Introduction to Reinforcement Learning

Aditya Das Analytics Club, IIT-M
Nihal John George Computer Vision Club, IIT-M

Logistics

- All queries in Zoom public chat, one of us will respond in due time. Requesting all participants to **mute their mics and switch off video**.
- Short doubt clearing sessions after each topic if needed, feel free to unmute and ask that time.
- For any software install issues, please consult our document (available on Shaastra.org -> Workshops -> Reinforcement Learning).
- We will walk through all code and gameplay on the presentation
- Please tell on chat if presentation is not visible/audible.
- Timings are 9-12, 1-4

Content Outline

Forenoon Session

1. Introduction, Clearing the Buzzwords
2. Sequential Decision Making - How to choose the best action?, Terminology
3. Markov Decision Process (MDP)
4. Value Functions, Bellman Equation
5. Q-Learning + Demo

Content Outline

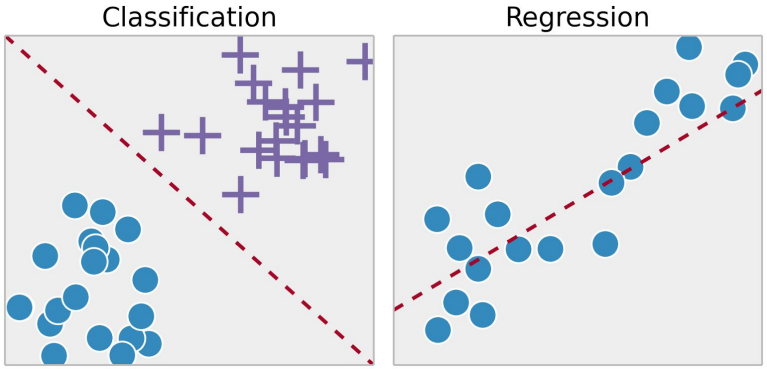
Forenoon Session

1. Introduction, Clearing the Buzzwords
2. Sequential Decision Making - How to choose the best action?, Terminology
3. Markov Decision Process (MDP)
4. Value Functions, Bellman Equation
5. Q-Learning + Demo

What are the different ways to **learn**?

Supervised Learning - Given input data and target value/class, have to predict the target for unseen input

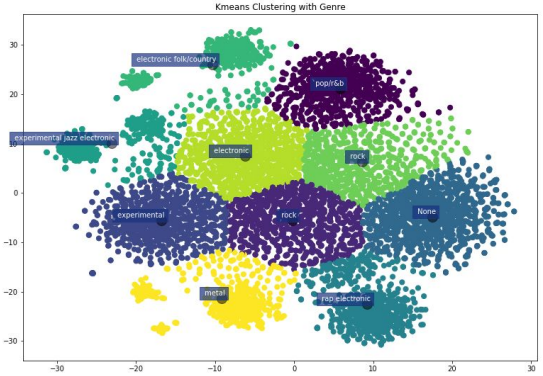
eg. Tumour classification given many scans with labelled tumours, House Price prediction given previous year trends)

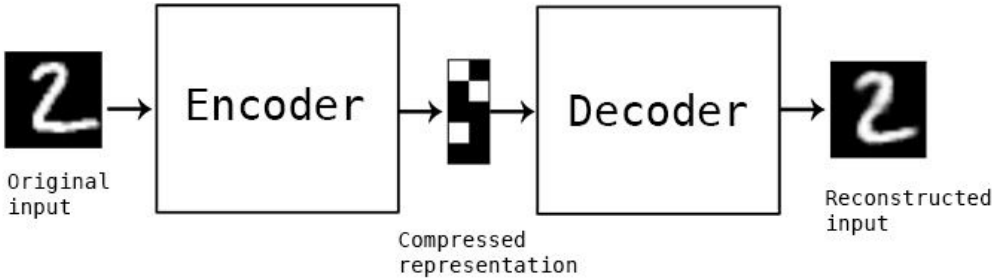


What are the different ways to **learn**? (cont.)

