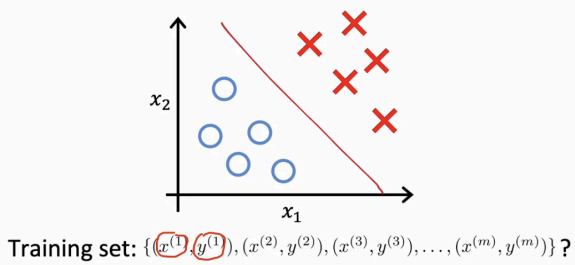
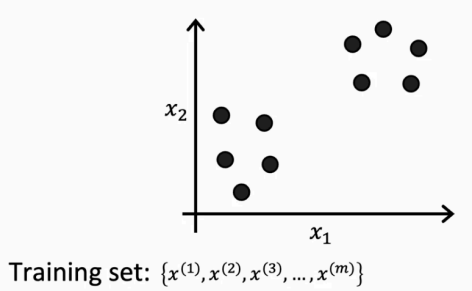
Week 1: Unsupervised Learning

Clustering algorithm

- Looks at a number of data points and evaluates which are similar

- In supervised learning, dataset includes input x and target output y



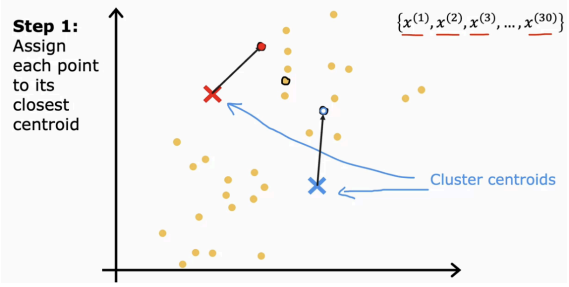


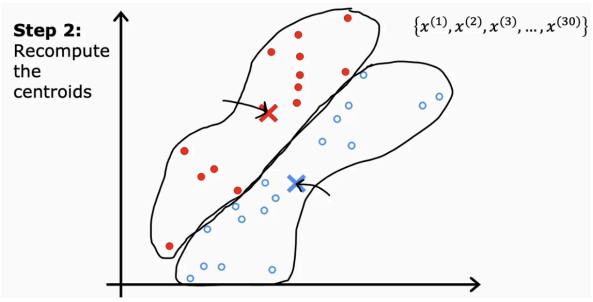
- Clustering algorithms look for clusters. Applications includes grouping similar news, market segmentation, DNA analysis, astronomical data analysis

- Most common clustering algorithm is *K-means*:
  - Will take a random initial guess at centres of two clusters.
  - Will do two things: assign cluster centroids, and move cluster centroids

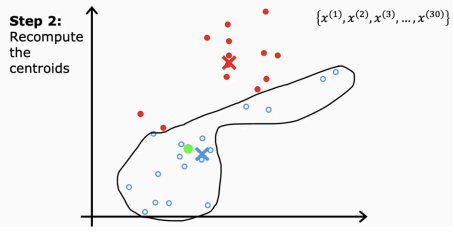
  1) After making an initial guess, goes through all examples x, and for each will check whether it is closer to red or blue cluster centroid, and will assign each point to its closer centroid

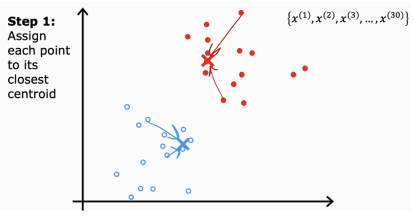
  2) Will look at all red points and take an average, and will move red cross to average location of all the red dots

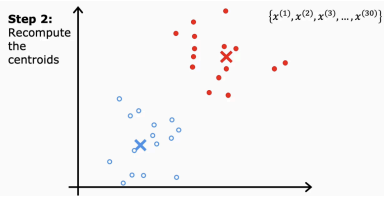




3) Then repeat, and check whether each point is closer to red or blue cluster centroid, and then compute average and move X again



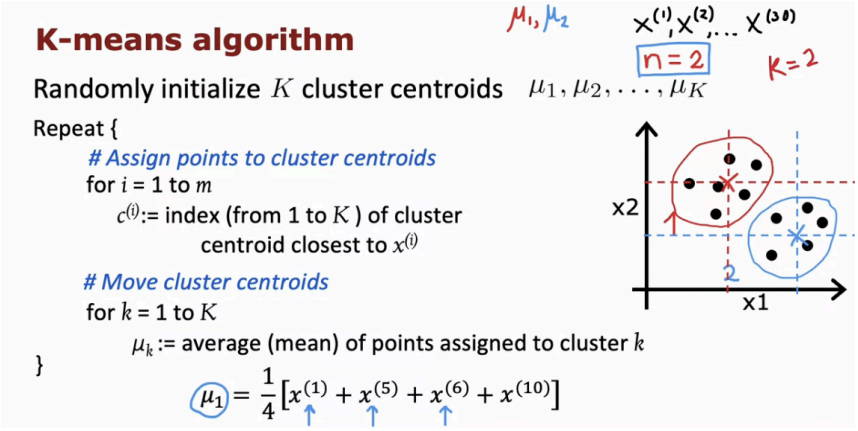




4) Eventually K-means clustering algorithm converges i.e. there are no more changes

Details of K-means algorithm

Technical steps:

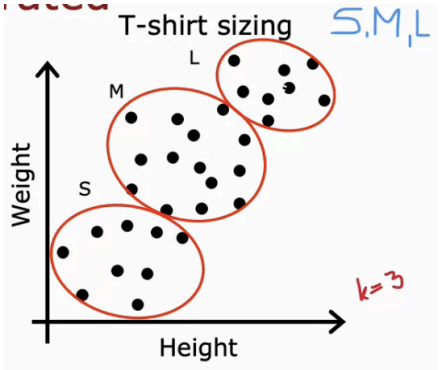


Notes;
- Mu1 and Mu2 have same dimensions as training data - here, they will be two-dimensional vectors

- If a cluster has zero training examples assigned - second step would be trying to compute average of zero points (not well defined). Commonly, just eliminated cluster to give K=k-1 cluster; or randomly re-initialise initial guess.

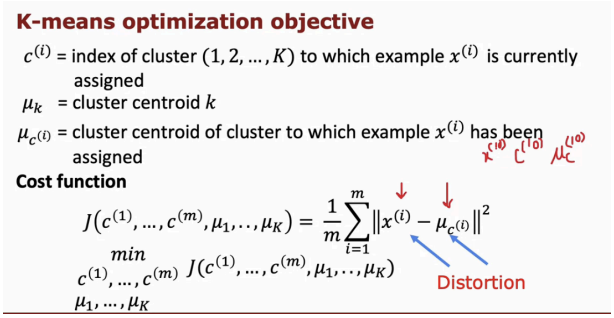
K-means for clusters that are not well separated

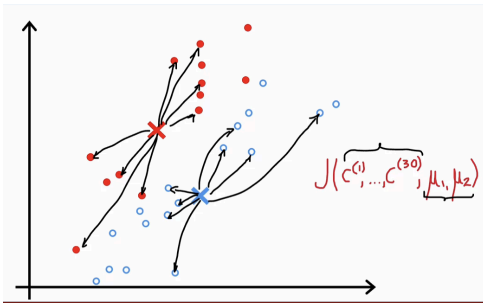
E.g. T-shirt size:



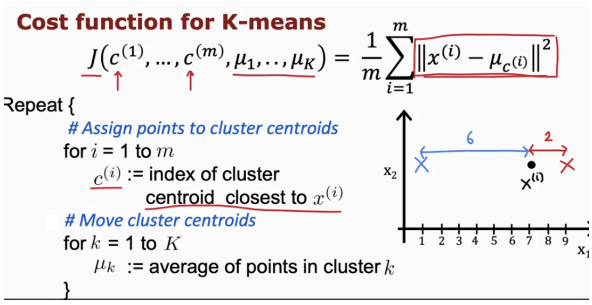
Optimising objective (Cost function for K-means)

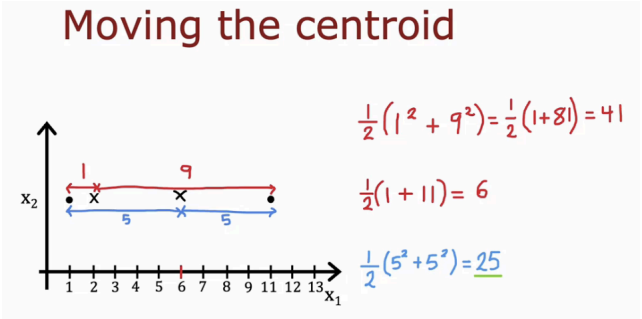
- Average square distance between every training examples x(i) and the location of the cluster centroid to which x(i) has been assigned





- In more detail:
  - First part of K-means (assign points to cluster centroids) - trying to update c(1) to c(m) to minimise J as much as possible while holding mu(1) to mu(k) fixed
  - Second part (move cluster centroids): trying to update mu(1) to mu(k) while keeping c(i) fixed

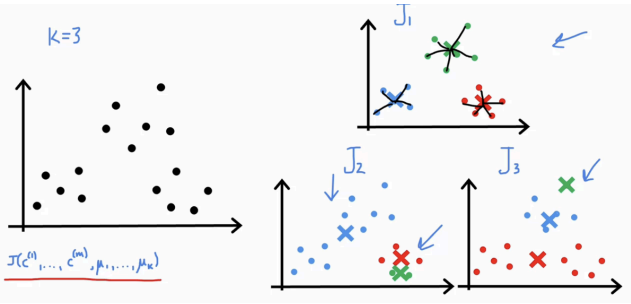




- Should always reduce (otherwise there will be a bug in the code)

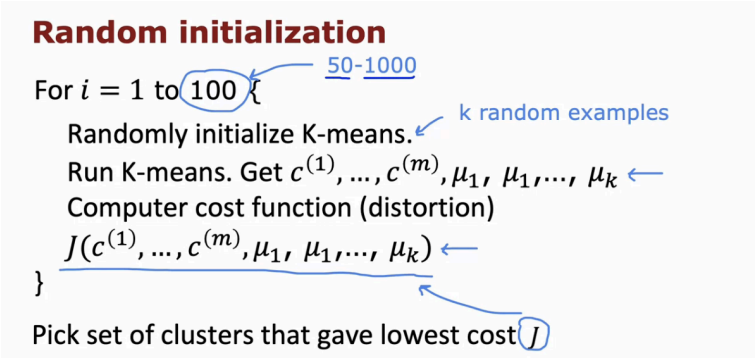
Initializing K-means: how to take the random guess

- Choose number of clusters to be less than training examples m (choose K < m)
- Randomly pick K training examples [not completely random]
- Set mu(1), mu(2), …, m(k) equal to these *K* examples

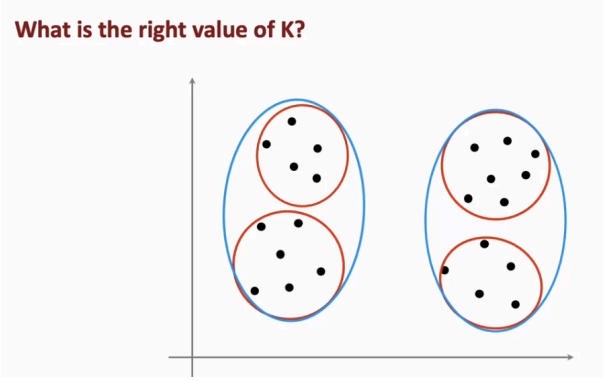


- Run initialisation multiple times to find best starting points - run cost function to evaluate these

The algorithm for this is:

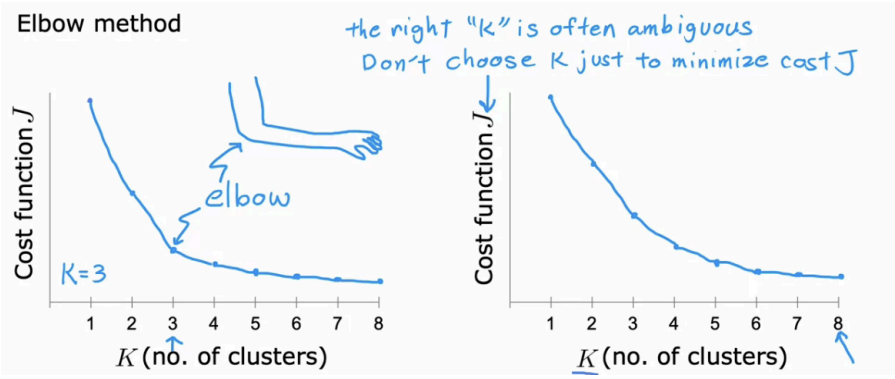


Choosing number of clusters (value of K)



