Clustering

Clustering looks at a dataset and tries to find points that are similar to each other and group them together.

The data has no labels, only $x^{(i)}$, no $y^{(i)}$. The algorithm has to figure out the structure on its own.

Some applications:

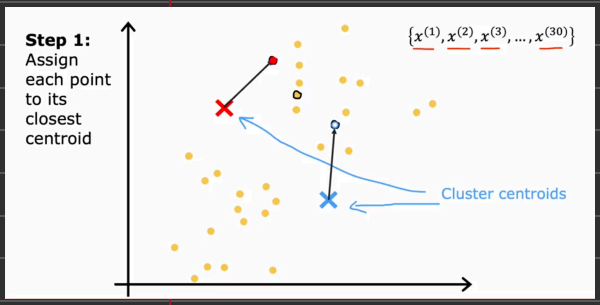
* Grouping similar news article
* Market segmentation
* DNA microarray data
* Astronomy

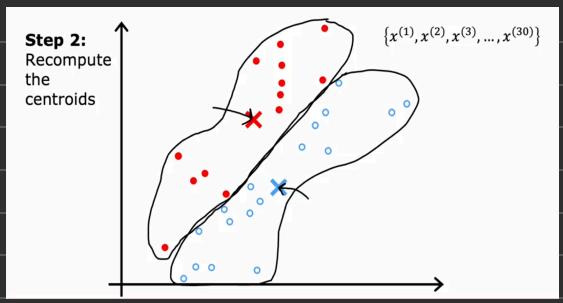
k-means intuition

K-means repeatedly does two things:

* Assign each point to the closest cluster centroid
* Move each centroid to the average of the points assigned to it.

We start with K randomly placed centroids. Color each point by which centroid is nearest. Then we move each centroid to the mean of its colored points. We then recolor and repeat. The centroids stop moving once assignments stabilize.





K-means Algorithm

We use the following notation:

* k is the number of clusters
* $x^{(i)} \in \mathbb{R}^n$
* μ_k is the position of the k centroid

We first randomly initialize $\mu_1, \mu_2, \ldots, \mu_k$

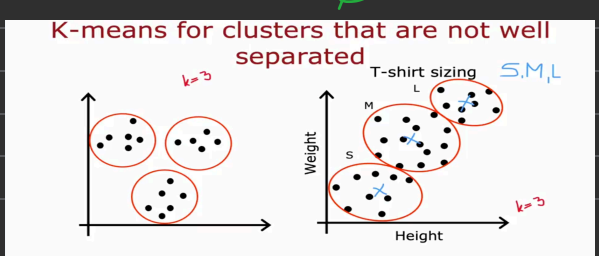
Repeat:

```
# Assign points to cluster centroids
for i = 1 to m:
    c^(i) := index k (from 1 to K) that minimizes ||x^(i) - μ_k||^2

# Move cluster centroids
for k = 1 to K:
    μ_k := average (mean) of points assigned to cluster k
```

Note: In practice, the squared distance $\|x^{(i)} - \mu_k\|^2$ is minimized, not the distance because it is cheaper to work with the squared distance in terms of compute and they have the same minimum.

Corner case: If a centroid ends up with zero points assigned the most common move is to delete it, ending with $K-1$ clusters. If we really need K, we re-initialized the centroid.



Optimization Objective.

Some notations

* $c^{(i)}$ is the index of cluster $(1, 2, \ldots, K)$ to which $x^{(i)}$ is currently assigned
* μ_k is the cluster centroid k
* $\mu_{c^{(i)}}$ is the cluster centroid of cluster to which example $x^{(i)}$ has been assigned.

K-means is minimizing a specific cost function, called the distortion function:

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

The average squared distance from each training example to the centroid it has been assigned to.

The two steps of K-means corresponds to minimizing J over different variables:

* The assignment step minimizes J over $c^{(i)}$ with μ_k held fixed. Closest centroid is the best choice.

* The move step minimizes J over μ_k with $c^{(i)}$ held fixed. The mean is the point that minimizes the sum of squared distance to a set of points.

So J goes down (or stays flat) every iteration. It should never go up. If implementation shows J increasing, there is a bug!

Convergence occurs when J stops decreasing or decreases very slowly.

Note: K-means converges to a local minimum, not necessarily the global one.

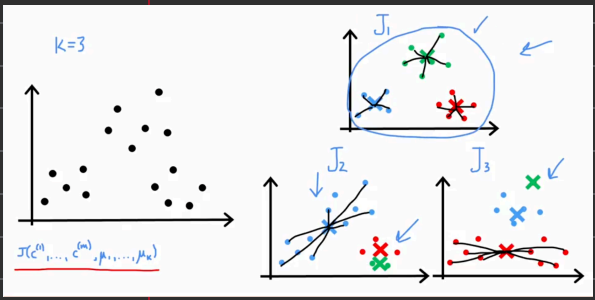
Initializing K-means

Random initialization:

* Choose $K < m$
* Randomly pick K distinct training examples.
* Set $\mu_1, \mu_2, \ldots, \mu_K$ equal to these K examples.

The clustering we end up with depends on initialization. Bad initial centroids give bad local minima of J. The fix is to run K-means multiple times with different random inits. Compute J at the end of each run. Keep the run with the lowest J.

50 to 1000 random restarts is typical. More than that gives diminishing returns.



