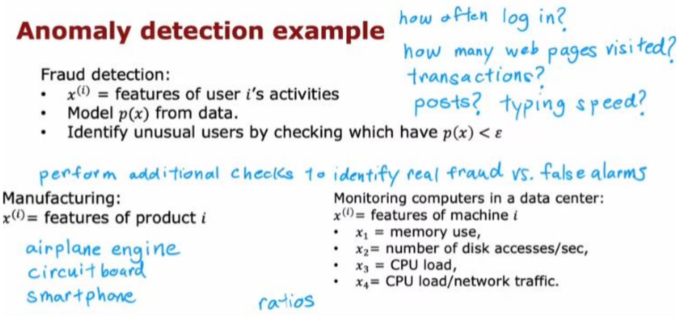
Model $p(x)$

Is x_{test} anomalous?

$p(x_{test}) \geq \varepsilon$

$p(x_{test}) < \varepsilon$

  - Examples of anomaly detection:



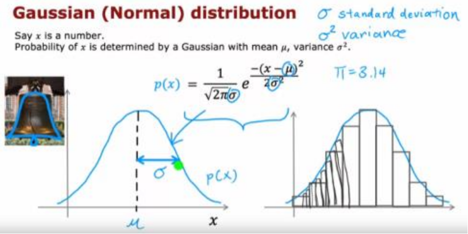
Anomaly detection example

Fraud detection:
- $x^{(i)}$ = features of user i's activities
- Model $p(x)$ from data.
- Identify unusual users by checking which have $p(x) < \varepsilon$

Manufacturing:
$x^{(i)}$= features of product i

Monitoring computers in a data center:
$x^{(i)}$= features of machine i
- x_1 = memory use,
- x_2= number of disk accesses/sec,
- x_3 = CPU load,
- x_4= CPU load/network traffic.

- Gaussian (normal) distribution.

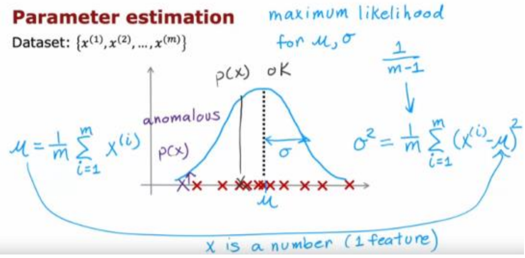


Gaussian (Normal) distribution

Say x is a number.
Probability of x is determined by a Gaussian with mean μ, variance σ^2.

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

  - Anomaly detection tries to estimate the parameters based on the training set.
  - Formula for computing parameters:

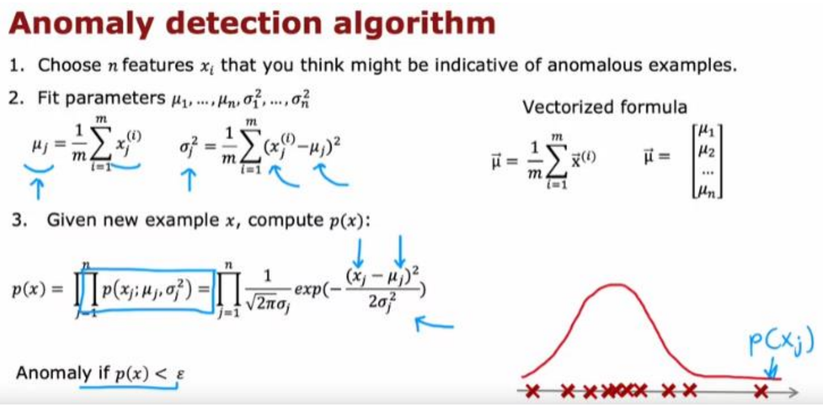


Parameter estimation

Dataset: $\{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}$

maximum likelihood for μ, σ
$\frac{1}{m-1}$
p(x) ok
anomalous
p(x)
σ
μ
$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$
$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} (x^{(i)}-\mu)^2$
x is a number (1 feature)

- Algorithm.



Anomaly detection algorithm

1. Choose n features x_i that you think might be indicative of anomalous examples.
2. Fit parameters $\mu_1, \ldots, \mu_n, \sigma_1^2, \ldots, \sigma_n^2$

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)} \qquad \sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m} (x_j^{(i)} - \mu_j)^2$$

Vectorized formula

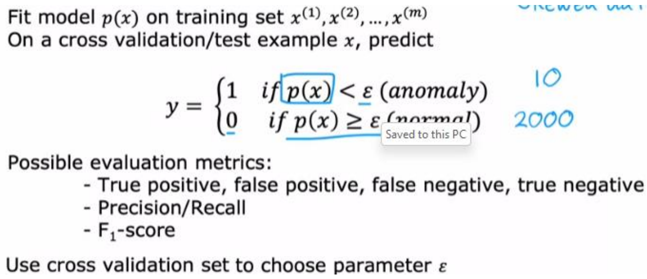
$$\vec{\mu} = \frac{1}{m}\sum_{i=1}^{m} \vec{x}^{(i)} \qquad \vec{\mu} = \begin{bmatrix} \mu_1 \\ \mu_2 \\ \cdots \\ \mu_n \end{bmatrix}$$

3. Given new example x, compute $p(x)$:

$$p(x) = \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2) = \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_j} exp(-\frac{(x_j - \mu_j)^2}{2\sigma_j^2})$$

Anomaly if $p(x) < \varepsilon$

  - Essentially computing the gaussian distribution for each feature and multiplying together the probabilities for a test data point.
- Practical tips.
  - It is helpful to (hopefully) be able to collect some test and cross validation data set with target labels.
  - Train algorithm on training set.
  - Use cross validation set to tune epsilon and x_j parameters.
  - Alternative: no test set, just tune epsilon and x_j parameters with a larger cross validation set.
    - Downside is after training; you don't have a test set.
    - But this is useful if you don't have enough data to create a separate test set.
  - Algorithm evaluation:



Fit model $p(x)$ on training set $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$

On a cross validation/test example x, predict

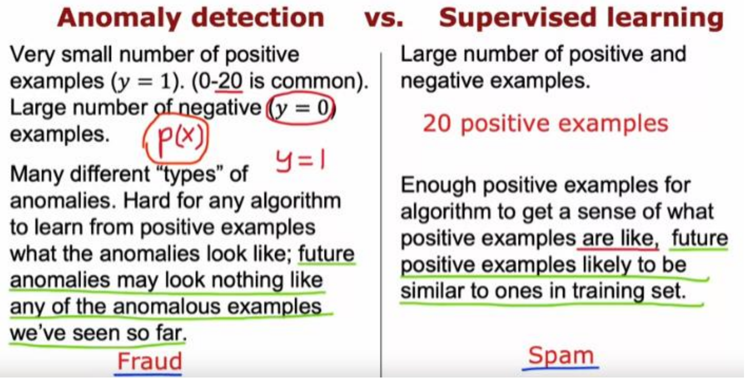
$$y = \begin{cases} 1 & if\ \boxed{p(x)} < \underline{\varepsilon}\ (anomaly) \\ \underline{0} & if\ \underline{p(x) \geq \varepsilon}\ (normal) \end{cases} \quad \begin{matrix} 10 \\ 2000 \end{matrix}$$

Possible evaluation metrics:

- True positive, false positive, false negative, true negative
- Precision/Recall
- F_1-score

Use cross validation set to choose parameter ε

- Anomaly Detection vs. Supervised Learning.



