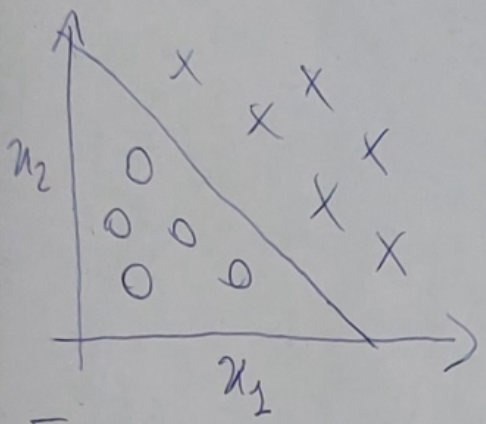
COURSES

Unsupervised Learning:

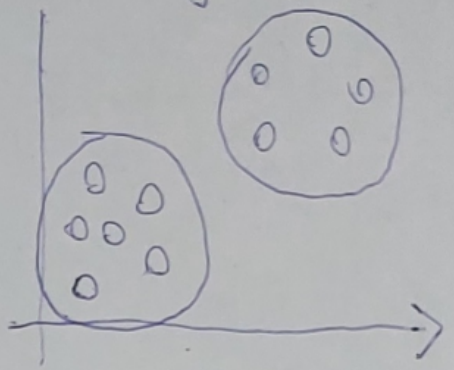
What is clustering: looks at a number of data points and automatically finds data points that are related or similar to each other.

Binary classification:



Training set: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}) \dots (x^{(m)}, y^{(m)})\}$

Clustering



Training set: $\{x^{(1)}, x^{(2)}, \dots x^{(m)}\}$

Since there is no y label, we can't tell the algorithm what the "right answer" is.

Instead, we are going to ask the algorithm to find something interesting (structure) about the data.

Applications of Clustering:

- Grouping similar news
- Market Segmentation
- DNA Analysis
- Astronomical data analysis

K-means

- Takes a random guess of the location of the centers of the clusters (cluster centroids) (assume there are 2)
- Go through all examples and checks if the point is closer to centroid 1 or 2. Assign each point to its closer centroid.
- Calculates the mean of the points assigned with the centroid and moves the centroid there.
- Go back to step 2, converged if there are no more changes

* If you end up with a cluster centroid with 0 points (you can't average that (/0)), just set $k = k-1$ (eliminate the cluster) or run it again and hope for a good result

Notation:

$c^{(i)}$: index of cluster $(1 \dots k)$ to which example $x^{(i)}$ is currently assigned.

μ_k : cluster centroid k

$\mu_{c^{(i)}}$: cluster centroid of cluster to which example $x^{(i)}$ has been assigned (when $k = c^{(i)}$) (example: $c^{(10)}$ corresponds to blue, what is the center of the cluster $x^{(10)}$ belongs to?)

Cost function:

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

It minimizes the distance between the points and the mean of their clusters (the one they belong to) by changing the cluster one point corresponds to and/or changing the mean of the clusters.

Also called: Distortion function

Repeat {
  # Assign points to cluster centroids
  for $i = 1$ to m: # keeping $\mu_1 \dots \mu_k$ fixed and varying $c^{(1)} \dots c^{(m)}$
    $c^{(i)} :=$ index of cluster centroid closest to $x^{(i)}$
  # move cluster centroids (keep $c^{(1)} \dots c^{(m)}$ fixed and vary $\mu_1 \dots \mu_k$
  for $k = 1$ to K:
    $\mu_k :=$ average of points in cluster k
}

Initializing K-means:

- Choose $K < m$.
- Randomly pick K training examples
- Set $\mu_1, \mu_2, \dots \mu_K$ equal to these K examples

Now the initial centroids won't be some random points

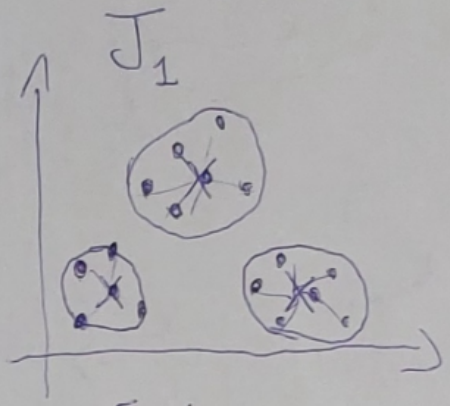


Fig. 1

A problem arises: Suppose 2 initial clusters are right next to each other. This will lead to different end results.

Fig 2

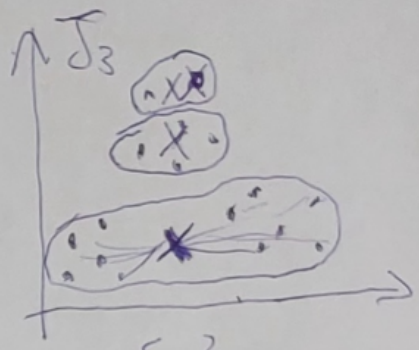


Fig. 3

We might get any of these 3 results if we run K-means once. Fig. 1 is a global minimum while Fig. 2 & 3 are local minima of the cost function J. What we do is run K-means multiple times and select the set of clusters with the lowest value for J.

Algorithm:

For $i = 1$ to 100 { ← number from 50 – 1000

Randomly initialize K-means ← k random examples

Run K-means. Get $c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K$

Compute cost function (distortion)

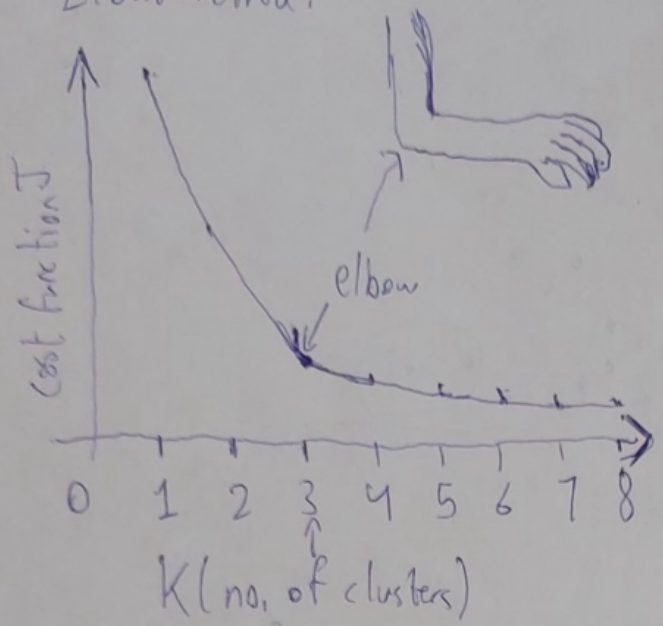
$J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K)$