Random initialization

For $i = 1$ to 100 {

Randomly initialize K-means. ← k random examples

Run K-means. Get $c^{(1)}, \ldots, c^{(m)}, \mu_1, \mu_1, \ldots, \mu_k$ ←

Computer cost function (distortion)

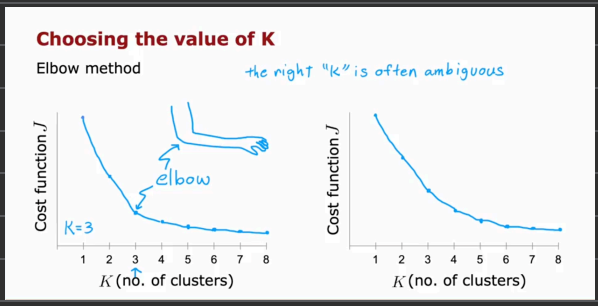
$J(c^{(1)}, \ldots, c^{(m)}, \mu_1, \mu_1, \ldots, \mu_k)$ ←

}

Pick set of clusters that gave lowest cost J

Choosing the number of clusters K

Elbow method: Plot J as a function of K. We look for a kink or elbow where J stops dropping sharply. The problem is that the curve is usually smooth with no clear elbow.



A wrong technique is choosing K to minimize J. J always decreases as K grows (more centroids means each point can sit closer to one). Following this rule always pick as large as possible which is bad.

The right approach is to pick K based on what clusters are for downstream.

Anomaly Detection.

Anomaly detection looks at unlabeled data of normal examples, learns what normal looks like, then flags new examples that look different.

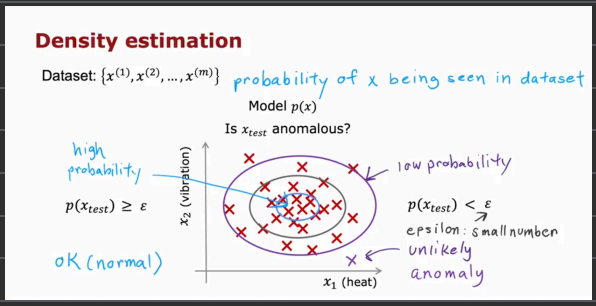
The technique used to do anomaly detection is called density estimation

Density estimation means building a model of the probability distribution that generated the training dataset. Given the dataset $\{x^{(1)}, \ldots, x^{(m)}\}$, you fit a model $p(x)$ that tells us, for any point x, how likely it is to come from the same distribution as the training dataset.

After fitting $p(x)$, anomaly detection becomes simple:

* Compute $p(x_{test})$ for a new example
* If $p(x_{test}) < \epsilon$, flag it as anomalous
* If $p(x_{test}) \geq \epsilon$, treat it as normal.

ϵ is a small threshold we have to choose.
Visually: regions where the training data is dense have high $p(x)$. Regions far from any training example have low $p(x)$. The anomalies sit in the low-density regions.



Gaussian (normal) distribution

To do density estimation, we need a model for $p(x)$. We use Gaussian here.

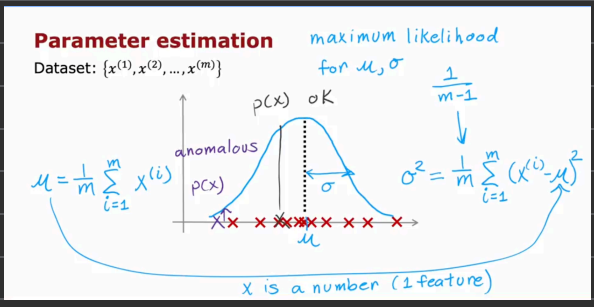
A random variable x follows a Gaussian distribution with mean μ and variance σ^2 if:

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

* Changing μ shifts the curve left or right.
* Smaller σ makes it taller or narrower.
* Larger σ makes it shorter and wider.
* Total area under curve is always 1.

Estimating the parameters from data (maximum likelihood):

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



Anomaly Detection Algorithm:

each training example has n features: $x^{(i)} \in R^n$

Model $p(x)$ by assuming the features are statistically independant and that each feature is Gaussian:

$$p(x) = p(x_1; \mu_1, \sigma_1^2) \cdot p(x_2; \mu_2, \sigma_2^2) \cdots p(x_n; \mu_n, \sigma_n^2) = \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2)$$

Algorithm:

* Choose n features x_j we think indicate anomalous examples
* Fit parameters $\mu_1, \ldots, \mu_n, \sigma_1^2, \ldots, \sigma_n^2$:

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

* Given a new example x, compute:

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

Flag as anomalous if $p(x) < \epsilon$

Intuition: $p(x)$ is large only where every x_j sits near its typical range. A single weird feature drags the whole product close to zero.

Anomaly detection algorithm

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

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

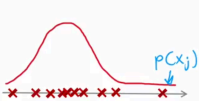
Vectorized formula

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

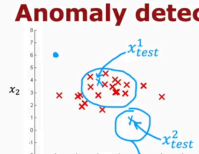
3. Given new example x, compute $p(x)$:

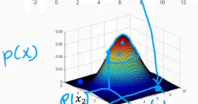
$$p(x) = \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2) = \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_j} exp\left(-\frac{(x_j - \mu_j)^2}{2\sigma_j^2}\right)$$

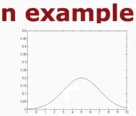
Anomaly if $p(x) < \varepsilon$



Anomaly detection example

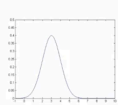






$\mu_1 = 5, \sigma_1 = 2$

$p(x_1; \mu_1, \sigma_1^2)$



$\mu_2 = 3, \sigma_2 = 1$

$p(x_2; \mu_2, \sigma_2^2)$

$\varepsilon = 0.02$

$p(x_{test}^{(1)}) = 0.0426 \longrightarrow$ "ok"

$p(x_{test}^{(2)}) = 0.0021 \longrightarrow$ anomaly

Developing and evaluating an anomaly detection system

To tune the system, we need need a real-number evaluation metric. This requires some anomalous examples (even if very few) to validate against.

