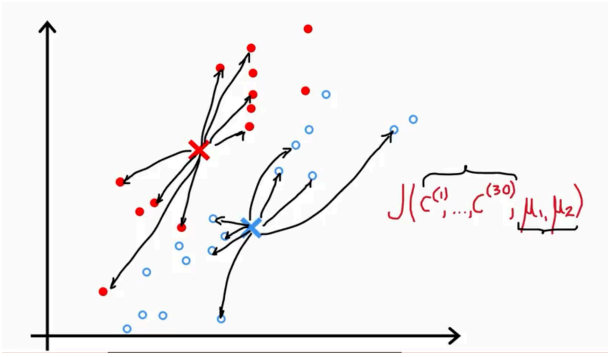
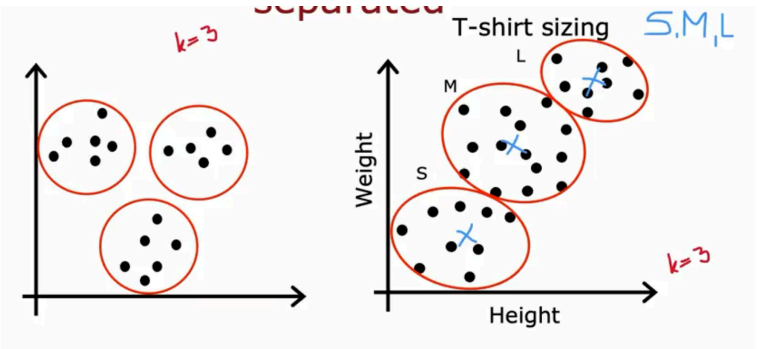
Course 3: Unsupervised Learning, Recommenders, Reinforcement Learning

Week 1:

Clustering

- K-Means Clustering:
  1. Initialize K amount of cluster centroid on random data points
  2. Find class membership (square distance) of *n* amount of object in relation to the nearest cluster centroid
  3. Relocate centroids to the average distance (mean) from all members of said centroid and reevaluate the objects until the membership (cost function) score no longer changes





- The cost function in k-means clustering is the aggregate of the distances computed between data points and their assigned centroids. It helps determine whether the overall clustering has improved in each iteration. A decreasing cost function indicates better-defined clusters, guiding convergence. Without it, there would be no clear way to measure or compare clustering quality across iterations.

Cost function for K-means

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

Repeat {

 # Assign points to cluster centroids

for i = 1 to m

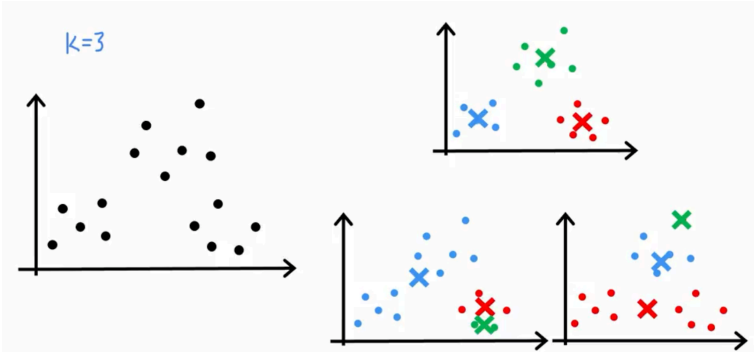
$c^{(i)}$:= index of cluster centroid closest to $x^{(i)}$

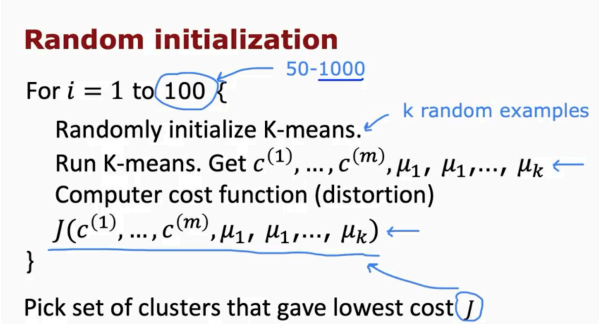
 # Move cluster centroids

for k = 1 to K

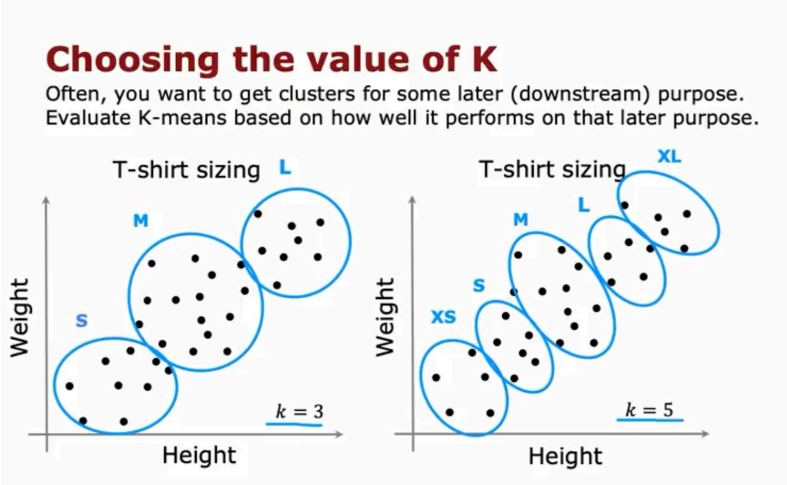
μ_k := average of points in cluster k

- Initializing K-means centroids randomly on data points helps overcome local minima and vastly different clusters.



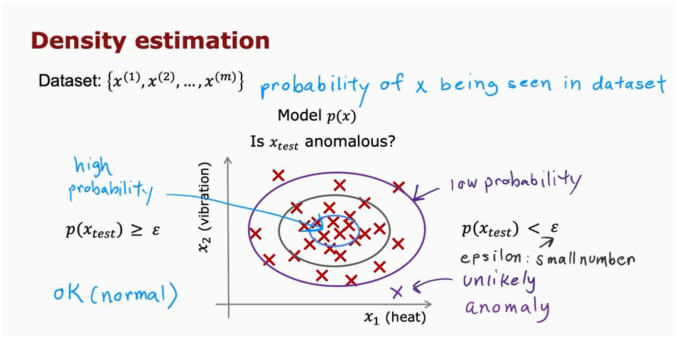


- Choosing amount of clusters:
    - Use square error to **1 to *n*** amount of clusters to determine the ideal amount of clusters (elbow method).
    - Evaluate based on overarching purpose of clustering **(Ideal)**