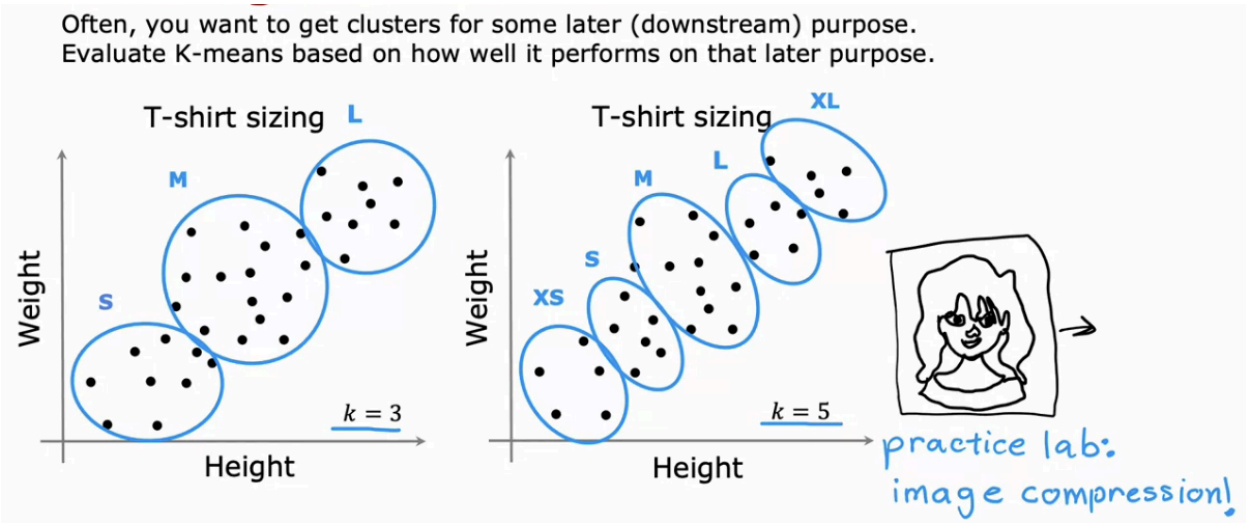
To automatically choose K:

1) Elbow method [but not v. good]
- Plot cost function against value of K
- But not always a clear elbow
- Don't choose K just to minimise J (as will generally just be largest K)



2) Evaluate K-means based on how well it performs on its later purpose

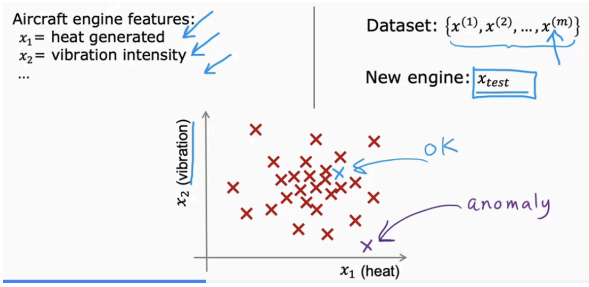
E.g. whether you want to make three or five sizes for T-shirts - see trade-off between better fit vs making more types of t-shirts



- With image compression, there is a trade off between quality of compressed image vs how much you can compress image to save space

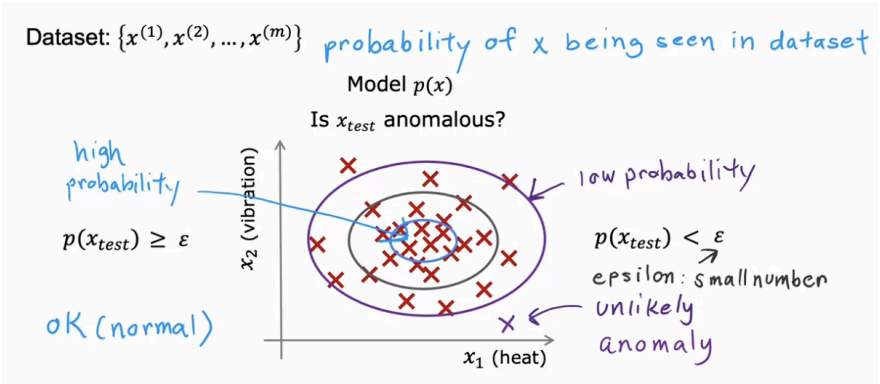
Anomaly detection

Example with aircraft engine manufacture:



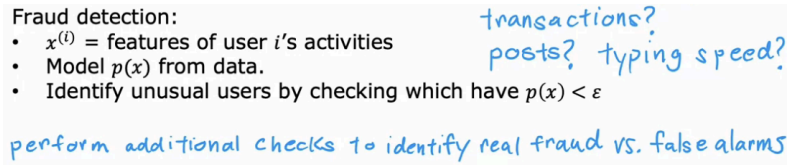
The most common way to carry out anomaly detection is to use *density estimation*:

- Probability of x being seen in the dataset



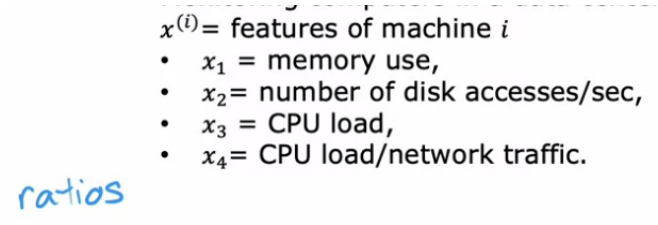
Other examples:

1) Fraud detection:
- How often do they log in?
- How many web pages visited?
- Transactions?
- Posts?
- Typing speed?

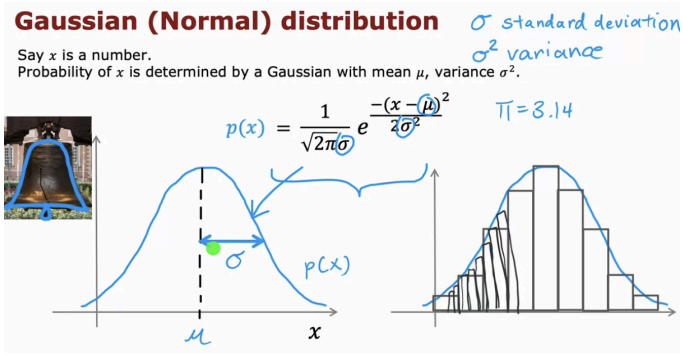


2) Manufacturing:
- To see if there is an issue that merits a closer look
- X[i] = features of product i

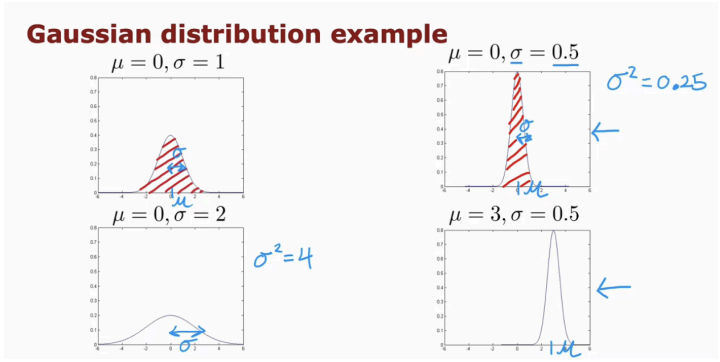
3) Monitoring computers in a data centre:



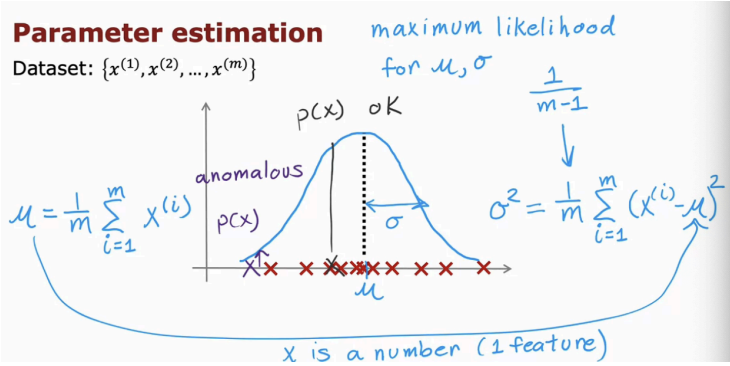
Gaussian (normal) distribution



Changing mu and sigma:
- Area under curve stays the same



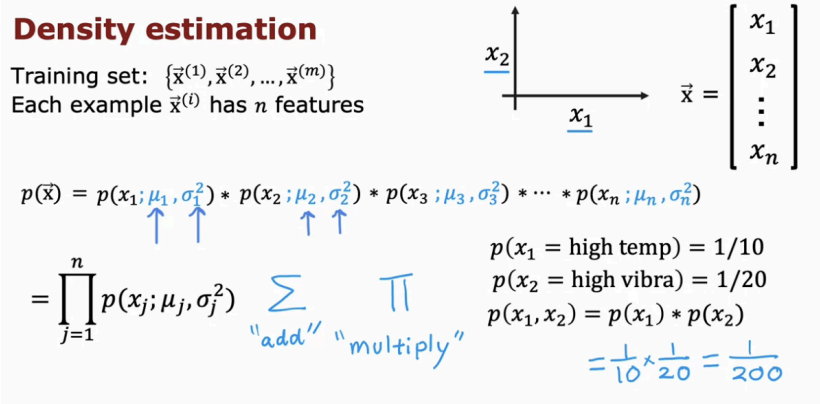
Applying this to anomaly detection: parameter estimation



- This is for if x is a single number (1 feature); but for practical applications you will have multiple features

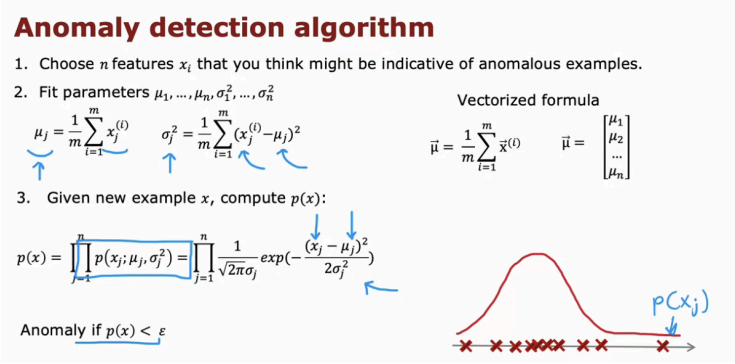
Density estimation

- Build a model for p(x)
- Assumes features are statistically independent

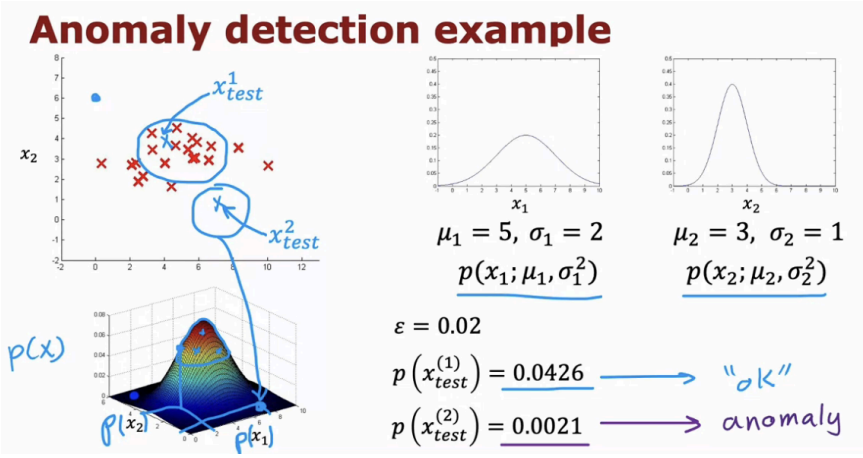


Putting it together into an anomaly detection algorithm

- If just one of p(xj) is v. small, as p(x) is a product, whole thing will be small