Setup: assume we have some normal examples ($y=0$) and a small number of known anomalies ($y=1$)

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

Cross validation set: $(x^{(1)}_{cv}, y^{(1)}_{cv}), \dots, (x^{(m_{cv})}_{cv}, y^{(m_{cv})}_{cv})$

Test set: $(x^{(1)}_{test}, y^{(1)}_{test}), \dots, (x^{(m_{test})}_{test}, y^{(m_{test})}_{test})$

} include a few anomalous examples. Mostly normal.

Aircraft engines monitoring example

10000 good (normal) engines — 2 to 50
20 flawed engines (anomalous) — 2 — $y=1$

Training set: 6000 good engines ($y=0$) — train algorithm on training set

CV: 2000 good engines ($y=0$) — 10 anomalous ($y=1$)
use cross validation set — tune ε — tune x_j

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

Alternative: No test set — Use if very few labeled anomalous examples
Training set: 6000 good engines 2 — higher risk of overfitting
CV: 4000 good engines ($y=0$), 20 anomalous ($y=1$)
tune ε tune x_j

Algorithm evaluation

course 2 week 3
skewed datasets

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

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

10
2000

Possible evaluation metrics:
- True positive, false positive, false negative, true negative
- Precision/Recall
- F_1-score

Use cross validation set to choose parameter ε

Anomaly detection vs Supervised Learning

If we have some labled anomalies, why not just train a supervised classifier? it depends on the problem.

Anomaly detection is the rigat choice when:

* Very small number of positive (anomalous) examples, say 0 to 20.
* Large number of negative (normal) examples
* Many different "types" of anomalies, hard for an algoritam to learn what they look like from the positive alone.
* Future anomalies may look nothing like the ones we have seen so far.

Use cases: fraud detection, manufacturing defects (especially new types), monitoring machines in a data center.

Supervised learning is the rigat choice when:

* Large numbers of positive and negative examples.
* Enough positive for the algoritam to learn what they look like.
* Future positives are likely to look similar to training positives.

Use cases: Spam classification, weather prediction, disease classification

Choosing what features to use

Feature choice matters more in anomaly detection than in supervised learning because supervised learning can downweight bad features using few labels. Anomaly detection has no labels at training time, so it has to trust whatever features we give it.

We can plot a histogram of each feature. If it looks bell-shaped, we leave it. If it is skewed, we transform it until it looks more Gaussian. Common transformations:

* $\log(x)$
* $\log(x+c)$ for some small constant c (handles zeros)
* $x^{1/2}$
* $x^{1/3}$

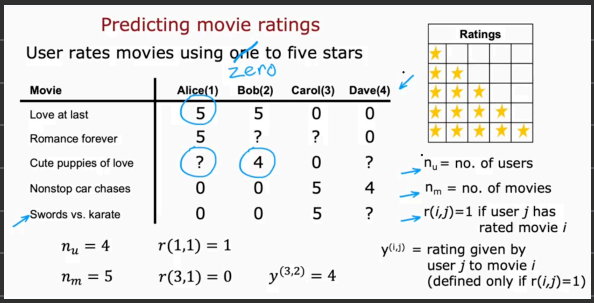
It is important to apply whatever transformation we apply to training set to CV and test set.

Recommender Systems

Collaborative filtering

Making recommendations

A recommender system predicts how a user would rate or respond to items they have not seen yet, then surface with the highest predicted score.



Predicting movie ratings

User rates movies using one to five stars

zero

Movie	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
Swords vs. karate	0	0	5	?

Ratings

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)

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

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

Using per-item features

We start with the easier case: assume someone has already given us features describing each items.

Movie example with two features:

* x_1 = how much romance
* x_2 = how much action

Each movie i has a feature vector $x^{(i)} \in \mathbb{R}^n$ (here $n=2$)
For each user j, learn parameters $w^{(j)} \in \mathbb{R}^n$ and $b^{(j)} \in \mathbb{R}$. The predicted rating user j gives movie i:

$$\hat{y}^{(i,j)} = w^{(j)} \cdot x^{(i)} + b^{(j)}$$

This is just linear regression, fit separately by user.

Cost function for a single user j (sum over movies j actually rated):

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

Cost function to learn parameters for all users at once:

$$J(w^{(1)}, \ldots, w^{(n_u)}, b^{(1)}, \ldots, b^{(n_u)}) = \frac{1}{2} \sum_{j=1}^{n_u} J(w^{(j)}, b^{(j)})$$

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

We then minimize this with gradient descent.
This works if we have item features. Usually we do not.

Note that $\frac{1}{m^{(j)}}$ normalization is dropped because it is just a constant.