}

Pick set of clusters with lowest J

Choosing the value of K:

Elbow method:



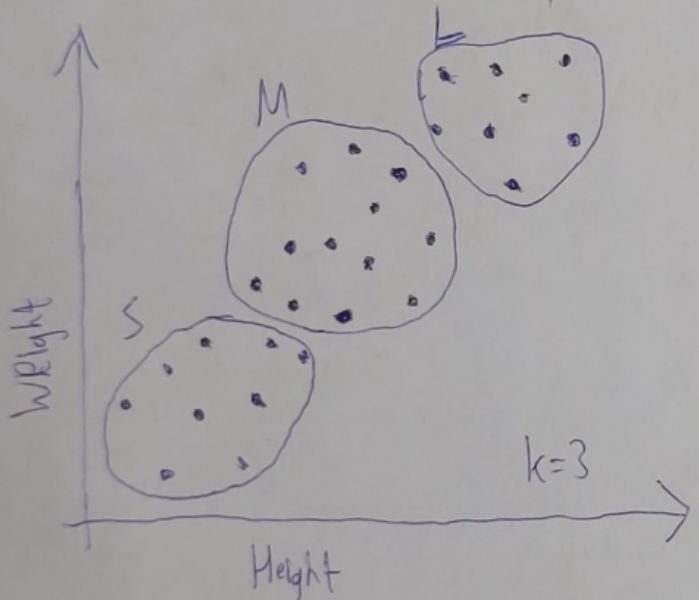
Andrew Ng does not use this method because he thinks that the right number of clusters is truly ambiguous, and you find that a lot of functions decrease more smoothly than the one to the left making it harder to detect an "elbow"

* Don't try to use the K that minimizes J. It is almost always choose the largest possible value for K.

Recommended approach:

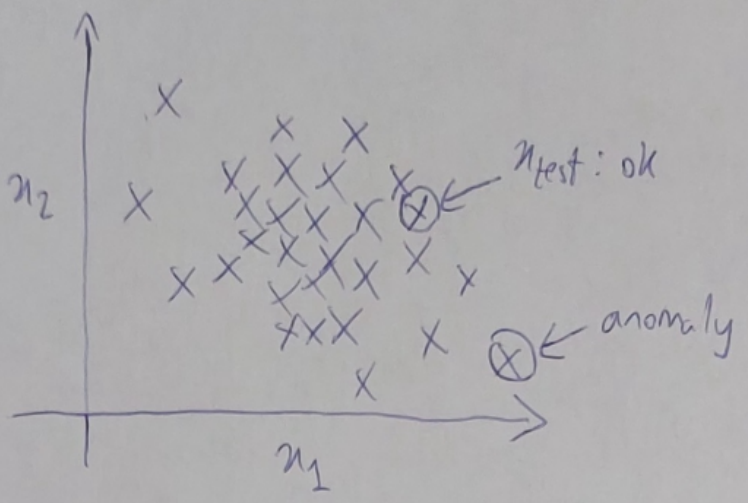
Often you want to get clusters for some later (downstream) purpose.

Evaluate K-means based on how well it performs on that later purpose.



Both groupings are fine, but whether you want to use 3 clusters or 5 clusters can now be decided based on what makes sense for your t-shirt business. There is a tradeoff between how well the t-shirts fit, and the cost for production, shipment. Decide based on this trade-off

Anomaly Detection: looks at an unlabeled dataset of normal events and thereby learns to detect an unusual or anomalous event.



Density Estimation:

Model $p(x)$: probability of X being seen in the dataset.

If $p(x_{test}) < \varepsilon$: anomaly

Otherwise: High/mid/ok probability

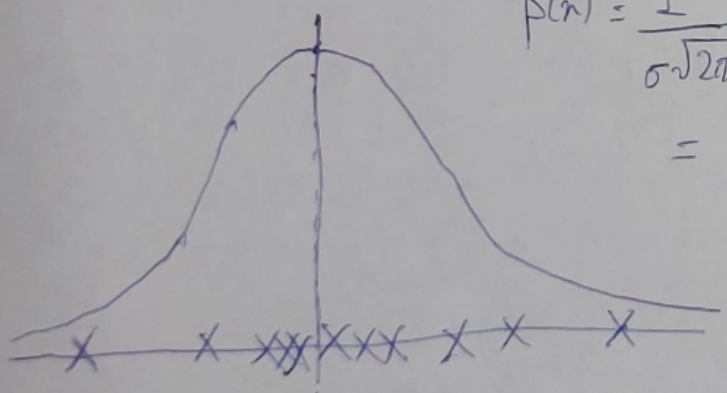
Real life Applications:

- Fraud detection (find fake accounts, identifying financial fraud)
- Manufacturing (flaws ...)
- Monitoring computers in data centers (memory use, no. of disk accesses/sec, CPU load ...)

We need to use the Gaussian (Normal) Distribution:

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

$$= \frac{1}{\sqrt{2\pi\sigma^2 e^{z^2}}} \quad \text{where } z = z\text{-score} = \frac{x-\mu}{\sigma}$$



(if x is a number (1 feature))

max. likelihood estimates for μ, σ^2 (can also use $\frac{1}{m-1}$)

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} (x^{(i)} - \mu)^2$$

Self-note: Think of the first graph as a 3 dimensional bell curve viewed from the third axis.

Training set: $\{\vec{x}^{(1)}, \vec{x}^{(2)}, \dots, \vec{x}^{(m)}\}$ $\quad \vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$

Each example $\vec{x}^{(i)}$ has n features

$$p(\vec{x}) = p(x_1; \mu_1, \sigma_1^2) \cdot p(x_2; \mu_2, \sigma_2^2) \cdot p(x_3; \mu_3, \sigma_3^2) \cdot \dots \cdot p(x_n; \mu_n, \sigma_n^2)$$

$$= \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2)$$

↑ ↑ parameters

(for independent probability but works well for dependent too)

Algorithm:

1. Choose n features x_i that you think might be indicative of anomalous examples.
2. Fit parameters $\mu_1, \dots, \mu_n, \sigma_1^2, \dots, \sigma_n^2$
3. Given new example x, compute $p(x)$.