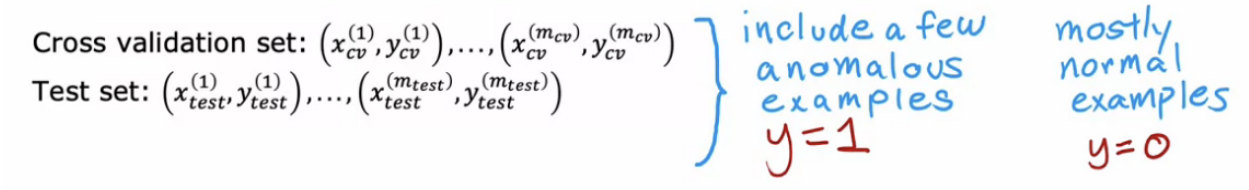
Anomaly detection example



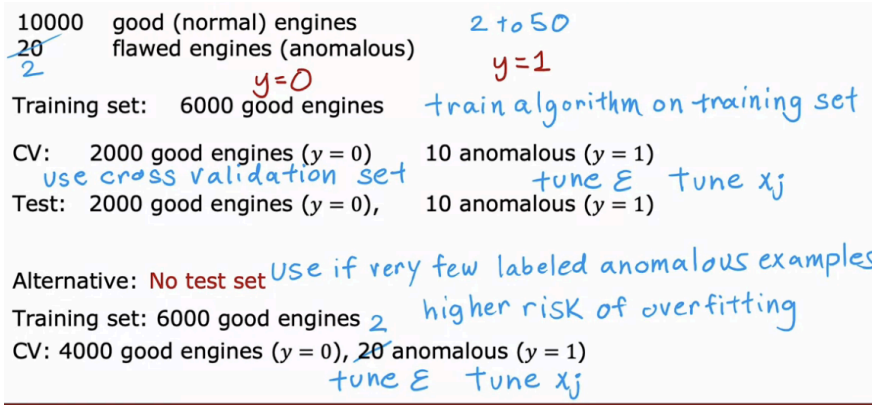
Developing and evaluating an anomaly detection system

The importance of real-number evaluation:

- When developing a learning algorithm (chosing features, etc.), making decisions is much easier if we have a way of evaluating our learning algorithm.
- Assume we have some labeled data, of anomalous (y=1) and non-anomalous (y=0) examples
- Training set: x(1), x(2), …, x(m) (assume normal examples/ not anomalous i.e. y=0 for all training examples)



E.g. aircraft engines monitoring example



Algorithm evaluation

Algorithm evaluation

Fit model $p(x)$ on training set $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$
On a cross validation/test example x, predict

$$y = \begin{cases} 1 & if\ p(x) < \varepsilon\ (anomaly) \\ 0 & if\ p(x) \geq \varepsilon\ (normal) \end{cases}$$

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

Fraud techniques changes a lot (every few months get a new type of fraud), but spam has not changed much.

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

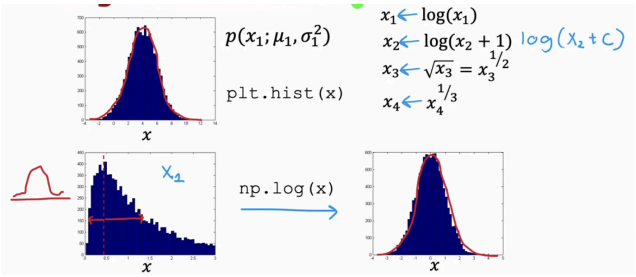
Anomaly detection	vs.	Supervised learning
Fraud detection		Email spam classification
Manufacturing - Finding new previously unseen defects in manufacturing.(e.g. aircraft engines)		Manufacturing - Finding known, previously seen defects (scratches)
Monitoring machines in a data center		Weather prediction (sunny/rainy/etc.)
⋮		Diseases classification
		⋮

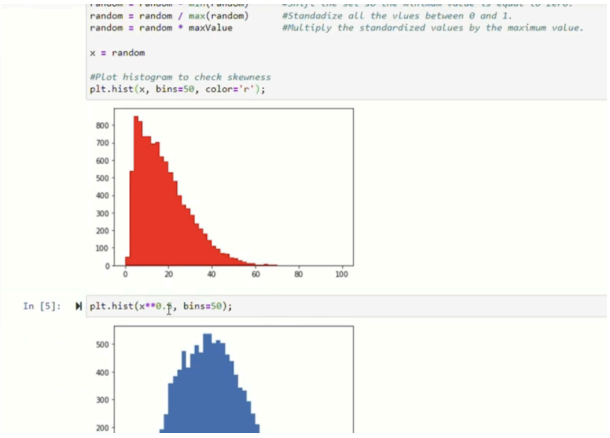
Choosing what features to use

- Harder to find what features to ignore for an unsupervised algorithm

Non-gaussian features

- Try to make features as gaussian as possible
    - Look at them using plt.hist(x)

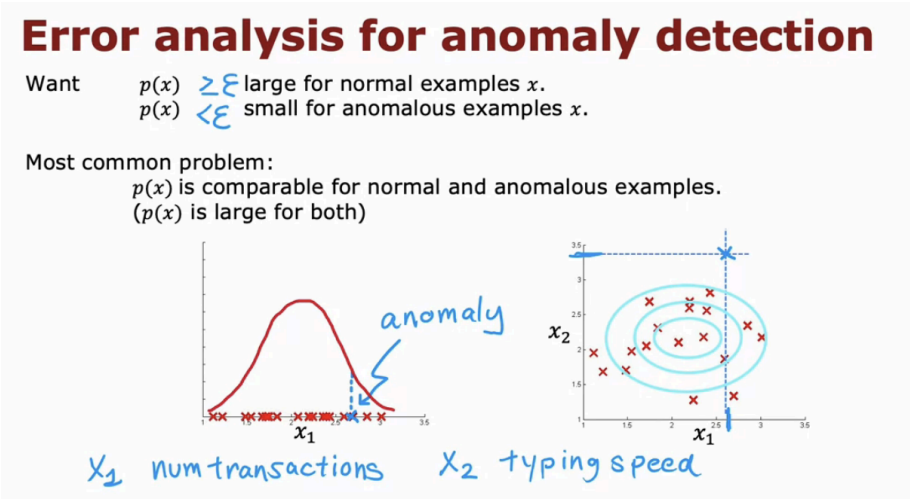




- Try things out. *Apply same transformation to training set <u>and</u> CV and test set*

Error analysis for anomaly detection

- May be useful to use a new feature
- Train a model, then see what anomalies in the CV set the algo has failed to detect. Then look at that examples to inspire new features that can flag these examples… (by giving unusually large or small values in the event of an anomaly)



Final example: monitoring computers in a data centre

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

high low

not unusual

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

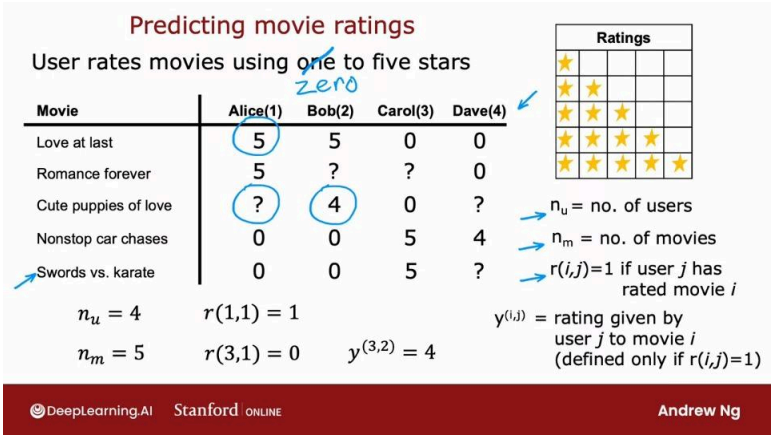
Deciding feature choice based on $p(x)$

Large for normal examples;

Becomes small for anomaly in the cross validation set

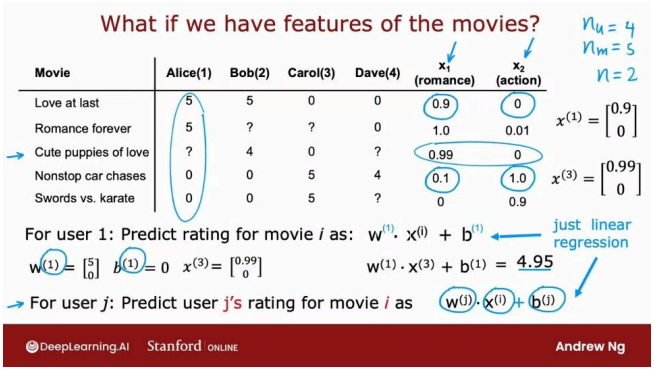
Recommender systems

Making Recommendations



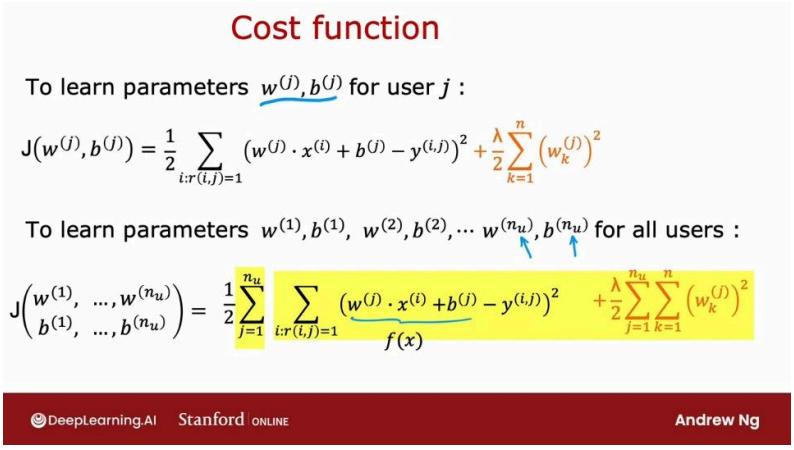
Using per-item features

- Adding features of the movies



How can we make a cost function for this algorithm?

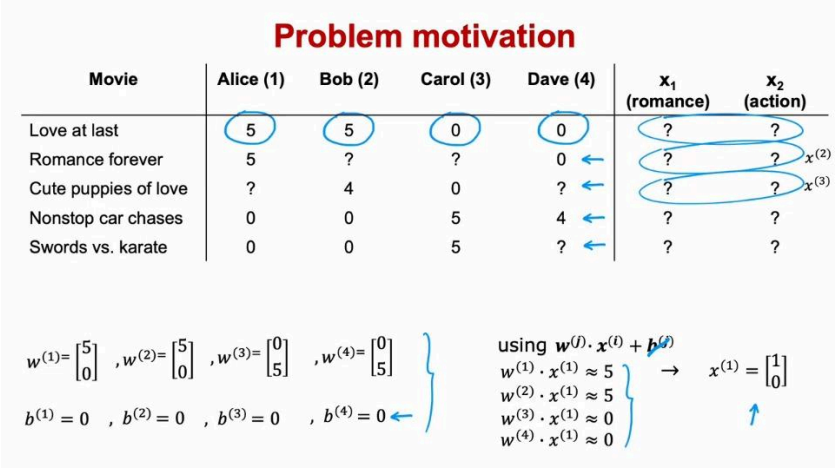
- Trying to minimise error between predicted rating and actual rating. Only summing over the movies that user j has actually rated…



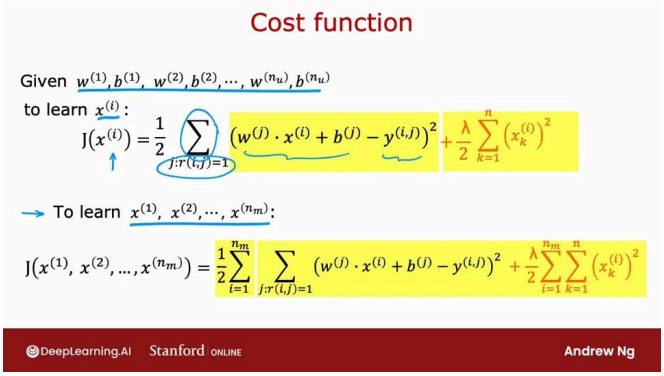
- Training a different linear regression model for each of the n(u) users

Collaborative filtering algorithm

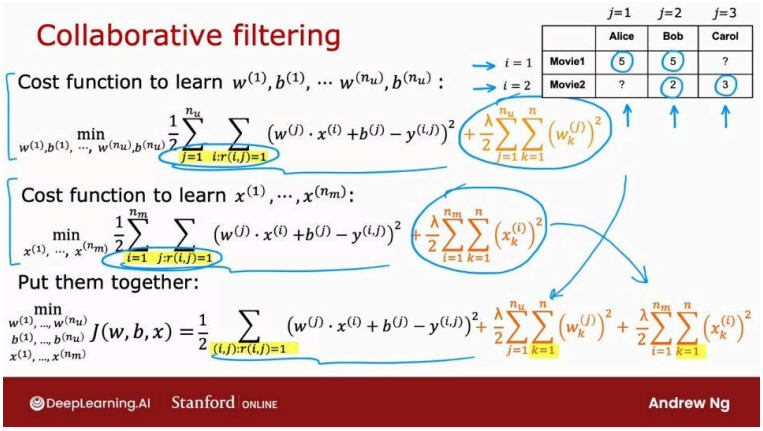
- What if you don't have the values of features x1 and x2?
- Given the values of w*x that we have, what choice for x(1) would cause these values to be right?...