Anomaly detection vs. Supervised learning

Very small number of positive examples ($y = 1$). (0-20 is common). Large number of negative ($y = 0$) examples.

p(x)

y=1

Many different "types" of anomalies. Hard for any algorithm to learn from positive examples what the anomalies look like; future anomalies may look nothing like any of the anomalous examples we've seen so far.

Fraud

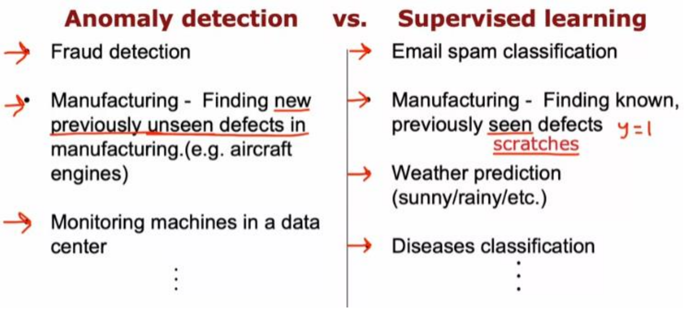
Large number of positive and negative examples.

20 positive examples

Enough positive examples for algorithm to get a sense of what positive examples are like, future positive examples likely to be similar to ones in training set.

Spam

  - Different use cases:



Anomaly detection vs. Supervised learning

→ Fraud detection

→ Manufacturing - Finding new previously unseen defects in manufacturing.(e.g. aircraft engines)

→ Monitoring machines in a data center

⋮

→ Email spam classification

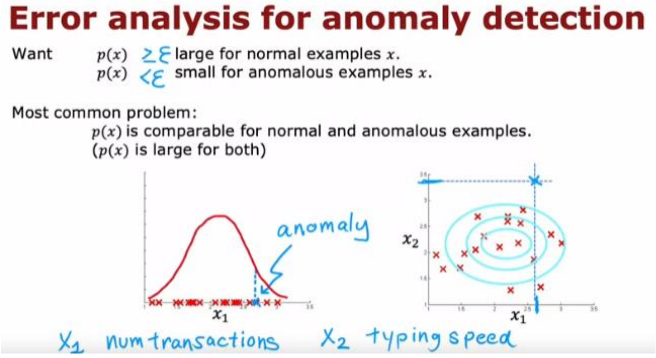
→ Manufacturing - Finding known, previously seen defects y=1 scratches

→ Weather prediction (sunny/rainy/etc.)

→ Diseases classification

⋮

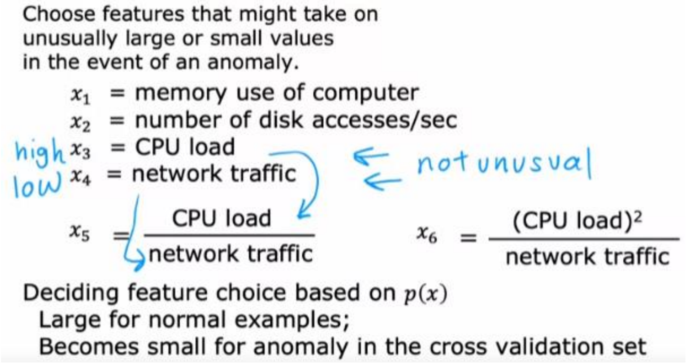
- Choosing the right features to use.
  - For unsupervised learning, it is harder for the algorithm to figure out which features to use and which to ignore, since it has no target label, y provided.
  - Carefully choosing features is more important for unsupervised learning than supervised learning algorithms.
  - Change your features to be more "gaussian" – transform skewed data sets (maybe by taking the log(x) or log(x+constant) or sqrt(x) of the x-axis values).
  - Error analysis for anomaly detection.
    - p(x) is comparable for both normal and anomalous examples.

- Can try to add new features which may help to separate out anomalous values from normal ones.
- 

  ## Error analysis for anomaly detection

  Want $p(x)$ $\geq \varepsilon$ large for normal examples x.
  $p(x)$ $< \varepsilon$ small for anomalous examples x.

  Most common problem:
  $p(x)$ is comparable for normal and anomalous examples.
  ($p(x)$ is large for both)

  X_1 num transactions X_2 typing speed
- An example of using new features for a data centre monitoring.
- 

  Choose features that might take on unusually large or small values in the event of an anomaly.

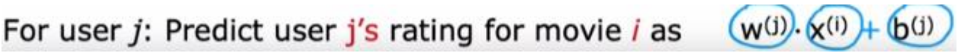
  x_1 = memory use of computer
  x_2 = number of disk accesses/sec
  x_3 = CPU load
  x_4 = network traffic

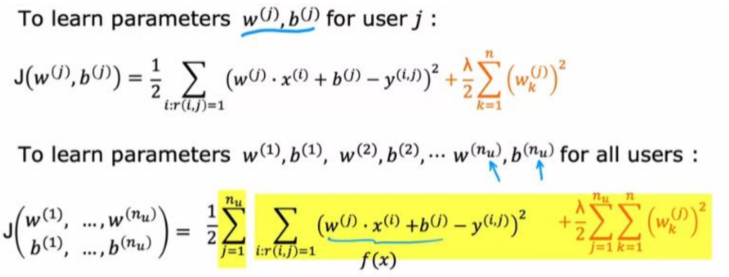
  $$x_5 = \frac{\text{CPU load}}{\text{network traffic}} \qquad x_6 = \frac{(\text{CPU load})^2}{\text{network traffic}}$$

  Deciding feature choice based on $p(x)$
  Large for normal examples;
  Becomes small for anomaly in the cross validation set

Week 9

Collaborative Filtering

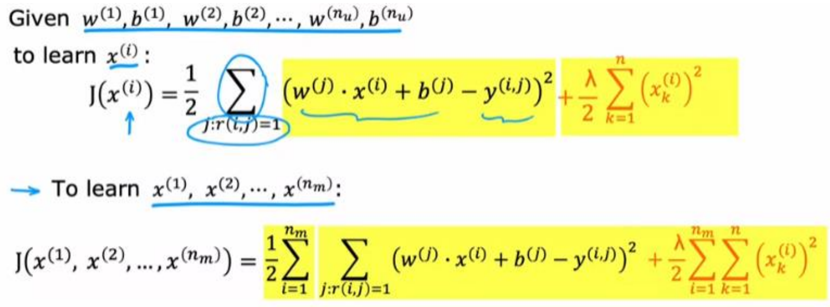
- Terminology:
    - n$_u$: number of users.
    - n$_m$: number of items (to recommend to the users).
    - r(i,j): value of 1 if user j has rated item i.
    - y$^{(i,j)}$: rating given by user j to item I (defined only if r(i,j)=1).
    - n: number of features for each item.
    - x$^{(i)}$: features for item i (stored as a vector of features for that item).
- 

  For user j: Predict user j's rating for movie i as $\boxed{w^{(j)}}\cdot\boxed{x^{(i)}}+\boxed{b^{(j)}}$
- w$^{(j)}$ and b$^{(j)}$ belong to a particular user.
- This is exactly like linear regression.
- Cost function:
    - 