Unsupervised Learning - Given input data but no targets, have to find patterns in the input data

eg. Clustering music into genres, Compressing images using autoencoders



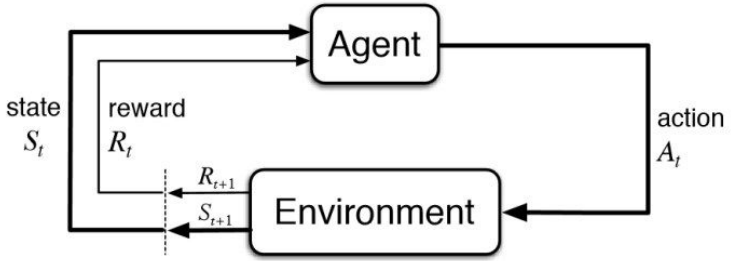


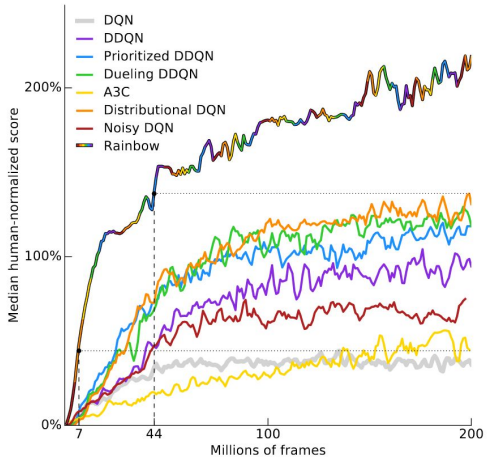
What are the different ways to **learn**? (cont.)

Reinforcement Learning - Model learns to take the correct actions to maximise reward, given the state of the environment.

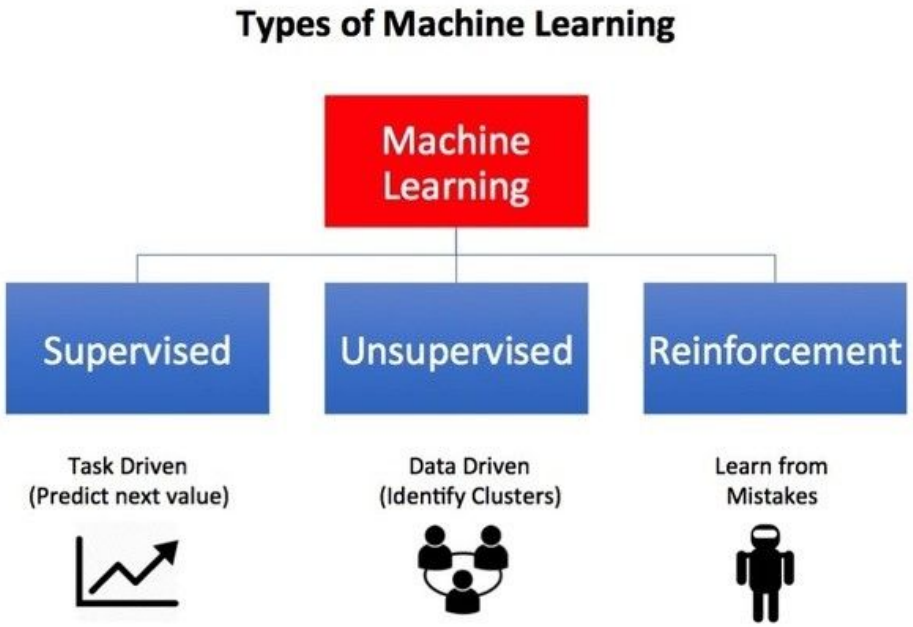
Input and a score is given, model takes action to increase that score. No particular score to achieve, just higher the score, the better.

