WHEN TO USE DECISION TREES

- Decision Trees & Tree Ensembles
  - Works well on tabular (structured) data
  - Not recommended for unstructured data (images, audio, text)
  - Fast to train
  - Small decision trees may be human ~~interp~~ interpretable [you can look @ small trees to know how it works]

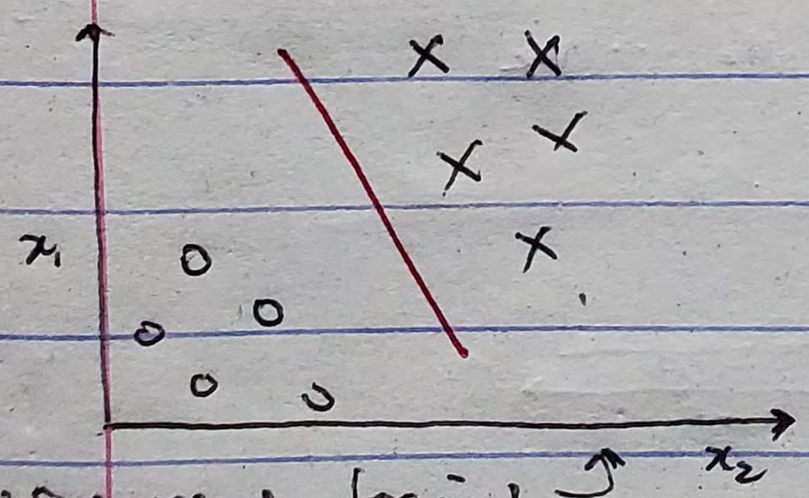
  <It's preferable to use XGBOOST>

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

- Check out CrossFeatch CV's documentation for hyperparameter tuning

UNSUPERVISED LEARNING, RECOMMENDERS, REINFORCEMENT LEARNING

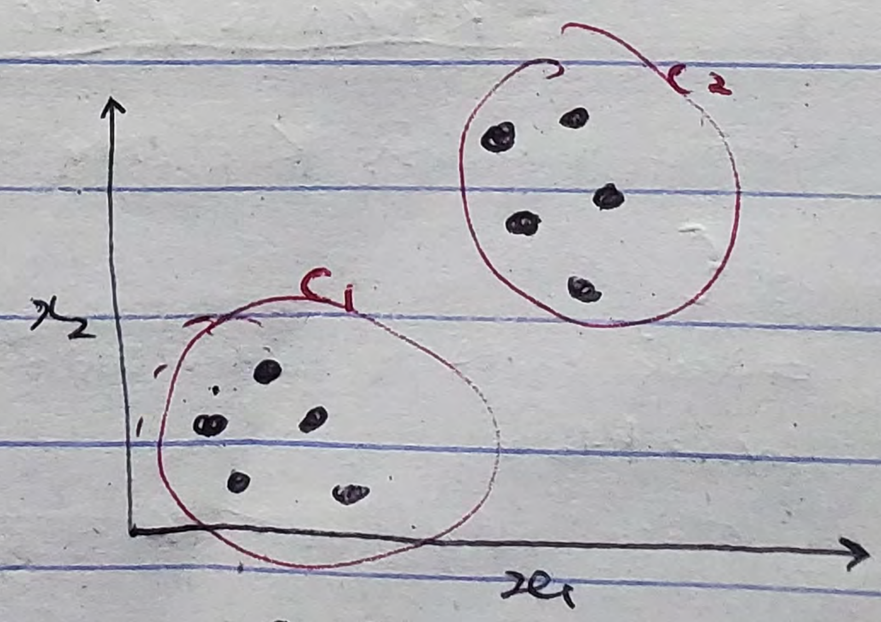
MK1 CLUSTERING



~~Sup~~ Supervised learning

contains input x & target output y

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



Training set : $\{x^{(1)}, x^{(2)}, x^{(3)} \dots x^{(m)}\}$

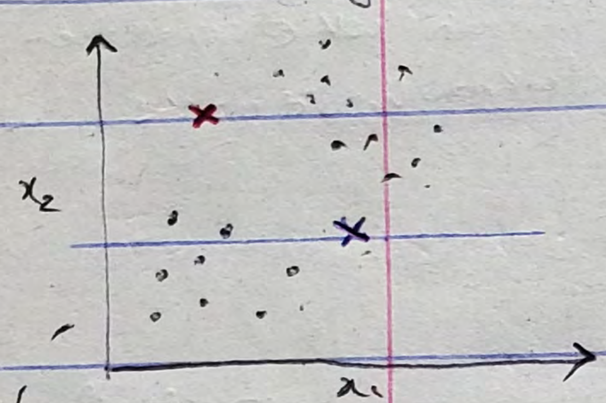
- No target label y, ∴ we can't ask the algorithm to tell us the right answer given x. Instead, we ask the algorithm to look for some thing interesting

∴ Clustering algorithm looks for ~~groups/clus~~ clusters i.e. groups of points in the data, that are similar to each other.

Applications of Clustering: (i) Grouping similar news (ii) Market segmentation

K-MEANS INTUITION

- We take an unlabeled dataset and the ~~k~~ thing that K-means does is to take a random guess of where the center of the $\boxed{\text{two}}$ clusters are. (2) → can be more if you like

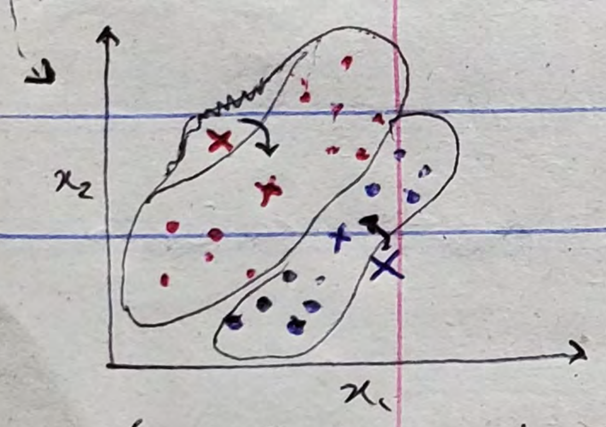


Note that K-means repeatedly does 2 things:

(i) - Assign points to cluster centroids

(ii) - Move cluster centroids

STEP 1 - The involves going through each of these points to find out which "cross" (× or ×) it is closer to. The center of the clusters = CLUSTER CENTROIDS



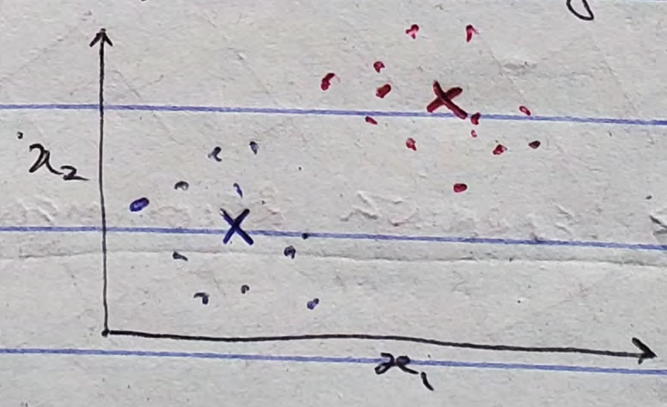
STEP 2: Kmeans looks at all of the red pts, takes an average of them, and moves the red cross to that average location. The same thing is done for the blue cross

After that, we keep on repeating step 1 & 2 till we get the true cluster centroids.

Eventually, we get

thus after getting the true centroids

↳ i.e. Kmeans has converged



K-MEANS ALGORITHM

(I) Randomly initialise K cluster centroids $\mu_1, \mu_2, \ldots, \mu_K$

→ our example, K=2 ∴ μ_1 = red cross, μ_2 = blue cross

(II) Repeat {

Assign points to cluster centroids

for i = 1 to m

$c^{(i)}$:= index (from 1 to K) of cluster centroid closest to $x^{(i)}$

↳ This finds which cluster, K, $x^{(i)}$ is closest to by minimizing the distance btw each < choosing the min distance >

$\min_k \| x^{(i)} - \mu_k \|^2$ → find the squared distance so as to avoid -ve values deceiving you.

Move cluster centroids
for $k = 1$ to K
$\mu_k :=$ average of points assigned to cluster k
}

Mean of x_1 values & mean of x_2 values will become value for μ_k

- If you should have a cluster having 0 pts assigned to it, the best thing to do is to eliminate that cluster $\Rightarrow$ $K = K - 1$

* You can also use K-means on a not-so-obviously-separated data.