      To learn parameters $\underline{w^{(j)}, b^{(j)}}$ for user j :

      $$J(w^{(j)},b^{(j)}) = \frac{1}{2}\sum_{i:r(i,j)=1}\left(w^{(j)}\cdot x^{(i)}+b^{(j)}-y^{(i,j)}\right)^2 + \frac{\lambda}{2}\sum_{k=1}^{n}\left(w_k^{(j)}\right)^2$$

      To learn parameters $w^{(1)},b^{(1)},\ w^{(2)},b^{(2)},\cdots\ w^{(n_u)},b^{(n_u)}$ for all users :

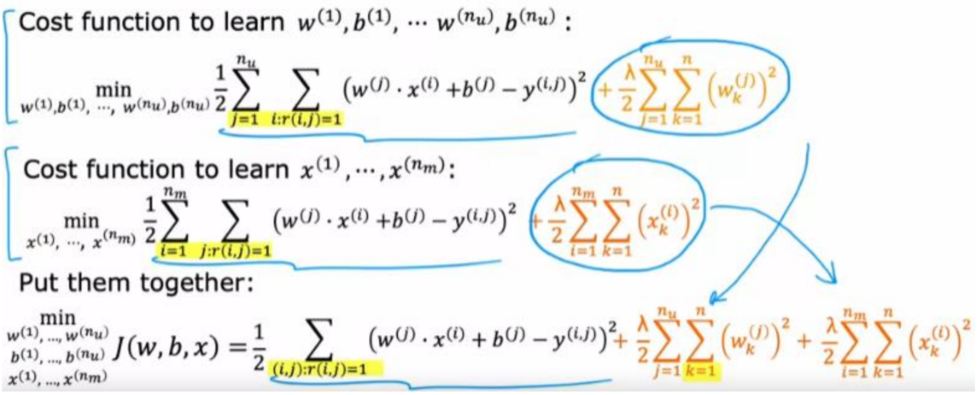
      $$J\begin{pmatrix}w^{(1)}, & \dots, w^{(n_u)}\\ b^{(1)}, & \dots, b^{(n_u)}\end{pmatrix} = \frac{1}{2}\sum_{j=1}^{n_u}\underbrace{\sum_{i:r(i,j)=1}\left(w^{(j)}\cdot x^{(i)}+b^{(j)}-y^{(i,j)}\right)^2}_{f(x)} + \frac{\lambda}{2}\sum_{j=1}^{n_u}\sum_{k=1}^{n}\left(w_k^{(j)}\right)^2$$
    - Like linear regression except without dividing by the m term.
    - Can use gradient descent or some other algorithm to minimize the cost function.
- What if you don't have values for features (x$^{(i)}$)?
    - Assuming you have w, b parameters for each user:
    - 

      Given $\underline{w^{(1)},b^{(1)},\ w^{(2)},b^{(2)},\cdots,\ w^{(n_u)},b^{(n_u)}}$

      to learn $\underline{x^{(i)}}$:

      $$J(x^{(i)}) = \frac{1}{2}\sum_{j:r(i,j)=1}\left(w^{(j)}\cdot x^{(i)}+b^{(j)}-y^{(i,j)}\right)^2 + \frac{\lambda}{2}\sum_{k=1}^{n}\left(x_k^{(i)}\right)^2$$

      → To learn $\underline{x^{(1)},\ x^{(2)},\cdots,\ x^{(n_m)}}$:

      $$J(x^{(1)},\ x^{(2)},\dots,x^{(n_m)}) = \frac{1}{2}\sum_{i=1}^{n_m}\sum_{j:r(i,j)=1}\left(w^{(j)}\cdot x^{(i)}+b^{(j)}-y^{(i,j)}\right)^2 + \frac{\lambda}{2}\sum_{i=1}^{n_m}\sum_{k=1}^{n}\left(x_k^{(i)}\right)^2$$
    - This helps to get features for each item.
- Collaborative Filtering.
    - Add the two cost functions above together to obtain:
    - 

      Cost function to learn $w^{(1)},b^{(1)},\cdots\ w^{(n_u)},b^{(n_u)}$:

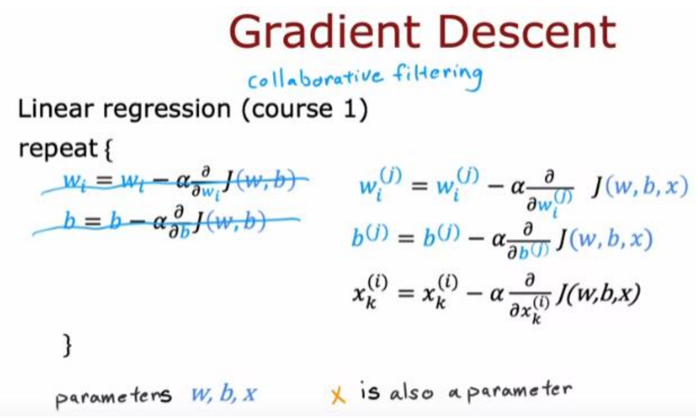
      $$\min_{w^{(1)},b^{(1)},\dots,w^{(n_u)},b^{(n_u)}}\frac{1}{2}\sum_{j=1}^{n_u}\sum_{i:r(i,j)=1}\left(w^{(j)}\cdot x^{(i)}+b^{(j)}-y^{(i,j)}\right)^2 + \frac{\lambda}{2}\sum_{j=1}^{n_u}\sum_{k=1}^{n}\left(w_k^{(j)}\right)^2$$

      Cost function to learn $x^{(1)},\cdots,x^{(n_m)}$:

      $$\min_{x^{(1)},\dots,x^{(n_m)}}\frac{1}{2}\sum_{i=1}^{n_m}\sum_{j:r(i,j)=1}\left(w^{(j)}\cdot x^{(i)}+b^{(j)}-y^{(i,j)}\right)^2 + \frac{\lambda}{2}\sum_{i=1}^{n_m}\sum_{k=1}^{n}\left(x_k^{(i)}\right)^2$$

      Put them together:

      $$\min_{\substack{w^{(1)},\dots,w^{(n_u)}\\ b^{(1)},\dots,b^{(n_u)}\\ x^{(1)},\dots,x^{(n_m)}}} J(w,b,x) = \frac{1}{2}\sum_{(i,j):r(i,j)=1}\left(w^{(j)}\cdot x^{(i)}+b^{(j)}-y^{(i,j)}\right)^2 + \frac{\lambda}{2}\sum_{j=1}^{n_u}\sum_{k=1}^{n}\left(w_k^{(j)}\right)^2 + \frac{\lambda}{2}\sum_{i=1}^{n_m}\sum_{k=1}^{n}\left(x_k^{(i)}\right)^2$$
    - Use gradient descent (as in linear regression) to minimize the cost function.