eg. Optimally playing Poker, Chess, Improving network load balancers





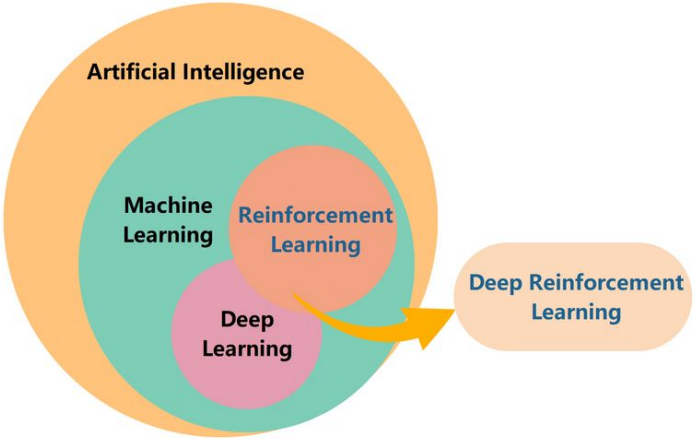
Side by Side Comparison



RL vs DL vs ML vs AI

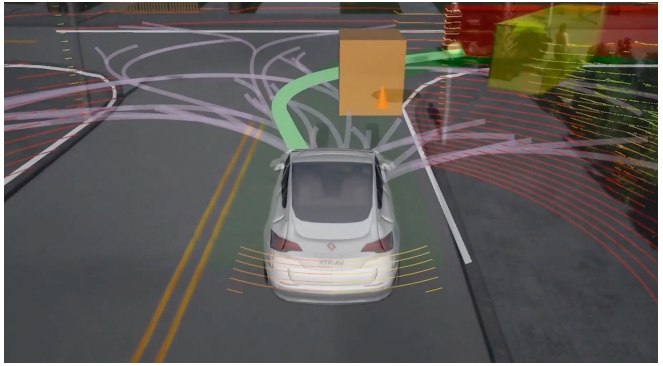
RL is a form of machine learning, which in turn is a subset of AI. This stems from the layman description of ML - a machine which improves performance on some task over time.

Deep learning involves the use of neural networks, which can be useful for RL in approximating value functions. This is called **Deep RL**

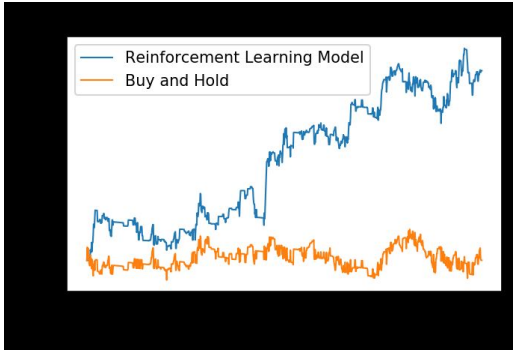


Real World Uses of RL (Other than fancy games)

Robotics and Self Driving Cars - Control, trajectory planning



Quantitative Trading - Decide portfolio, buy and sell actions, maximise earnings



Recommender Systems and Advertising - Using activity history, recommend options to user, maximise engagement

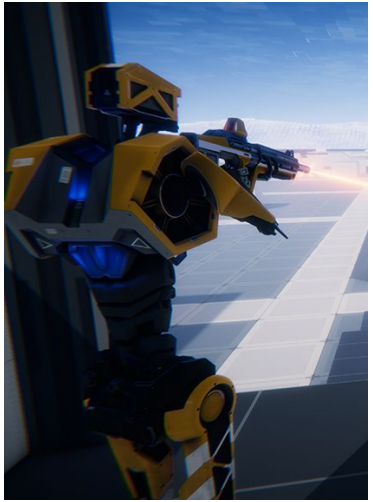
Content Outline

Forenoon Session

1. Introduction, Clearing the Buzzwords
2. Sequential Decision Making - How to choose the best action?, Terminology
3. Markov Decision Process (MDP)
4. Value Functions, Bellman Equation
5. Q-Learning + Demo

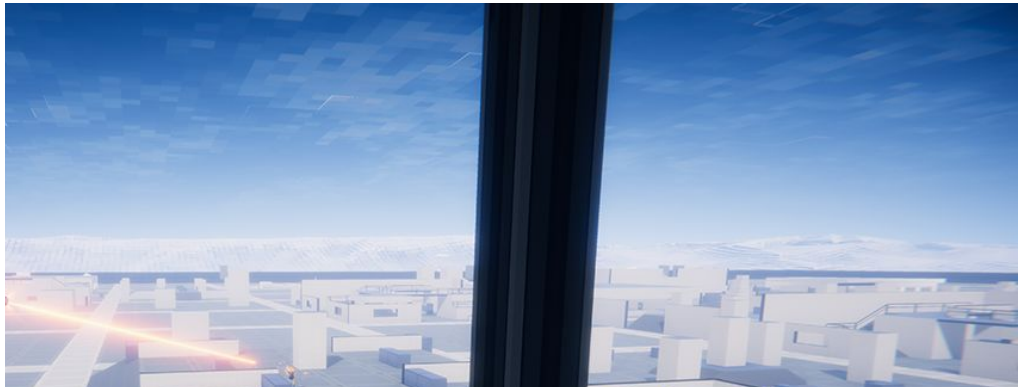
An Example Situation

- Consider a shooting game (like Counter-Strike, Call Of Duty)
- We can break down the game into the following parts