OPTIMIZATION OBJECTIVE

$c^{(i)}$ = index of cluster (1, 2, ..., 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. → ranges from 1 to K

COST FUNCTION; ... aka called DISTORTION COST FUNCTION

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

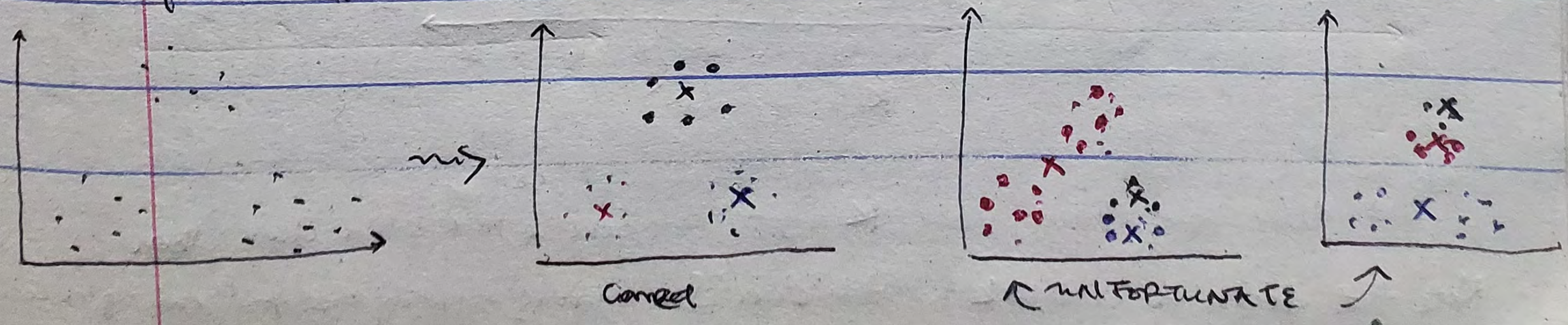
* Let's take a look @ the K-means algorithm; in relation to the cost function

Repeat {
# Assign points to cluster centroids * → This updates $c^{(1)}$ to $c^{(m)}$ to min J while holding μ_1 to μ_K fixed
# Move cluster centroids * → This leaves c_1 to c_m fixed but updates μ_1 to μ_K to min. distortion.
}

INITIALIZATION K-MEANS

* Therefore we'll look @ how we take the random guess for the initial position of the cluster centroids.

- The points you choose as your initialization matters alot because if chosen wrongly, you can end up with different clusters



- If you want to give K-means a better chance of finding the best local optima, one of the things you can do, is to run it multiple times and choosing the best local optima. This can be done by computing the cost function J of each initialization and picking which has the lowest.
- For the example plot previously given, the first initialization would be best because it will have the least cost J, seeing as the squared distances from each cluster centroid is the least when compared to the rest.

Tip that I forgot: To randomly initialize, pick any K training examples. That is, if you have 2 clusters, pick any 2 training examples out of the m training examples.

» RANDOM INITIALIZATION ALGORITHM

For i = 1 to 100 {

Randomly initialize K-means.

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

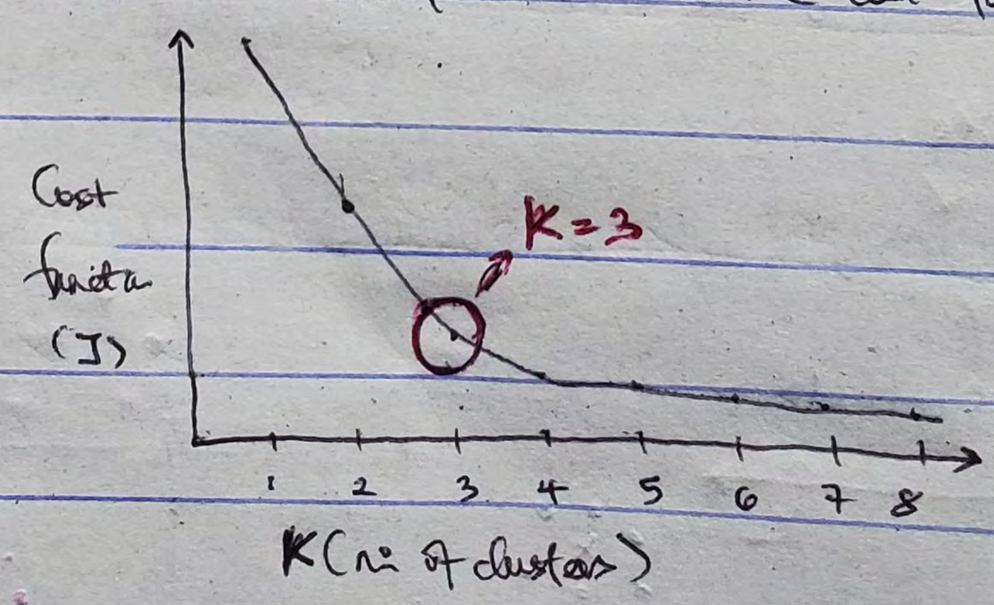
Compute cost function (distortion), $J(c^{(1)}, \dots, c^{(m)}, \mu_1, \dots, \mu_K)$

}

Pick set of clusters that gave lowest cost J

CHOOSING the NUMBER of CLUSTERS

One can use the elbow method. It is one where the cost function J for different values of K is plotted and the ideal K is the point where the cost function J stops rapidly decreasing



But there are times when K is just ambiguous so there isn't a clear elbow

Note that choosing the value of K that minimizes J is a wrong approach since you'll end up choosing

the largest value of K

So, How do we choose the value of K in practice?

[Often, you want to get clusters for some later <downstream> purpose. Evaluate K-means based on how well it performs on that later purpose. See how the K clusters you've chosen actually play out in real life.

WK 1 ANOMALY DETECTION

- Anomaly detection algorithms look @ an unlabeled ~~set~~ data set of "normal" events and learn to raise red flags if an "abnormal" event is encountered.

Example: Let's look @ an aircraft engine manufacturer who wants to identify ~~fa~~ likely faulty products

Aircraft engine features:

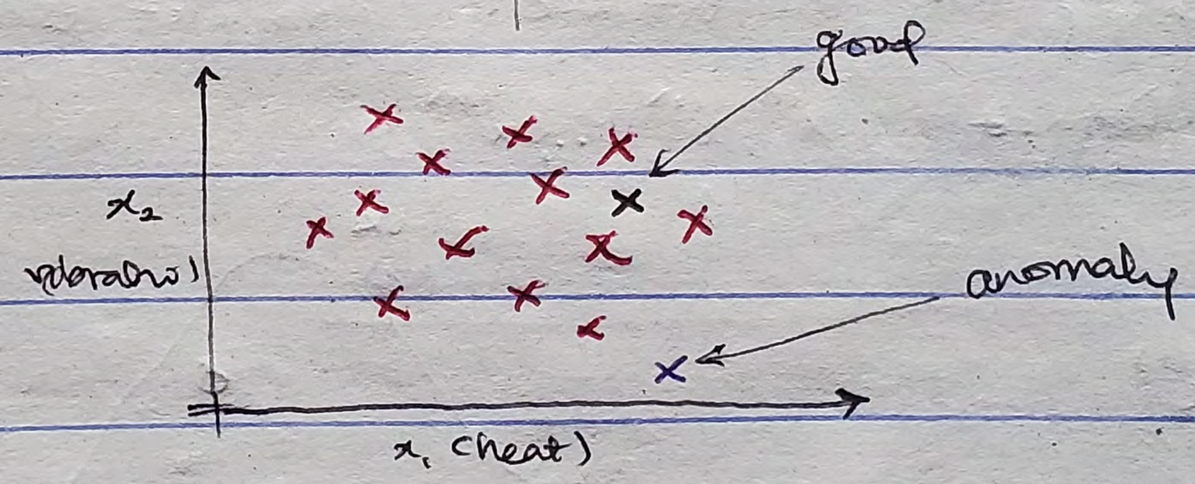
x_1 = heat generated
x_2 = vibration intensity
... etc

we'll use this 2 for now

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

data from okay m engines

New engine: x_{test}



- The most common way to carry out anomaly detection is through DENSITY ESTIMATION

[Density estimation involves

① Firstly building a model $p(x)$ which is the probability of x given the dataset $[\{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}]$. Here the model assigns a high/low probability to each value based on the dataset.

② Next we calculate the probability of x_{test} → $P(x_{test})$ and compare it to a small number, ε. ∴ if $p(x_{test}) < \varepsilon$, then x_{test} is likely an anomaly

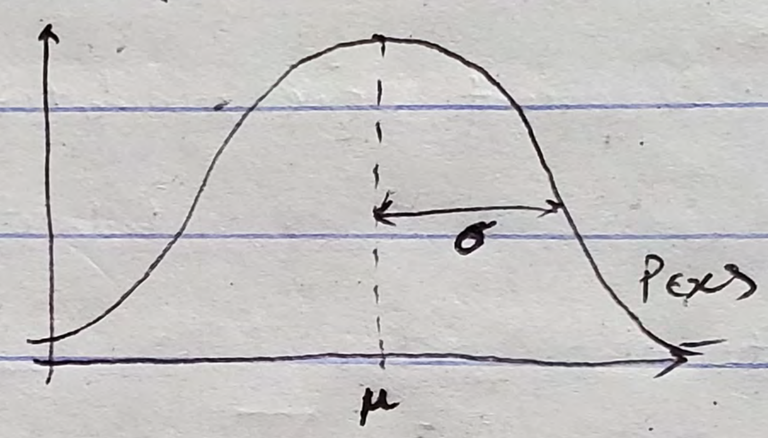
For example, when dealing with a fraud detection problem

- $x^{(i)}$ = features of user i's activities
  - How often do ~~the~~ ex user log in
  - How many webpages visited
  - # of transaction
  - Any posts?
  - What is the typing speed?
- Model $p(x)$ from data
- Identify unusual users by checking which have $p(x) < \varepsilon$

GAUSSIAN <NORMAL> DISTRIBUTION

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

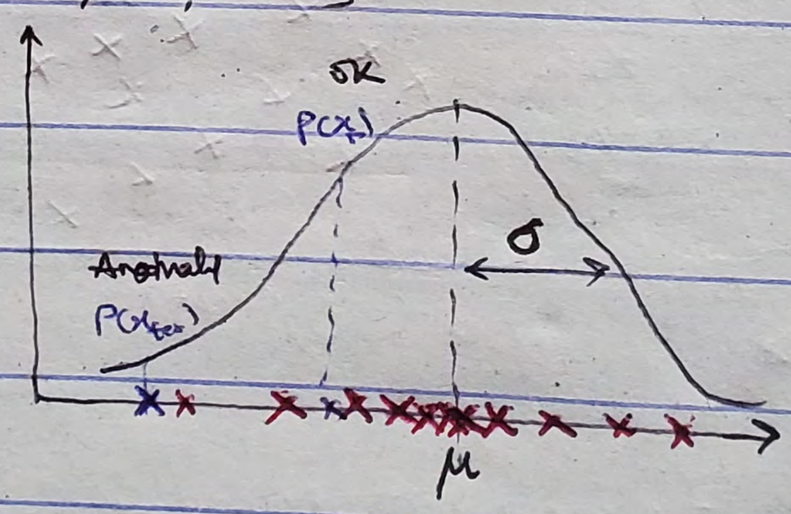
$$p(x) = \frac{1}{\sqrt{2\pi}\cdot\sigma} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$



When implementing this distribution in parameter estimation,

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

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



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

* The above method is only for when x is a single feature <a single number> but in the real world, x is a lot of ~~numbers~~ n features.