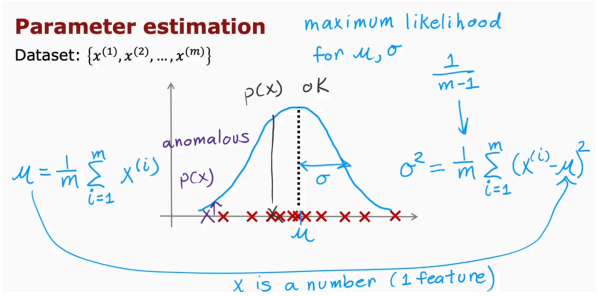


Anomaly Detection:

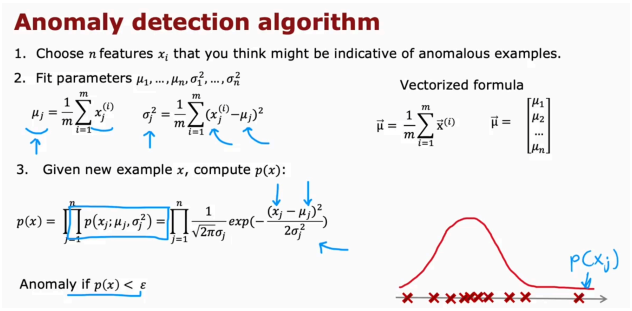
- Anomaly detection places data instances based off of density which is dependent on high probability to low probability. By then comparing test data with the density ranges we can determine if the test data is an anomaly or not.

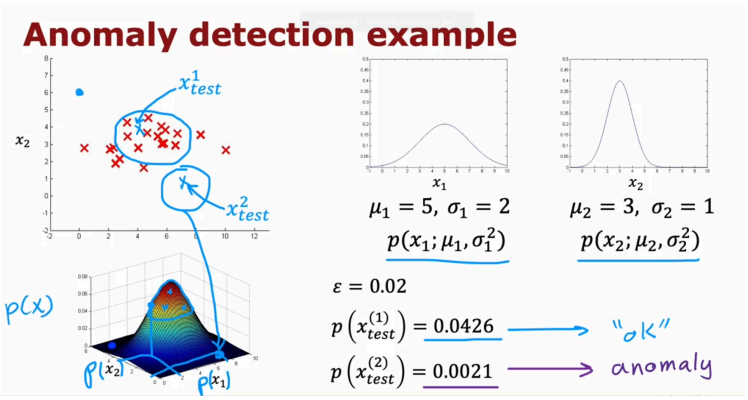


- Normal Distribution leverages standard deviation and variance as parameters to plot data points. The farther away from the center you get, the higher the likelihood of the data point being *anomalous.*



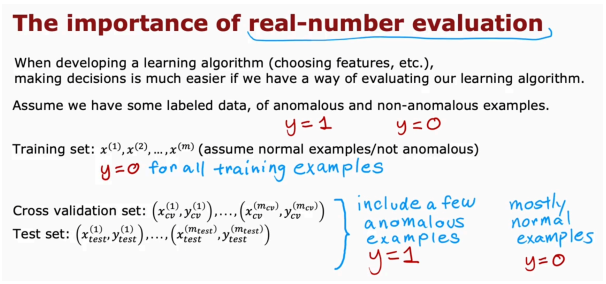
- To determine if a new instance of *x* is an anomaly or not, we determine its probability '*p(x)*'. If *p(x)* < *epsilon (a small number)* we consider the new instance *x* as anomalous.

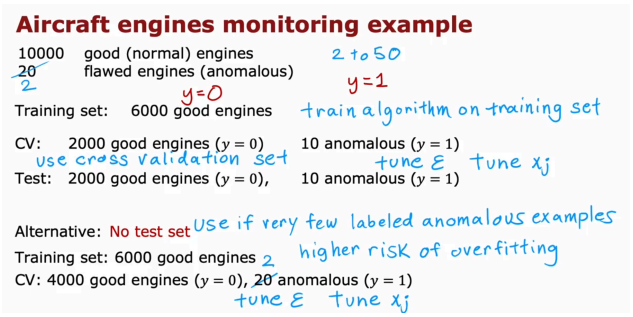




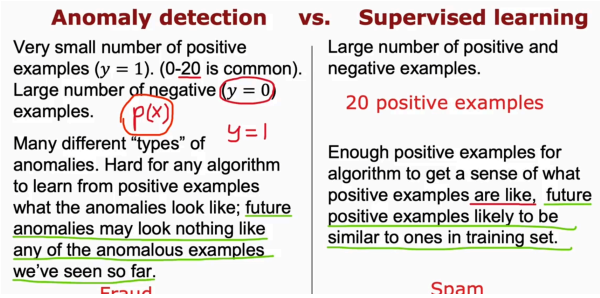
- Evaluating an Anomaly Algorithm by using CV and Test sets to tune *epsilon* and predict *anomaly* of data instance using *p(x)*:

- Use *precision, recall and F1-scores* as possible evaluation metrics for the data.

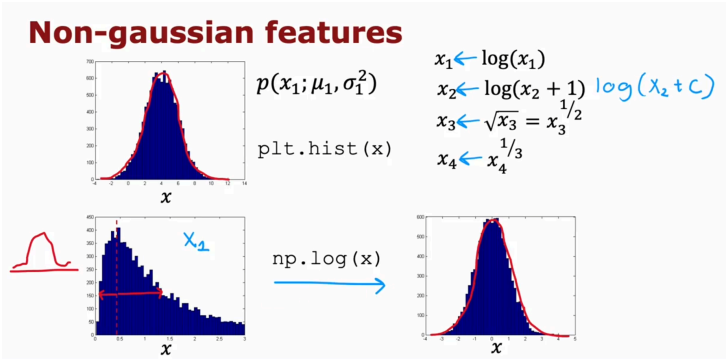




- Anomaly Detection vs Supervised Learning
  - Anomaly detection models are modeled based off of the normal examples in the dataset, as such anything that deviates from the normal is considered an anomaly allowing for brand new instances of deviations to be quickly flagged as anomalies even if the model has never seen it before. (Detecting previously unseen/new issues that may arise in a products (data centers, engines, phones) lifetime)
  - Supervised models are trained to detect similar types of data or “anomalies”. As such supervised learning models wont perform well on new variations of data it has never seen before as it can't predict what it has never experienced during training. (Detecting known issues, instances (weather).