   Anomaly if $p(x) < \varepsilon$

Even if 1 of the features of the new example was abnormally large or small, then $p(x)$ would be small.

If you can have a way to evaluate a system even as it is being developped, you'll be able to make decisions and change the system and improve it much more quickly

Real-number evaluation: Changing an algorithm and computing a number that tells you if the algorithm got better or worse.

Assume we have some labeled data, of anomalous ($y=1$) and non-anomalous ($y=0$) examples.

Training set: $y=0$ for all training examples $\{x^{(1)}, x^{(2)}, x^{(3)}, \dots x^{(m)}\}$

Dev set: $\{(x_{cv}^{(1)}, y_{cv}^{(1)}), (x_{cv}^{(2)}, y_{cv}^{(2)}), \dots, (x_{cv}^{(m_{cv})}, y_{cv}^{(m_{cv})})\}$

Test set: $\{(x_{test}^{(1)}, y_{test}^{(1)}), (x_{test}^{(2)}, y_{test}^{(2)}), \dots, (x_{test}^{(m_{test})}, y_{test}^{(m_{test})})\}$

} Include a few anomalous examples (mostly normal ($y=0$))

Example: Aircraft Engines monitoring

10000 good engines

20 flawed engines $y=1$ (apply algorithm with 2-50 known anomalies)

Training set: 6000 good engines

CV: 2000 good engines ($y=0$) 10 anomalous ($y=1$)

Test: 2000 good engines ($y=0$) 10 anomalous ($y=1$)

Use CV set to see how many of the anomalous engines it correctly flags, then tune ε, x_j

Then use it on the test set to see how many it detects and mistakes.

This algorithm is still primarily an unsupervised learning algorithm because the training has no labels (or are all assumed to be $y=0$)

Algorithm evaluation:

- Fit $p(x)$ on training set
- On CV/test examples: predict $y = \begin{cases} 1 & \text{if } p(x) < \varepsilon \\ 0 & \text{if } p(x) \geq \varepsilon \end{cases}$

Evaluation metrics: - TP, FP, FN, TN

- Precision, Recall
- F_1-score

When to use Anomaly Detection vs Supervised Learning

- Very small no. of +ve examples. (0-20) is common. Large no. of -ve examples.
- Many different "types" of anomalies. Hard for any type of algorithm to learn from the +ve examples what the anomalies look like; future anomalies may look nothing like any of the anomalous examples seen so far. eg: financial fraud

- Large no. of +ve and -ve examples
- Enough +ve examples for the algorithm to get a sense of what +ve examples are like; future +ve examples are likely to be similar to ones in training set. e.g.: spam

Choosing what feature to use:

Non-gaussian features: make them gaussian.

Use plt.hist(x) to check the distribution.

Transform the data: $x_1 \rightarrow \log x_1$

$x_2 \rightarrow \log(x_2 + c)$

$x_3 \rightarrow \sqrt{x_3}$.

$x_4 \rightarrow x_4^{1/3}$

Sometimes, after using the CV set, you might need to add new features that are combinations of other features (ratios, products, ...) so that the model detects the example labeled anomalous in the CV set.

Recommender System:

Predicting movie ratings

Movie	Alice (1)	Bob (2)	Carol (3)	Dave (4)
Movie 1	5	5	0	0
Movie 2	5	?	?	0
Movie 3	?	4	0	?
Movie 4	0	0	5	4
Movie 5	0	0	5	?

$n_u = 4$ $\quad r(1,1) = 1$

$n_m = 5$ $\quad r(3,1) = 0$ $\quad y^{(3,2)} = 4$

n = number of features for the movies

n_u = no. of users

n_m = no. of movies

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

$y^{(i,j)}$ = rating given by user j to movie i (defined only if $r(i,j) = 1$)

$m^{(j)}$ = no. of movies rated by user j

Assume we have n movie features (x_1 and x_2: romance and action).

$x^{(1)} = \begin{bmatrix} 0.9 \\ 0 \end{bmatrix}$

$x^{(3)} = \begin{bmatrix} 0.99 \\ 0 \end{bmatrix}$

For user 1: predict rating for movie i as: $w^{(1)} \cdot x^{(i)} + b^{(1)}$

user 1 → $w^{(1)} = \begin{bmatrix} 5 \\ 0 \end{bmatrix}$; $b^{(1)} = 0$; movie 3 → $x^{(3)} = \begin{bmatrix} 0.99 \\ 0 \end{bmatrix}$

$w^{(1)} \cdot x^{(3)} + b^{(1)} = 4.95$

↳ makes sense since she rated romance movies 5/5 and action movies 0/5

More generally, for user j's rating for movie i:

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

We want to find $w^{(j)}$ and $b^{(j)}$

Cost function:

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

Turns out it is better not to divide by $m^{(j)}$. It is a constant. ||||

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

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

We had features x_1 and x_2 to ~~help us~~ give us details about the movies. We basically used linear regression to learn to predict movie ratings. Now assume we don't know the movie features x_1 and x_2.

Movie	Alice(1)	Bob(2)	Carol(3)	Dave(4)	(romance) x_1	(action) x_2
Love at last	5	5	0	0	?	?
Romance forever	5	?	?	0	?	?
Cute puppies of love	?	4	0	?	?	?
Nonstop car chases	0	0	5	4	?	?
Swords vs. karate	0	0	5	?	?	?