Collaborative filtering algorithm

What if we don't have features for romance and action in the movie example? But if we already know users' parameters $w^{(j)}, b^{(j)}$ and their actual rating, we can work backwards to learn the features $x^{(i)}$

Given $w^{(1)}, \dots w^{(n_u)}$ and $b^{(1)}, \dots b^{(n_u)}$, we learn $x^{(i)}$ by minimizing:

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

For all movies at once

$$J(x^{(1)}, \dots, x^{(n_m)}) = \frac{1}{2} \sum_{i=1}^{n_m} J(x^{(i)})$$

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

Now the trick: we don't have w's and b's either and we don't have x's. But if we fix w,b we can solve for x and if we fix x we can solve for w,b. So we just learn both simultaneously by writing one combined cost function:

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

Minimizing with gradient descent over $w^{(j)}, b^{(j)}, x^{(i)}$ all together. The features $x^{(i)}$ are now learned parameters, not given inputs. We pick n (number of features) ourselves.

The name "collaborating filtering" comes from this, many users collaborating on rating movies effectively teach the algorithm what feature each movie has, which then helps predict ratings for other users.

$$w_k^{(j)} := w_k^{(j)} - \alpha \frac{\partial J}{\partial w_k^{(j)}}, \quad b^{(j)} := b^{(j)} - \alpha \frac{\partial J}{\partial b^{(j)}}, \quad x_k^{(i)} := x_k^{(i)} - \alpha \frac{\partial J}{\partial x_k^{(i)}}$$

Binary Labels (faves, likes and clicks)

Many real systems do not have 1 to 5 star ratings. They have binary signals. We can generalize the algoritam by switching from linear regression to logistic regression

Predicted probability that user j gives a positive label on item i:

$$\hat{y} = g(w^{(j)} \cdot x^{(i)} + b^{(j)}) \; , \; g(z) = \frac{1}{1+e^{-z}}$$

Loss for a single example uses binary cross entropy:

$$L(\hat{y}^{(i,j)}, y^{(i,j)}) = -y^{(i,j)} \log(\hat{y}^{(i,j)}) - (1 - y^{(i,j)}) \log(1 - \hat{y}^{(i,j)})$$

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

Curious...

What about softmax? for multiclass...

Mean normalization

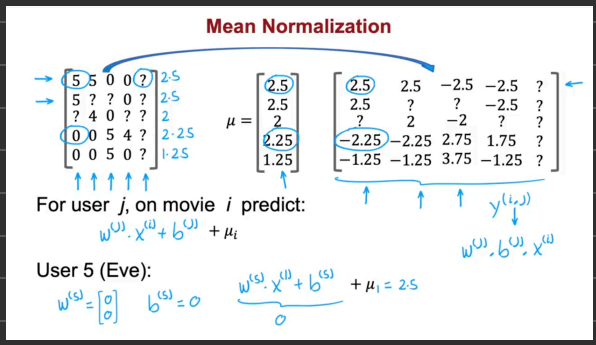
Introducing a new use who has rated nothing causes a problem. The cost function only has the regularization terms left for that user, which pushes all $w^{(i)}$ to zero. Predicted ratings becomes $0 + b^{(i)} = 0$ for every movie.

We fix this using mean normalization. We build the rating matrix Y where row i is movie i and column j is user j. For each movie i, we compute the average rating μ_i across users who actually rated it. Subtract μ_i from every observed rating in that row to get Y_{norm}.

We train collaboratively on Y_{norm}, learning $w^{(i)}, b^{(i)}, x^{(i)}$ as before. When predicting, we add the mean back:

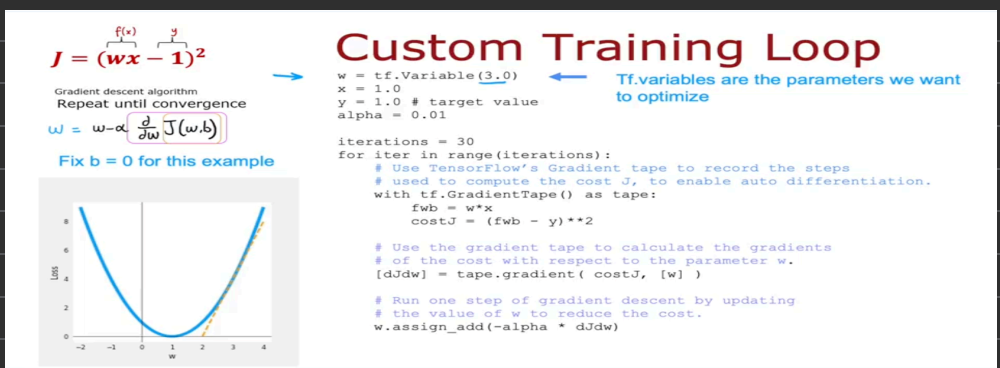
$$\hat{y}^{(i,j)} = w^{(i)} x^{(i)} + b^{(i)} + \mu_i$$

For a brand new user who rates no ratings, $w^{(i)} = 0$ and $b^{(i)} = 0$, so the prediction collapses to μ_i.



Tensorflow Implementation of Collaborative filtering.