- Features to choose:
  - Turn non-gaussian features into gaussian using relevant mathematical transformations (transformation mustang be same for train/CV/test sets)
  - Find or create a new feature (by combining features) to fit gaussian better (sometimes p(x) can be comparable to normal and anomalous examples

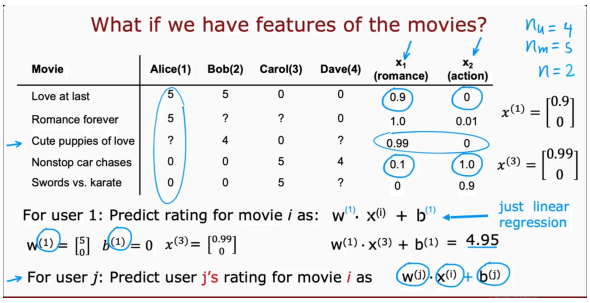


Week 2:

Collaborative filtering

Note: For the following notes, consider the example of users rating various movies. This way the notes will be easier to understand.

- **To predict the rating of a movie: *W*** is the weight parameter determined using previous ratings of movies, ***x*** is the feature data of how much a certain movie is considered romance or action. **(User must have rated the movie, unrated movies don't count)**



- Cost function will only sum over movies that have been rated by user ***j.*** Note: ***n*** in the regularization term is the number of features.

Notation:

$r(i,j) = 1$ if user j has rated movie i (0 otherwise)
$y^{(i,j)}$ = rating given by user j on movie i (if defined)
$w^{(j)}$, $b^{(j)}$ = parameters for user j
$x^{(i)}$ = feature vector for movie i

For user j and movie i, predict rating: $w^{(j)} \cdot x^{(i)} + b^{(j)}$
$m^{(j)}$ = no. of movies rated by user j
To learn $w^{(j)}$, $b^{(j)}$

Cost function

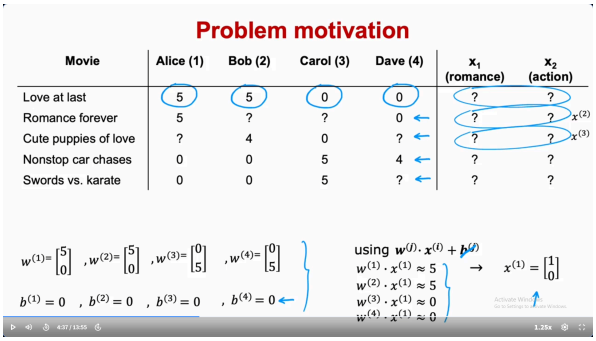
To learn parameters $w^{(j)}, b^{(j)}$ for user j :

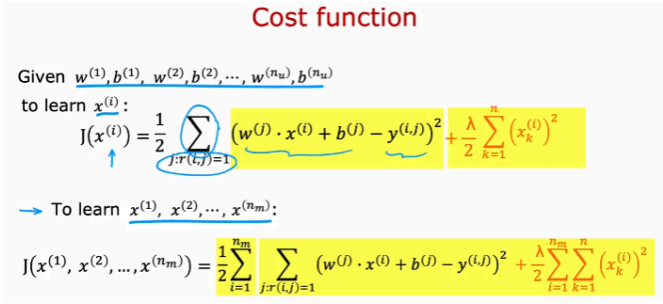
$$J(w^{(j)}, b^{(j)}) = \frac{1}{2} \sum_{i:r(i,j)=1} \left(w^{(j)} \cdot x^{(i)} + b^{(j)} - y^{(i,j)}\right)^2 + \frac{\lambda}{2} \sum_{k=1}^{n} \left(w_k^{(j)}\right)^2$$

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

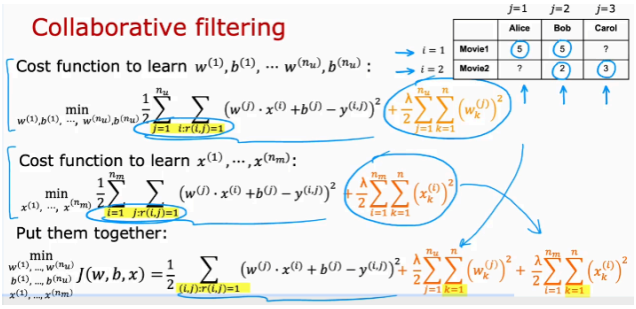
$$J\begin{pmatrix} w^{(1)}, \ldots, w^{(n_u)} \\ b^{(1)}, \ldots, b^{(n_u)} \end{pmatrix} = \frac{1}{2} \sum_{j=1}^{n_u} \underbrace{\sum_{i:r(i,j)=1} \left(w^{(j)} \cdot x^{(i)} + b^{(j)} - y^{(i,j)}\right)^2}_{f(x)} + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} \left(w_k^{(j)}\right)^2$$

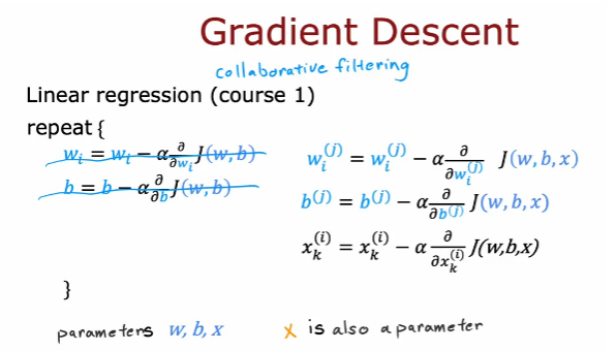
- In a scenario where features are not available, we can use parameter ***w, x*** and ***b*** to learn what values of the features **(this is considering that the parameters have been learnt in advance).**
- In collaborative filtering, we can come up with the values of features thanks to having data from multiple sources (Ex: Ratings from various users), as such this methodology can't work in typical linear regression as we lack the necessary information to make such deductions.





- In order to get the **parameters** for different users, we put together the algorithm to learn parameters and the algorithm to learn features.
- In this aggregated algorithm **(Collaborative Filtering)** we sum over **each movie** for **each user** (not the other way around) for **only the movies that have been rated**.
- Helps us find ***w, b and x*** all at the same time (***x*** also becomes a parameter).