Assume we know:

$w^{(1)} = \begin{bmatrix} 5 \\ 0 \end{bmatrix}$, $w^{(2)} = \begin{bmatrix} 5 \\ 0 \end{bmatrix}$, $w^{(3)} = \begin{bmatrix} 0 \\ 5 \end{bmatrix}$, $w^{(4)} = \begin{bmatrix} 0 \\ 5 \end{bmatrix}$

$b^{(1)} = 0$, $b^{(2)} = 0$, $b^{(3)} = 0$, $b^{(4)} = 0$

using $w^{(j)} \cdot x^{(i)} + b^{(j)} \approx 0$

$w^{(2)} \cdot x^{(1)} \approx 5$

$w^{(2)} \cdot x^{(1)} \approx 5$

$w^{(3)} \cdot x^{(1)} \approx 0$

$w^{(4)} \cdot x^{(1)} \approx 0$

$\longrightarrow x^{(1)} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$

In collaborative filtering, you have ratings from multiple users of the same item (movie). That's what makes it possible to try to guess the possible values for the features

So,

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

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

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

To learn $x^{(1)}, x^{(2)}, \dots, x^{(n_m)}$;

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

Putting together the cost function to learn $w^{(1)}, b^{(1)}, \dots w^{(n_u)}, b^{(n_u)}$ and the one to learn $x^{(1)}, \dots, x^{(n_m)}$:

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

To minimize this new cost function, you need to add 1 line to gradient descent:

repeat {

$$\vdots$$

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

}

Many of the important applications of recommender systems or collaborative filtering algorithms involve binary labels (like/dislike, upvote/downvote).

1: engaged after being shown item

0: did not engage after being shown item

?: item not yet shown to the user.

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

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

Cost function for binary application:

Previous cost function:

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

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

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

Binary cross entropy Loss function

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

If you have a new user who hasn't rated any movie, the algorithm would set $w=\begin{bmatrix}0\\0\end{bmatrix}$ (if we set $b=0$), predicting the user would rate all movies 0.

Ratings:

$$\begin{bmatrix} 5 & 5 & 0 & 0 & ? \\ 5 & ? & ? & 0 & ? \\ ? & 4 & 0 & ? & ? \\ 0 & 0 & 5 & 4 & ? \\ 0 & 0 & 5 & 0 & ? \end{bmatrix}$$

$\mu = $ (np.mean(..., axis=1))

$$\begin{bmatrix} 2.5 \\ 2.5 \\ 2 \\ 2.25 \\ 1.25 \end{bmatrix}$$

mean normalization:

$$\begin{bmatrix} 2.5 & 2.5 & -2.5 & -2.5 & ? \\ 2.5 & ? & -2.5 & -2.5 & ? \\ ? & 2 & -2 & ? & ? \\ -2.25 & -2.25 & 2.75 & 1.75 & ? \\ -1.25 & -1.25 & 3.75 & -1.25 & ? \end{bmatrix}$$

For user j, on movie i, predict:

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

Since for new user (user 5): $w^{(5)} = \begin{bmatrix}0\\0\end{bmatrix}$ & $b^{(5)}=0$: prediction $= 0+0+0+\mu_i = \mu_i$, so the initial prediction would be the average rating of the movie

You would normalize the columns if there is a new movie. We just normalized the rows.

Implementation in Tensorflow:

```
w = tf.Variable(3.0)  # tf.variable are the parameters we want to optimize.
x = 1.0
y = 1.0  # target value
alpha = 0.01
iterations = 30
for iter in range(iterations):  # Auto Diff, also called Auto Grad (although it's wrong)
    with tf.GradientTape() as tape:
        fwb = w * x
        costJ = (fwb - y) ** 2
    [dJdw] = tape.gradient(costJ, [w])
    w.assign_add(-alpha * dJdw)  # tf.variables require special functions to modify
```

Finding related Items:

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

To find other items related to it,

find item k with $x^{(k)}$ similar to $x^{(i)}$

i.e. with smallest distance $\sum_{l=1}^{n} \left(x_l^{(k)} - x_l^{(i)}\right)^2$

- Limitations of Collaborative Filtering:

- Cold start problem. How to
  - Rank new items that few users have rated
  - Show something reasonable to new users who have rated few items
- Use side information about items or users:
  - Item: Genre, movie stars, studio, ---
  - User: Demographics (age, gender, location), expressed preferences, ---

Content Based filtering algorithm / 2nd type of recommender system:

- Collaborative filtering:
  Recommends items to you based on ratings of users who gave similar ratings to you
- Content based filtering:
  Recommends items to you based on features of user and item to find good match.

User features = $x_u^{(j)}$ for user j

Movie features = $x_m^{(i)}$ for movie i

Predicting the rating of user j on movie i as:

$w^{(j)} \cdot x^{(i)} + b^{(j)}$ ← not needed

$\downarrow$ $\downarrow$

$V_u^{(j)} \cdot V_m^{(i)}$ where V stands for vector, u for user, and m for movie

$\uparrow$ $\uparrow$

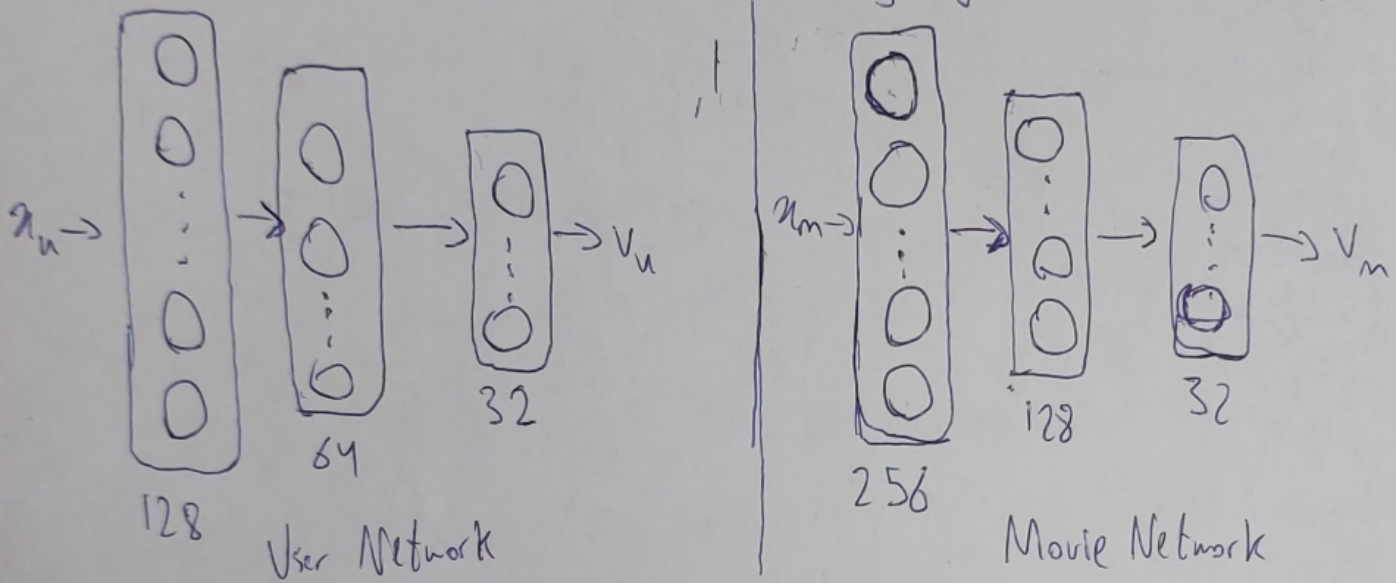
computed from $x_u^{(j)}$ computed from $x_m^{(i)}$