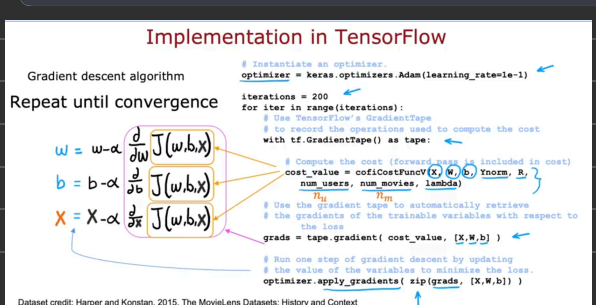
Collaborative filtering's gradients are non trivial to derive by hand. Tensorflow's auto-diff (also called auto-grad) computes them automatically




```
w = tf.Variable(...)
x = tf.Variable(...)
b = tf.Variable(...)

optimizer =
tf.keras.optimizers.Adam(learning_rate=1e-1)

for iter in range(iterations):
    with tf.GradientTape() as tape:
        cost = cofi_cost_func(X, W, b, Y, R, lambda_)
    grads = tape.gradient(cost, [X, W, b])
    optimizer.apply_gradients(zip(grads, [X, W, b]))
```



Finding related items

After training, the learned features $x^{(i)}$ characterizes each item. They are not human readable (the algorithm did not learn romance and action, it learned something), but items with similar x vectors play similar roles in user preferences.

To find similar items to i: compute the squared distance

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

for every other item k, we sort ascending, return the top few. That gives "more like this" recommendations.

Limitations of collaborative filtering

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

Two ways to build recommender:

* Collaborative filtering recommends item i to user j by looking at what other users with similar ratings to user j thought of item i. It uses the matrix only. It does not look at who the users are or what the items are.

* Content-based filtering recommends item i to user j based on features of user j and features of item i, asking whether they are good match. It uses side information about users and items, not just past ratings.

Notation stays the same: $r(i,j) = 1$ if user j rated item i, $y^{(i,j)}$ is their rating.

Content-based filtering needs feature vectors for both users and items.

User features $x_u^{(j)}$ for user j might include:
* Age (often binned or normalized)
* Gender (one-hot, or three-way to allow unknown)
* Country (one-hot)
* Movies watched
* Average rating given per genre

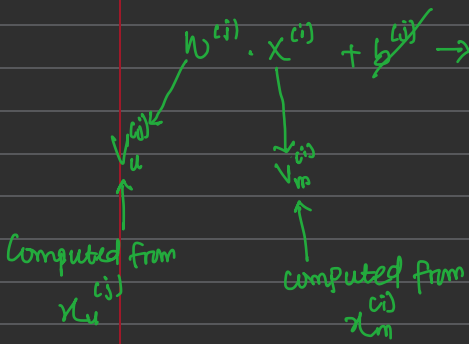
Movie features $x_m^{(i)}$ for movie i might include:
* Year
* Genre (one-hot or multi-hot, a movie can be multiple genres)
* Reviews / Critic scores
* Average rating across all users
* Cast

The two vectors need not have the same dimension.

The Model

Predict rating of user j on movie i as:

$w^{(j)} \cdot x^{(i)} + b^{(j)}$ → we remove $b^{(j)}$ as it doesn't hurt performance of content-based filtering at all



So we have:

$$\hat{y}^{(i,j)} = v_u^{(j)} \cdot v_m^{(i)}$$

where $v_u^{(j)}$ is a vector computed from $x_u^{(j)}$ and $v_m^{(i)}$ is a vector computed from $x_m^{(i)}$. Both v's must have the same dimension (otherwise we cannot dot-product them)

Deep learning for content based filtering

How do we compute v_u from x_u and v_m from x_m? We will use two neural networks. One for users, one for items.

User network: takes $x_u^{(j)}$ (say 1500 numbers), passes it through hidden layers, outputs $v_u^{(j)}$ (say 32 numbers)

Movie network: takes $x_m^{(i)}$ (say 50 numbers), passes it through hidden layers, outputs $v_m^{(i)}$ (say 32 numbers)

The two networks are independent (different weights, possibly different architectures), but their output layers must have the same number of units, because prediction is:

$$\hat{y}^{(i,j)} = v_u^{(j)} \cdot v_m^{(i)}$$

For binary labels, we wrap the dot product in a sigmoid:

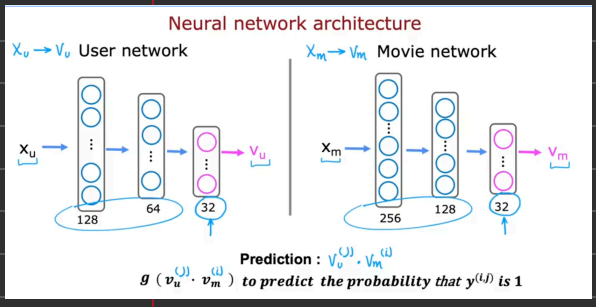
$$\hat{y}^{(i,j)} = g\left(V_u^{(j)} \cdot V_m^{(i)}\right), \quad g(z) = \frac{1}{1+e^{-z}}$$

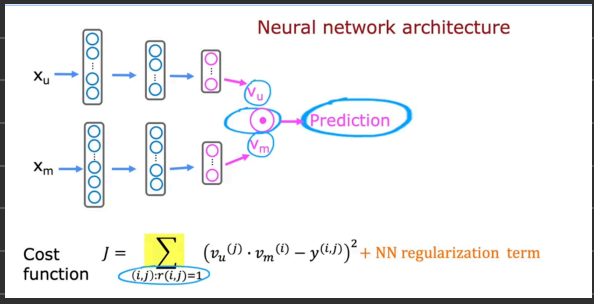
Training the model

Both networks are trained together as a single combined model.

The cost function for the regression version is:

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





Finding similar items

After training, $V_m^{(i)}$ describes the character of items i in the learned space. Items with similar V_m vectors are similar to each other.