- 

  Gradient Descent

  collaborative filtering

  Linear regression (course 1)

  repeat {

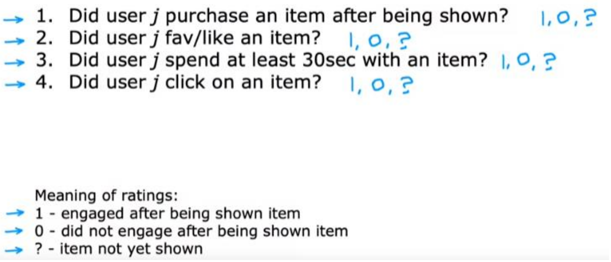
  ~~$w_i = w_i - \alpha \frac{\partial}{\partial w_i} J(w,b)$~~ $\quad w_i^{(j)} = w_i^{(j)} - \alpha \frac{\partial}{\partial w_i^{(j)}} J(w,b,x)$

  ~~$b = b - \alpha \frac{\partial}{\partial b} J(w,b)$~~ $\quad b^{(j)} = b^{(j)} - \alpha \frac{\partial}{\partial b^{(j)}} J(w,b,x)$

  $x_k^{(i)} = x_k^{(i)} - \alpha \frac{\partial}{\partial x_k^{(i)}} J(w,b,x)$

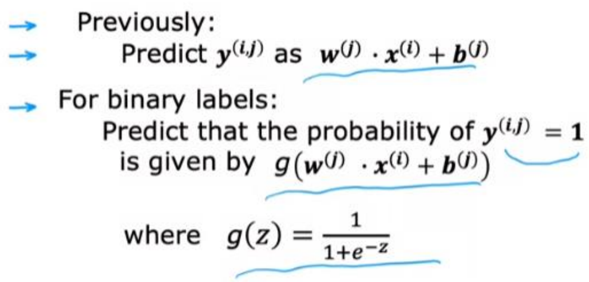
  }

  parameters w, b, x $\quad$ x is also a parameter

- Binary labels.
  - Example applications:
    - 

      1. Did user j purchase an item after being shown? 1, 0, ?
      2. Did user j fav/like an item? 1, 0, ?
      3. Did user j spend at least 30sec with an item? 1, 0, ?
      4. Did user j click on an item? 1, 0, ?

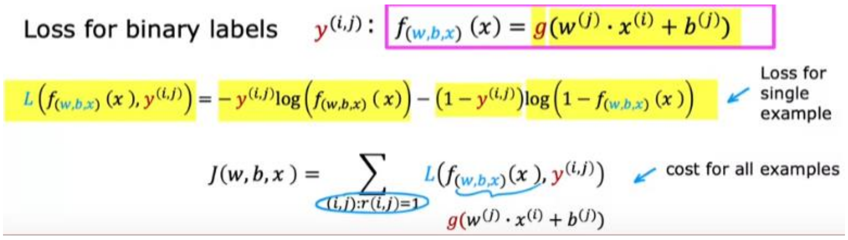
      Meaning of ratings:
      1 - engaged after being shown item
      0 - did not engage after being shown item
      ? - item not yet shown

  - From regression to binary classification:
    - 

      Previously:
      Predict $y^{(i,j)}$ as $\underline{w^{(j)} \cdot x^{(i)} + b^{(j)}}$

      For binary labels:
      Predict that the probability of $y^{(i,j)} = 1$
      is given by $\underline{g(w^{(j)} \cdot x^{(i)} + b^{(j)})}$

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

  - Cost function for binary application.
  - 

    Loss for binary labels $y^{(i,j)}$: $\boxed{f_{(w,b,x)}(x) = g(w^{(j)} \cdot x^{(i)} + b^{(j)})}$

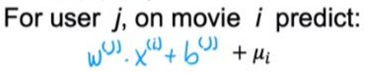
    $$L\left(f_{(w,b,x)}(x), y^{(i,j)}\right) = -y^{(i,j)} \log\left(f_{(w,b,x)}(x)\right) - \left(1 - y^{(i,j)}\right) \log\left(1 - f_{(w,b,x)}(x)\right)$$

    Loss for single example

    $$J(w,b,x) = \sum_{(i,j):r(i,j)=1} L\left(\underbrace{f_{(w,b,x)}(x)}_{g(w^{(j)} \cdot x^{(i)} + b^{(j)})}, y^{(i,j)}\right)$$

    cost for all examples

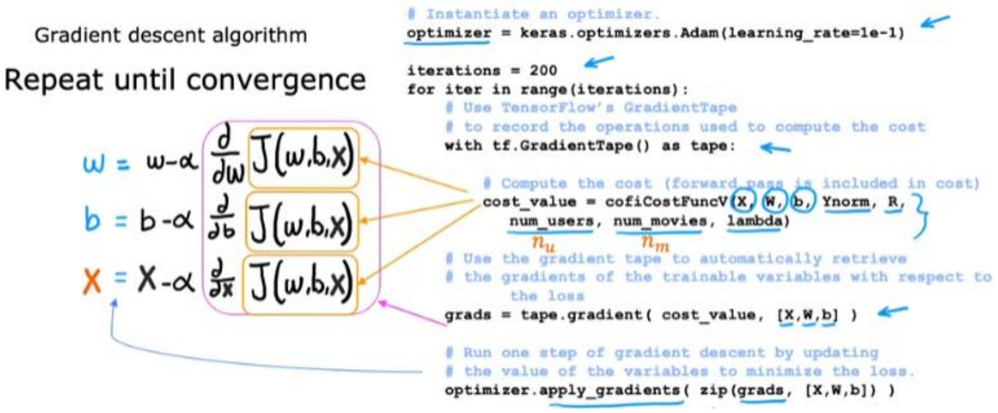
Recommender Systems Implementation Details

- As in the case for linear regression, feature normalization can help the algorithm perform better.
- Mean normalization:
  - Subtract each item’s average rating from each of its score given to it by a user.
  - This helps the algorithm to perform better.
  - Also, when a new user with parameters **w**, b is set to 0, it predicts the rating of a movie will be the mean.
  - 

    For user j, on movie i predict:

    $w^{(j)} \cdot x^{(i)} + b^{(j)} + \mu_i$

- Tensorflow implementation of collaborative filtering.
  - Example of a training loop:



Gradient descent algorithm

Repeat until convergence

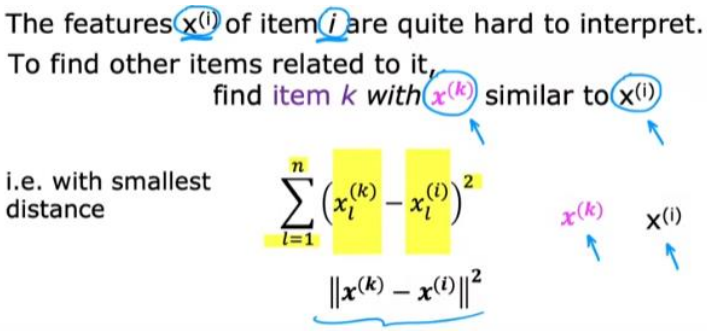
$$w = w - \alpha \frac{\partial}{\partial w} J(w,b,x)$$

$$b = b - \alpha \frac{\partial}{\partial b} J(w,b,x)$$

$$x = x - \alpha \frac{\partial}{\partial x} J(w,b,x)$$

```
# Instantiate an optimizer.
optimizer = keras.optimizers.Adam(learning_rate=1e-1)

iterations = 200
for iter in range(iterations):
    # Use TensorFlow's GradientTape
    # to record the operations used to compute the cost
    with tf.GradientTape() as tape:

        # Compute the cost (forward pass is included in cost)
        cost_value = cofiCostFuncV(X, W, b, Ynorm, R,
          num_users, num_movies, lambda)

    # Use the gradient tape to automatically retrieve
    # the gradients of the trainable variables with respect to
        the loss
    grads = tape.gradient( cost_value, [X,W,b] )

    # Run one step of gradient descent by updating
    # the value of the variables to minimize the loss.
    optimizer.apply_gradients( zip(grads, [X,W,b]) )
```

- Finding related items.