ANOMALY DETECTION ALGORITHM

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

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

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

∴ running density estimation to find probabilities,

NEXT PAGE

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

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

Π works just like Σ. while Σ is summation, Π means multiplication

$\therefore$ Thus of the algorithm;

1) Choose n features x_i that you think might be indicative of anomalous examples

2) Fit parameters $\mu_1, \dots, \mu_n, \sigma_1^2, \dots, \sigma_n^2$

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

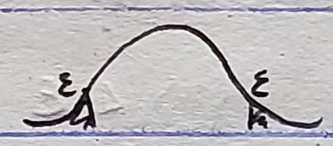
vectorized formula

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

3) Given new example x, Compute $p(x)$:

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

4) Anomaly if $p(x) < \varepsilon$ $\quad x \rightarrow$



~~This might be~~

* This might not be correct

DEVELOPING & EVALUATING AN ANOMALY DETECTION SYSTEM

When developing a learning algorithm (choosing 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: $x^{(1)}, x^{(2)}, \dots, x^{(m)}$ (assume normal examples (not anomalous)) $\leftarrow y=0$ for all examples

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

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

} Include a few anomalous examples $y=1$; mostly normal examples $y=0$

Let's look @ an ~~Aircraft~~ engines monitoring example:

We have 10000 good (normal) engines $y=0$

20 flawed engines (anomalous) $y=1$

A) We can split the data as follows

Training set : 6000 good engines ~~was~~ This will be used to train the algorithm.

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

→ use this to tune ε & tune the features; x_j

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

B) An alternative for when we have little (y=1) anomalous data; is to have no test set

- Training set : 6000 good engines
- CV : 4000 good engines (y=0) ; 20 anomalous (y=1)

downside is that we have no test set to see how our algorithm would work on new data

∴ Algorithm Evaluation:

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

ii) On a cross validation/test example x, predict

$$y = \begin{cases} 1 & \text{if } p(x) < \varepsilon \text{ (anomaly)} \quad \text{e.g } 10 \\ 0 & \text{if } p(x) \geq \varepsilon \text{ (normal)} \quad \text{e.g } 2000 \end{cases}$$

Since data is highly skewed, consider ~~pos~~ other following possible evaluation metrics ~~metrics~~

- True positive, False positive, False negative, True negative
- Precision/Recall
- F_1- score.

ANOMALY DETECTION vs SUPERVISED LEARNING

- When should you use one over the other?

A.D	S.L
i) Very small number of positive examples (y=1). (0-20 is common). Large number of negative (y=0) examples.	i) Large number of positive and negative examples.

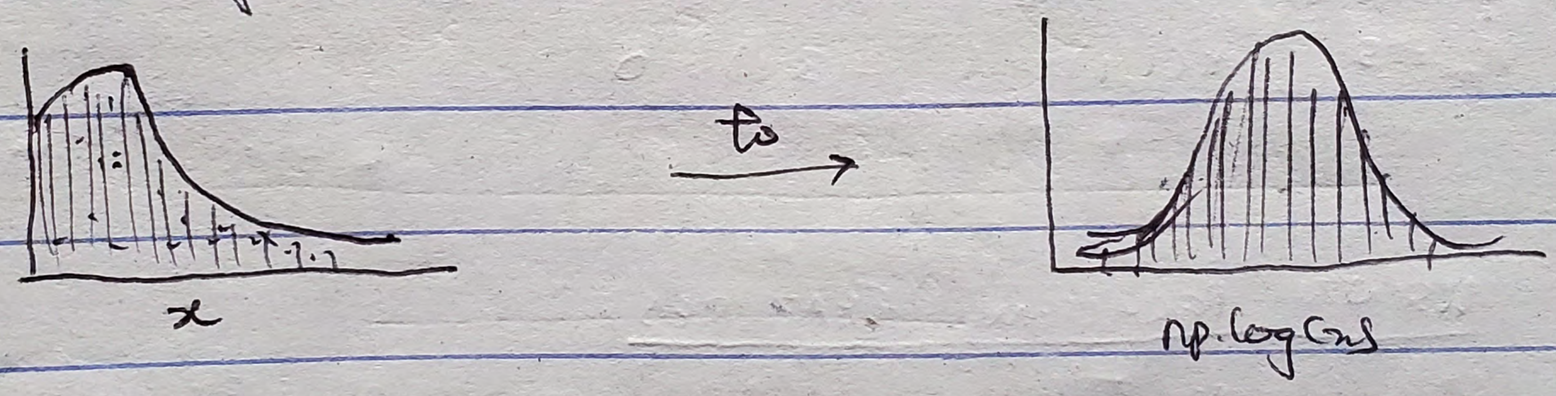
A.D.	S.L.
(i) Many different "types" of anomalies. Hard for any algorithm to learn from positive examples what the anomalies look like; future anomalies may look nothing like any of the anomalous examples we've seen so far → A.D. shines here because it looks @ "good" and "what isn't good" e.g. Fraud detection, Manufacturing - finding new previously unseen defects in manufacturing, monitoring machines in a data.	(ii) 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. E.g. Spam emails are likely to be the same. Manufacturing - finding known, previously seen defects, Weather prediction (sunny/rainy/snowy etc), disease classification.

CHOOSING WHAT FEATURES to USE

- **Tip 1:** Make sure you use GAUSSIAN features.

  How to handle non-gaussian features.

Transform



- You can use a log transform or $-\log(x_1)$ or $\log(x_1 + c)$

  square root transform: $\sqrt{x_1}$

  cube root: $\sqrt[3]{x_1}$

  If you get an error due to the presence of 0, use $\log(x_1 + 0.001)$

Just try log go till you get a gaussian shape.

- Always remember to apply the same transformation to the CV & Test set.

- **Tip 2:** Carry out Error Analysis for Anomaly Detection.

  After training a model, see what anomalies in the CV set that the algorithm is failing to detect and then look at them to see if they can inspire the creation of new features that will allow the algorithm to spot that example. You can create new features by combining current features.

What we want: $p(x) \geq \varepsilon$ large for normal examples x,
$p(x) < \varepsilon$ small for anomalous examples x

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

WK 2 COLLABORATIVE FILTERING

- Making RECOMMENDATIONS → Example: Predicting movie ratings.
- A user is expected to rate movies using zero to five stars

Movies	Alice (1)	Bob (2)	Carol (3)	Dave (4)
Love at last	5	5	0	0
Romance forever	5	?	?	0
Cute puppies of love	?	4	0	?
Nonstop car chases	0	0	5	4
Sword vs. Karate	0	0	5	?

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 $r(i,j) = 1$)

$n_u = 4$; $n_m = 5$, $r(1,1) = 1$, $r(3,1) = 0$

$y^{(3,2)} = 4$

USING PER-ITEM FEATURES

What if we have features of the movies? <romance (x_1) & action (x_2)>

Movie	Alice (1)	Bob (2)	Carol (3)	Dave (4)	x_1 (romance)	x_2 (action)
Love at last	5	5	0	0	0.9	0
Romance forever	5	?	?	0	1.0	0.01
Cute puppies of love	?	4	0	?	0.99	0
Nonstop car chases	0	0	5	4	0.1	1.0
Sword vs. karate	0	0	5	?	0	0.9

$n_u = 4$, $n_m = 5$, $n = 2$, $x^{(1)} = \begin{bmatrix} 0.9 \\ 0 \end{bmatrix}$, $x^{(3)} = \begin{bmatrix} 0.99 \\ 0 \end{bmatrix}$

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

$w^{(1)} = \begin{bmatrix} 5 \\ 0 \end{bmatrix}$ $b^{(1)} = 0$, $x^{(3)} = \begin{bmatrix} 0.99 \\ 0 \end{bmatrix}$

$w^{(1)} \cdot x^{(3)} + b^{(1)} = \begin{bmatrix} 5 \\ 0 \end{bmatrix} \cdot \begin{bmatrix} 0.99 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

$= 4.95$

For user j: Predict user j's rating for movie i as: $w^{(j)} \cdot x^{(i)} + b^{(j)}$

~~Extra Notations~~: $w^{(j)}, b^{(j)}$ = parameters for user j | $x^{(i)}$ = feature vector for movie i

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

∴ The cost function for learning $w^{(j)}, b^{(j)}$ for user j

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

$i:r(i,j)=1$ → we're summing only movies that user j has rated

no. of features

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

for recommender systems, for convenience sake, we can remove the constant $m^{(j)}$

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 $w^{(1)}, b^{(1)}, w^{(2)}, b^{(2)}, \ldots, w^{(n_u)}, b^{(n_u)}$ for all users:

Cost Function

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

COLLABORATIVE FILTERING ALGORITHM

× Used when we don't have the features x_1 & x_2. Exa

i	x_1	x_2
	?	?
	?	?
	?	?
	?	?