- Let's find a cost function for learning the values x1 and x2 (assuming you have the parameters for the different movies):



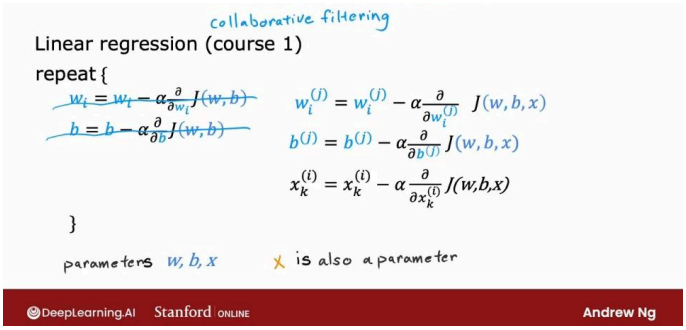
- Putting together to get collaborative filtering algorithm:



So how do we minimise this cost function?

Option 1: Gradient descent

- Note that x is also a parameter here, so must also be minimised



Binary labels: favourites, likes and clicks

- Generalise the algorithm to this system (cf linear → logistic regression)
- Predict how likely the people are to like the movies they have not yet rated → Then can decide whether to recommend these movies or not

Binary labels

Movie	Alice(1)	Bob(2)	Carol(3)	Dave(4)
Love at last	1	1	0	0
Romance forever	1	?	?	0
Cute puppies of love	?	1	0	?
Nonstop car chases	0	0	1	1
Swords vs. karate	0	0	1	?

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

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

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

Adapting cost function for binary application:

Previous cost function:

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

$f(x)$

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

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

Loss for single example

$$J(w,b,x) = \sum_{(i,j):r(i,j)=1} L(f_{(w,b,x)}(x), y^{(i,j)})$$

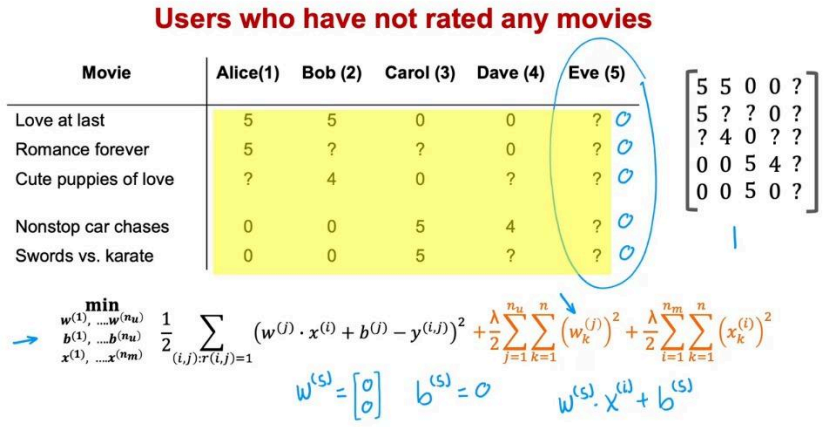
$g(w^{(j)} \cdot x^{(i)} + b^{(j)})$

cost for all examples

Recommender systems implementation detail

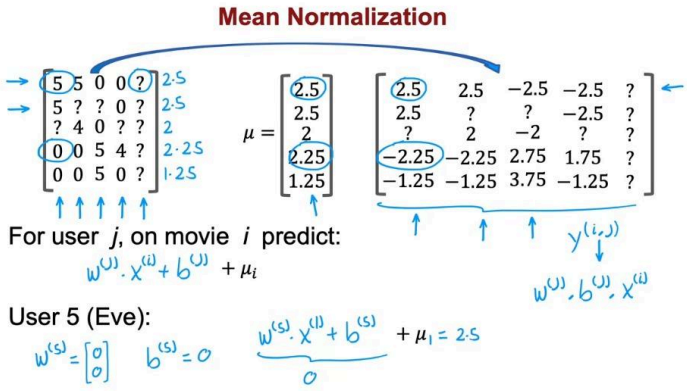
Mean Normalisation

Mean normalisation (i.e. normalise movie ratings to have a consistent average value) can speed up recommender system algorithms.



Computing average of rating of each movie:

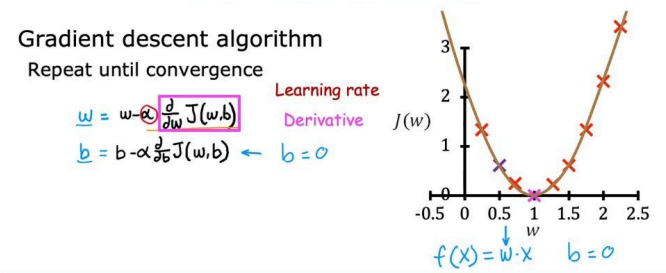
- Subtracting mean from ratings. Can then year w,b,x for the modified dataset.
- Possible to also normalise columns; however, here rows make more sense. Normalising columns would work if it were a brand new movie no one had rated yet, rather than a new user

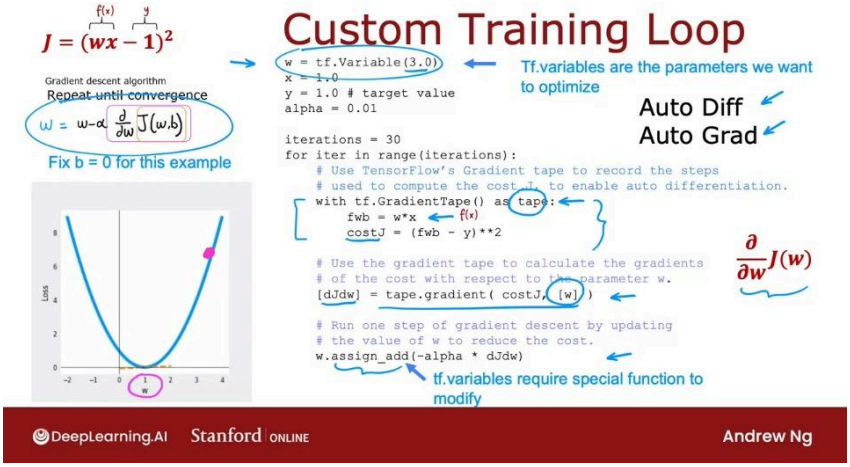


TensorFlow implementation of collaborative filtering

Derivatives in ML:

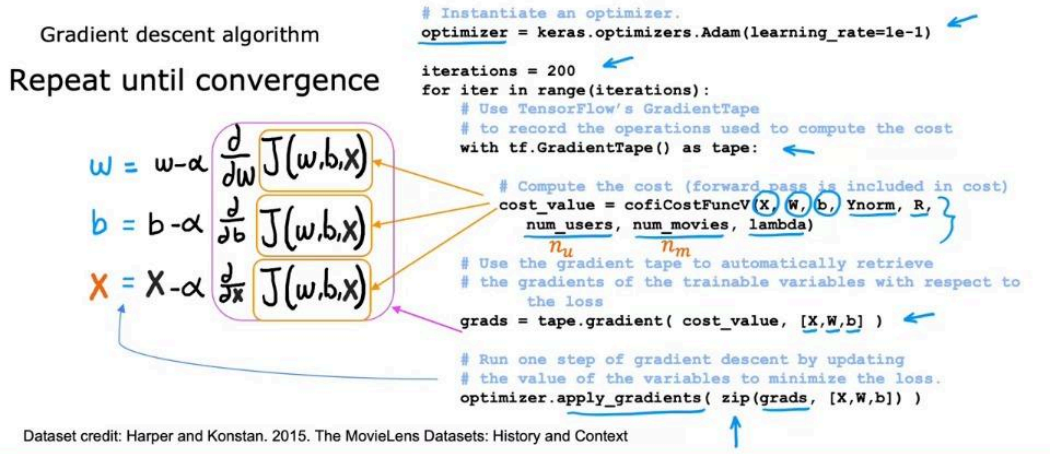
- Calculating update derivative for b can be complex - TensorFlow can help with this… automatically computing it
- Procedure below allows you how to implement gradient descent without having to know how to figure out derivative:



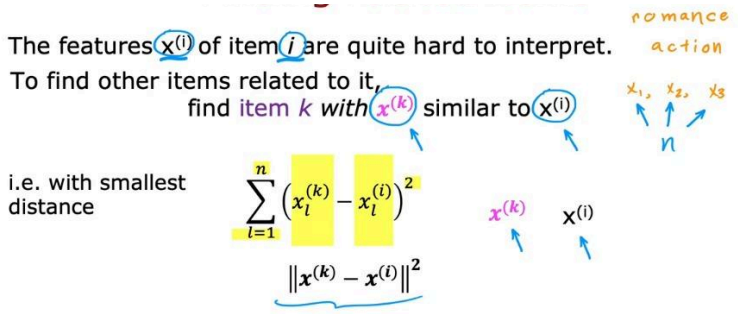


Implementing collaborative filtering algorithm in TensorFlow (with adam):

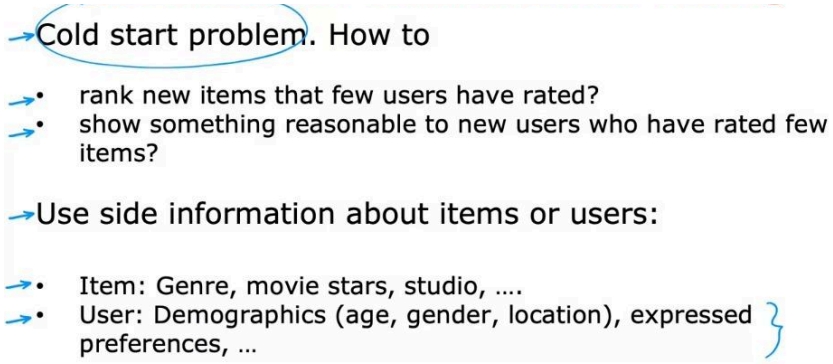
- Using TensorFlow, you are not limited to gradient descent - can use a more powerful optimization algorithm such as adam