- Using collaborative data (ratings) from multiple users for a single/multiple subject (movies), gives us the necessary information to appropriately guess the features of the subject (movie), which in turn helps us predict what potential user's (who have not rated the movie) may rate the movie in the future.
- Recommender systems are particularly efficient when working with **binary labels**
- In order to generalize a linear regression collaborative filtering model to work with binary labels we predict the probability of the **output label** being **1** (meaning the subject has been engaged, i.e being shown a movie), whereas before we'd predict whatever the **value of the output label is.**

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

Cost function for binary application

Previous cost function:

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

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

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

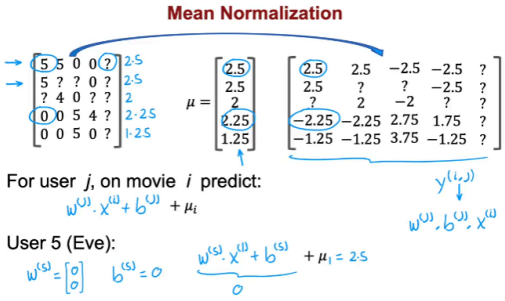
Loss for single example

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

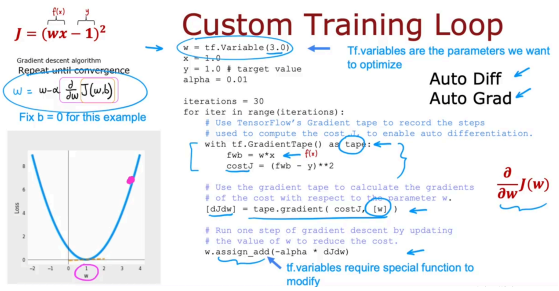
$g(w^{(j)} \cdot x^{(i)} + b^{(j)})$

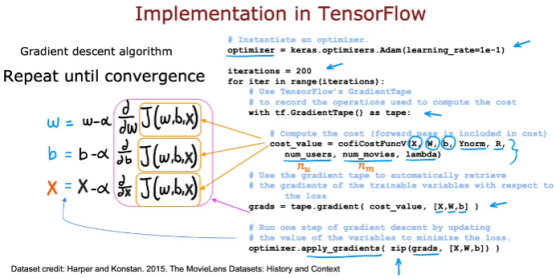
Recommender system implementation:

- Normalization also helps recommendation systems through the usage of mean normalization. This is important as without normalization, for a user without any ratings (considering movie example), the algorithm will predict '**0**', for all movies due to their parameters **w** and **b** being initialized as **'0'**
- In **mean normalization**, you convert all data points into a matrix and discover the mean of each class (in our case, each movie's mean rating). The mean is then subtracted from all the original data points (ratings) unless it is unknown/missing/empty.
- By implementing the mean into the algorithm's calculations, we gain a baseline prediction (for a movie's rating) when a user has no prior/reference data for said subject (rating)
- Normalizing works regardless of application on the column or the row, but its better to choose based on context (choosing row for users with no prior ratings, choosing column for movies with no prior rating).

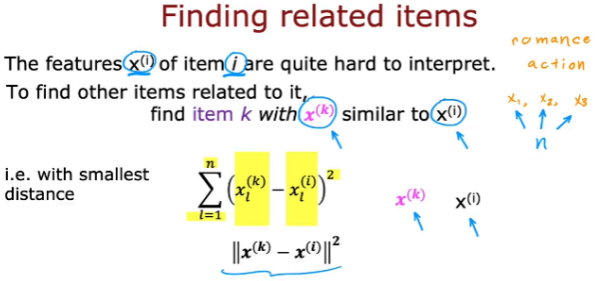


- Tensorflow implementation of gradient descent and differentiation followed by **implementation of collaborative filtering:**





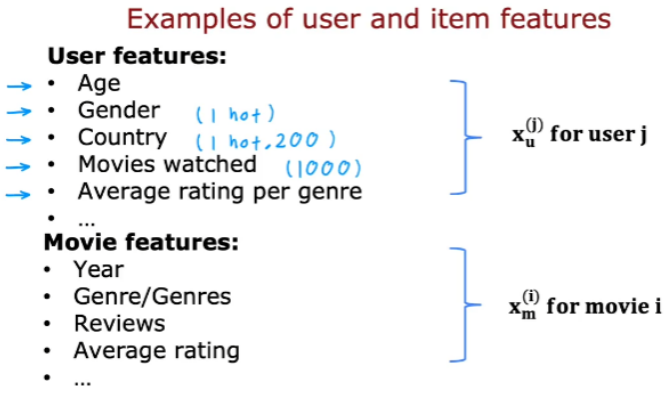
- Collaborative filtering algorithms can also help to find related items by finding features of item ***k*** similar to features of item ***i***. This is achieved with the help of distance, as the shortest distances between the features of two items, the more related they are.



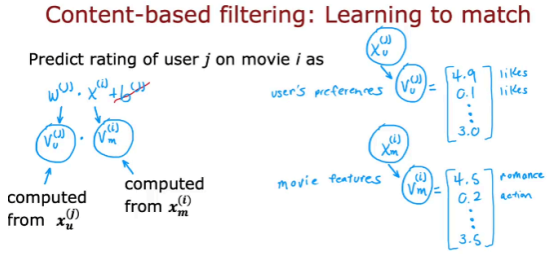
- Such an algorithm also suffers from **cold start** (How to rank new items that have very little data such as ratings? How to show new users with few rated items a reasonable set of items?) and **the inability to use side/additional information about items or users** (genre, actors, genders, location, age).

Content-based filtering:

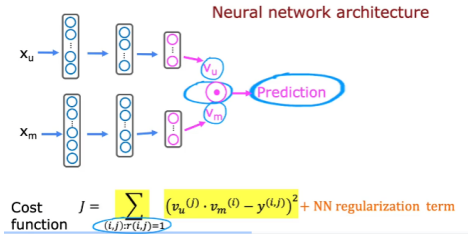
- While **collaborative filtering** recommends items based off of similarity of ratings between users, **content-based filtering** recommends based on the features of the user and items to find a good match.

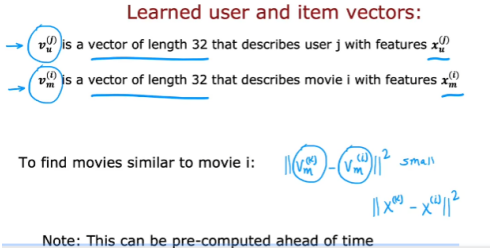


- Try to determine if movie ***j*** will be a good match for user ***i***. [**V_m** (Movie vector, parameter captures the likelihood of a set of genres) and **V_u** (User vector, parameter may capture preferences, etc.)]must be the same size].