Players

- Objective - Maximise score (by shooting all opponents and not dying)
- Take actions to achieve goal
- Are bounded by rules of their environment



Environment

- Consists of the game area and its points system
- Enforce rules on player actions (movement and attack)
- Provides the objective to players
- Records the actions of players at each timestep, computes new positions and reward for each agents.

Terminology

- **Environment** - (*Game arena and points allocation code*) - The world in which the agent operates. It defines what actions an agent can take and how the situation of the agent evolves over time.

- **Agent** - (*Players*) - The entity which interacts with the environment in order to achieve its objective.

- **Action** - (*Shoot/Buy/Hide*) - The methods by which agents interact with the environment.

- **Reward** - (*Points*) Feedback given by the environment to an agent when the agent takes an action.

- **Objective** - (*Win the game*) - Goal pursued by all agents => Maximise reward.

Devising a Strategy - What are States?

- Some things you'll keep in mind
  - Where am I currently located in the battlefield?
  - Where is everyone else (teammates and opponents) on the battlefield?
  - How much ammunition (bullets/grenades etc.) do I have left?
  - What is my health condition?
  - *For now, let's settle with this information*

- **Observation** - Information of the environment given to the agent.

- **State** - Encompasses the current situation of the agent.

- For certain RL problems, state is completely defined by the observations (for eg, Tic Tac Toe, Chess).

- For others, state may not be completely defined by the observations (eg. Poker - can't observe other players cards). We will not delve into the formalities of these problems.

- Taking an action causes the agent to transition between states (sometimes back to the same state).

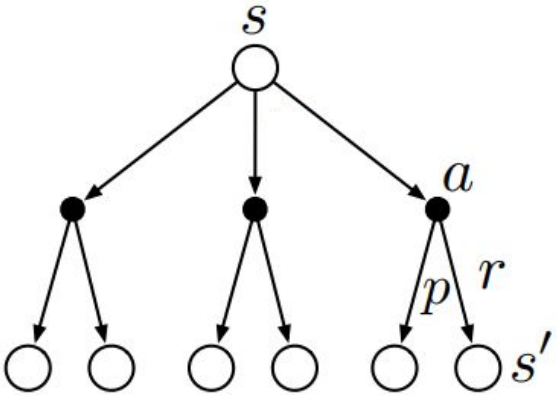
Content Outline

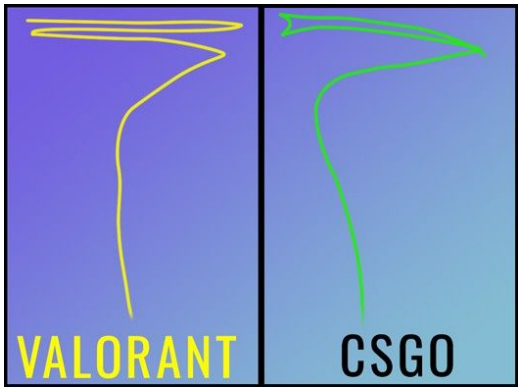
Forenoon Session

1. Introduction, Clearing the Buzzwords
2. Sequential Decision Making - How to choose the best action?, Terminology
3. Markov Decision Process (MDP)
4. Value Functions, Bellman Equation
5. Q-Learning + Demo

Markov Decision Process (MDP)

- Mathematical construct for defining RL problems.
- Given an environment and an agent,
  - Environment is in some current state *s*
  - Agent receives state information and decides to take an action *a*
  - Action causes environment to transition to state *s'*
  - Reward *r* is given to agent during this transition
- But the transition may be random!
  - May end up at $s' = s_2$ instead of $s' = s_1$
  - Different probabilities of transitioning to different states
  - Rewards also may be random for the same state transition
- In the shooter game, the gun has some recoil (upward tilt due to bullet momentum) which can be modelled using probability distributions. So an opponent may or may not be hit even for the same action.