Assume that we already have the regression parameters $w^{(j)}, b^{(j)}$

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

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

→ using $w^{(j)} \cdot x^{(i)} + \cancel{b^{(j)}}$ since $b^{(j)} = 0$

$w^{(1)} \cdot x^{(1)} \approx 5$
$w^{(2)} \cdot x^{(2)} \approx 5$
$w^{(3)} \cdot x^{(3)} \approx 0$
$w^{(4)} \cdot x^{(4)} \approx 0$

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

P.T.O.

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

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

to learn $x^{(1)}, x^{(2)}, \dots, x^{(n_m)}$:

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

$\therefore$ To get the collaborative filtering algorithm, we combine the cost functions used to learn $[w^{(1)}, b^{(1)}, \dots, w^{(n_u)}, b^{(n_u)}]$ & $[x^{(1)}, \dots, x^{(n_m)}]$;

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

- cost function for collabo filtering

For Gradient descent;

repeat {

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

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

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

} parameters w, b, x

The name collaborative filtering refers to the sense ~~of what~~ that because multiple users have rated the same movie, collaboratively, that allows you to guess what are appropriate features for that movie and this in turn allows you to predict how other users that haven't rated that same movie may decide to rate it in the future

<u>BINARY LABELS : FAVS, LIKES & CLICKS.</u>

* Here we generalize the collab filtering algorithm to cute settings where the users rate on a binary scale <like or don't like, 0 or 1, etc>.

Example of data

Movie	Alice (1)	Bob (2)	Carol (3)	Dave (4)
→ Love @ last	1	1	0	0
Romance 4ever	1	?	?	0
Cute puppies of love	?	1	0	?
Nonstop car chases	0	0	1	1
Sword vs Karate	0	0	1	?

- By predicting how the users would likely rate the movies they haven't seen, we can decide whether or not to recommend it to them.
- Depending on the situation, it's important to know what "1", "0" & "?" mean.

Previously: we to 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 $g(z) = \frac{1}{1+e^{-z}}$

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

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

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

for all examples.

HW2 RECOMMENDER SYSTEMS IMPLEMENTATION DETAIL

MEAN NORMALIZATION

Normalizing your ratings to have a consistent average values improves the efficiency of your algorithm. To explain mean normalization, we will add a 5th user, Eve, who has not yet rated any movie

Movie	Alice (1)	Bob (2)	Carol (3)	Dave (4)	Eve (5)
Love @ last	5	5	0	0	?
Romance 4ever	5	?	?	0	?
Cute puppies of love	?	4	0	?	?
Nonstop car chases	0	0	5	4	?
Sword vs Karate	0	0	5	?	?

If you want to predict $w^{(5)}$ & $b^{(5)}$ that is, ratings for Eve, you'd most likely get 0s as her rating because none of her values play a role in the cost function, so minimizing this means making the parameters w as small as possible

~~Ia~~ Mean normalization will help this algorithm come up w/ better predictions of the movie ratings for a new user that has not yet rated any movie.

To explain mean normalization;

(i) We'll express the values in form of a matrix

(ii) We'll compute the average ~~mov~~ rating for each movie, μ

(iii) For every rating given per movie, subtract its average rating

(iv) This gives us our new values $\langle y^{(i,j)}$. With this we predict the normal

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

(v) In order to not predict a -ve rating, we add back μ_i

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

Basically, this algorithm causes the initial preds for the new user to be equal to the average movie rating.

Example: $\rightsquigarrow$

$$\begin{bmatrix} 5 & 5 & 0 & 0 & ? \\ 5 & ? & ? & 0 & ? \\ ? & 4 & 0 & ? & ? \\ 0 & 0 & 5 & 4 & ? \\ 0 & 0 & 5 & 0 & ? \end{bmatrix} ; \mu = \begin{bmatrix} 2.5 \\ 2.5 \\ 2 \\ 2.25 \\ 1.25 \end{bmatrix} \xrightarrow{(-\mu)} \underbrace{\begin{bmatrix} 2.5 & 2.5 & -2.5 & -2.5 & ? \\ 2.5 & ? & ? & -2.5 & ? \\ ? & 2 & -2 & ? & ? \\ -2.25 & -2.25 & 2.75 & 1.75 & ? \\ -1.25 & -1.25 & 3.75 & -1.25 & ? \end{bmatrix}}_{Y^{(i,j)}}$$

for user j, on movie i predict:

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

user 5 (Eve), $w^{(5)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$; $b^{(5)} = 0 \rightsquigarrow w^{(5)} \cdot x^{(1)} + b^{(5)} + \mu_1 = 2.5$

* Tip:
~~Normali~~ Normalize rows to help with initial ratings for each movie for a new user

There's no need to normalize columns to find out a movie rating. This one doesn't make sense ~~becau~~

∴ Mean Normalization helps the algorithm run faster and predict better when there are few users who have rated a movie.

$$Y_{norm}^{(i,j)} = y^{(i,j)} - \mu_i$$

where $\mu_i = \dfrac{1}{\sum_j r(i,j)} \sum_{j:r(i,j)=1} y^{(i,j)}$

TENSOR FLOW IMPLEMENTATION of COLLABORATIVE FILTERING

- Apart from its use in Neural networks, tensorflow also works well in collabo. filtering. It can be used to compute the derivatives of the cost function which in turn will be used to optimize the cost.

- Recall that gradient descent algorithm in ML involved calculating the derivatives and simultaneously updating w & b. This is repeated until the algorithm reaches convergence i.e. when there isn't that much change in param w & b w/ every update.

- Now, let's look @ a Custom training loop for collab. filtering.

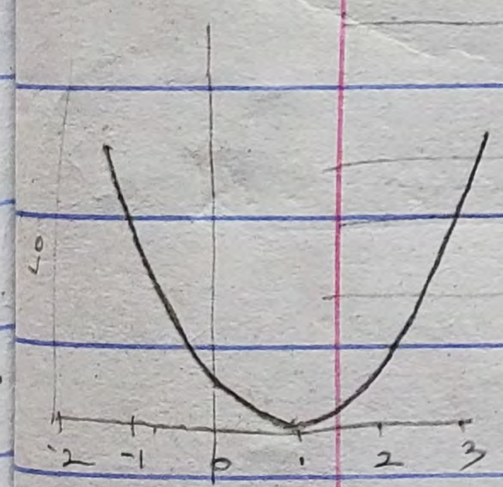
We'll assume a cost function, $J = (\underbrace{wx}_{fwb} - \underbrace{1}_{y})^2$ and we will carry out the gradient descent algorithm, updating w while fixing b at 0.

(i)
```
w = tf.Variable(3.0)
x = 1.0
y = 1.0 # target variable
alpha = 0.01

iterations = 30
for iter in range(iterations):
    # use Tensorflow's gradient tape to record the steps
    # used to compute the cost J, to enable auto differentiation
    with tf.GradientTape() as tape:
        fwb = w*x
        cost J = (fwb - y)**2
    # Use the gradient tape to calculate the gradients
    # of the cost with respect to the parameter w.
(iii) [dJdw] = tape.gradient(cost J, [w])

    # Run one step of gradient descent by updating
    # the value of w to reduce the cost
(iv) w.assign_add(-alpha * dJdw)
```

(ii)



- Tf-variables are the parameters that we want to optimize. We initialise w as 3.0

(iv). The syntax for GradientTape, tensorflow records the steps taken to compute the cost J. This is needed to enable automatic differentiation. Tensorflow saves the sequence of operations as "tape".

(ii) ~~blank~~ "tape" containing the recorded steps; ~~line~~ block (iii) allows tensorflow to compute the derivate of J w/respect to w. (iii) knows that we want to calculate dJ/dw because we specified w in —(i)

(i) —(iii) ... ient (cost J, [w])

(iv) Handles the update of w ⇝ $\langle w = w - \alpha \frac{\partial J}{\partial w} \rangle$. ~~These~~ tf variables require special function to modify. That is why we use the "assign_add" function

Auto Diff
Auto Grad } → Automatic differential / gradient in tensorflow. We just show tensorflow how to calculate the cost and it ~~auto~~ automatically gives us the gradients when called

FINDING RELATED ITEMS

The features $x^{(i)}$ of item i are quite hard to ~~interp~~ interpret. Nonetheless, the n features describe the item; i. To find other items related to it, find item k with features $x^{(k)}$ similar to $x^{(i)}$.