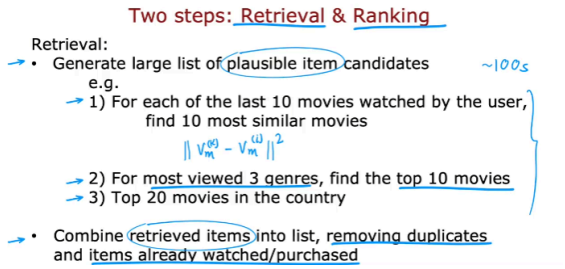


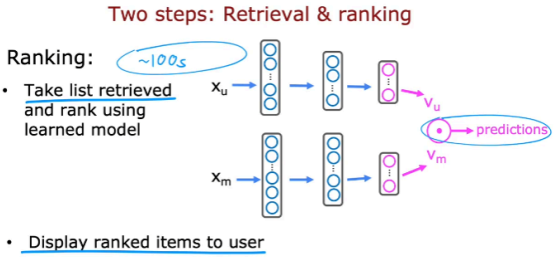
- To compute ***V_u*** and ***V_m*** using ***X_u*** and ***X_m***, we make use of neural networks with ***X_u*** and ***X_m*** as inputs and ***V_u*** and ***V_m*** as outputs. The dot product of ***V_u*** and ***V_m*** is then used to predict the probability of the output (likelihood of a good match) being **1.**

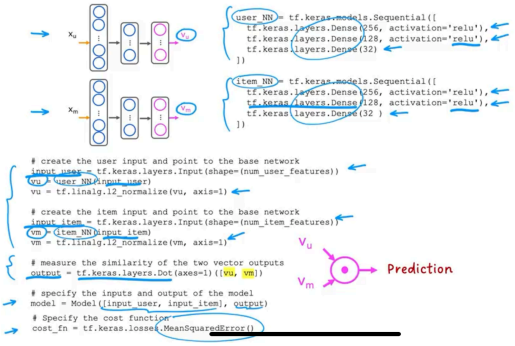




- To efficiently recommend a handful of relevant items we employ **retrieval & ranking.** Retrieving more items yields better performance at slower recommendations. Optimize trade-off by checking if retrieving additional items yields better recommendations

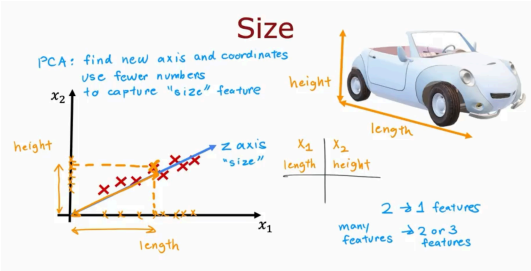


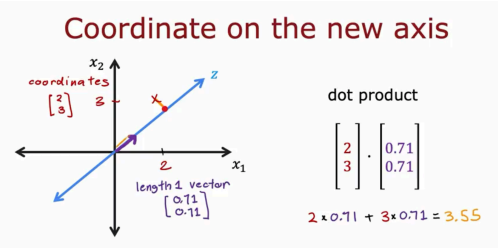


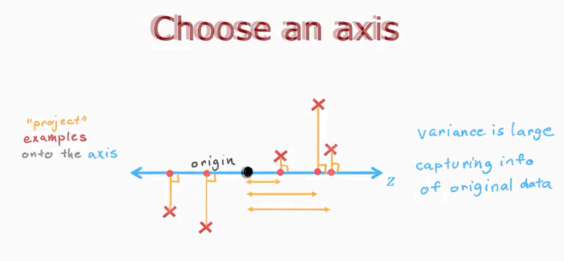


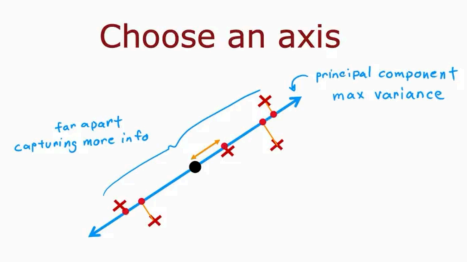
Principal Component Analysis:

- **PCA** is an unsupervised learning algorithm that can help **convert 3d or multi dimensional plots to 2d for easier visualization**
- PCA tends to **compress features to decrease large amounts of features into a smaller amount**
- Ensure to **normalize** and **feature scale** the features before finding the new axis
- The new axis is similar to a best fit line where **variance is maximum,** and we **capture the most amount of information** from the original feature data. This new axis is called the **principal component**

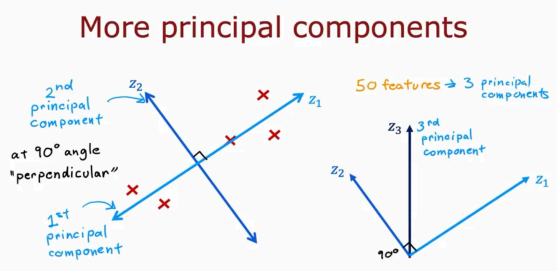




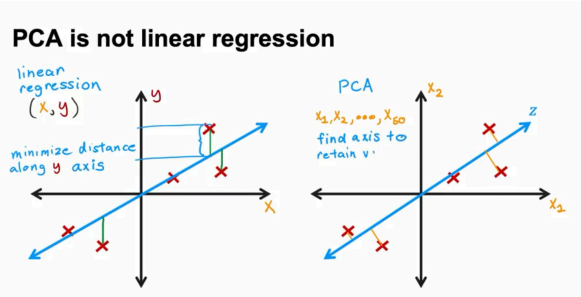


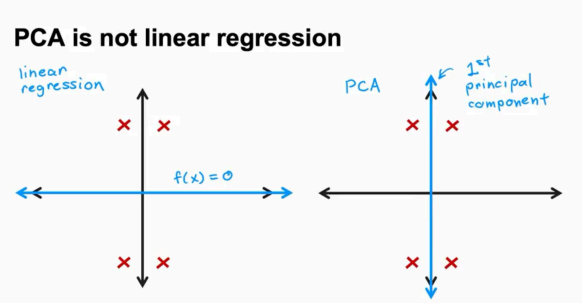


- A second principal component will be **90 degree to the first principal component** (similar for additional principal components)

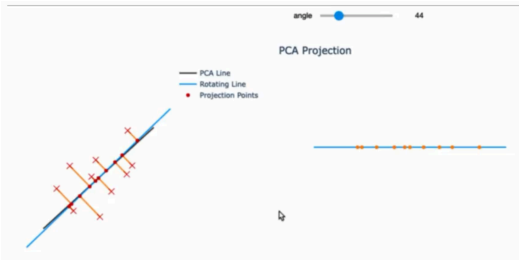


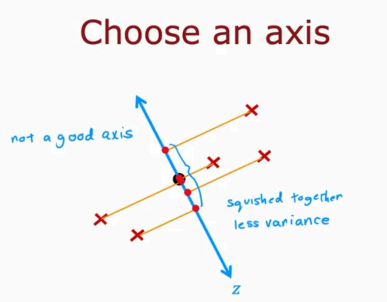
- Linear regression however focuses on finding the best fit line to predict output class ***y***. In PCA there is no output class, as such the new axis is treating all features equally and trying to retain as much information from all features as possible. So even if the 2D lines may seem the same, they are in truth, not the same.