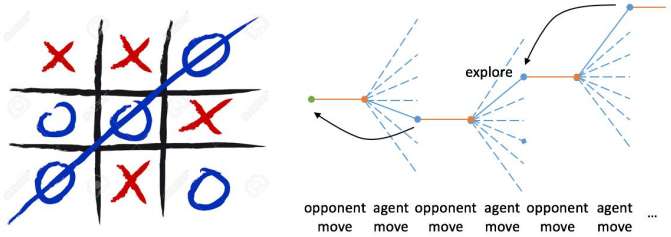


MDPs (cont.)

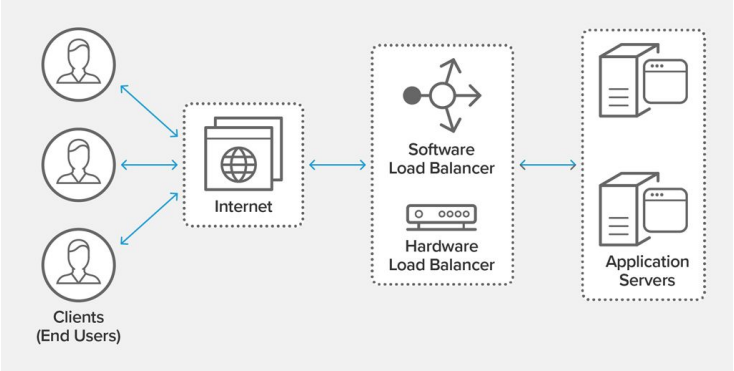
- This process of transitioning is called Markov Decision Process (MDP)
- Probability function *P(s', r | s, a)* read as “Probability of getting reward *r* and going to state *s'* given the current state *s* and action *a* taken in this state”.
- **Markov Property** - The dynamics of the environment depend only on *P(s', r | s, a)* - no dependence on states before *s*
- In other words, the state *s* encompasses all information about the present and past situation of the environment, after undergoing the actions taken by the agent

Examples

Tic-Tac-Toe: Define the state as which cells are occupied by which sign. Actions are drawing an X for the human, the environment responds randomly by putting an O somewhere. No reward is given during the game, except +1 when the human wins, or -1 if the human loses, or 0 if it's a draw.



Load balancing: Given a queue of requests from users and a number of servers (may not be identical), assign requests to servers. States are the occupancy of each server along with the current queue length. Rewards are -1 for each timestep, -100 for failed response, +10 for successful response



Try Making Your Own MDP!

Using the rules of the MDP, try to define your own real-life MDP and put it on the Zoom chat

For reference, the rules are -

1. Define the environment, states, actions and rewards
2. Explain why the Markov property holds for your example

We will stick to finite MDPs (finite number of states and actions). In case of continuous variables like position, we “discretize” to to minimum quantity to make it finite.

Content Outline

Forenoon Session

1. Introduction, Clearing the Buzzwords
2. Sequential Decision Making - How to choose the best action?, Terminology
3. Markov Decision Process (MDP)
4. Value Functions, Bellman Equation
5. Q-Learning + Demo

Policy

Strategy of the agent is called policy

Formally, the policy is defined as a mapping of states to actions. It is represented with $\pi(s) : S \rightarrow A$

The reinforcement learning problem is to achieve the optimal policy for a given environment

Finding this policy may be difficult to do directly

Eg. Tic Tac Toe's optimal policy is already found - using exhaustive search of all possible games

How do we quantify the 'goodness' of a state or an action?

Expected Rewards

Define G_t as the total reward obtained for all time instants t+1, t+2, … , T

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T$$

T is considered as the final timestep

One example: points you get at each time step of a shooting game

But for unending tasks, T tends to infinity, and G_t may also blow up

So we use a different formulation, known as the **discounted** return

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Where $0 \leq \gamma \leq 1$ is known as the **discount rate**

Quick Task

Suppose I give you an unfair coin P(heads) = 0.4, P(tails) = 0.6

Suppose to toss the coin you have to pay 0.5$. Also assume that you will toss till you run out of money. If I give you $1 for each time you get heads, what is the expected amount of money you get?

Here your choice of actions are - toss or do nothing. I enforced a policy which always chooses to toss till the wallet is empty.