- To find / calculate similarity between $x^{(i)}$ & $x^{(k)}$, we look @ those squared distances;

$$\sum_{i=1}^{n} \left(x_i^{(k)} - x_i^{(i)}\right)^2 = \left\| x^{(k)} - x^{(i)} \right\|^2$$

↳ we look at those with the smallest distances

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

WK 2 CONTENT-BASED FILTERING

* Collaborative filtering: This recommends items to you based on ratings of users who gave similar ratings as you.
* Content-based filtering: Recommends items to you based on features of users & items to find a good match

Here's an example of user & ~~feature~~ item features for a movie recommender system

USER FEATURES

- Age
- Gender
- Country
- Movies watched
- Avg rating per genre

$\rightarrow x_u^{(j)}$ for user j

MOVIE FEATURES

- Year
- Genre / Genres
- Reviews
- Avg rating

$x_m^{(i)}$ for movie i

* Vector size of $\vec{x}_u^{(j)}$ & $\vec{x}_m$ could be different

— We'd be predicting the rating of user j on movie i as

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

• we'll get rid of $b^{(j)}$ & replace $w^{(j)}$ & $x^{(i)}$

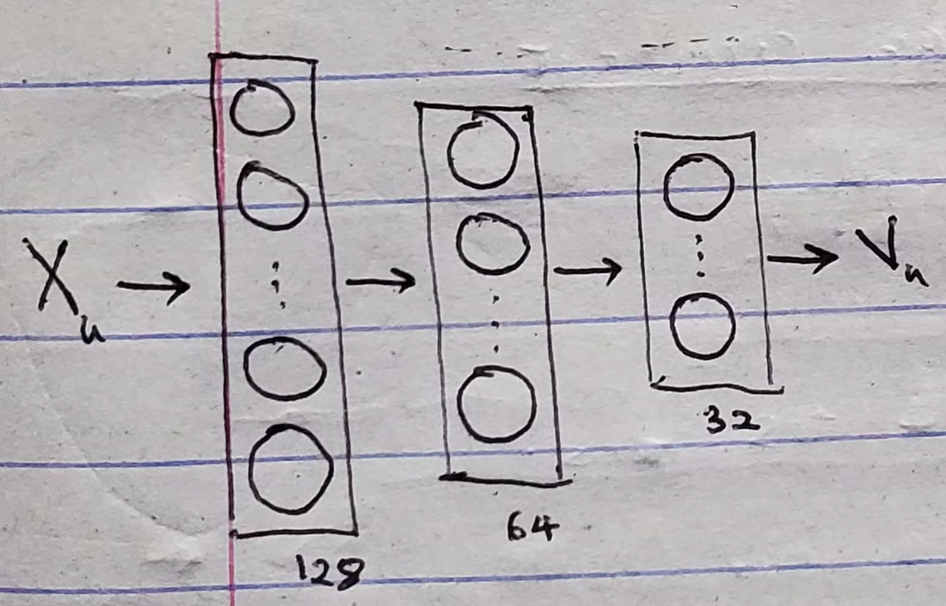
vector computed from $x_u^{(j)}$ ← $v_u^{(j)} \cdot v_m^{(i)}$ → vector computed from $x_m^{(i)}$

⇒ Even though $x_u^{(j)}$ & $x_m^{(i)}$ could be a lot different in size, $v_u^{(j)}$ & $v_m^{(i)}$ have to be the same size.

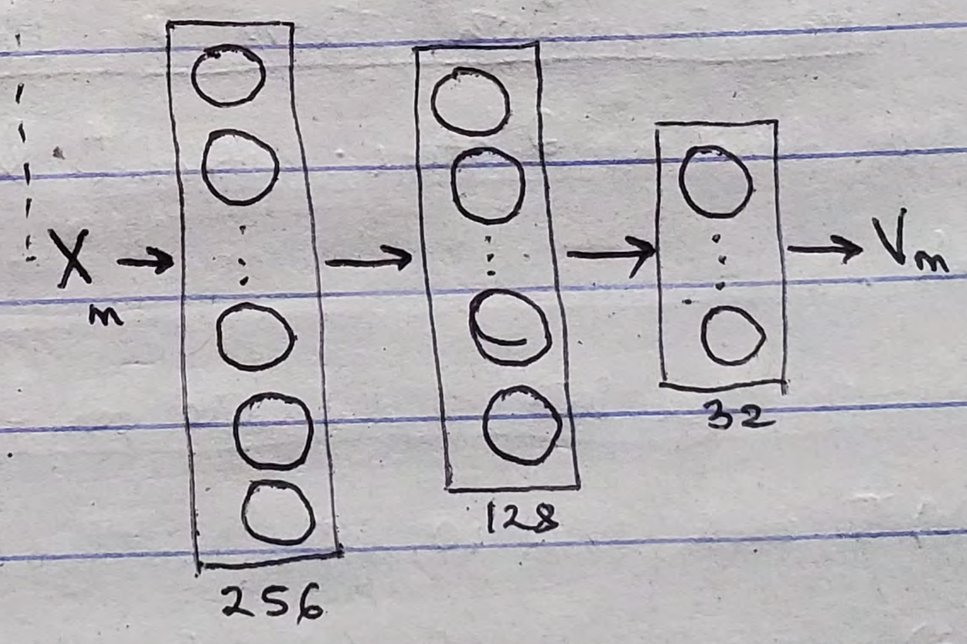
DEEP LEARNING FOR CONTENT BASED FILTERING

• For getting v_u & v_m from user features $x_u^{(j)}$ & movie features x_m, we make use of Neural Network Architect..

$x_u \rightarrow v_u$ **User Network**

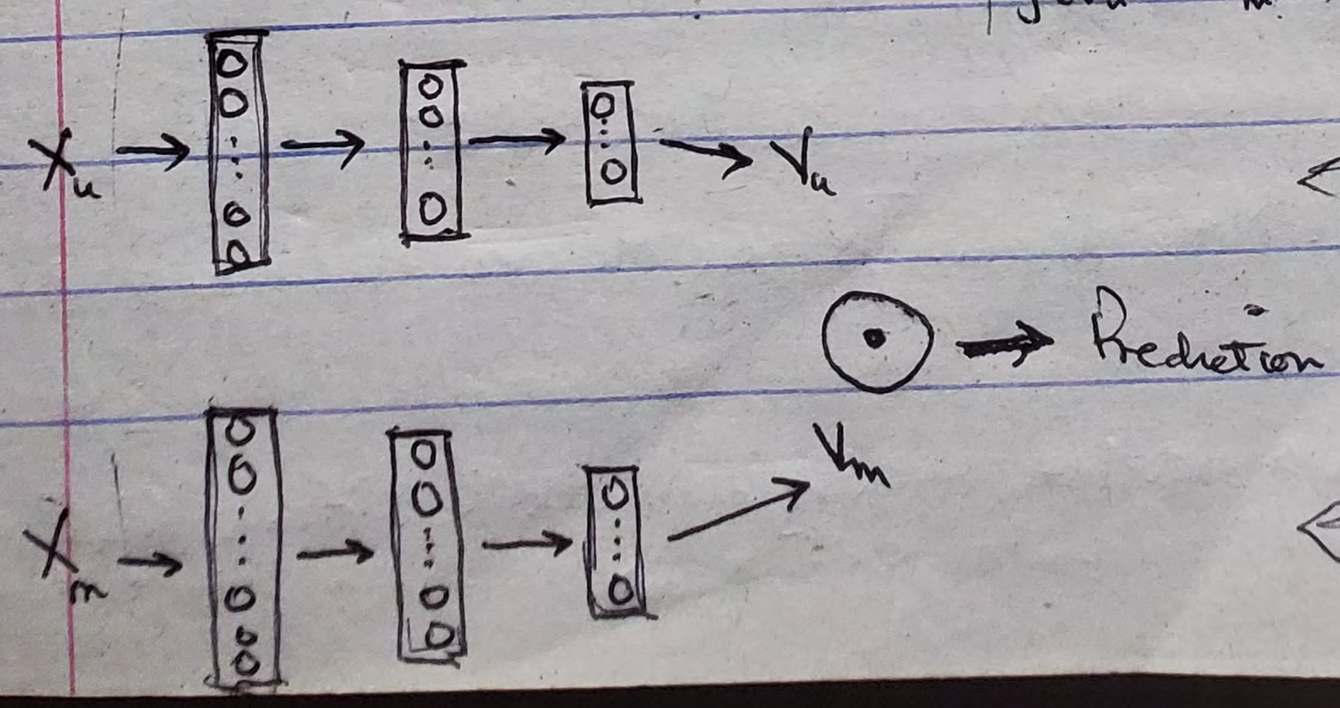


$x_m \rightarrow v_m$ **Movie Network**



Prediction: $v_u \cdot v_m$

For binary labels, $g(v_u^{(j)} \cdot v_m^{(i)})$ to predict the $P(y^{(i,j)} = 1)$



Combining the User Network & the Movie Network

• This Neural Network Architecture has a lot of parameters. So how do we train all the parameters for the user network & the movie network.

Here are the cost function J;

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

Gradient descent, finds the parameters that'll ~~make the~~ minimize the loss (squared error in this case) for any optimization algorithm. Regularization can be added to encourage the NN to keep the values of their parameters small.

LEARNED USER & ITEM VECTORS

- $v_u^{(j)}$ is a vector of length 32 that describes user j with features $x_u^{(j)}$
- $v_m^{(i)}$ is a vector of length 32 that describes movie i w/ features $x_m^{(i)}$

To find similar movies we look for the movies that have the least square distance between them → $\| v_m^{(k)} - v_m^{(i)} \|^2$ ... small

Note: This can be computed ahead of time.

RECOMMENDING FROM A LARGE CATALOGUE

This can be done in 2 steps: Retrieval & Ranking

(i) Retrieval:

- Generate large list of plausible item candidates [its okay to include less likely items]
  - 1) For each of the 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

  This can generate about 100s of movies. Some the user might like & some the user might not like. Not a problem. The goal is to cover a broad range of plausible movies.

- Combine retrieved items into list, removing duplicates and items already watched or purchased.