The features $x^{(i)}$ of item i are quite hard to interpret.

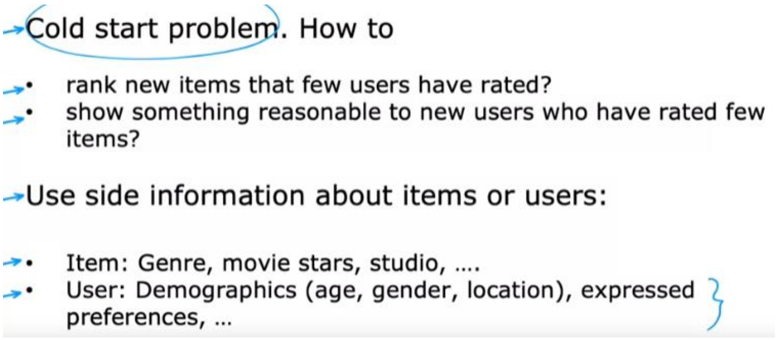
To find other items related to it,
find item k with $x^{(k)}$ similar to $x^{(i)}$

i.e. with smallest distance

$$\sum_{l=1}^{n}\left(x_l^{(k)} - x_l^{(i)}\right)^2$$

$$\left\|x^{(k)} - x^{(i)}\right\|^2$$

- Limitations of collaborative filtering:



Cold start problem. How to

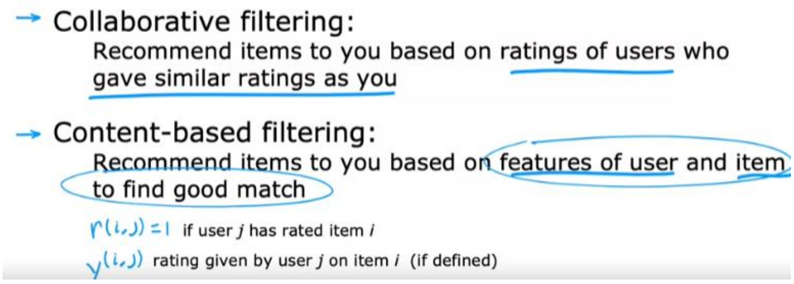
- rank new items that few users have rated?
- show something reasonable to new users who have rated few items?

Use side information about items or users:

- Item: Genre, movie stars, studio, ….
- User: Demographics (age, gender, location), expressed preferences, ...

Content-based Filtering

- Difference with collaborative filtering:



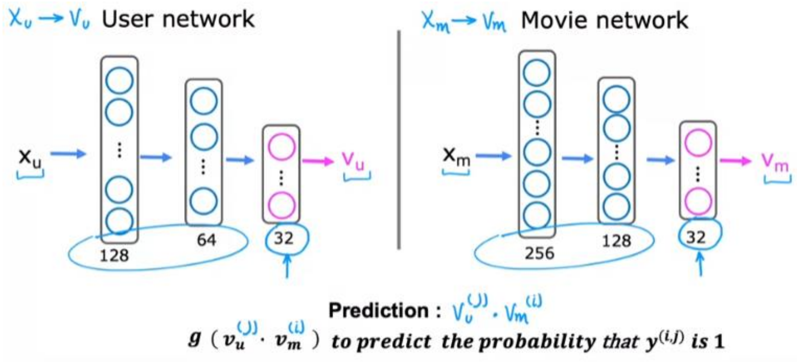
Collaborative filtering:
Recommend items to you based on ratings of users who gave similar ratings as you

Content-based filtering:
Recommend items to you based on features of user and item to find good match

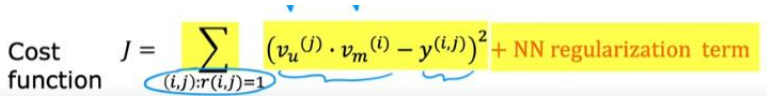
$r(i,j) = 1$ if user j has rated item i

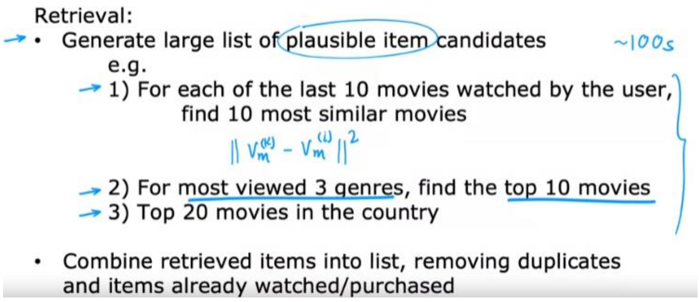
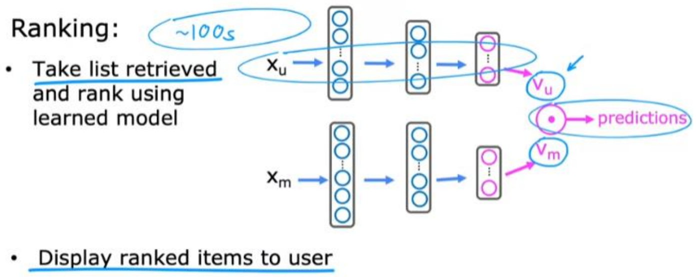
$y^{(i,j)}$ rating given by user j on item i (if defined)

- Algorithm learns to match users to movies.
- Deep learning for content-based filtering.
    - Use two separate neural networks, one for user, one for the item (movie).
    - Dot product both the vector outputs to obtain v_u (dot product) v_m.
    - The output dimensions of both networks must be the same.

- 

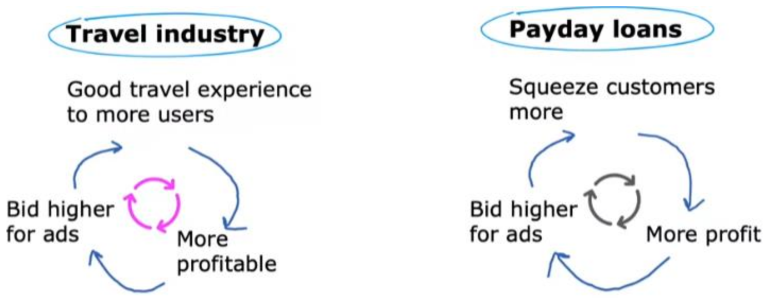
$X_u \rightarrow V_u$ User network
x_u
128
64
32
v_u
$X_m \rightarrow V_m$ Movie network
x_m
256
128
32
v_m

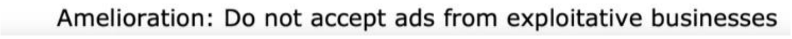
Prediction : $V_u^{(j)} \cdot V_m^{(i)}$
$g\ (v_u^{(j)} \cdot v_m^{(i)})$ *to predict the probability that* $y^{(i,j)}$ *is* 1
- The cost function can be used with an algorithm like gradient descent to tune the parameters of the model.
- 
Cost function $J = \sum_{(i,j):r(i,j)=1} (v_u^{(j)} \cdot v_m^{(i)} - y^{(i,j)})^2$ + NN regularization term
- To find items (movies) that are similar to each other, compute the squared difference of two item (movie) vectors, $v_m^{(k)}$.
- Note that this difference can be computed ahead of time (say overnight) or while the user is not using the service.