User's preferences $V_u = \begin{bmatrix} 4.9 \\ 0.1 \\ \vdots \\ 3.0 \end{bmatrix}$ likes action / likes romance

Movie features $V_m = \begin{bmatrix} 4.5 \\ 0.2 \\ \vdots \\ 3.5 \end{bmatrix}$ Action / Romance

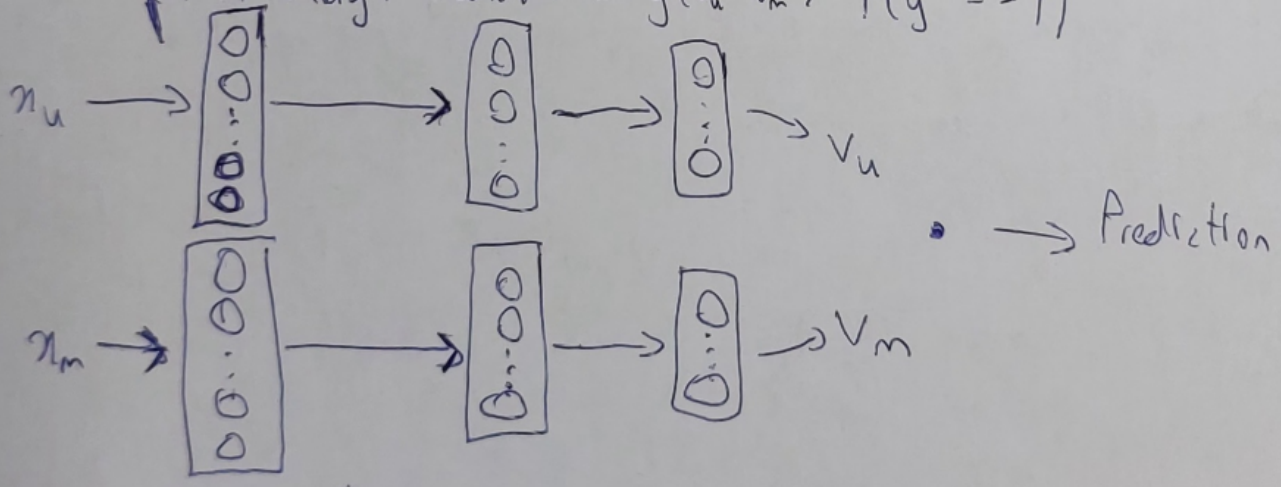
Notice that $V_u^{(j)}$ and $V_m^{(i)}$ have to be the same size because we're taking their dot product. $x_u^{(j)}$ and $x_m^{(i)}$ can be of any size, distinct or not.

A good way to develop a content based filtering algorithm is to use deep learning.



Prediction: $V_u^{(j)} \cdot V_m^{(i)}$

(or if binary: Prediction $= g(V_u^{(j)} \cdot V_m^{(i)}) = P(y^{(j,i)} = 1)$)



Merged into 1 neural network

Cost function $J = \sum_{(i,j):r(i,j)=1} \left(v_u^{(j)} \cdot v_m^{(i)} - y^{(i,j)} \right)^2 +$ NN regularization term

← can be precomputed

To find movies similar to movie i: $\| v_m^{(k)} - v_m^{(i)} \|^2$ small

One limitation is that the algorithm is computationally very expensive to run if we have a large catalog of different items.

Making it more efficient (Assume you already precomputed $v_m^{(i)}$ for all movies i):

Retrieval (broad step):

- Generate a list of plausible item candidates

  eg. 1) for each of the last 10 movies watched by the user, find 10 most similar movies

  2) for most viewed 3 genres, find the top 10 movies

  3) top 20 movies in the country

- Combine retrieved items into a list, removing duplicates and items already watched/purchased

Ranking:

- Take retrieved list and rank using learned model

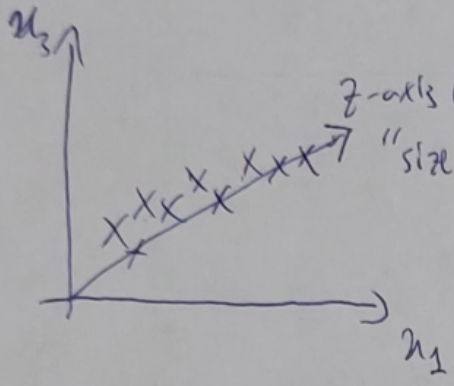
  Since we already have all v_m's, just do inference on the upper part of the model and find v_u.

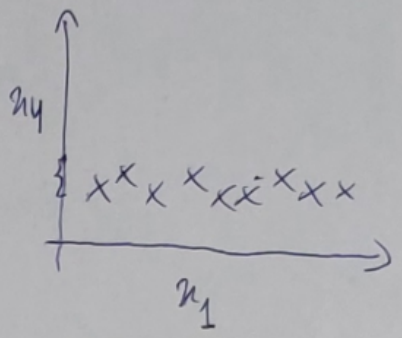
- Display ranked items to user.

Principal Component Analysis (PCA): reduces number of features to 2 or 3.

Say for a car: x_1 = length, x_2 = width, x_3 = height, x_4 = wheel diameter.

It turns out that due to regulations, the width doesn't vary that much. Also, wheel diameter doesn't. So we can remove both x_2 and x_4.