(ii) Ranking:

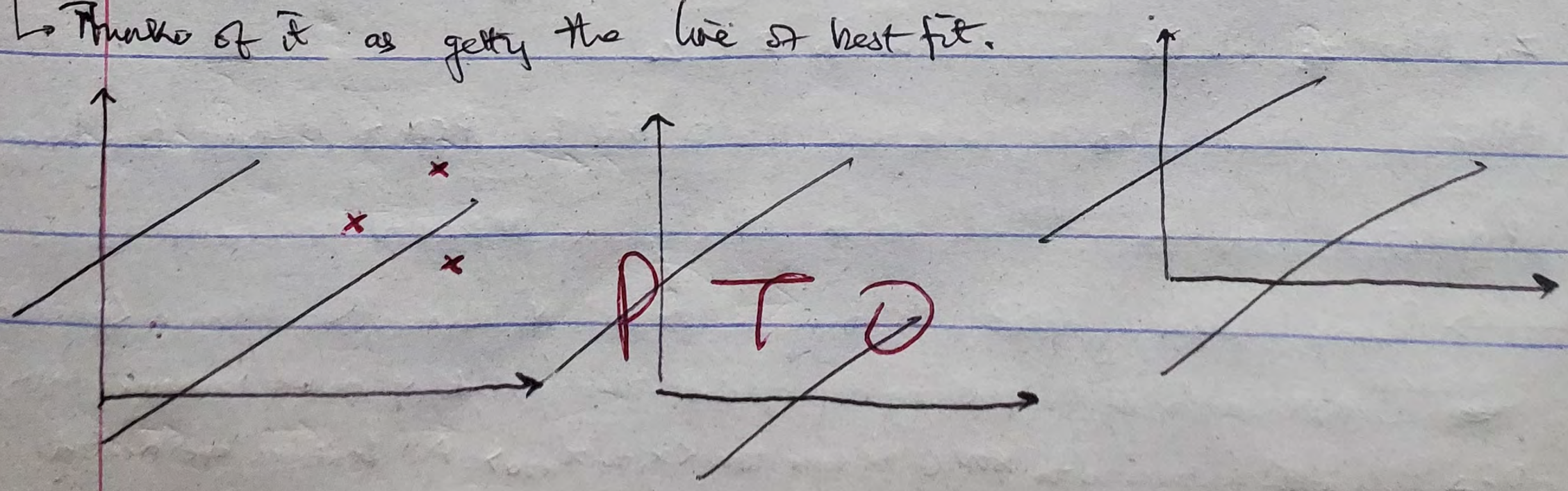
- Take list retrieved and rank using learned model
- Display ranked items to user.

* Retrieving more items results in better performance but slower recommendations.
* To analyze/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)

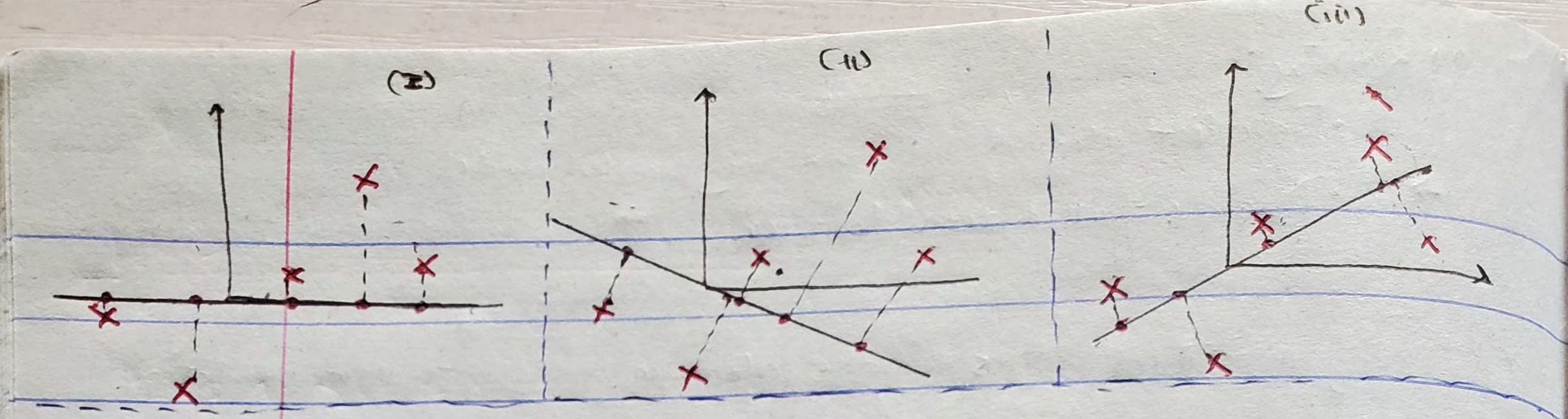
Note: To speed up the response time of your recommendation system, you can ~~pre-comp~~ pre-compute the vectors V_m for all the items you might recommend. This can be done even before a user logs in to your website and even before you know x_u or V_u vector. This is because the output of the item/movie neural network V_m is not dependent on the user network when making predictions. Precomputing the results speeds up the prediction process

Wk 2 PRINCIPAL COMPONENT ANALYSIS

* An algorithm mostly used for visualization. It helps plot data with many features by reducing the features down to two.
  - → The idea of PCA is to find new axis and coordinates. This helps us use few numbers to describe the ~~feature~~ item
- PCA can reduce 20 to +1000 features to 2 or 3 features.
- We're looking for a line (or an axis) that captures the variance in the dataset
  - → Think of it as getting the line of best fit.



PTO



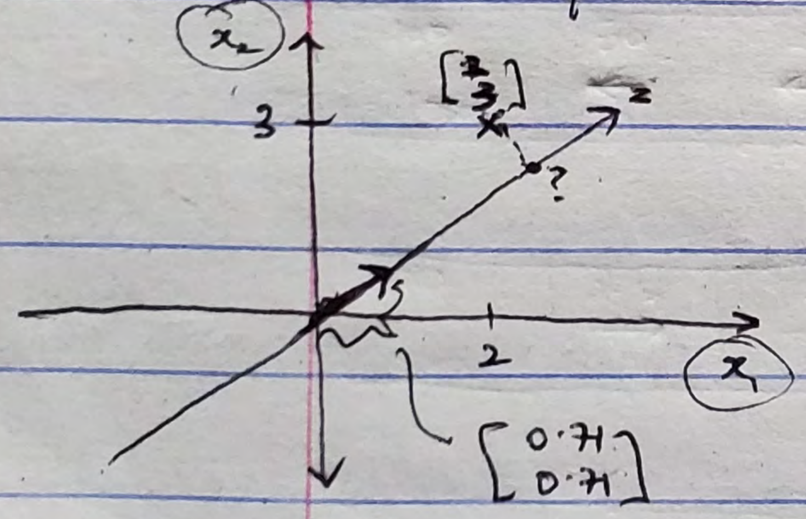
using "x" axis as "z"

Aim is to capture variance along axis (I) is also better choice because it captures majority of the variability which was 2 axes. This axis is the Principal Component

→ The PRINCIPAL COMPONENT is the axis that when you project your data unto it, you end up with the largest possible amount of variance.

Note: The principal components must always pass through the centre of the data. The ~~mf~~ PCs for diagram ~~of~~ ~~II~~ is much ~~not~~ more wrong because they do [technically, it correct if the data was normalized around 0]

>> How do we find/calculate the projection on the z axis?



- We know the coordinate x_1, x_2 $\begin{bmatrix}2\\3\end{bmatrix}$ & we know the coordinate of length 1 on the z axis from origin $\begin{bmatrix}0.71\\0.71\end{bmatrix}$

∴ new coordinate of projection;

$$\begin{bmatrix}2\\3\end{bmatrix} \cdot \begin{bmatrix}0.71\\0.71\end{bmatrix} = 3.55$$

∴ distance of the projection from origin is 3.55

→ <Make sure your data is normalized around the origin (0)>

- The 2nd axis is perpendicular to the 1st axis. This means that z_2 is at 90° to z_1. Also, z_3 will be perpendicular to z_1 & z_2
- Although I used PCA to find the line of best fit, PCA is different from Linear Regression

→ Simple linear regression, we're trying to predict Y, hence, the distance (projection) is measured in the direction of the y axis, giving special treatment to Y as we minimize them

- PCA deals with features up to 1000 @ the same time <x_1 to x_{1000}>. All features are treated

equally. All we're looking for is an axis Z that will minimize variability for all features.

- PCA is in no way similar to Linear Regression!!

APPROXIMATION → A way of getting the ~~factors~~ coordinates of the features used to get the values on the Z axis

Recall, from our previous example, we had 3.55. ~~Therefore~~ Therefore to get our approx x_1 & x_2,

$$3.55 \times \begin{bmatrix} 0.71 \\ 0.71 \end{bmatrix} = \begin{bmatrix} 2.52 \\ 2.52 \end{bmatrix} \rightarrow \text{approximate } \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

Z value

length 1 on Z axis

PCA IN SCIKIT-LEARN

- Optional preprocessing: Perform Feature Scaling

(1) "Fit" the data to obtain 2 (or 3) new axes (principal components

└ fit <also includes mean normalization>

(2) Optionally examine how much variance is explained by each principal component

└ explained_variance_ratio

(3) Transform (project) the data onto new axes → transform

Shows how many % of the variance is explained by each PCA.