- Large catalogues.
    - If you have many products, re-running the computation for many users becomes computationally infeasible.
    - Many large-scale recommender systems are implemented as two steps: Retrieval & Ranking.
        - The retrieval step (relatively fast):
        - 
Retrieval:
• Generate large list of plausible item candidates ~100s
e.g.
1) For each of the last 10 movies watched by the user, find 10 most similar movies
$\| v_m^{(k)} - v_m^{(i)} \|^2$
2) For most viewed 3 genres, find the top 10 movies
3) Top 20 movies in the country
• Combine retrieved items into list, removing duplicates and items already watched/purchased
        - The ranking step:
        - 
Ranking: ~100s
• Take list retrieved and rank using learned model

• Display ranked items to user
        - If you have v_m computed ahead of time, all you need to do is compute vu which shouldn't take too long.
    - Retrieving more items results in better performance, but slower recommendations.
    - To analyse/optimize trade-offs, carry out offline experiments to see if retrieving additional items are beneficial (i.e., $p(y^{(i,j)})=1$ of items displayed to user are higher).

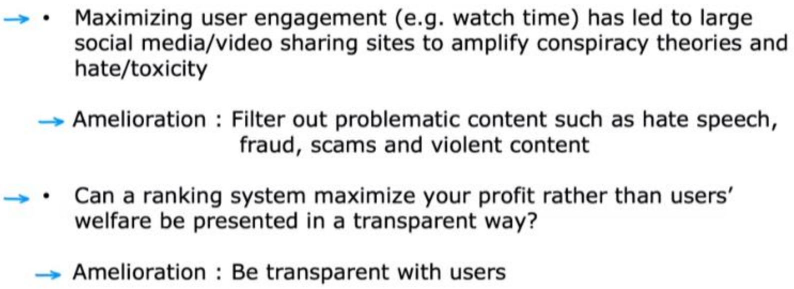
- Ethical considerations.
    - Two different use cases using the same system.



- 

  Amelioration: Do not accept ads from exploitative businesses

- Other problematic cases:



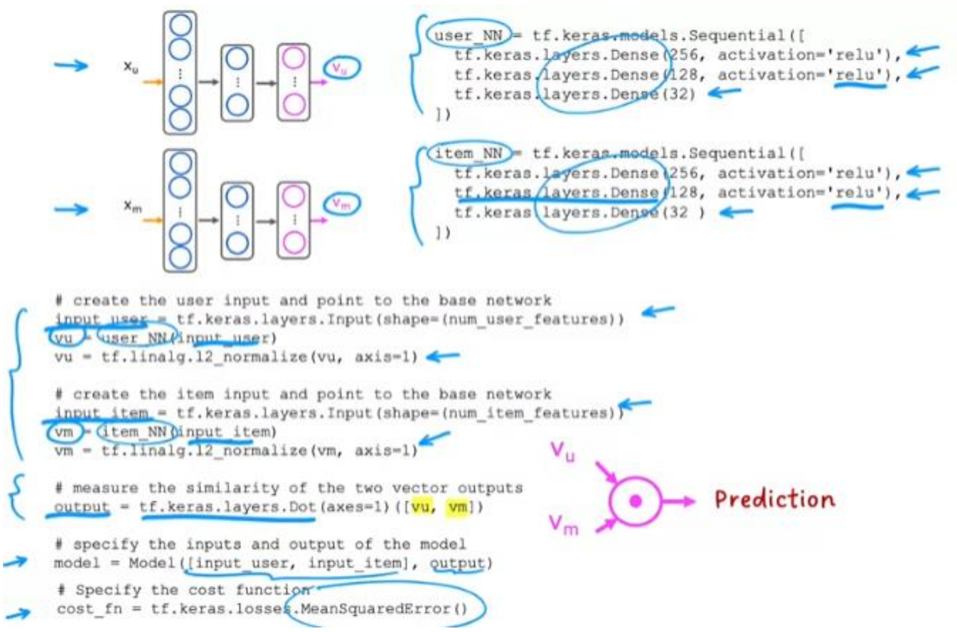
- Maximizing user engagement (e.g. watch time) has led to large social media/video sharing sites to amplify conspiracy theories and hate/toxicity

  Amelioration : Filter out problematic content such as hate speech, fraud, scams and violent content

- Can a ranking system maximize your profit rather than users' welfare be presented in a transparent way?

  Amelioration : Be transparent with users

- Tensorflow implementation.

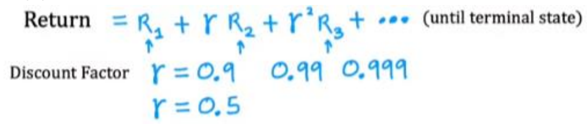


```
user_NN = tf.keras.models.Sequential([
  tf.keras.layers.Dense(256, activation='relu'),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(32)
])

item_NN = tf.keras.models.Sequential([
  tf.keras.layers.Dense(256, activation='relu'),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(32 )
])

# create the user input and point to the base network
input_user = tf.keras.layers.Input(shape=(num_user_features))
vu = user_NN(input_user)
vu = tf.linalg.l2_normalize(vu, axis=1)

# create the item input and point to the base network
input_item = tf.keras.layers.Input(shape=(num_item_features))
vm = item_NN(input_item)
vm = tf.linalg.l2_normalize(vm, axis=1)

# measure the similarity of the two vector outputs
output = tf.keras.layers.Dot(axes=1)([vu, vm])

# specify the inputs and output of the model
model = Model([input_user, input_item], output)

# Specify the cost function
cost_fn = tf.keras.losses.MeanSquaredError()
```

Week 10

Reinforcement Learning

- Don’t tell the algorithm how to do it, just reward positive behaviour and give negative reward for bad behaviour.
- Applications:
    - Controlling robots.
    - Factory optimization.
    - Financial trading.
    - Playing games.
- In reinforcement learning, every state is represented by a 4-tuple: (current state, action, reward at the current state, next state).
- To encourage faster reward gains, the Return rewards actions which lead to rewards that happen sooner.

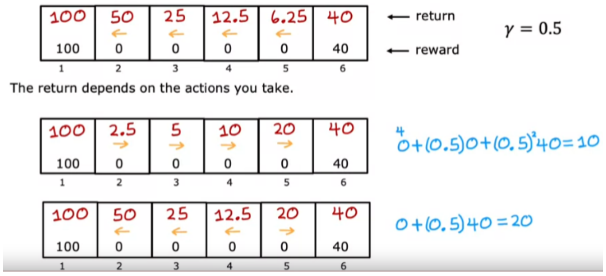


Return $= R_1 + \gamma R_2 + \gamma^2 R_3 + \cdots$ (until terminal state)

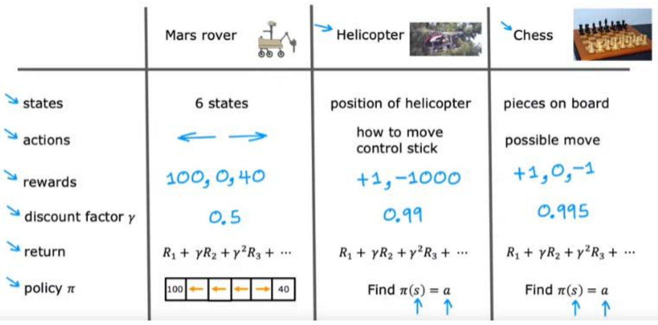
Discount Factor $\gamma = 0.9 \quad 0.99 \quad 0.999$

$\gamma = 0.5$

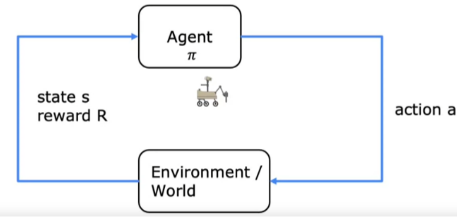
- The Return depends on the actions you take, and which state you start in.