To find items similar to item i:

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

for every other items k, sort ascending and return the top few.

Note that this can be precomputed. We do not need to wait until a user shows up to compute similar items. Build the list ahead of time and serve it from cache.

Recommending from a large catalogue.

For a service with millions of items (YouTube videos, products, news articles), running the full neural network on every item every time a user shows up is too slow. The standard fix is a two-stage architecture.

Retrieval: Generate a large list of plausible candidates, several hundred to a few thousand, using fast and approximate methods. For the movie example:

* For each of the last 10 movies the user watched, fetch the 10 most similar movies (precomputed using v_m distances)
* Top movies in the user's three most watched genres
* Top movies in the user's country

We combine these lists, deduplicate, remove items the user has already seen. End up with anywhere 100 to 1000 candidates.

Ranking: Take that shortlist, run the full content-based model on every candidate, sort by predicting rating $\hat{y}_{(i,j)}$, return the top k to the user.

The point of two stages is that retrieval is cheap and inexact, ranking is expensive and accurate. We only pay the ranking cost on a small set.

Trade-off in retrieval is that more candidate means better final recommendations but slower response. We tune the size on our held-out data, measuring offline metrics like recall@k or directly through A/B tests.

Tensorflow Implementation

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

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

# Inputs
input_user =
tf.keras.layers.Input(shape=(num_user_features,))
vu = user_NN(input_user)
vu = tf.linalg.l2_normalize(vu, axis=1)

input_item =
tf.keras.layers.Input(shape=(num_item_features,))
vm = item_NN(input_item)
vm = tf.linalg.l2_normalize(vm, axis=1)

# Dot product
output = tf.keras.layers.Dot(axes=1)([vu, vm])

model = tf.keras.Model([input_user, input_item],
output)
model.compile(optimizer=tf.keras.optimizers.Adam(0.
01), loss=tf.keras.losses.MeanSquaredError())
```

Principal Component Analysis

PCA allows us to take data with a lot of features say hundreds or thousands of features and reduce it to very small number of features, say 2 or 5 features. It is commonly used to visualize data to figure out what might be going on in the data.

PCA Algorithm

Step 1: Normalize features. PCA is sensitive to feature scales because it works with variance. Standardize: Subtract the mean of each feature, divide by its standard deviation. Each feature now has mean 0 and variance 1.

Step 2: Pick the first principal component. The first PC is the direction (unit vector) along which the projected data has maximum variance.

Step 3: Pick subsequent components. The second principal component is the direction of maximum variance among directions perpendicular to the first. The third is perpendicular to the first two. And so on. The PCs form an orthogonal basis.

Step 4: Project data. To reduce n features to k features, keep the first k principal components and project each data point onto them. The projection of point x onto a unit vector u is:

$$z = x \cdot u$$

Step 5 (optional): Reconstruct. We can approximately recover the original point from its low dimensional representation.

$$x_{approx} = z \cdot u$$

PCA in code

```
from sklearn.decomposition import PCA

# Step 1: optionally normalize features first
# (StandardScaler if features have very different
scales)

# Step 2: fit
pca = PCA(n_components=2)
pca.fit(X)

# Step 3: how much variance does each component
explain?
pca.explained_variance_ratio_
# e.g. array([0.992, 0.008]) means PC1 explains
99.2% of variance