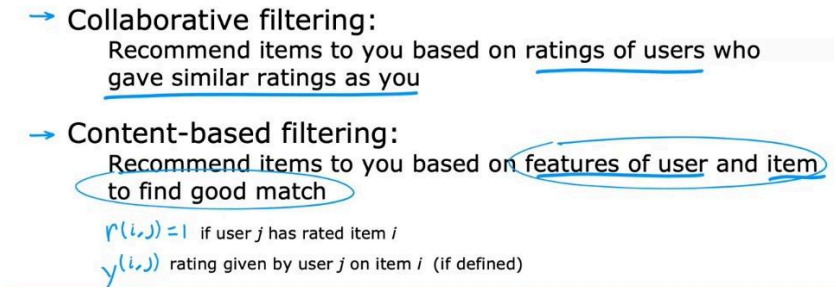
Finding related items



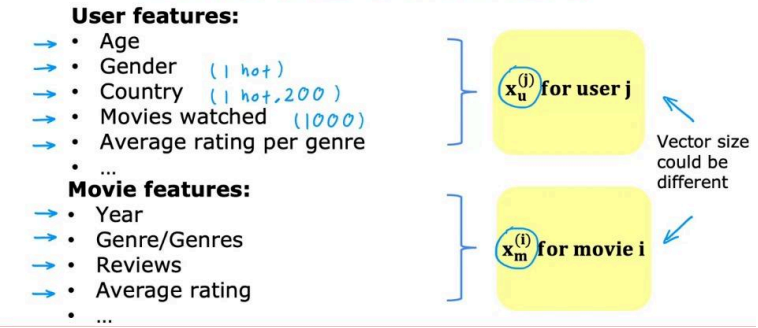
Limitations of Collaborative Filtering



Collaborative filtering vs Content-based filtering

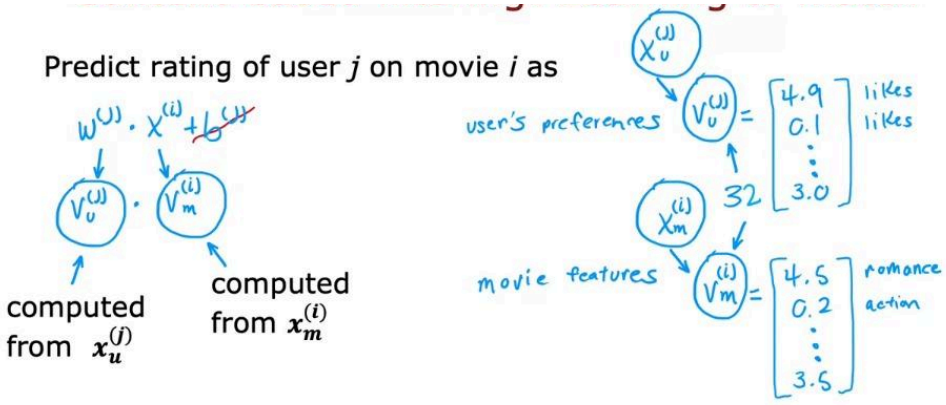


Examples of user and item features:



So how do we match user and movie in content-based filtering:

- Given features of a user x(j)u, how can we create the vector for the user's preferences vector and similarly, given features of a movie, how to compute movie features vector
- Xs can be different in size, vs must be same size so you can take a dot product

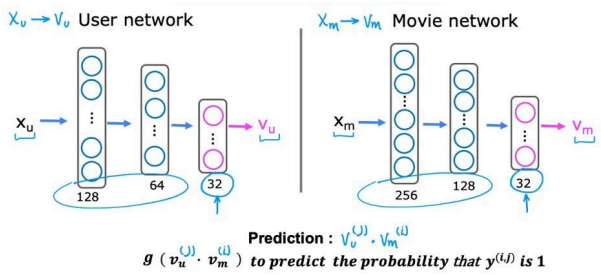


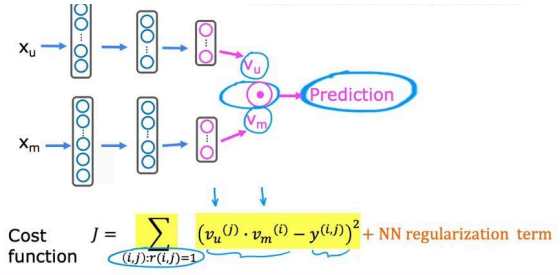
Summarising:

- In collaborative filtering, we have number of users give ratings of different items
- Vs in content-based filtering, we have features of users and features or items and want to find good matches by computing vectors for both and taking dot products
- But how to compute these vectors?...

Deep learning for content-based filtering

- Neural network architecture



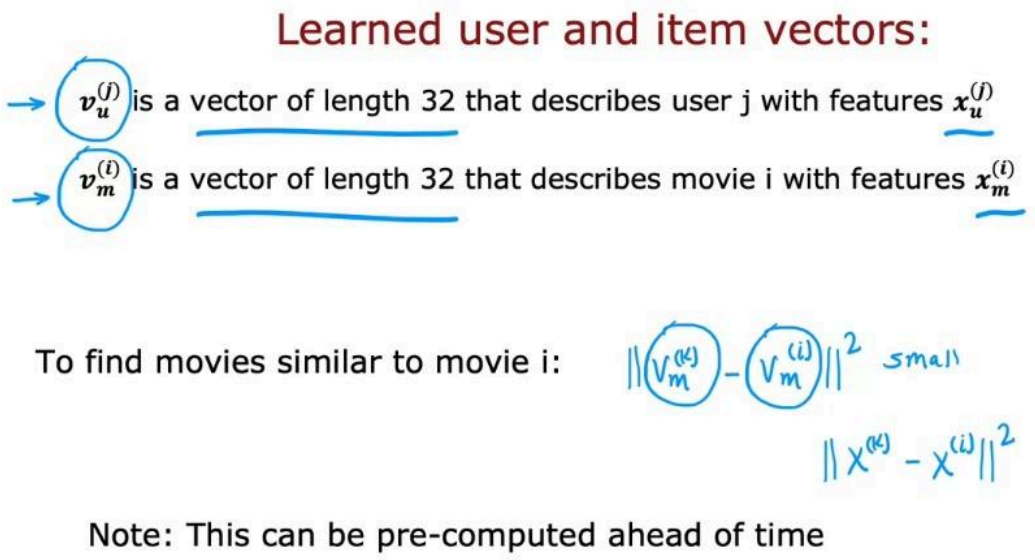


Train parameters of neural network so result in vectors of users and movies to minimise square error.

- Regularisation encourages NN to keep parameters of NN small

After training this model, can also use it to find similar items… (can do this ahead of time)

- I.e. where the distance between two movie vectors is small



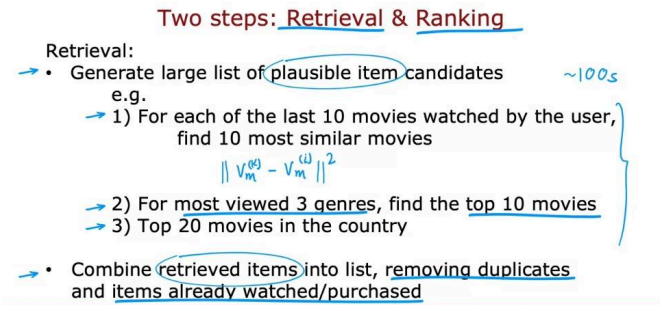
Note that this is reflective of NN benefits over decision trees - easier to take multiple neural networks and put them together to work in tandem.

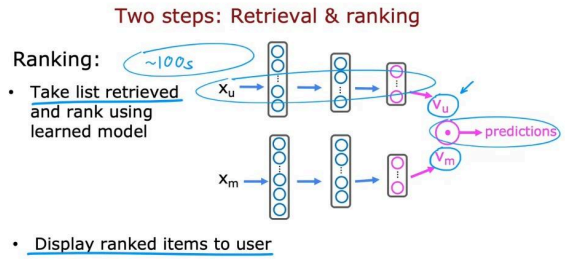
We now consider practical issues of scaling up to very large item catalogues…

Recommending from a large catalogue

How can we efficiently find recommendation from a large set of items?

- Computationally unfeasible to run neural network influence huge numbers of times every time a user comes on website
- Instead, use two steps: retrieval and ranking…





- One additional optimization is that if you have computed VM for all the movies in advance, then all you need to do is to do inference on this part of the neural network a single time to compute VU, and then take that VU they just computed

for the user on your website right now and take the inner product between VU and VM for the movies that you have retrieved during the retrieval step.

Trade off:

- Retrieving more items results in better performance, but slower recommendations.
- To analyse/optimize the trade-off, carry out offline experiments to see if retrieving additional items results in more relevant recommendations (i.e., $p(y^{(i,j)}) = 1$ of items displayed to user are higher).

100 500

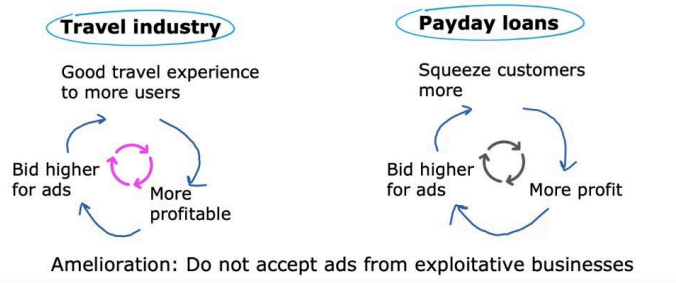
Ethical use of recommender systems

What is the goal of the recommender system?

Recommend:

- Movies most likely to be rated 5 stars by user
- Products most likely to be purchased
- Ads most likely to be clicked on + high bid
- Products generating the largest profit
- Video leading to maximum watch time

Ethical considerations with recommender systems:



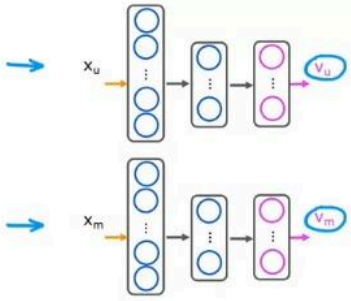
Amelioration: Do not accept ads from exploitative businesses

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 of content-based filtering:

- axis=1 normalises vector to length 1
- keras.layers.Dot just takes dot product between vu and vm to give output
- Can also L2 normalise vector

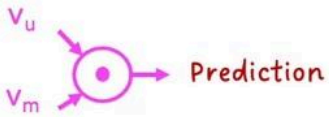



```
user_NN = tf.keras.models.Sequential([
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(32)
])
```