Let us find expected R_t
$E[R_t]$= sum(profit*Probability of getting that profit) = 0.5*0.4 + (-0.5)*0.6 = -0.1 for all t due to independence of tosses

Now G_t = sum(R_i) for all i≥t
= -0.1 -0.1 -0.1 .... = -∞
But if the wallet is limited, you eventually run out of money
So G_t = wallet money at time t

Discounting

Notice that G_t has now become a weighted sum of rewards. If $\gamma < 1$, rewards closer in the future receive higher weight than the ones farther away.

In a way, we are forcing the agent to consider short term gains more than long term ones. This is helpful in very long MDPs where discovering the values of states and actions is tedious, and a high gamma may not lead our model to the correct policy in reasonable time

If $\gamma = 0$, then the agent only considers the next reward.

If $\gamma = 1$, then all the rewards are given equal weight, so the agent has become far-sighted.

This γ is a hyperparameter (set by us according to our liking), and can be tuned for different behaviours

Value Functions - How Good Are States and Actions?

The expected return $E[G_t]$ of a state (or) state-action pair. Higher expected reward means higher value, and so we should choose an action that is most valuable

Value function - The value of a state under a policy π is denoted as $V_\pi(s) : S \rightarrow R$

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right], \text{ for all } s \in \mathcal{S}$$

Action Value function - The value of a state-action pair under a policy π is denoted as $Q_\pi(s,a) : S \times A \rightarrow R$

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right]$$

The Gridworld

For illustrating the process of determining value functions, we are going to use an environment called Gridworld

In this environment, the states are the cells of this table. The actions are moving up, down, left or right.

The starting state is Agent. The two terminal states are End (Winning) and Pit (Losing).

The reward for falling into the pit is -5, for reaching the end is +10 and for every other step is -1

Now how do we decide a policy to move?

Value Assignment

End	1	0.9
0.5	0.7	0.8
Pit	0.3	Agent

In the beginning, let's assign values to the states according to our understanding

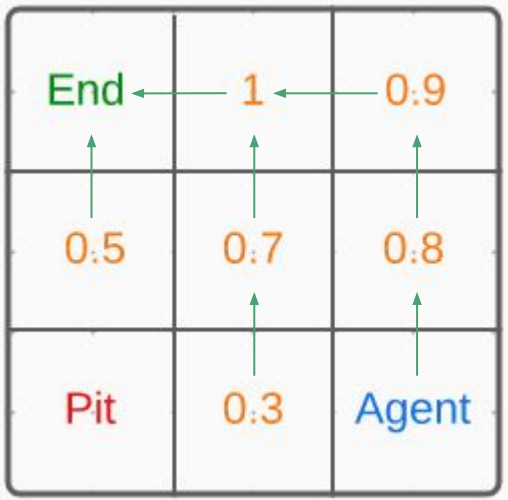
Given these values, how do we devise a strategy for moves?

Notice that you would always want to maximise your return.

Since value V(s) denotes the expected return after going to state s, we can choose the direction of the most valuable neighbour.

Now let's place the arrows indicating the strategy

Policy



Here is the policy. Notice that we have always chosen the best neighbour or the End state for each cell

Mathematically, given all the values, we can construct the policy π(a|s) as

$\pi(a|s) = 1$ if $a = \text{argmax}_{(a)} V(s')$ [i.e, the best action]
$= 0$ otherwise

So we set a probability of 1 to choose the best action, so we see only one outgoing arrow from each non-terminal state

Determining the Value Function

Now we have a way of making the policy once we have a value function

But our previous value function was estimated by us.

How do we get the agent to find the true value function, in terms of expected return G_t?

For this, we use the **Bellman Equation.** It is derived by noticing a recursive expansion. Suppose you do an action in state s that leads to state s',

$V(s) = E[G_t|S_t=s] = E[R_{t+1} + \gamma G_{t+1}| S_t=s]$

($G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots$

$G_t = R_{t+1} + \gamma (R_{t+2} + \gamma R_{t+3} + \ldots)$

$V(s) = E[R_{t+1}| S_t=s] + \gamma E[G_{t+1}| S_t=s]$

$V(s) = E_a[R(s,a)] + \gamma V(s')$ **[The Bellman Equation for V]**

We Are Close to The Solution! But ....