# Step 4: project to lower dimension
X_reduced = pca.transform(X)

# Step 5: reconstruct back to original space (lossy)
X_reconstructed = pca.inverse_transform(X_reduced)
```

PCA is mostly used for data visualization nowadays.
Used historically, less so now!

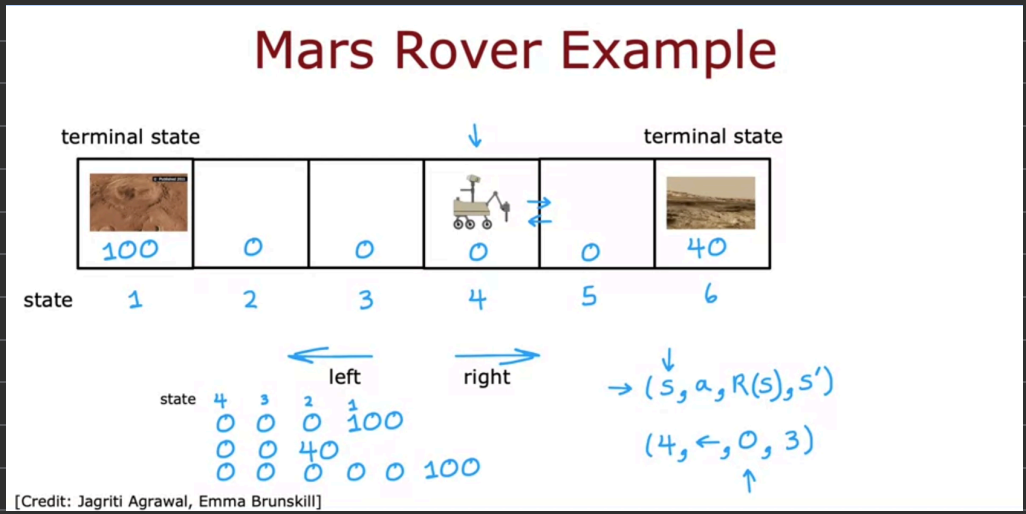
* Data compression
* Speeding up supervised learning: Reduce features before training a model. Largely superseded by modern algorithms that handle high-dimensional data fine! And by feature engineering, regularization, deep learning that learns its own representations)

Reinforcement Learning.

Reinforcement learning (RL) trains an agent to take actions in an environment. The agent does not get told what action is correct. It gets a reward signal instead: a number that says how well things are going.

We can compare this with supervised learning which needs (x, y) pairs where y is the correct output. For tasks like flying helicopters or running a mars rover, nobody can write down what the correct action at this state is for every state. But it is easy to score the outcome. Helicopter flying smoothly = good (positive reward like 10), helicopter crashed = bad (large negative reward). RL takes that score and figures actions on its own.

Mars Rover example



The Return in RL

The return is the sum of rewards the agent collects from a given starting point, but with future rewards discounted.

$$G = R_1 + \gamma R_2 + \gamma^2 R_3 + \gamma^3 R_4 + \cdots$$

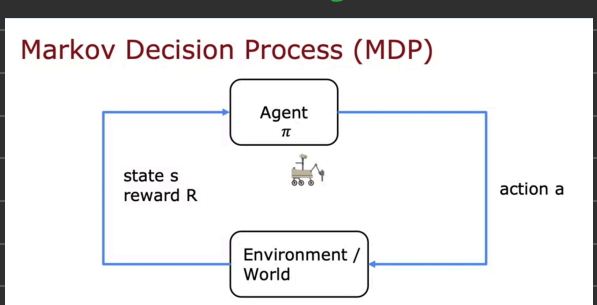
Where R_t is the reward received at step t and $\gamma \in [0, 1)$ is the discount factor. Reasons for the discount is that the infinite sum converges and a reward is worth more than the same reward in 100 steps. In financial application, discount factor is the interest rate.

Making decisions: Policies in RL

A policy π is a function that says, given a state s, what actions to take: $\pi(s) = a$.

So the goal of RL is to find a policy π that maximizes expected return.

This framework is called a Markov Decision Process (MDP). Markov because the next state depends only on the current state and the action, not the history.

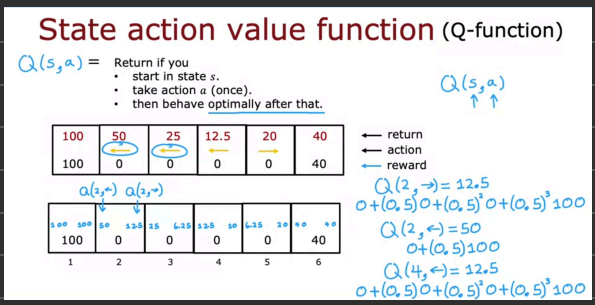


State-action Value Function (Q-function)

The state-action value function, written Q(s,a), is the return we get if we:

* start in state s
* Take action a once
* Then behave optimally from then on.

Sometimes called the Q-function or optimal Q-function (Q*)



If we know Q(s,a) for every state and action, we would know the optimal policy. We just pick the highest Q

$$\pi(s) = \arg\max_a Q(s,a)$$

Large γ makes it patient and wants to take more steps to get the best reward while small γ makes it impatient and wants to take fewer steps to get to the terminal state even if it gets less reward.

Bellman Equation

The bellman equation expresses $Q(s,a)$ recursively. It says: the value of taking an action a in state s equals the immediate reward plus the discounted best value from the next step.

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

Where:

* s is the current state and a is the current action.
* $R(s)$ is the reward in state s
* s' is the state we land in after taking action a
* a' is the action we take in state s'
* $\max_{a'} Q(s',a')$ is the best value achievable from s'

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

Q has a recursive structure that lets us compute it iteratively or learn it from data.

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

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

$$Q(s,a) = R_1 + \gamma R_2 + \gamma^2 R_3 + \gamma^3 R_4 + \cdots$$

Random (stochastic) environment.

Real environments are noisy, so many external factors will affect our simplified RL model (eg 90% of time go left, 10% go right).

We can generalize the return to its expected value because outcomes are random:

$$E[G] = E[R_1 + \gamma R_2 + \gamma^2 R_3 + \cdots] \longrightarrow \text{average value.}$$

The Bellman equation becomes:

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

where the expectation is still to maximize expected return, given that it can control noise.

Continous State Spaces

Our Mars rover example has 6 discrete states, real problems have continous states.

A self driving cars state vector might be:

$$S = (x, y, \theta, \dot{x}, \dot{y}, \dot{\theta})$$