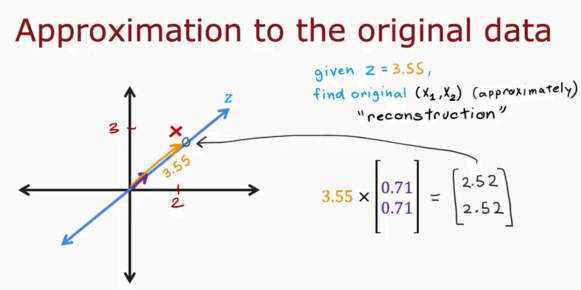


- Another way of thinking is that PCA will find an axis that distributes the features such that the features don't **overlap** or aren't **squished** together (if they are then we can't draw much information from that new axis)

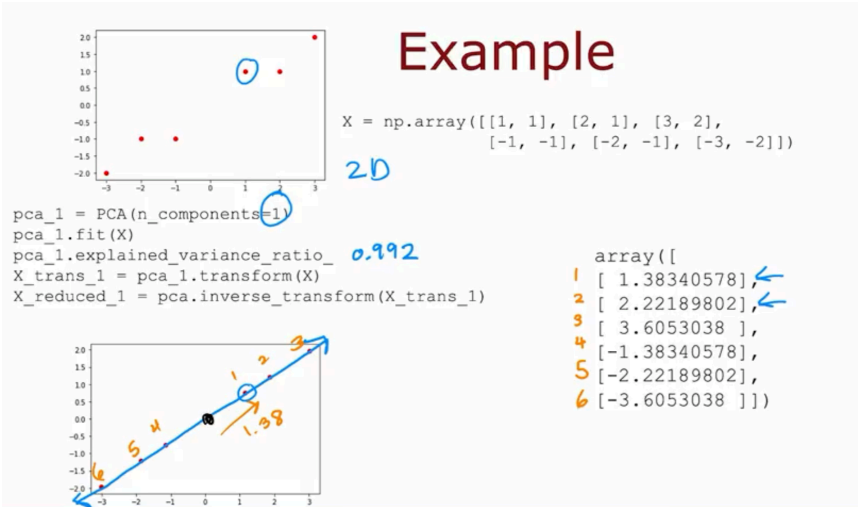




- We can reconstruct the original training example using the distance from origin to the new coordinate point on the principal component and multiplying with the vector of said principal component. This way we get an approximation of the original training examples from the original axis



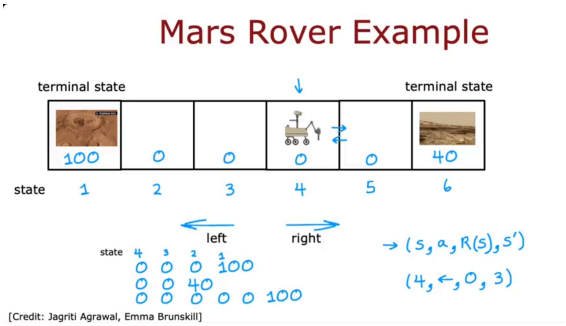
- Implementation:



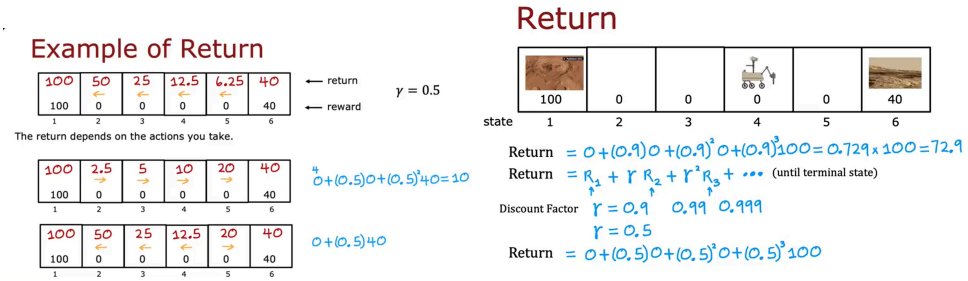
Week 3:

Reinforcement learning:

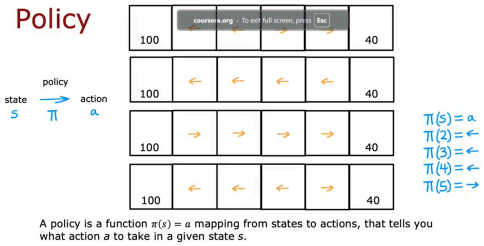
- We know reinforcement learning uses reward systems to incentivize models to train themselves in a manner that progresses them towards a goal. This is achieved by creating **states *s***, with each state having a set **reward *R(s)*,** and the model starts at a certain state and can then take an **action *a*,** to move to some new state ***s'*** at every **time step**. Combined we get **(s, a, R(s), s')**.



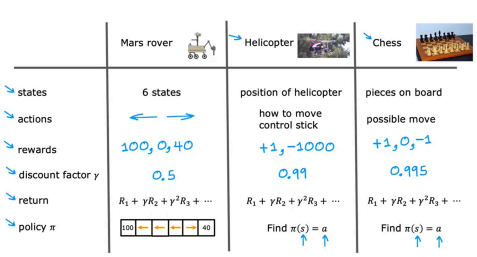
- The **Return** is the aggregate of all rewards experienced in each state until reaching the terminal state, with each reward being multiplied by a weight factor called the **discount factor.**
- The **discount factor** makes the algorithm less patient by decreasing the **value** or **credit** of the rewards, with the value decreasing the more states/time the algorithm takes to reach the terminal. This urges the algorithm to pick greater rewards sooner for higher overall yield.
- If **negative rewards** are part of the system, the discount factor is actually incentivized to **push out the negative rewards as far into the future as possible**.