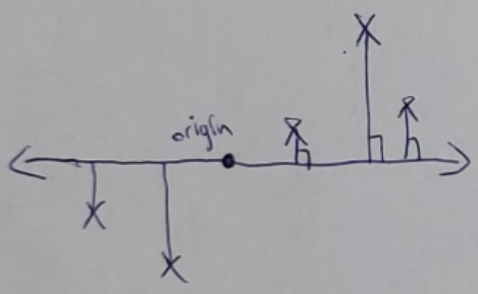
PCA: Find new axis and coordinates; use fewer numbers

It is an unsupervised learning algorithm

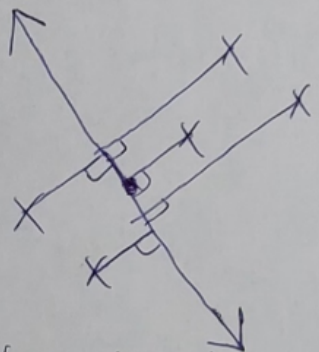
Algorithm:

Preprocess features, Normalized to have 0 mean, feature scaling

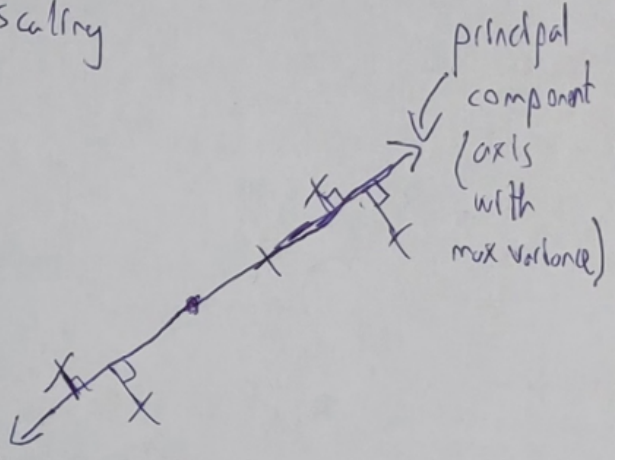
Choose an axis



Variance is large (capturing info of original data)



Variance is small squished together Bad axis



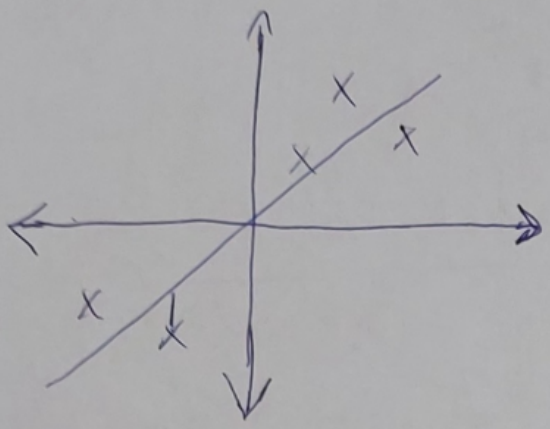
Variance is large For apart capturing more info

length 1 vector (unit): Assume $\begin{bmatrix} 0.71 \\ 0.71 \end{bmatrix}$, and assume coordinates$_{old}$ $= \begin{bmatrix} 2 \\ 3 \end{bmatrix}$

coordinates$_{new}$ $= \begin{bmatrix} 2 \\ 3 \end{bmatrix} \cdot \begin{bmatrix} 0.71 \\ 0.71 \end{bmatrix} = 2 \times 0.71 + 3 \times 0.71 = 3.55$

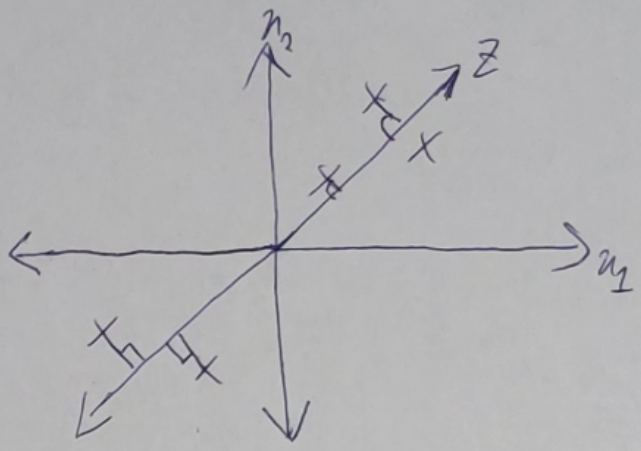
The 2nd principal component is perpendicular to the 1st, and the 3rd principal component is perpendicular to both the 1st and the 2nd

Difference between Linear regression and PCA:



we have label y.

We minimize the distance along y axis (and therefore the cost function)



Finding an axis to retain variance (info)

No y label

x_1 and x_2 are treated equally

given $z = 3.55$, find original (x_1, x_2) (approximately):

"Reconstruction"

$$3.55 \times \begin{bmatrix} 0.71 \\ 0.71 \end{bmatrix} = \begin{bmatrix} 2.52 \\ 2.52 \end{bmatrix} \text{ which isn't far from } \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

Implementation:

```
pca_2 = PCA(n_components= 2) # or whatever
pca_2.fit(X) # includes mean normalization
pca_2.explained_variance_ratio # how much of the data is explained by variance
X_trans_2 = pca.transform(X)
X_reduced_2 = pca.inverse_transform(X_trans_2)
```

Other applications of PCA: (Used 15-20 years ago, not nowadays).

- Data compression (to reduce storage or transmission costs)
- Speeding up training of a supervised learning model.