x and y position, heading θ, and their three velocities. Six continous numbers.

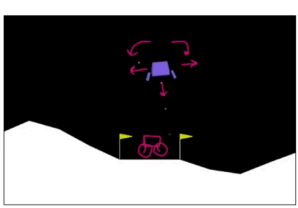
Lunar lander has 8-dimensional continous state:

$$S = (x, y, \dot{x}, \dot{y}, \theta, \dot{\theta}, l, r)$$

x and y position, x and y velocity, angle, angular velocity, and two binary flags for whether left and right legs are touching the ground. We cannot store $Q(s,a)$ as a table of state-action pairs when s is continous. There are infinitely many steps. We approximate $Q(s,a)$ with a neural network instead.

Lunar Lander

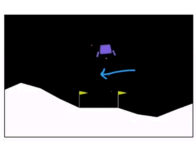
Lunar Lander



actions:
- do nothing
- left thruster
- main thruster
- right thruster

$S = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \\ \theta \\ \dot{\theta} \\ l \\ r \end{bmatrix}$ 0 or 1

Reward Function

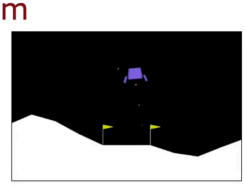


- Getting to landing pad: 100 – 140
- Additional reward for moving toward/away from pad.
- Crash: -100
- Soft landing: +100
- Leg grounded: +10
- Fire main engine: -0.3
- Fire side thruster: -0.03

Lunar Lander Problem

Learn a policy π that, given

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



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

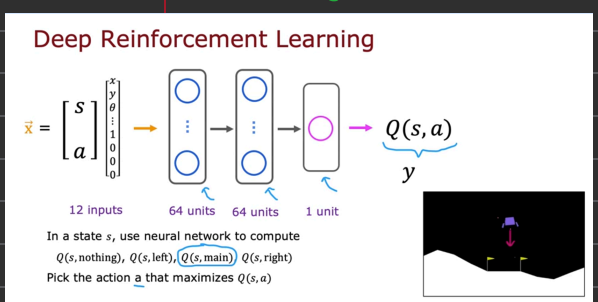
$\gamma = 0.985$

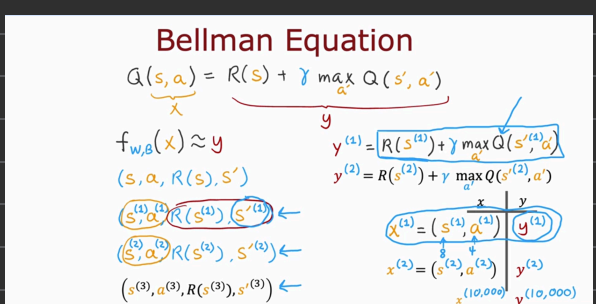
Deep Q-Network (DQN)

We want to train a neural network to compute or approximate the state-action value function $Q(s,a)$ and that in turn will let us pick good actions.

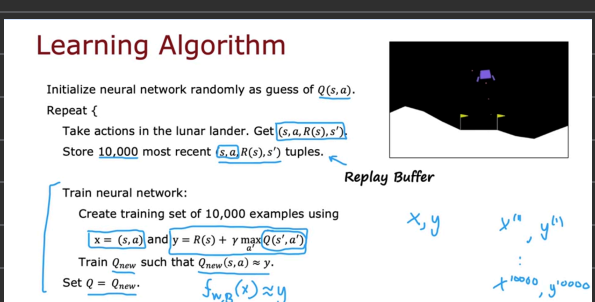
Input: State s (8 numbers for lunar lander) plus action a (one-hot encoded as 4 numbers), so input dimension is 12.
Output: scalar predicting $Q(s,a)$.
Architecture:

* Input layer: 12 units
* Hidden layer 1: 64 units, ReLU
* Hidden layer 2: 64 units, ReLU
* Output layer: 1 unit, linear (no activation).



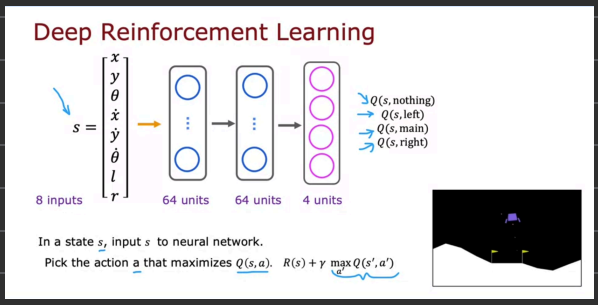


Training Q-network.
Training data is built by interacting with the environment.



Algorithm refinement: Improved neural network architecture.

For the previous architecture, we have to carry out inference for each of the four actions, this is inefficient as we have to do inference four times for each actions, We can just train a single neural network to output all four actions at once.



ϵ-greedy policy

How does the agent decide which action to take while collecting experience?

* Pure exploitation: always pick $\arg\max_a Q(s,a)$, the problem is that if the network is wrong about some action being bad, the agent never tries it and never learns.

* Pure exploration: random actions. The agent never uses what it has learned.

ϵ-greedy balances the two:
* With probability $1-\epsilon$, pick $\arg\max_a Q(s,a)$ (greedy/exploit)
* With probability ϵ, pick a random action (explore)

We typically start ϵ high (say 1.0) and decay it over training to a small value (say 0.01). Early on, we mostly explore. Later, mostly exploit.

RL algorithms are more finicky than supervised learning. Hyperparameter tuning matters more and a bad ϵ schedule or learning rate can kill training.

Mini-batch and soft update.

We can use two more refinements:

* Mini-batch gradient descent: Instead of computing the gradient over all training data, we use a small random subset (say 64 examples) per step. Faster per step, noisier per step, usually converges faster overall. This is standard for any neural network training.

* Soft update: Naively, the target $y = R + \gamma \max_{a'} Q(s', a')$ use the same network that is being trained, which makes the targets move every step. This causes instability.
The fix is to use two networks. A Q-network that gets trained and a target Q-network used to compute the targets y. The target network is updated slowly:

$$W_{target} := \tau W + [1 - \tau] W_{target}.$$

With small τ (e.g. 0.001). The target network drifts slowly toward the Q-network, stabilizing the target. Without this, training often diverges.

Soft Update

Set $Q = Q_{new}$. ← $Q(s, a)$

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

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

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

State of RL

RL is great at simulated environments, games, robotics in controlled settings.

Limitations of Reinforcement Learning

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