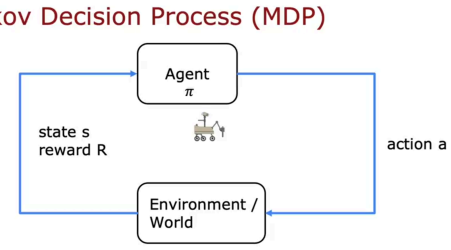


- We make use of a **policy *pi***, to map what action to take for a certain step. Thus: **pi(s) = a**



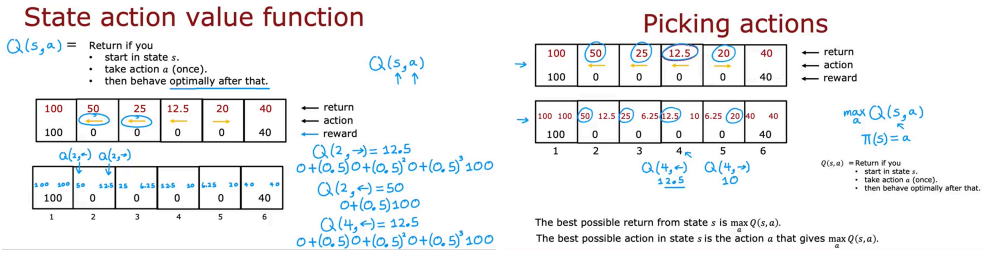
- The formalization of **states, rewards, actions, discount factor, return and policy** are formally known as **Markov Decision Process (MDP)** where the future is dependent only on the current state, the past matters not.



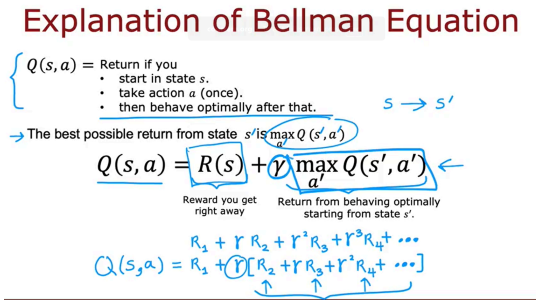


State-action value function

- State-action value function will provide a **return** based on the state and singular action specified, considering that the system behaves optimally after executing the specified action.
- This requires a system with defined policies and rewards to work.
- If, however, the **State-action value is defined *Q(s, a)***, we can use it to determine the best action for a given state based on the maximum state-action value in a given state.



- **Bellman equation** is the simplification of the **state-action value function** to calculate the state-action values for a given state with different initial actions as long as we know the max state-action value for the next state. **(Ex: In the Mars rover's case, we calculate the state-action value for state 4 with action for going left and right. If we choose to go left, we need the MAX state-action value of state 3 and if we go right we will need that of state 5)**



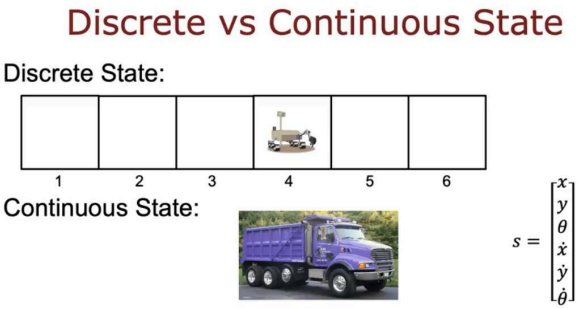
Note: While bellman's equation and the state-action value function looks similar and are virtually the same, notice that bellman's requires us to know the maximum state-action value of proceeding steps or, at the very least, calculate it.

- Sometimes going in the best direction may not be the most reliable/feasible (going left might be the quickest way to get home, but I might slip and go right instead).
- To account for this, we may assign a probability to the potential states (a higher probability to the direction with higher state-action value and vise-versa)
- Over many iterations we will receive varying kinds of reward sequences/state action values

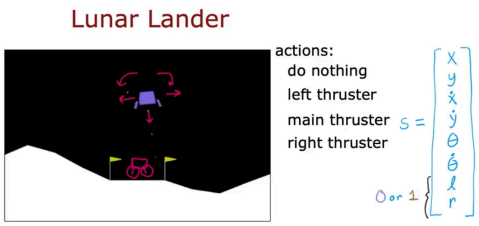
- In a **stochastic reinforcement learning** problem, what we're interested in is not maximizing the return because that's a random number. What we're interested in is **maximizing the average value of the sum of discounted rewards**. Say, if we were to take a policy and try it out a large amount of times, we'd get **lots of different reward sequences**. If we were to take the **average** o**ver all of these different sequences of the sum of discounted rewards**, then that's what we call the **expected return**.

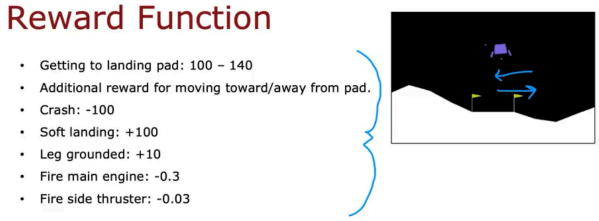
Continuous state reinforcement learning algorithm

- In a continuous state reinforcement learning algorithm (**Continuous MTP**) we use vectors of numbers or **continuous state spaces** to quantify states.
- Instead of dealing with **discrete** set of states (rover being in one of six positions), we consider a large number of **continuous** value positions (rover could have a position anywhere between 0 kn to 6 km, such as 4.2 km)
- Additionally, a state can consist of many numbers (vector of numbers) that may track multiple factors such as the velocity of the rover in ***x*** and ***y*** directions, its coordinates ***x'*** and ***y'***, its angle ***theta***.
- For any specific state, any if these numbers can take on any value in a **valid range**



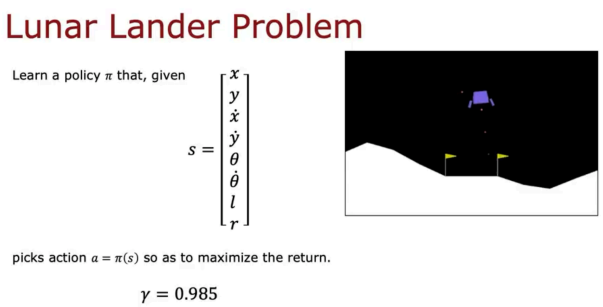
- By creating a complex reward function we can incentivize correct behaviors and actions while demotivating incorrect ones, doing this if far less laborious than specifying the exact correct action for every state



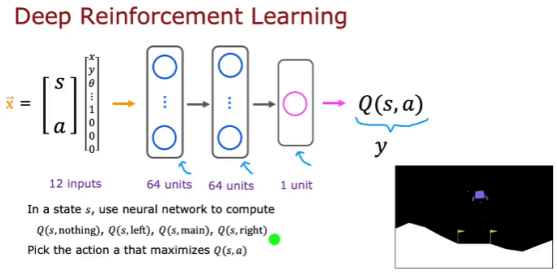


Note - In the lunar landing example, the **state spaces comprise** of:

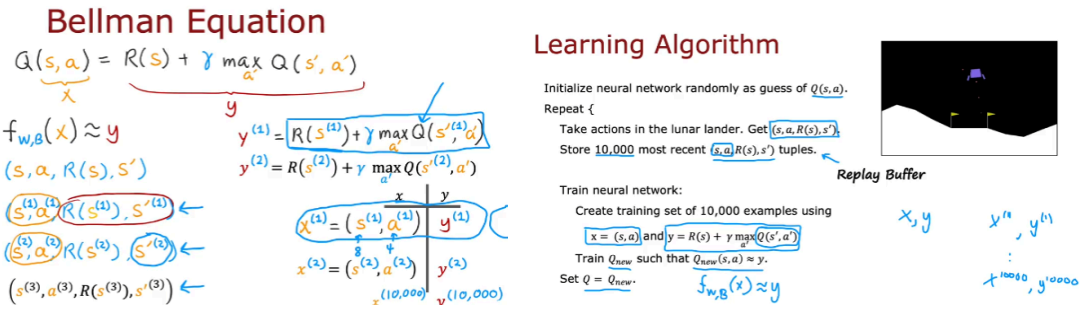
- ***Position x*** *(Left/Right)*
- ***Position y*** *(Up/Down)*
- ***x'*** *(Horizontal velocity)*
- ***y'*** *(Vertical velocity)*
- ***Theta*** *(Angle: Left/Right Tilt)*
- ***Theta'*** *(Angle Velocity)*
- ***L*** *(Left leg grounded)*
- ***R*** *(Right leg grounded)*



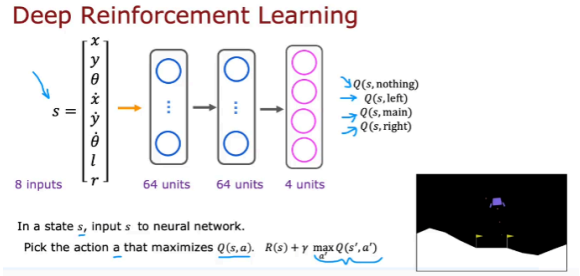
- By feeding a Deep Reinforcement Learning Model **a combination of vectorized state values (8 numbers as seen previously) and one-hot encoded actions (one-hot encoded 4 numbers corresponding to the action)**, we try to receive a **continuous state-action value, *Q(s,a)***.
- By computing all actions ***a*** for a constant state ***s*** (so the state remains the same, but we simply change the actions), we can determine which action yields the highest state-action value, helping us find the maximum *Q(s, a)* or *y*.



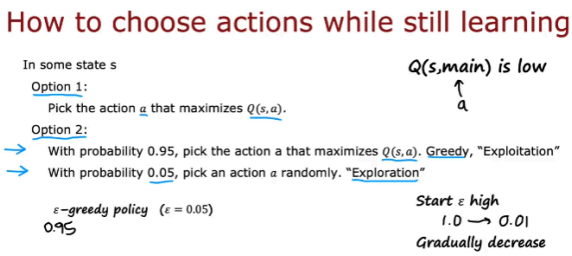
- We can use the **bellman equation** to generate *a* **training set** of inputs *x* to corresponding outputs *y* by either taking the **policy based actions** or **random actions** if no policy is set. We also take a **random initial guess** for the **state-action value of the next state *Q(s', a')*** as we don't know it yet***.***
- By taking different actions, we will get multiple sets of: state ***s***, action ***a***, reward for state ***R(s)*** and next state ***s'*.**
- Training the model will slowly give us a more **accurate *Q*** for us to replace during initialization instead of **randomly guessing it**



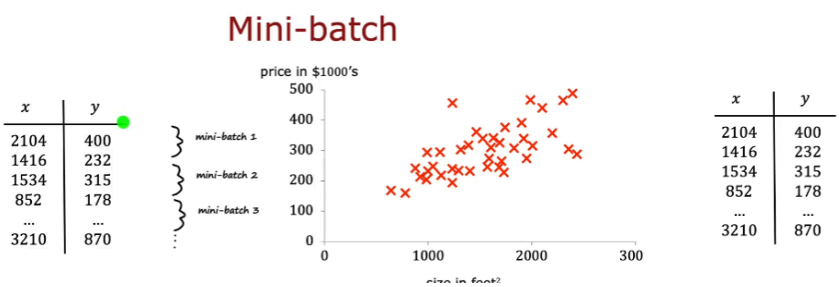
- Carrying out inference/computation for each action separately in the deep reinforcement learning model is **inefficient**. **The updated model can compute all actions for a singular state, simultaneously**

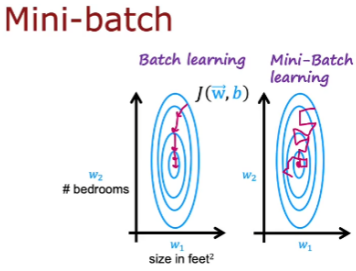


- Taking **initial actions** (does not need to be optimal) is one of the most **important** of the reinforcement algorithms, as without doing so, we cannot generate a training set to train the model on. **But what is the best way to choose these actions?**



- Always trying to maximize *Q(s,a)* is not a good idea as it can trap us in a local minimum. **Ex:** Assume that parameters were initialized such that *Q(s, Thruster)* is always low, the model will never fire the Thruster while trying to maximize *Q(s, a)* because it thinks its a bad decision, as such it will never discover that firing the Thrusters could possibly be a good action that would lead it to a better *Q(s,a).*
- Thus, **Option 2 is better,** known as **e-greedy policy**. It is good to start with a epsilon value thats initially hing and gradually decrease it until the ideal performance is found
- We can use **mini-batches** to speed-up a reinforcement learning algorithm, where we set samples/batches from an entire set of training examples (For a training set of size 100,000,000, we may create batches for every 1000 examples).
- For every **one iteration** we choose one unique/different batch that has not been previously used





- Mini-batch gradient descent is useful when dataset is exceptionally large, allowing for faster training and computation, Batch gradient descent is useful when we have smaller datasets and it converges to the minima consistently
- Going back to the lunar landing, we may use 1000 examples per training iteration for our training set out of the 10,000
- Additionally updating Q to the newly computed Q is not always ideal as it may overwrite it with a worse one
- To mitigate use a **soft update** where we may set hyperparameters which determine how aggressively ***Q*** will move towards ***Q_new***. This soft update helps to converge more reliably