$n = 1000 \rightarrow 100$

Last week:

Reinforcement learning: You don't tell the algorithm what is the right output y for every single input, all you have to do is to specify a reward function that tells it when it is doing poorly and when it is doing well. It is the algorithm's job to figure out how to choose good actions

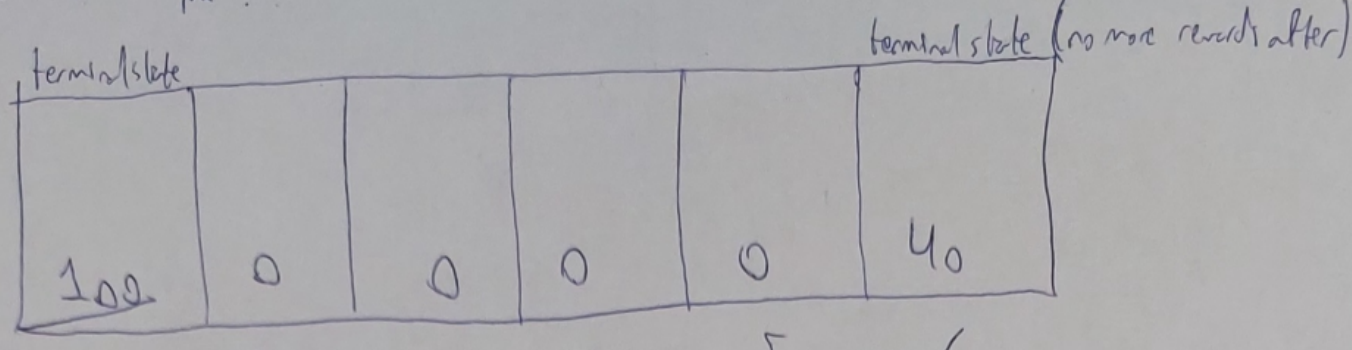
state s ——————→ action a

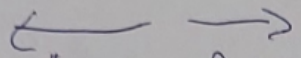
Supervised learning isn't very effective as you might not have exactly the right actions for every state

Applications:

- Controlling Robots
- Factory Optimization
- Financial (stock) exchange
- Playing games (including video games)

Mars Rover Example





associated

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

$(4, \leftarrow, 0, 3)$

Return: Get $5 now or walk half an hour across town for $10?

$$\text{Return} = \gamma^0 R_1 + \gamma R_2 + \gamma^2 R_3 + \dots \text{ (until terminal state)}$$

where γ = discount factor (making the algorithm a little bit important).

γ gives the first reward full credit then gradually gives less credit for other rewards.

since $\gamma \in (0, 1)$

In financial applications, the discount factor has a very natural interpretation as the interest rate or the time value of money (a dollar today > dollar tomorrow because you can take it and invest it.)

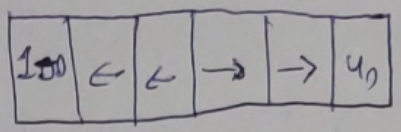
Q function: $Q(s, a)$ = Return if you
- start in state s
- take action a (once).
- then behave optimally after that

Also called: Q^*, Optimal Q function, state-action value function

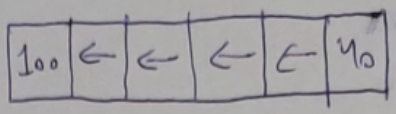
Policy (controller):

state $\xrightarrow{\text{policy}}$ action

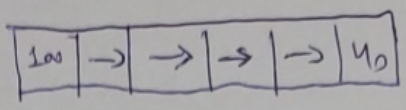
$s \quad \pi \quad a$



Go to nearer

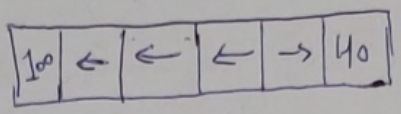


Go to larger



Go to smaller

$\pi(2) = \leftarrow$
$\pi(3) = \leftarrow$
$\pi(4) = \leftarrow$
$\pi(5) = \rightarrow$



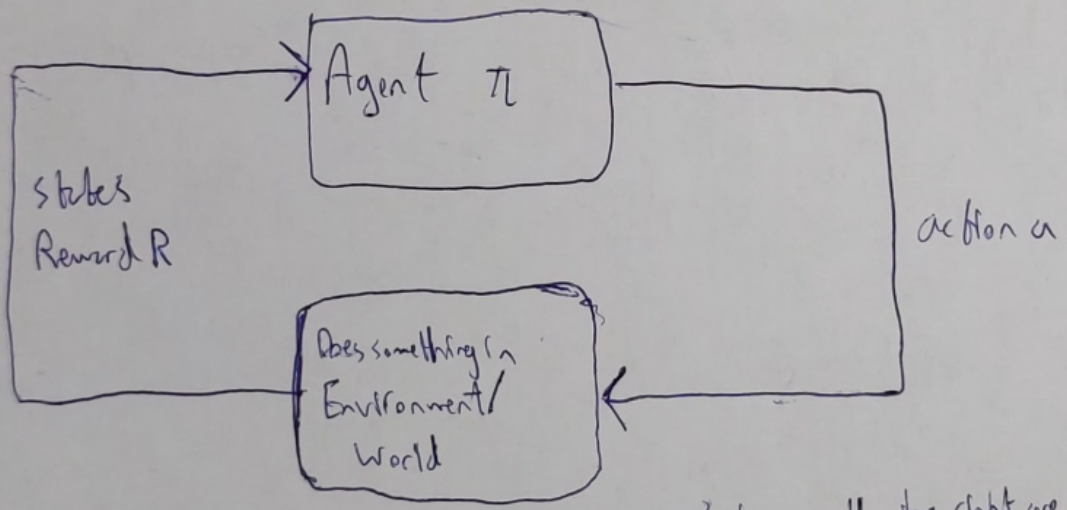
Go to the larger unless you're 1 step away from the smaller in which you go to it

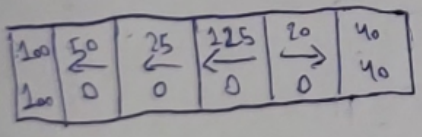
A policy is a function $\pi(s) =$ a mapping from states to actions, that tells you what action a to take in a given state s.

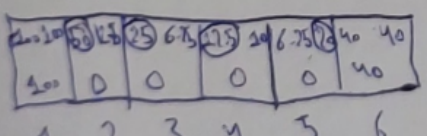
Goal of Reinforcement Learning: Find a policy π that maximizes the return.

The formalism of Reinforcement Learning is called Markov Decision Process (MDP):

↖ The future depends only on the current state, not the past







Note: number in state s's box on the top right are $Q(s, \rightarrow)$

number in state s's box on the top left are $Q(s, \leftarrow)$

The best possible return from state s is $\max_a Q(s,a)$

The best possible action in state s is the action a that gives $\max_a Q(s,a)$

Bellman Equation: Used to compute values of $Q(s,a)$

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

($R(s)$: Reward of state s)

If in terminal state: $Q(s,a) = R(s)$

$R(s)$ is the reward you get right away.

The other term is the return from behaving optimally starting from state s'

Random (stochastic) environment:

In practice many robots don't always manage to do exactly what you tell them, because of wind blowing, off course, wheel slipping, ...