- Reinforcement learning aims find a policy which takes a state and outputs an action – while maximizing reward.
    - Find a policy pi that tells you what action to take in every state to maximize the return.
- A policy is a function pi(s)=a mapping from states to actions, that tells you what action a to take in each state s.
- Examples of using reinforcement learning:

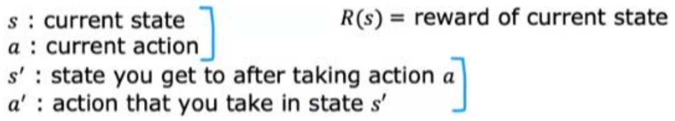


	Mars rover	Helicopter	Chess
states	6 states	position of helicopter	pieces on board
actions	← →	how to move control stick	possible move
rewards	100, 0, 40	+1, −1000	+1, 0, −1
discount factor γ	0.5	0.99	0.995
return	$R_1 + \gamma R_2 + \gamma^2 R_3 + \cdots$	$R_1 + \gamma R_2 + \gamma^2 R_3 + \cdots$	$R_1 + \gamma R_2 + \gamma^2 R_3 + \cdots$
policy π	100 ← ← ← → 40	Find $\pi(s) = a$	Find $\pi(s) = a$

- Markov Decision Process (MDP)
    - The future only depends on the current state.
    - 

State-Action Value Function

- Q(s,a) is the Return you get if you start in state s, take action a (once), and behave optimally after that.
- The best possible Return from state s is the maximum Q over all possible actions.
- The optimal value of Q is also known as Q* in some literature.
- Bellman Equation

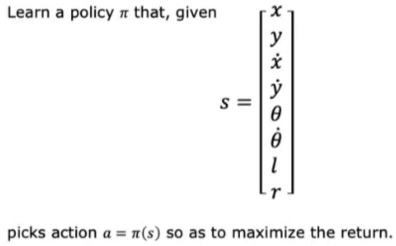


s : current state
a : current action
s' : state you get to after taking action a
a' : action that you take in state s'
$R(s)$ = reward of current state

  - $$Q(s,a) = R(s) + \gamma \max_{a'} Q(s',a')$$
  - Recursively defined.
- Random (stochastic) environment.
  - What if there was a non-zero probability that the robot would take another action rather than the optimal one?
  - This results in a modified version of the Bellman equation, which maximizes the expected average Return over multiple runs of the algorithm.
  - $$Q(\underset{\uparrow}{s},\underset{\uparrow}{a}) = \underbrace{R(s)} + \gamma\, E[\max_{a'} Q(\underset{\uparrow}{s'},a')\,]$$

Continuous State Spaces

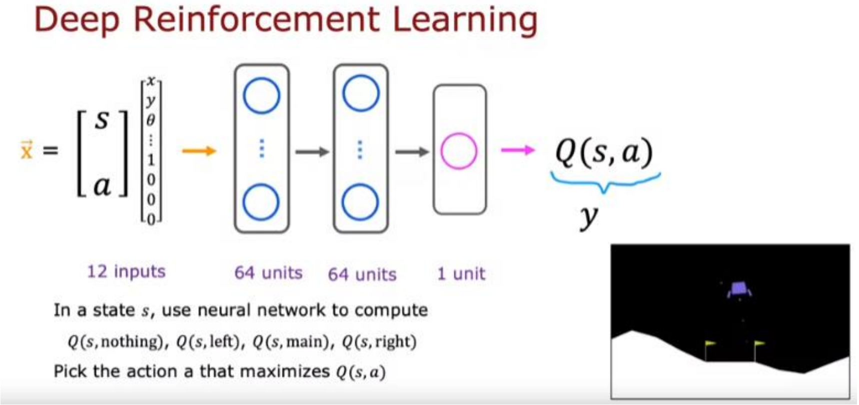
- States could be continuous values such as orientation (in degrees) or position (in meters).
- Complex applications like autonomous vehicles could have a state represented by a vector of continuous values (like speed).
- The Lunar Lander Problem:



Learn a policy π that, given

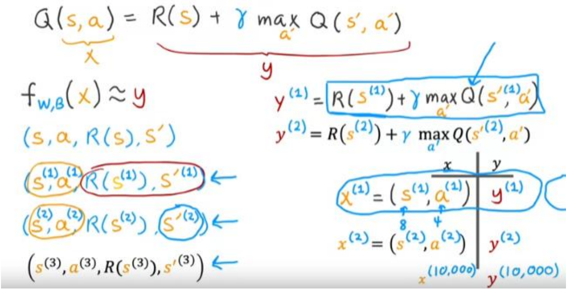
$$s = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \\ \theta \\ \dot{\theta} \\ l \\ r \end{bmatrix}$$

picks action $a = \pi(s)$ so as to maximize the return.

- $\gamma = 0.985$
- Where the symbols are the x-position, y-position, x-speed, y-speed, tilt, angular velocity, is the left leg on ground, is the right leg on the ground.
- Learning algorithm:
  - Uses a neural network.
  - 

Deep Reinforcement Learning

In a state s, use neural network to compute
$Q(s,\text{nothing})$, $Q(s,\text{left})$, $Q(s,\text{main})$, $Q(s,\text{right})$
Pick the action a that maximizes $Q(s,a)$

- How do we generate training examples for the neural networks?
    - We can use the Bellman Equation.
    - 

$$\underbrace{Q(s,a)}_{x} = \underbrace{R(s) + \gamma \max_{a'} Q(s', a')}_{y}$$

$f_{w,B}(x) \approx y$

$(s, a, R(s), s')$

$(s^{(1)}, a^{(1)}, R(s^{(1)}), s'^{(1)}) \leftarrow$

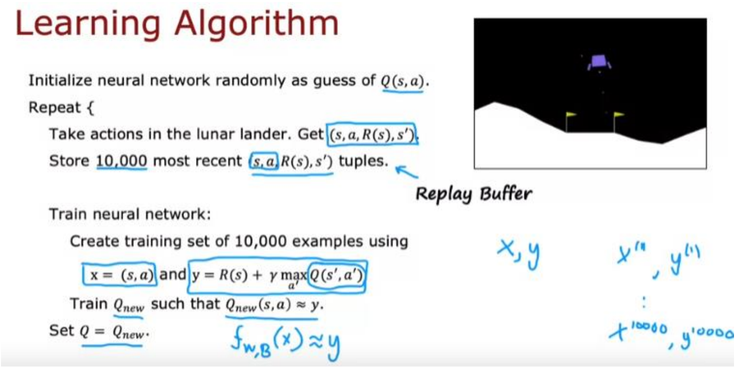
$(s^{(2)}, a^{(2)}, R(s^{(2)}), s'^{(2)}) \leftarrow$