Applications of PCA

(i) For visualization

└ Less frequent uses: (a) Data compression <to reduce storage or transmission costs>

└ (b) Speed up training of a supervised learning model

WK3 REINFORCEMENT LEARNING

- Supervised learning doesn't work well for autonomous systems.
- We work with reward functions in this case. We don't tell the system how to do it, but tell it what to do & then reward it +vely -vely

Applications
- (i) Controlling robots
- (ii) Factory optimization
- (iii) Financial (stock) trading
- (iv) Playing games (including video games)

The core elements of any reinforcement learning applications:

→ $(s, a, R(s), s')$ — s = state, a = action

(iii) $R(s)$ = Reward of that state (iv) s' = Next state

THE RETURN in REINFORCEMENT LEARNING

The concept of the return: Rewards that can be gotten quicker are better than ones that take a longer time

Return = sum of returns, weighted by the discount factor

→ $R_1 + \gamma R_2 + \gamma^2 R_3 + \dots$ (until terminal state) → end of operation / behavior

Discount factor $\gamma = 0.9$ (gamma) <most times this value is closer to 1 e.g. 0.99, 0.999>

Examples:

100	50	25	12.5	6.25	40	← return
100	← 0	← 0	← 0	← 0	40	← reward

$\gamma = 0.5$

*moving to the left

100	2.5	5	10	20	40
100	→ 0	→ 0	→ 0	→ 0	40

*moving to the right

100	50	25	12.5	20	40
100	← 0	← 0	← 0	→ 0	40

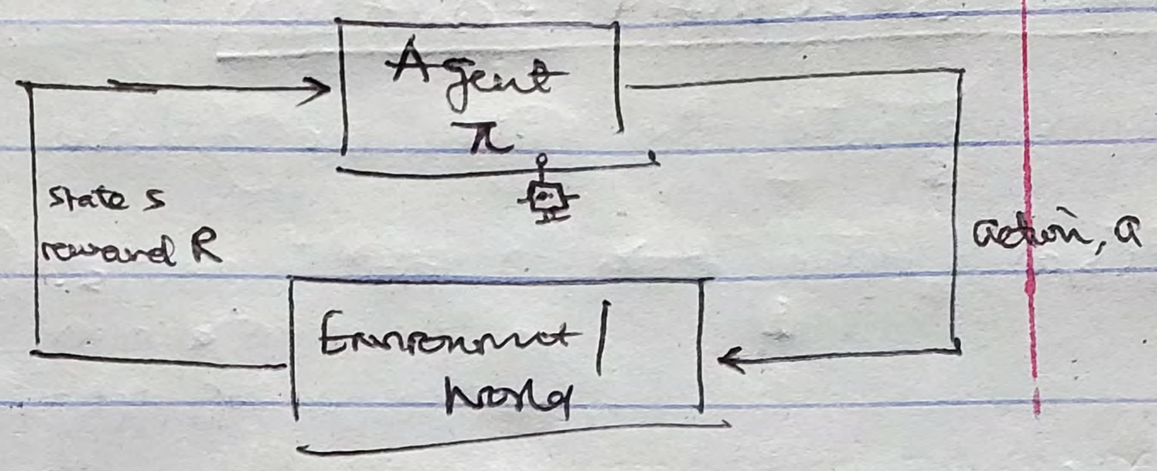
MAKING DECISIONS: POLICIES IN REINFORCEMENT LEARNING

- Our goal in reinforcement learning is to come up with a function which is called a policy (π) which takes as input any state (s) and map it to an action (a) so as to maximize the return
- The whole process of having a set of states, a set of actions, rewards, discount factor, & policy π, all to maximize the return refers to the MARKOV DECISION PROCESS

The term "Markov" in MDP refers to the thought that the future only depends on the current state and not on anything that might have occured prior to getting to the current state.

- This means that in a Markov Decision Process, the future depends only on where you are now, not on how you got there.

Another way to see this → We have an agent with which we get to choose action a. And as a result of that action, something happens in the world



Action a is chosen using policy π and based on what happens in the world, we get to observe what state(s) we're in & what reward (R) that we get.

Wk 3 STATE-ACTION VALUE FUNCTION ---- aka (Q function)

- This is the key quantity that reinforcement learning tries to compute $\langle Q(s,a) \rangle$

$Q(s,a)$ — A function of the state you're in & action chosen to take

└ Returns a number that equals the return if you
- start in state s
- take action a once
- then behave optimally after that.

For example: using the optimal policy for our mars rover example

100	50	25	12.5	20	40
↑	←	←	←	→	
100	0	0	0	0	40

... return
- - - - - action
... reward

Let's find the state action value function, $Q(s,a)$ @ different states + actions

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

$Q(2,\leftarrow)$ $Q(2,\rightarrow)$ $Q(4,\leftarrow)$

$$Q(2,\rightarrow) = \overset{(3)}{0} + 0.5\overset{(2)}{(0)} + 0.5^2\overset{(1)}{(0)} + 0.5^3(100) = 12.5$$

$$Q(2,\leftarrow) = \overset{(2)}{0} + 0.5\overset{(1)}{(100)} = 50$$

$$Q(4,\leftarrow) = \overset{(4)}{0} + 0.5\overset{(3)}{(0)} + 0.5^2\overset{(2)}{(0)} + 0.5^3\overset{(1)}{(100)} = 12.5$$

- It's also important to note that 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 $\max_a Q(s,a)$.

Other names of State Action Value function: Q function; Optimal Q function; Q^*

KIK 3 BELLMAN EQUATION

- Helps us compute the state value function ($Q(s,a)$)

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

Notations for the equation: (a) s: current state (b) $R(s)$ = reward of current state

(c) a: current action (d) s': state you get to after taking action a

(e) a': action that you take in state s'

Reward you get right away [Immediate reward/return]

Equation: $Q(s,a) = R(s) + \gamma \max_{a'} Q(s',a')$ → Return from behaving optimally starting from state s'

... using our prev. example

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

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

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

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

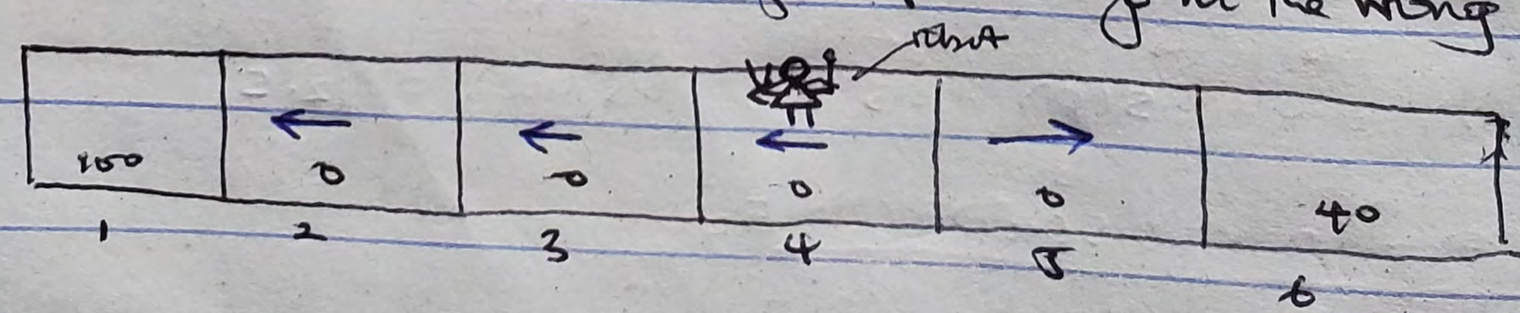
$$Q(4,\leftarrow) = R(4) + 0.5 \max_{a'} Q(3,a') = 0 + (0.5)25 = 12.5$$

- At the terminal state (1 & 6), $Q(s,a) = R(s)$ because there's no s'

RANDOM (STOCHASTIC) ENVIRONMENT

* Here, we'll look at the generalization of the reinforcement learning framework where models random environment.
  - Most times our robot might slip and go in the wrong direction

Example:



ideal movement: 4-3-2-1; return: $0 + \gamma 0 + \gamma^2 0 + \gamma^3 100$

random slip @ 2: 4-3-2-3-2-1; return: $0 + \gamma 0 + \gamma^2 0 + \gamma^3 0 + \gamma^4 0 + \gamma^5 100$

random slip @ 4: 4-5-6; return: $0 + \gamma 0 + \gamma^2 40$

In essence in stochastic environments, there isn't one sequence of rewards that we're absolutely sure of

∴ What we're interested in, for stochastic problems, is maximizing the average values of all possible ~~rewards~~ sequences of rewards

Expected Return = Average $(R_1 + \gamma R_2 + \gamma^2 R_3 + \gamma^3 R_4 + \dots)$

$$= E[R_1 + \gamma R_2 + \gamma^2 R_3 + \gamma^3 R_4 + \dots]$$

- In statistics "expected value" ~~is also~~ can also be seen as average of the values

∴ The modified Bellman Equation;

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

In the normal Bellman Eq, s' ~~can be either the~~ left value is fixed since it takes the optimal value. In a random env. the s' is also random as it can take any of the values (left or right in this case)

KK3 CONTINUOUS STATE SPACES

- Whilst our ~~example~~ previous example made use of a discrete state i.e. we could only be at one ~~state~~ position out of six ~~or~~ possibilities, real world examples could have continuous states. For example, for a autonomous ~~car~~ truck, we would like to know the x value, y value, orientation (angle @ which we turn), speed in the x direction, speed in the y direction and speed of turn.