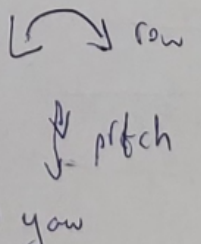
Imagine that, when you command a robot to move to the left, it does so 90% of the time, and goes right the other 10%.

$$\text{Expected Return} = \text{Average}\left(R_1 + \gamma R_2 + \gamma^2 R_3 + \dots\right) = E\left[R_1 + \gamma R_2 + \gamma^2 R_3 + \dots\right]$$

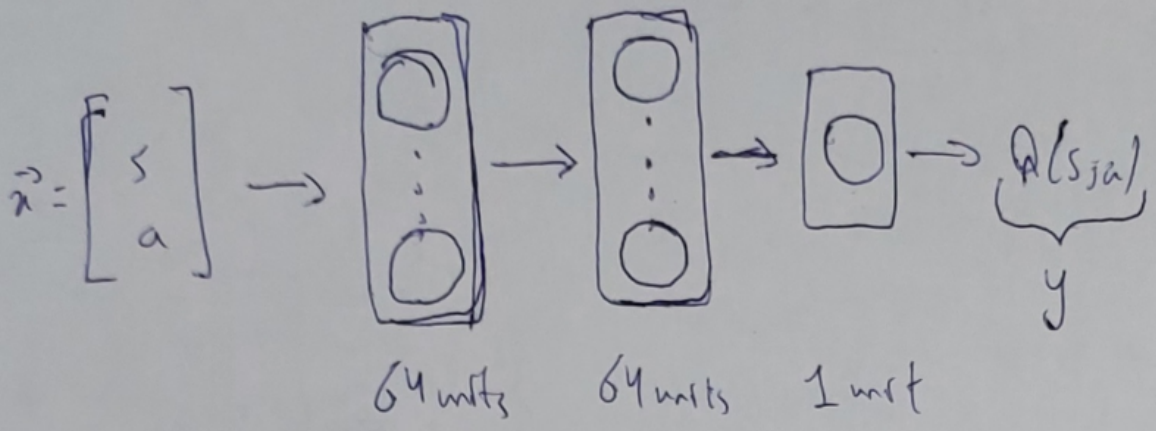
Bellman Equation: $Q(s,a) = R(s) + \gamma E\left[\max_{a'} Q(s',a')\right]$

Continuous state space: (Come on, it is clear)

$$S = \begin{bmatrix} x \\ y \\ \theta \\ \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix}$$



Turns out we need to use deep learning (Neural Networks) to learn the policy, so the process becomes "Deep Reinforcement Learning"



In a state s, use NN to compute:

$Q(s, \text{nothing})$, $Q(s, \text{left})$, $Q(s, \text{main})$, $Q(s, \text{right})$.

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

Algorithm: Initialize Neural Network randomly as a guess of $Q(s,a)$

Repeat{

Take actions in the lunar lander. Get $(s, a, R(s), s')$

Store 10000 most recent $(s, a, R(s), s')$ tuples ← Replay Buffer

Train Neural Network:

Create training set of 10,000 examples using

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

Train Q_{new} such that $Q_{new}(s,a) \approx y$.

Set $Q = Q_{new}$

}

Algorithm is called DQN algorithm, DQ Network, Deep Q-Network

For lunar lander:

$$\begin{bmatrix} s \\ a \end{bmatrix} = \begin{bmatrix} x \\ y \\ \theta \\ \vdots \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

actions: nothing
left
main
right
and we'll use one hot encoding

A more efficient implementation would have 4 units in the output layer. This is faster than inferencing 4 different times. Also, you can compute $Q(s,a)$ faster because you can directly plug in $Q(s',a')$

How to choose actions while still learning:

In some states.

Option 1:

Pick the action a that maximizes $Q(s,a)$ (even if it is a bad guess for Q)

Option 2 (Better option):

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

With probability 0.05, pick action a randomly (What if the NN was stuck and thought firing the main engine is bad).
"Exploration step"

"Exploration vs Exploitation tradeoff"

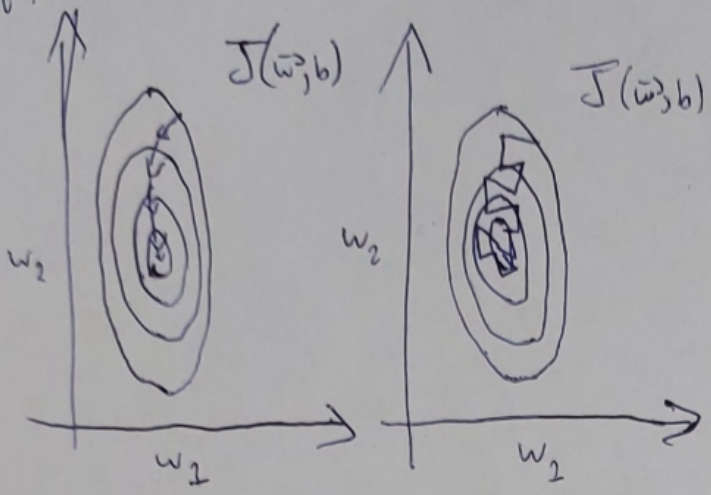
Option 2's approach is called an ε-greedy policy ($\varepsilon = 0.05$)

Another trip: Start ε high then gradually decrease it

Reinforcement algorithms are much more finicky to little choices of parameters.

Mini-batch: Every time you have a very large data set ($m = 100,000,000$)

Then take $m' = 1000$ and for every iteration in gradient descent, you have a different set:



x	y	
2104	400	} mini-batch 1
1416	232	
1534	315	} mini-batch 2
852	178	} mini-batch 3
...	...	
3210	870	

Batch learning

Mini-Batch learning
(much faster but less accurate)

Soft Update: In the learning algorithm,

Instead of Set $Q = Q_{new}$

Set $W = 0.01 W_{new} + 0.99 W$

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

We do this since we don't know whether Q_{new} is a better function than Q because Mini-batch introduced some "noise"

Limitations of Reinforcement Learning:

- Much easier to get to work in a simulation than a real robot
- Fewer applications than Supervised and Unsupervised learning.
- But... exciting research direction with potential for future applications.