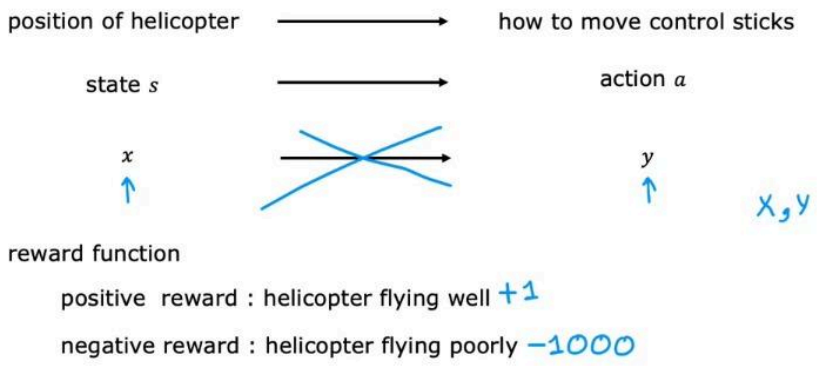
```
item_NN = tf.keras.models.Sequential([
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(32 )
])
```

```
# 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 3: Reinforcement Learning

- Could use supervised learning to learn states and then output actions; however, hard to identify an exact state
- Reinforcement learning - use a reward function

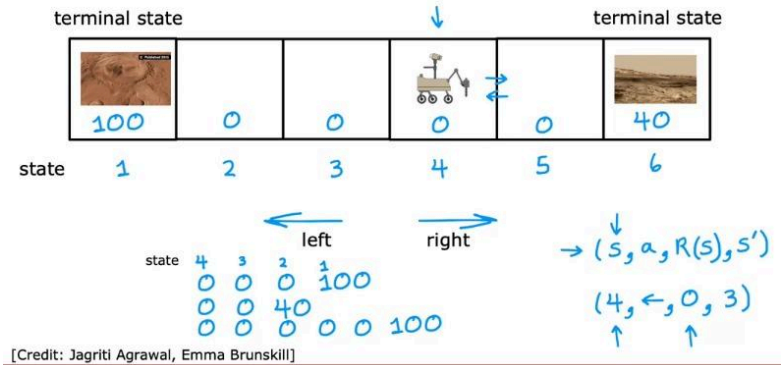


Applications:

- Controlling robots
- Factory optimization (maximising throughput)
- Financial (stock) trading
- Playing games (including video games)

The Mars rover example

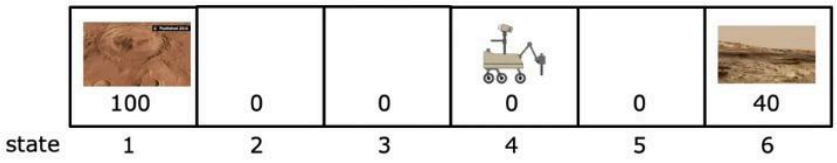
- Rover can be any of six positions (states)
- In this example, state 1 has a more interesting sample than 6 but is further away. To incentivise, give a higher reward at state 1.
- At each step, rover can either go to left or go to the right.
- 1 and 6 are terminal states - game ends/ nothing more happens after
- S = state, a = action, R(s) = reward, s' = next state



The Return in reinforcement learning

How do you know if a particular set of rewards is better than another?

- I.e. rewards that you can get quicker may be better to take
- *Return* is the sum of rewards, with a discount factor applied (normally about 0.9, 0.99 etc - i.e. close to one)



Return $= 0 + (0.9)0 + (0.9)^2 0 + (0.9)^3 100 = 0.729 \times 100 = 72.9$

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

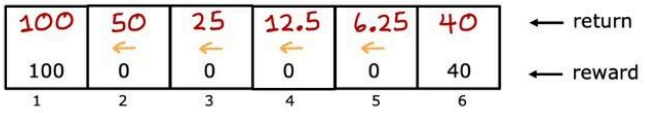
Discount Factor $\gamma = 0.9 \quad 0.99 \quad 0.999$

$\gamma = 0.5$

Return $= 0 + (0.5)0 + (0.5)^2 0 + (0.5)^3 100$

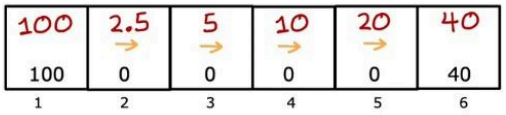
More concrete examples of returns:

- Return depends on the rewards, and the rewards depend on the actions you take - so the return depends on the actions you take
- Starting at different states and only going in one direction…

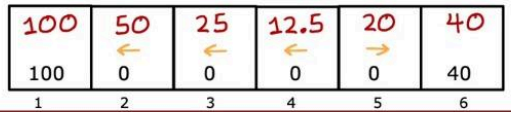


$\gamma = 0.5$

The return depends on the actions you take.



$0 + (0.5)0 + (0.5)^2 40 = 10$



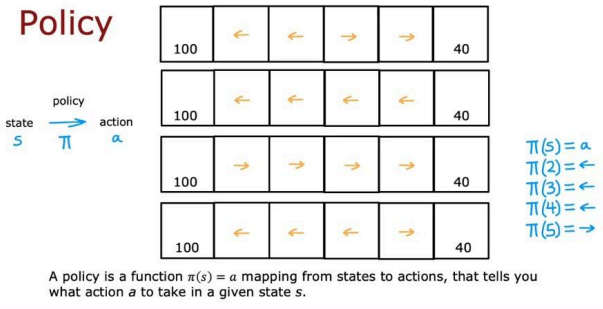
$0 + (0.5)40 = 20$

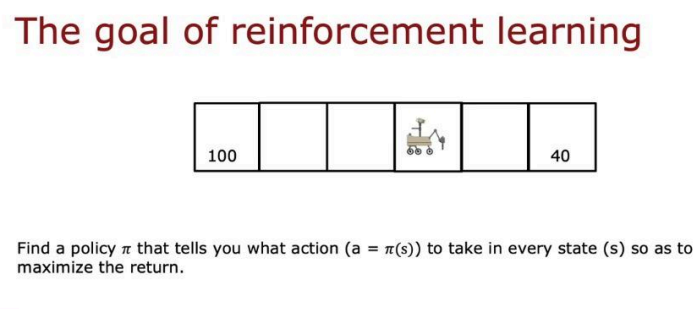
With negative rewards, this discount factor incentivises system to push out negative rewards as far into the future as possible.

Making decisions: Policies in reinforcement learning

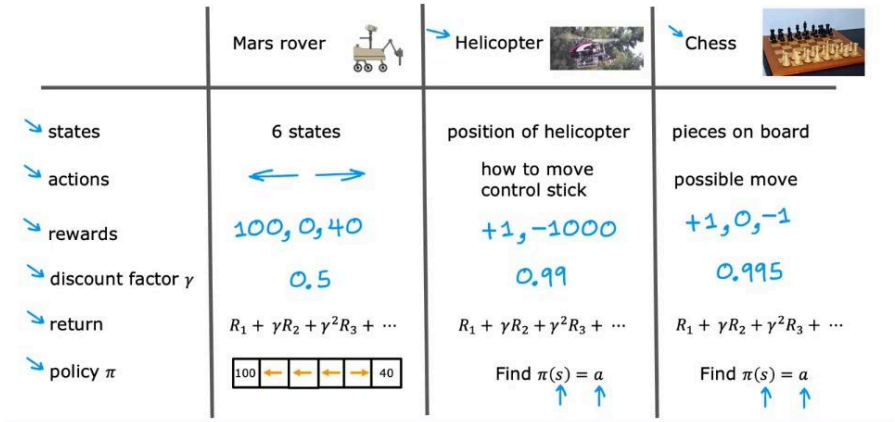
Policies:

- Could decide to always go to a nearer/larger/smaller one etc.
- **A policy maps a state to an action**



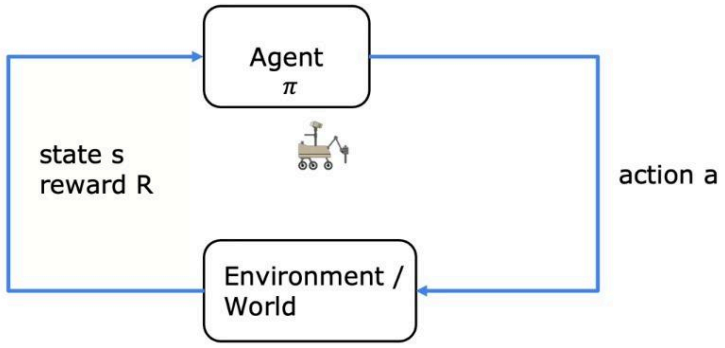


Summary



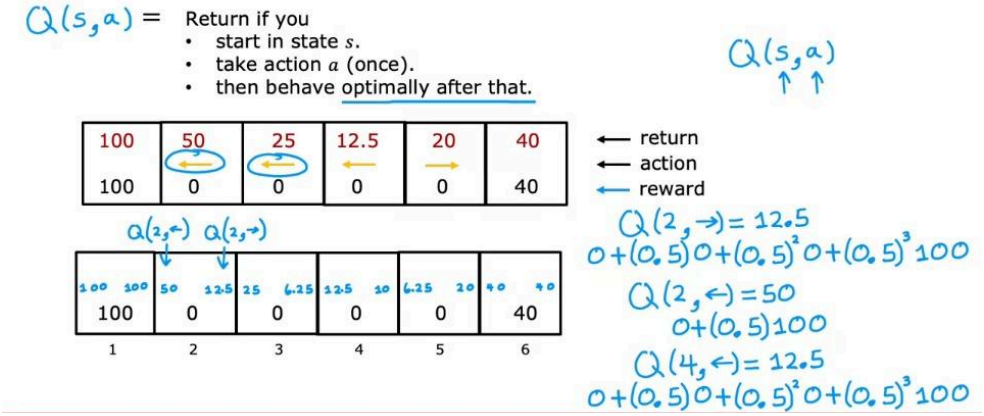
Markov Decision Process (MDP):

- Markov - future only depends on current state, not on how you got here
- Can think of it as: you have an agent, choose action a, based on action a something will happen in world, we then see what state we in and what reward R we get



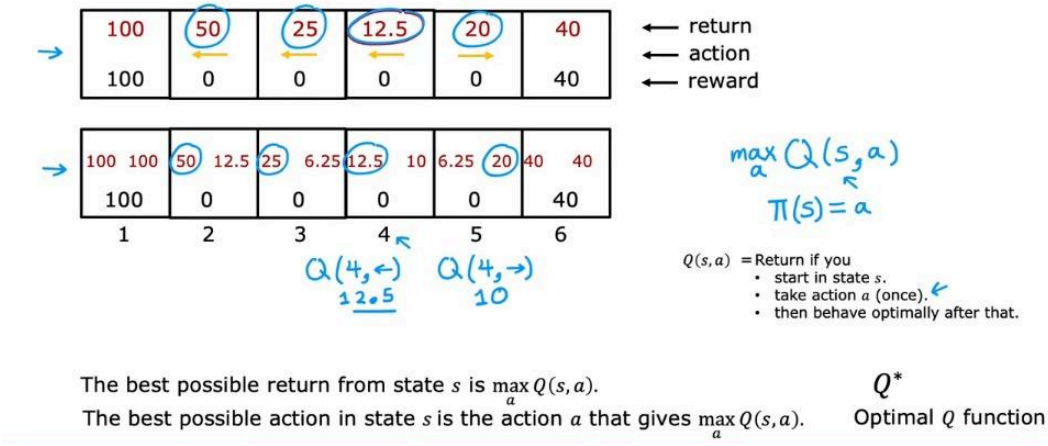
State-action value function, Q

Q(s,a) - function of state and action



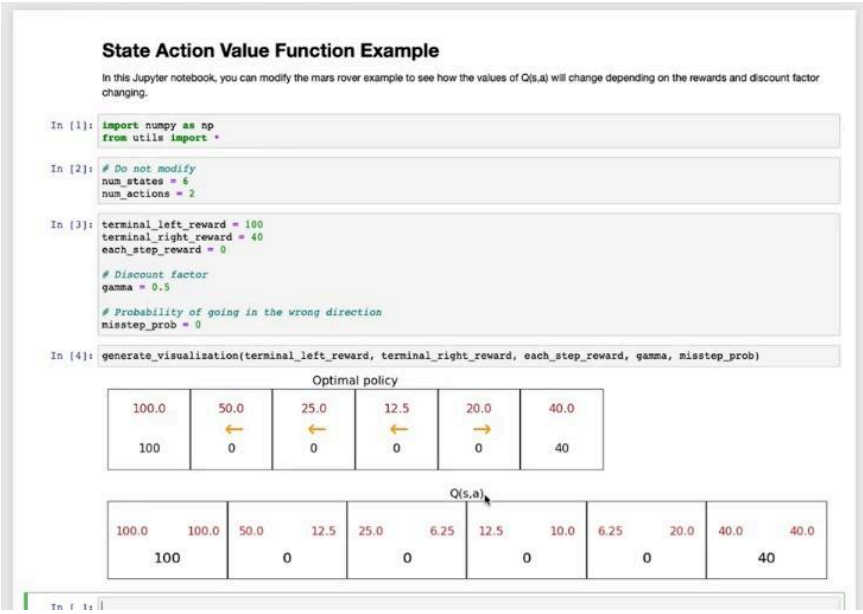
Q function gives you a way to pick actions too…

- **The best possible return from state s is max Q(s,a)**



Example of state-action value function:

- Discount factor is a measure of patience - how beneficial it is to wait
- See how values of Q(s,a) change and how optimal policy changes



State Action Value Function Example

In this Jupyter notebook, you can modify the mars rover example to see how the values of Q(s,a) will change depending on the rewards and discount factor changing.