$(s^{(3)}, a^{(3)}, R(s^{(3)}), s'^{(3)}) \leftarrow$

$y^{(1)} = \boxed{R(s^{(1)}) + \gamma \max_{a'} Q(s'^{(1)}, a')}$

$y^{(2)} = R(s^{(2)}) + \gamma \max_{a'} Q(s'^{(2)}, a')$

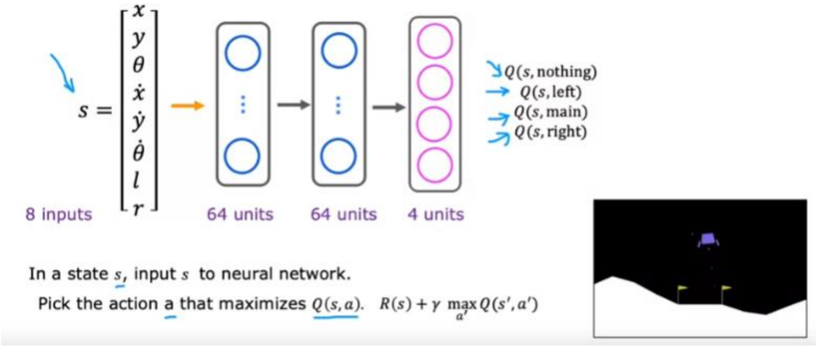
x	y
$x^{(1)} = (s^{(1)}, a^{(1)})$	$y^{(1)}$
$x^{(2)} = (s^{(2)}, a^{(2)})$	$y^{(2)}$
$x^{(10,000)}$	$y^{(10,000)}$

- Full algorithm:
    - 

Learning Algorithm

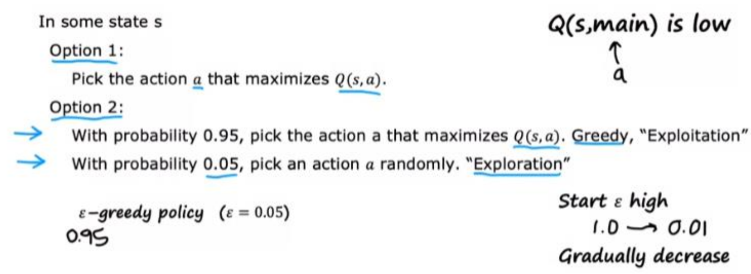
```
Initialize neural network randomly as guess of $Q(s,a)$.
Repeat {
    Take actions in the lunar lander. Get $(s, a, R(s), s')$.
    Store 10,000 most recent $(s, a, R(s), s')$ tuples.
    Train neural network:
        Create training set of 10,000 examples using
        $x = (s,a)$ and $y = R(s) + \gamma \max_{a'} Q(s',a')$
        Train $Q_{new}$ such that $Q_{new}(s,a) \approx y$.
    Set $Q = Q_{new}$.
```

    - Sometimes called a DQN algorithm (Deep Q-Network), where Q is the Q function.

- Making DQN more efficient:
    - Improved NN architecture.
        - Rather than carry out inference for each of the 4 (or n) actions available, output the Q values for all four (or n) states in one inference.
        - 

In a state s, input s to neural network.

Pick the action a that maximizes $Q(s,a)$. $R(s) + \gamma \max_{a'} Q(s', a')$

    - Epsilon-greedy policy.
        - How can we take a better action in each iteration of the algorithm while it is still learning (originally guessed value of Q)?
        - Pick actions with a certain probability that maximizes Q.
        - This helps to overcome any preconceptions the algorithm might have because of a randomly initialized Q value.
        - 

In some state s

Option 1:

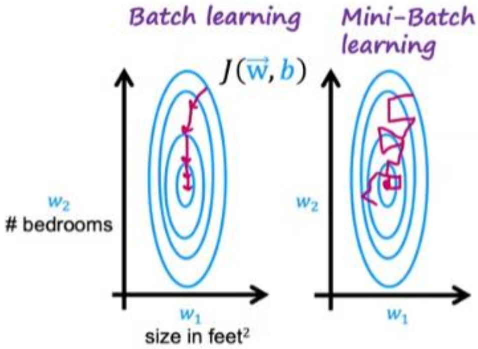
Pick the action a that maximizes $Q(s,a)$.

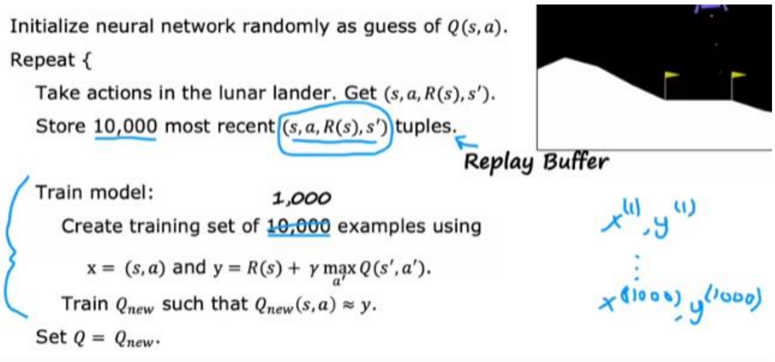
Option 2:

→ With probability 0.95, pick the action a that maximizes $Q(s,a)$. Greedy, "Exploitation"

→ With probability 0.05, pick an action a randomly. "Exploration"

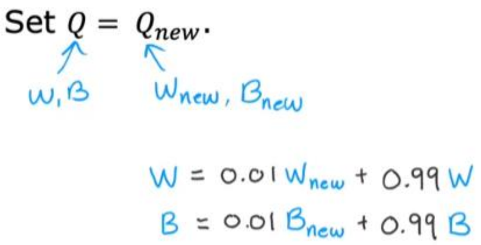
ε-greedy policy ($\varepsilon = 0.05$)

- Mini-batching.
  - Can be applied to supervised algorithms as well.
  - Don't use all training examples on every step of gradient descent if you have an excessive amount of training examples.
  - Makes each iteration compute faster and more efficient.
  - Divide the training examples into groups of batches to be used at each iteration (or alternatively, pick a few random examples at each iteration).
  - There is a trade-off with the path taken, but it tends to be faster overall due to the smaller number of training examples.
  - 

  - Lunar Lander example:
  - 

```
Initialize neural network randomly as guess of $Q(s,a)$.
Repeat {
    Take actions in the lunar lander. Get $(s,a,R(s),s')$.
    Store 10,000 most recent $(s,a,R(s),s')$ tuples.  ← Replay Buffer
    Train model:
        Create training set of ~~10,000~~ 1,000 examples using
            x = $(s,a)$ and y = $R(s) + \gamma \max_{a'} Q(s',a')$.
        Train $Q_{new}$ such that $Q_{new}(s,a) \approx y$.
    Set $Q = Q_{new}$.
```

$x^{(1)}, y^{(1)}$
⋮
$x^{(1000)}, y^{(1000)}$

- Soft updates.
  - Helps reinforcement learning application converge to a solution faster.
  - In the Set Q = Q$_{new}$ step of the algorithm.
  - 

Set $\underset{W,B}{Q} = \underset{W_{new},\,B_{new}}{Q_{new}}$.

$W = 0.01\,W_{new} + 0.99\,W$

$B = 0.01\,B_{new} + 0.99\,B$

- State of Reinforcement Learning.
  - Limitations:
    - Much easier to get to work in a simulation than a real robot.
    - Far fewer applications than supervised and unsupervised learning.
    - Exciting research direction with potential for future applications.