THE LUNAR PROBLEM <2>

- Learn a policy π that given; $s =$

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

- x — how far to the left or right
- y — how high up we are
- $\dot{x}$, $\dot{y}$ — The velocities @ x & y direction
- θ — tilt angle to the left or right
- $\dot{\theta}$ — angular velocity
- l, r — 1/0 for if the left or right leg is grounded or not

→ pick action $a = \pi(s)$ so as to maximize the return. $[\gamma = 0.985]$

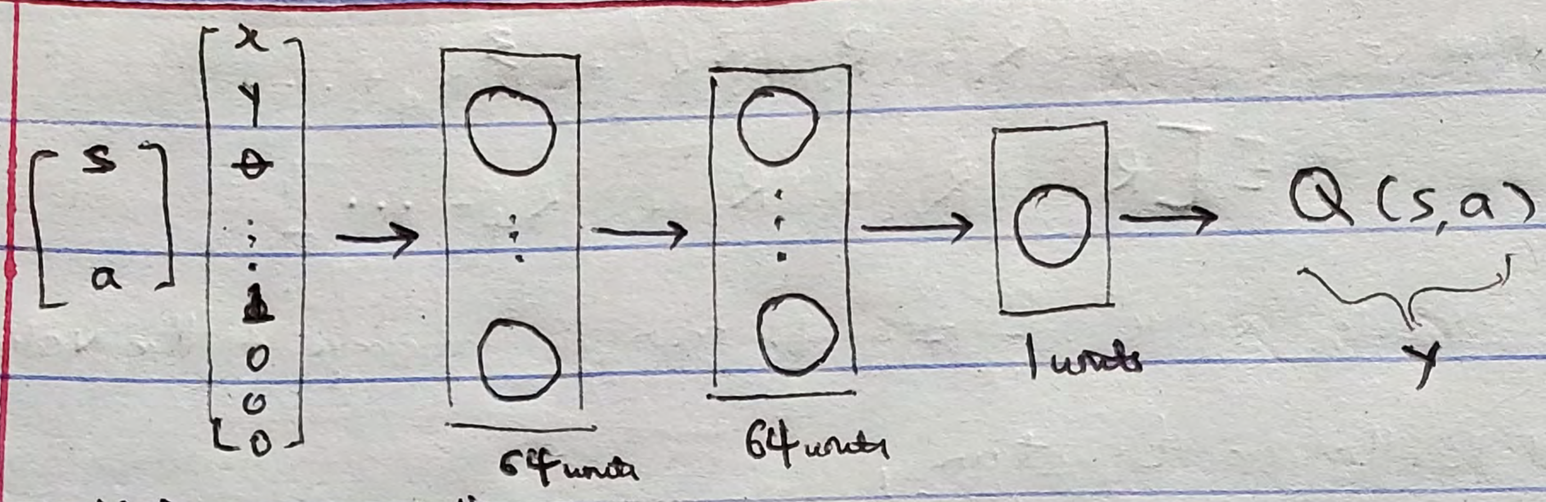
Actions = (i) do nothing; (ii) left thruster; (iii) main thruster, (iv) right thruster

Reward function

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

KK3 DEEP REINFORCEMENT LEARNING

$\vec{x} = \begin{bmatrix} s \\ a \end{bmatrix}$



12 inputs

8 vector values of s & one hot encoder of actions

nothing (1,0,0,0)

- In a state s, use neural network to compute
  - Q(s, nothing)
  - Q(s, left)
  - Q(s, main)
  - Q(s, right)
- Pick the action a that maximizes Q(s,a)

Note: The neural network doesn't predict an action based on the state, Instead it takes the state and action pair & predicts the state-action values function for that action in that state.

How then do we train this NN?

- (i) Using the Bellmans equation & x values (states & actions), we find the y values (Q(s,a)). This gives us training examples x, y.
- (ii) Using supervised learning, we learn a mapping from x (the state action pair) to y (Q(s,a))

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

- We use a lunar lander simulator to take random actions at various states, even though we have no optimal policy yet. With this, we get the training example as seen below

$(s, a, R(s), s') \rightarrow y = \ldots$

x	y
$x^{(1)} = (s^{(1)}, a^{(1)})$	$y^{(1)}$
$x^{(2)} = (s^{(2)}, a^{(2)})$	$y^{(2)}$
⋮	⋮
$x^{(100000)} = (s^{(100000)}, a^{(100000)})$	$y^{(100000)}$

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

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

⋮

$(s^{(100000)}, a^{(100000)}, R(s^{(100000)}), s'^{(100000)})$

* Note that $\max_{a'} Q(s',a')$ for now is a random guess which gets better over time. Don't stress yourself, I no sabi too. Just take it like that.

LEARNING ALGORITHM

$x = s, a$, $y = Q(s,a)$

Initialize neural network randomly as guesses of $Q(s,a)$

Repeat {

- Take actions in the lunar lander. Get $(s, a, R(s), s')$. → Done to minimize computer memory usage

- Store 10,000 most recent $(s, a, R(s), s')$ tuples. ← aka $\boxed{\text{Replay Buffer}}$

  ↳ experience for our learning algorithm

- Train Neural Network:

  Create training set of 10,000 examples using; — still a guess ??

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

  Immediate return ← $R(s)$; γ = discount factor

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

- Set $Q = Q_{new}$ → This improves our guesses/estimates of the Q function so that as you run the algorithm more, it gets far better

* This algorithm is called the DQN algorithm: Deep Q-Network

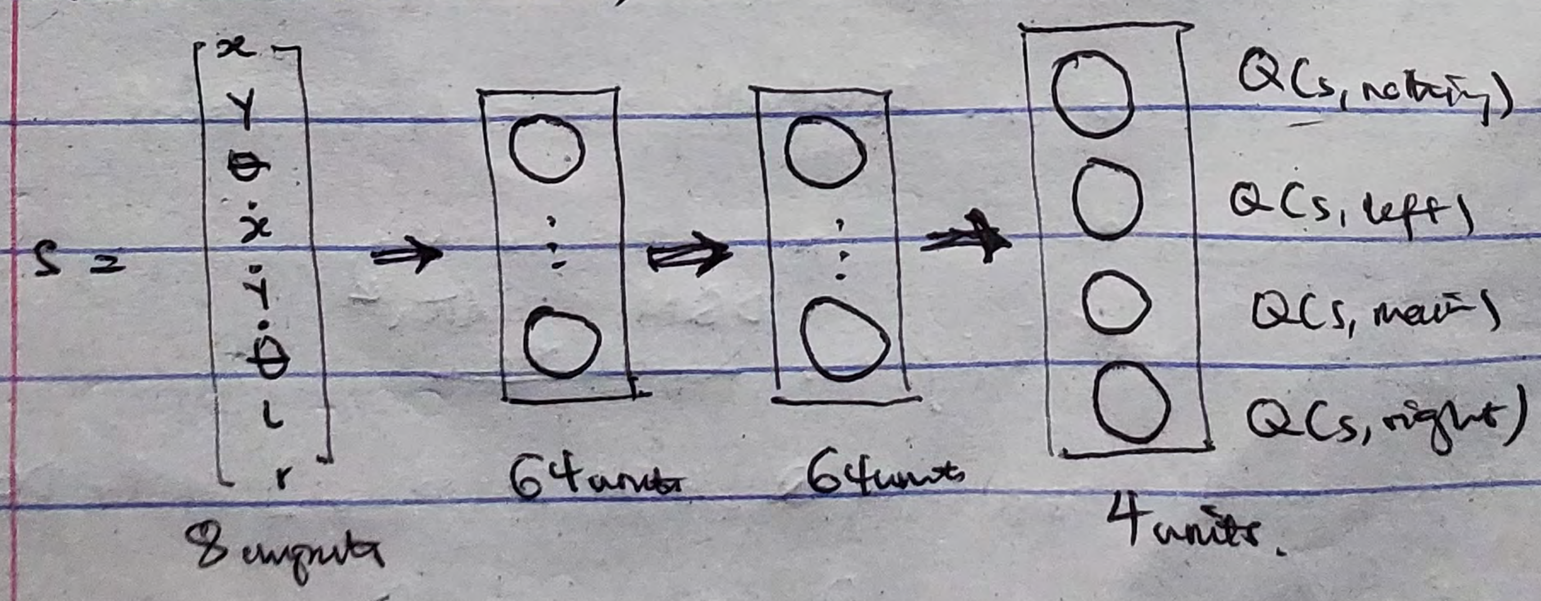
ALGORITHM REFINEMENT: IMPROVED NEURAL NETWORK ARCHI.

- Recall that the Deep Q-Network @ state s, we would need to compute $Q(s,a)$, len(a) number of times i.e. the number of actions that we have. And then pick the action a that gives the largest $Q(s,a)$ value.

  This is inefficient.

- It's more efficient to train a NN to output all values simultaneously.

Modified NN architecture;



ALGORITHM REFINEMENT: ε-GREEDY POLICY

* This helps us ~~the~~ take action in the "take action" step of the DQN algorithm

There are two options to pick when taking action

- Option 1: Pick the action a that maximizes $Q(s,a)$ [aka Greedy ~ Exploitation]
- Option 2: - With probability 0.95, pick the action a that maximizes $Q(s,a)$
  - With probability 0.05, pick an action a randomly [aka Exploration]

~ Option 2 is better because since the model maximize $Q(s,a)$, it'll tend to ignore actions ~~that were at a spot~~ that were initialized at a low value. Allowing the model to randomly pick an action makes it learn that sometimes that action (which was purposefully initialized @ a low value perhaps ~~been~~ because too much of it is bad) is sometimes useful

- Option 2 is called "ε-greedy policy" with ($\varepsilon = 0.05$) (picking an action randomly)
- One trick used: is to start ε high & gradually decrease it

ALGORITHM REFINEMENT: MINI BATCH & SOFT UPDATES

- Mini batch: When running the gradient descent algorithm, even in supervised learning, recall that we needed to take the average of all m training examples @ every iteration of the gradient descent algorithm. This becomes an issue when m is really large, ex. m = 100000. In order to improve the computational performance you can take ~~pc~~ batches of 1000 (x^i) which will be used for every calculation. The first iteration use the first 1000, the next iteration uses the next 1000 and so on. Eventually you reach the global minimum and its less computationally expensive.
  - This can also be applied when training our Neural Network in DQN algorithm ~~Neural Net~~
- ~~The~~ Soft update: Focuses on the update of Q ["Set Q = Q_{new}"] in the DQN alg
  - This is to prevent overwriting a good estimate of Q with a wrong Q_{new} ~~since~~ iteration is done → The parameters in the NN for Q = W, B and that of Q_{new} = W_{new}, B_{new}

With the soft update, what we do is make W 99% the old W + 1% of the new W

Same thing with the B parameter.

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

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

0.99 & 0.01 are hyperparam that can be changed