```
In [1]: import numpy as np
        from utils import *

In [2]: # Do not modify
        num_states = 6
        num_actions = 2

In [3]: terminal_left_reward = 100
        terminal_right_reward = 40
        each_step_reward = 0

        # Discount factor
        gamma = 0.5

        # Probability of going in the wrong direction
        misstep_prob = 0

In [4]: generate_visualization(terminal_left_reward, terminal_right_reward, each_step_reward, gamma, misstep_prob)
```

Optimal policy

100.0	50.0	25.0	12.5	20.0	40.0
100	← 0	← 0	← 0	→ 0	40

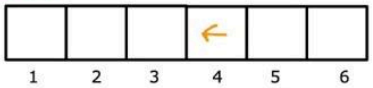
Q(s,a)

100.0 100.0	50.0 12.5	25.0 6.25	12.5 10.0	6.25 20.0	40.0 40.0
100	0	0	0	0	40

Bellman equation:

- Allows you to compute Q(s,a)...

$Q(s,a)$ = Return if you
- start in state s.
- take action a (once).
- then behave optimally after that.



$R(1)=100$ $R(2)=0$ ... $R(6)=40$

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

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

$Q(s,a) = R(s)$

100 100	50 12.5	25 6.25	12.5 10	6.25 20	40 40
100	0	0	0	0	40
1	2	3	4	5	6

$s = 2$
$a = \rightarrow$
$s' = 3$

$Q(2,\rightarrow) = R(2) + 0.5 \max_{a'} Q(3,a')$
$= 0 + (0.5)25 = 12.5$

$s = 4$
$a = \leftarrow$
$s' = 3$

$Q(4,\leftarrow) = R(4) + 0.5 \max_{a'} Q(3,a')$
$= 0 + (0.5)25 = 12.5$

Looking at an example:

The best possible return from state s is max Qa(s,a).
The best possible return from state s' is max(a')Q(s', a').

$Q(s,a)$ = Return if you
- start in state s.
- take action a (once).
- then behave optimally after that.

$s \rightarrow s'$

→ The best possible return from state s' is $\max_{a'} Q(s', a')$

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

Reward you get right away

Return from behaving optimally starting from state s'.

$$Q(s,a) = R_1 + \gamma [R_2 + \gamma R_3 + \gamma^2 R_4 + \dots]$$

To relate this back to the earlier example…

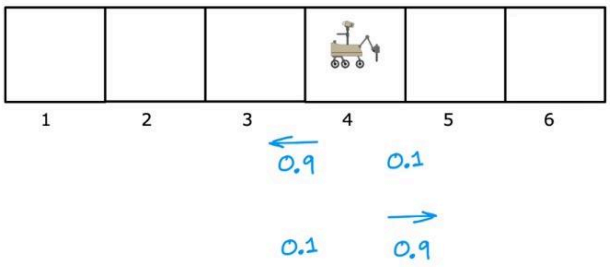
$$Q(s,a) = R(s) + \gamma \max_{a'} Q(s', a')$$

100 100	50 12.5	25 6.25	12.5 10	6.25 20	40 40
100	0	0	0	0	40
1	2	3	4	5	6

$Q(4, \leftarrow)$
$= 0 + (0.5)0 + (0.5)^2 0 + (0.5)^3 100$
$= R(4) + (0.5)[0 + (0.5)0 + (0.5)^2 100]$
$= R(4) + (0.5) \max_{a'} Q(3, a')$

Random (stochastic) environment

- In some applications, when you take an action, outcome is not completely reliable (e.g. wind blowing or wheel slipping with robots etc.)...
- E.g. if you command it to go left, 10% of the time it might slip and go in the other direction, and vice versa



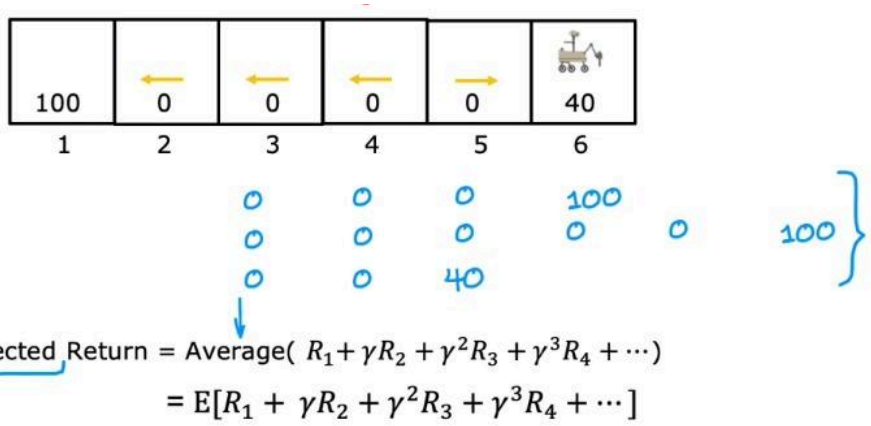
Expected Return:

- E.g. use policy where you go left in states 4,3,2 and right in state 5
- E.g.1 states you visit will be a little random due to these percentages (0.9, 0.1) - might start in state 4, get a bit lucky and go to state 3, then to state 2, and to state 1 (giving 0 0 0 100)
- E.g.2 but might get less lucky - tell it to go left, and works; but then tell it to go right, and it slips and go right, and then goes left for the rest (0 0 0 0 0 100)
- E.g.3 start in four, say go left and it slips and goes right, then command it to go right (0 0 40)

We had previously written out return as sum of discounted rewards. But when the reinforcement learning problem is stochastic, there isnt one sequence of rewards you see for sure, but lots of different ones. Thus, in a stochastic learning problem, we dont maximise return (as this is a random number) - instead we want to maximise the <u>average value of the sum of discounted rewards</u> (i.e. if we take policy and try it out thousands of time, get lots of different sequences and then take average to get expected (average) return.)

(E = expected value)

Learning algorithm aims to choose a policy pi to maximise the average of the expected sum of discounted rewards.



Here: s' is random, so we put an average E:

Goal of Reinforcement Learning:

Choose a policy $\pi(s) = a$ that will tell us what action a to take in state s so as to maximize the expected return.

Bellman Equation:

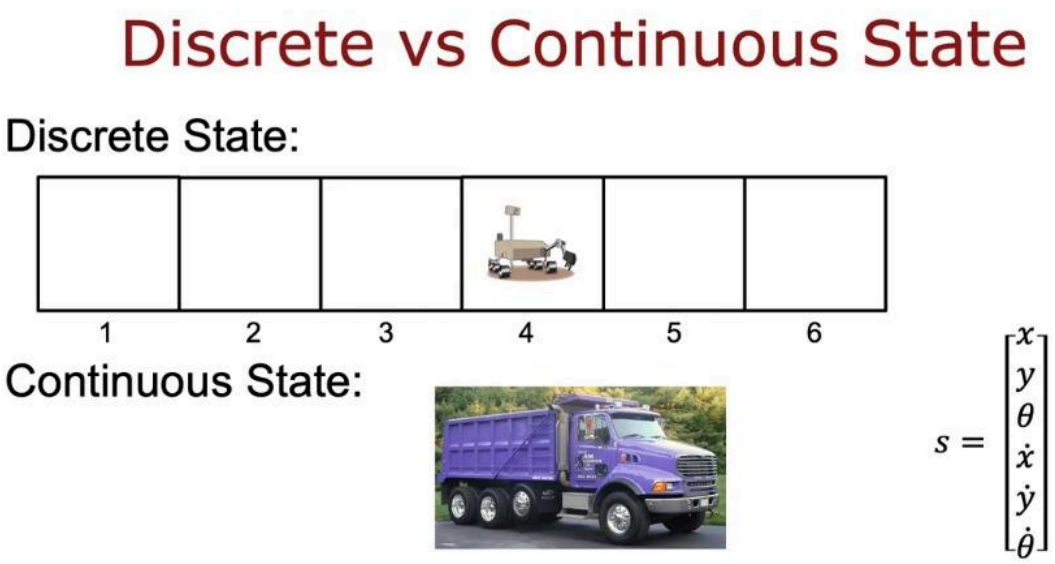
$$Q(s,a) = R(s) + \gamma \, E[\max_{a'} Q(s',a')]$$

3 ← 2 or 4

- In the lab, this probability of slipping etc. is called the “misstep_prob”

<u>Example of continuous state space applications</u>

- How to generalise to a continuous state instead of six discrete states (like the mars rover) e.g. if it could be anywhere along a line between 0 and 6km
- E.g. building a self-driving a car (xy position, orientation (angle theta), speed in a direction, and how quickly it is turning) - i.e. becomes a vector:



E.g.2 Position of autonomous helicopter
X (how far north or south), y (how far east or west), z (height above group) + orientation (row (rowing left to right), pitch (up or back), yor (which compass direction it is facing)

$s = \begin{bmatrix} x \\ y \\ z \\ \phi \\ \theta \\ \omega \\ \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\phi} \\ \dot{\theta} \\ \dot{\omega} \end{bmatrix}$

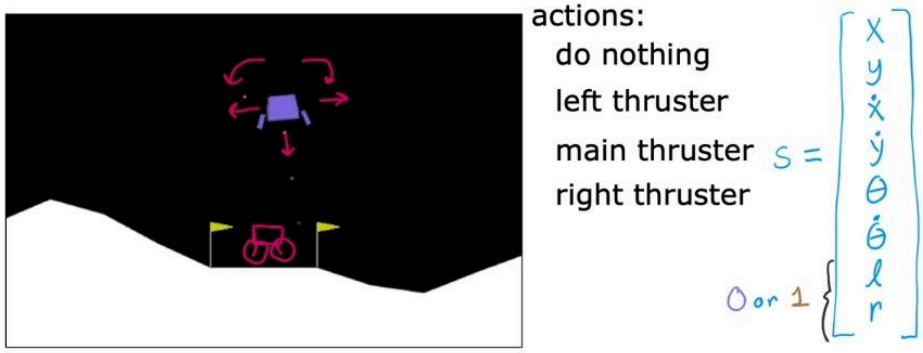
Position (x,y,z); Row (phi), pitch (theta), yor (omega); speed (in x, y, z directions), rate of turning (how fast is row, pitch, and yor changing)

- This set of 12 numbers is input into a policy, which decides what the appropriate next action is

Thus, in a continuous state MDP, the state of the problem is a vector of continuous numbers.

Lunar Lander

Four possible actions:



- In vector, theta is tilt to left or right (angular velocity); l and r correspond to whether left or right leg is sitting on ground (binary values)

Reward function:

- Getting to landing pad: 100 – 140
- Additional reward for moving toward/away from pad.
- Crash: -100
- Soft landing: +100
- Leg grounded: +10
- Fire main engine: -0.3
- Fire side thruster: -0.03

So overall problem is:

Learn a policy π that, given

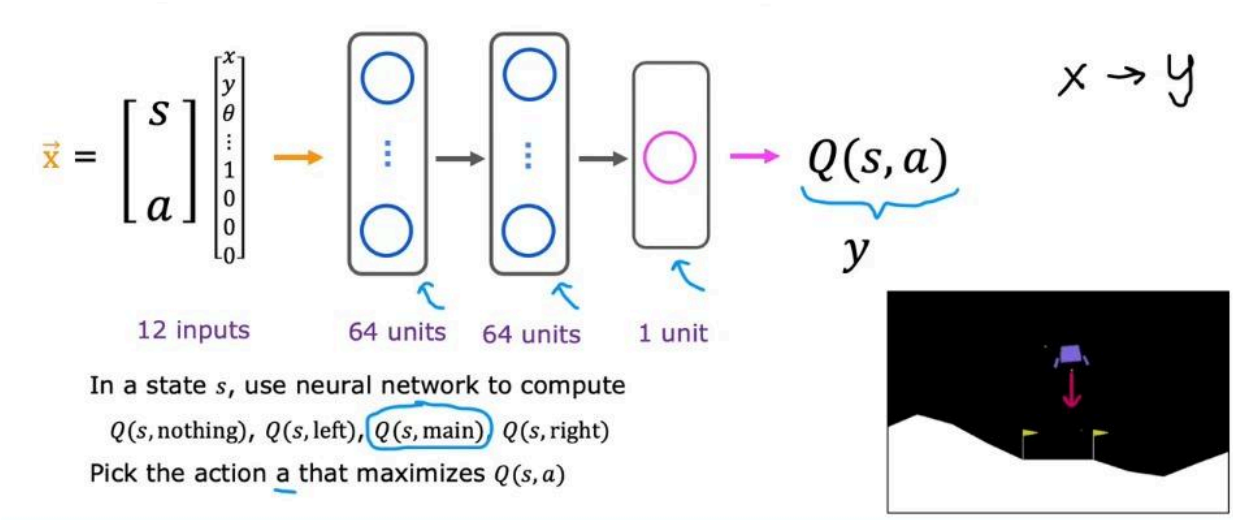
$$s = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \\ \theta \\ \dot{\theta} \\ l \\ r \end{bmatrix}$$

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

$$\gamma = 0.985$$

Deep Reinforcement Learning: Learning the state-value function

Train a neural network to approximate Q(s,a), which will in turn allow us to pick good actions.



Encode any of four actions using a one-hot feature vector (e.g. for “do nothing”: encode with 1 0 0 0)

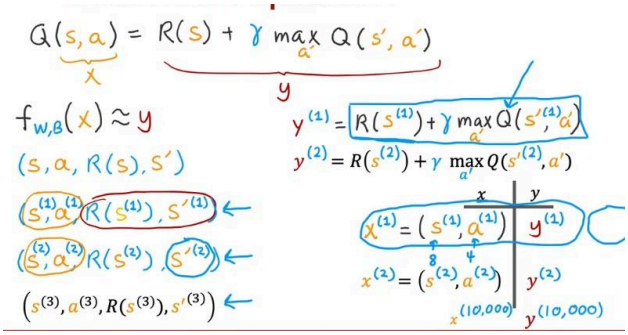
- Input a state-action pair and use it to output Q(s,a) for each of the four actions, and pick the action a that maximises Q(s,a)

So, how do you train this neural network?

- Use Bellman’s equation to create a training set with lots of examples x and y, and then use supervised learning to map from state-action pair to Q(s,a) (i.e. accurately predict x→y)

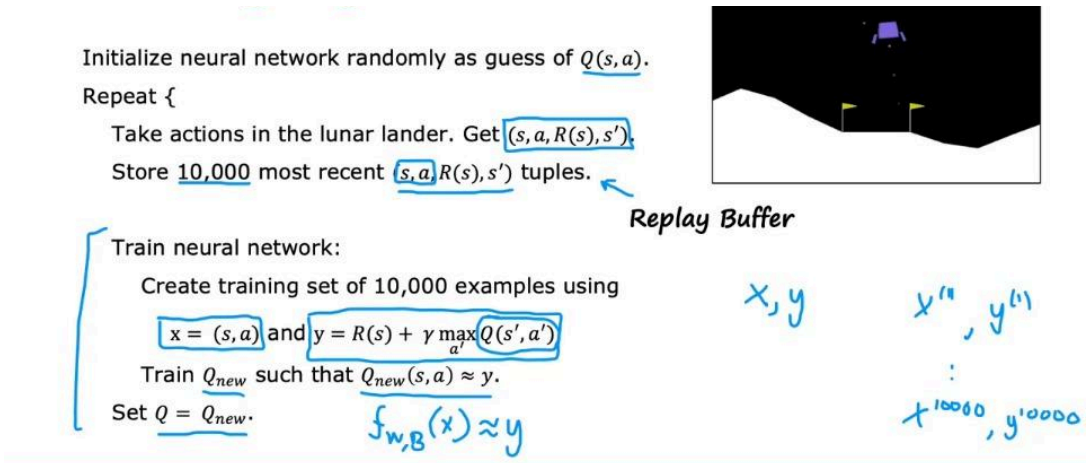
So how do you get the training set with values of x and y?

- Take random actions to start with, to see what rewards you get. Each tuple can represent a single training example
- Xs are inputs with 12 features, ys are just numbers, then train neural network with the mean squared error loss to try to predict y as a function of the input x



Where does Q(s',a') come from? Can take a random guess of the actual Q function, even though we do not know.

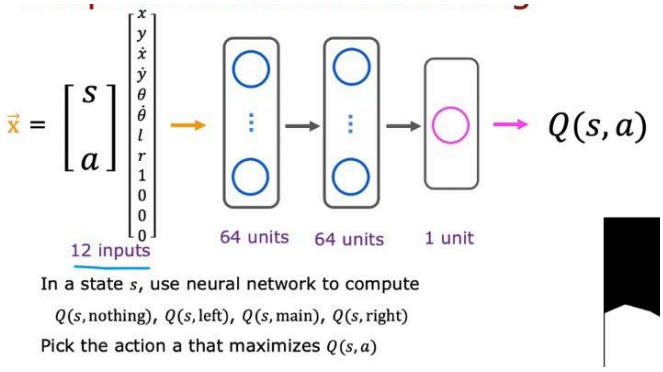
Full Learning Algorithm (DQN algorithm)



Starting a random guess of Q, using Bellman's equations to repeatedly improve guess of Q, slightly improves guess, so for next model you train will have a slightly better estimate of Q, which then updates Q.

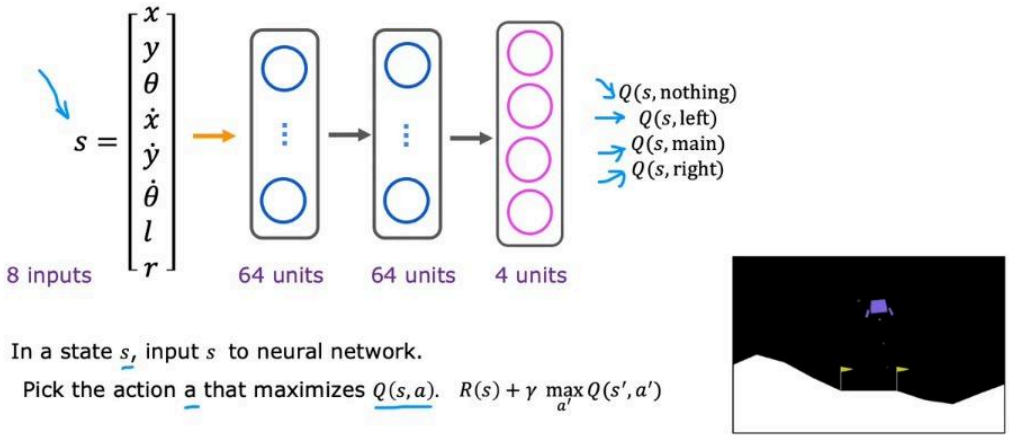
Algorithm refinement: Improved neural network architecture

- Possible change to architecture to make algo much more efficient…



Here, whenever we are in some state s, have to carry out inferences in the NN separately four times to compute four values of Q (nothing, right etc.), to pick action a that maximises Q(s,a).

- More efficient to output all four simultaneously



This NN also makes it much more efficient to compute R(s) + …, because we are getting Q(s',a') for all actions at the same time, so just pick the max to compute this value.

Algorithm refinement: epsilon-greedy policy

To pick actions while you're still learning (i.e. don't have a good estimate of Q(s,a) yet), use the epsilon-greedy policy…

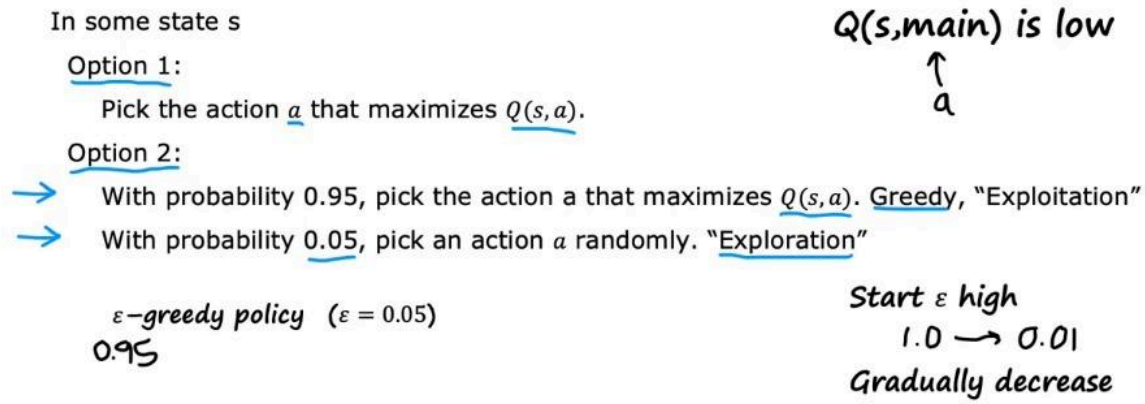
Initialize neural network randomly as guess of $Q(s,a)$.
Repeat {
Take actions in the lunar lander. Get $(s, a, R(s), s')$.
Store 10,000 most recent $(s, a, R(s), s')$ tuples.

Train model:
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 $Q_{new}(s,a) \approx y$. $f_{w,b}(x) \approx y$
Set $Q = Q_{new}$.

How to choose actions while still learning? Some options… (Option 2 is best)

- Occasionally pick an action randomly. Suppose that Q(s,a) was initialised randomly so that Q(s,main) is always low - then NN will never try to fire main thruster, and thus will never learn that firing main thruster is a good idea. Randomness allows it to try out new actions, so NN can overcome its own possible preconceptions - "exploration step".
- Exploration vs exploitation trade-off

- Trick: start with epsilon high, and then gradually decrease so that over time you are more likely to use improving estimate of Q to decide on actions instead of random actions

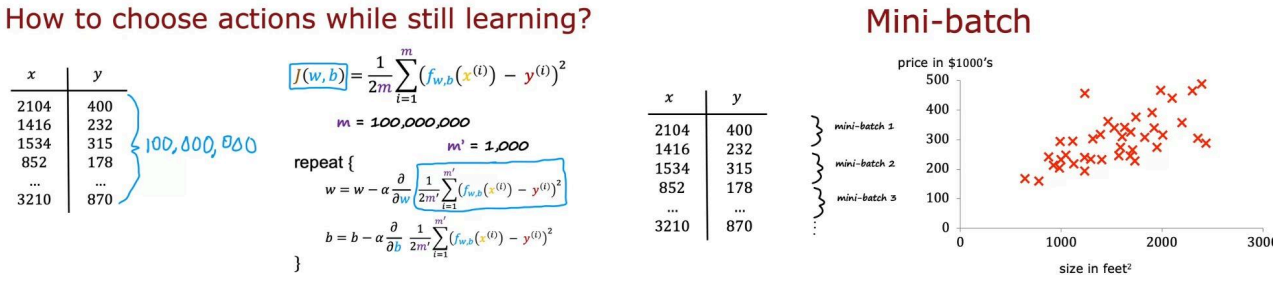


Algorithm refinement: Mini-batch and soft updates

Can speed up supervised learning algorithms as well.

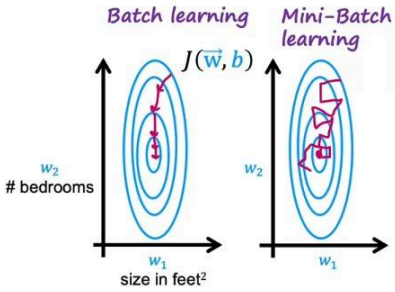
Mini-batches

- Problem with gradient descent (below) when dataset is very big is that every step of grad descent requires computing sum of this average over 10000000 examples. Then have to do it again after a tiny grad descent step.
- Mini-batch gradient descent does not use all training examples on every iteration, but instead use a smaller number m' (a subset) - much more efficient



Recall *batch gradient descent*: every step causes parameters to reliably get closer to global minimum. In contrast, *mini-batch* will take a more convoluted route towards global minimum, but each iteration is much less computationally expensive.

- For a large dataset, mini batch (with other optimisation algos like adam, for example) is much more common than batch grad descent



Updated previous learning algorithm

- Just take a subset of the examples of the tuples to train a fewer examples of the NN - accelerates it (even if it makes it a bit more noisy)
- Other refinement: Set Q = Qnew can make an abrupt change to Q (as can be overwritten by a worse algorithm… use Soft Update to remedy this:)

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:

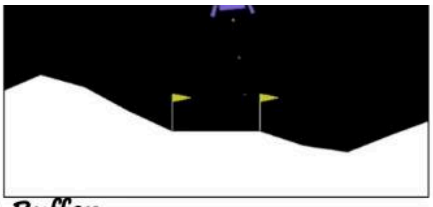
1,000

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 $Q = Q_{new}$.



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

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

Soft Update

- Soft update only adds a small amount of Wnew (the new value) - controls how aggressively you move towards Wnew and Bnew from W and B (soft update allows a more gradual change)
- Allows more reliable convergence

Set $Q = Q_{new}$. ← $Q(s,a)$

W, B W_{new}, B_{new}

$W = 0.01\, W_{new} + 0.99\, W$ $\quad W = 1 W_{new} + 0 W$

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

- NB mini batching also applies well to supervised learning applications

Limitations of Reinforcement Learning

- Much easier to get to work in a simulation than a real robot!
- Far fewer applications than supervised and unsupervised learning.
- But … exciting research direction with potential for future applications.