Solving a system of |S| equations can run into tens of thousands of equations for complex environments

Equation solving takes time of the order $|S|^3$, which can take days or months to solve

We need a way of at least getting close to the optimal value functions with relatively few iterations

For this, we explore a method **Q-Learning**

Content Outline

Forenoon Session

1. Introduction, Clearing the Buzzwords
2. Sequential Decision Making - How to choose the best action?, Terminology
3. Markov Decision Process (MDP)
4. Value Functions, Bellman Equation
5. Q-Learning + Demo

Q Learning

For Q learning, we use the Q function

Recall that Q(s,a) gives the value of taking action *a* at state *s*

This is simpler to use than the V function, since computing the policy is rather simple --- For a particular state, check Q of all possible actions, and choose maximum

Previously, we had to compute the policy using V. So we need to know which action takes you to which state. For Gridworld, it is simple --- UP takes you to the upper neighbour etc.

But for complex environments, the same action can take us to different states.

So we choose not to bother with V, instead Q will directly tell us what to do at a state s.

Q-Learning

The Bellman Equation for Q has a similar form as V

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

However, we want to iteratively improve our Q(s,a) estimate --- our estimate should become closer to satisfying the Bellman Equation --- our error wrt Bellman RHS should reduce.

We will use a sampling version of the Bellman equation to avoid the expectation function.
The expectations above will be computed using many samples, which will eventually converge to the mean.

$Q(s,a) = R(s,a) + \gamma \max_{a'} Q(s',a')$ **[Sampling version of Bellman]**

Temporal Difference (TD) error

Now when we are trying to learn from the game, like most ML algorithms, we need a cost function or an error that we need to minimize. In Q learning this is the TD error.

Suppose you perform an action a in a state s and reach state s' with a reward r. From the previous equation, you compute Q_calc = r + $\max_{a'}$(s',a'). This is the target value. According to what you have learnt so far, you know the predicted value as Q(s,a).

The difference between these two is called the TD error, which we are trying to minimize

TDerror = Q_calc - Q(s,a)

Exploration vs. Exploitation

When we are training our agent, we would want it to explore all possible states and figure out which are the best, and with enough exploration it can build a very accurate q table

So what is a good exploration strategy? Should we always just choose random actions? The problem with that is you have a very low chance of ever moving forward in the game. We do of course want to move forward in the game, but we want to explore along the way

This method is called **epsilon-greedy** strategy. We fix a base probability to explore which is called epsilon. Suppose epsilon is 0.2, so whenever we have to choose an action, there is a 20% chance of choosing a random action (**exploration**) and an 80% chance of choosing the best action (**exploitation**) as dictated by the qtable.

Q-Learning Algorithm

1. Start the game with a Q-table (table with S rows and A columns) filled with 0s
2. Repeat the following till TD < threshold or for a fixed number of episodes:

    a. With probability ϵ (epsilon), sample a random action, with 1-ϵ, choose the best action using Q(s,a)

    b. Send action to env, get reward R(s,a) and new state s'

    c. $Q_{target}(s,a) \leftarrow R(s,a) + \gamma \max_{a'} Q(s',a')$ [Target according to Bellman Sample]

    d. $TD \leftarrow Q_{target}(s,a) - Q(s,a)$ [TD between Bellman and current Q]

    e. $Q(s,a) \leftarrow Q(s,a) + \alpha * TD$ [Update current Q by reducing the error]

Time for a Demo!

Clone/Download Zip of the Github Repo

Link: https://github.com/AdityaDas-IITM/Shaastra-2021

We will be doing a ‘Code-Along’, which means I will share my screen showing myself typing the code and explaining the lines, and the participants can type the code on their computer

Google Colab will be used since platform issues are minimal

Visit https://colab.research.google.com

Problems with Q Learning

The major limitation of Q-learning is the memory requirement.

We have simplified the state space of the snake game such that there will never be more that 256 states and the q table will never be bigger than 256x4.

But what if that was not possible? Imagine we are playing a shooting game where your state is the screen you can see. It's not possible to make an easy state representation in this case and so you will have millions of states

Can you make such a big Q-Table and expect your laptop to survive?

So what do we do now?

You will find out in the next session

That marks the end of the forenoon session, and the basics of Reinforcement Learning!

The afternoon session deals with Deep RL, where we use neural networks to solve the RL problem

Do provide feedback on the session through Whatsapp/Zoom Private Chat

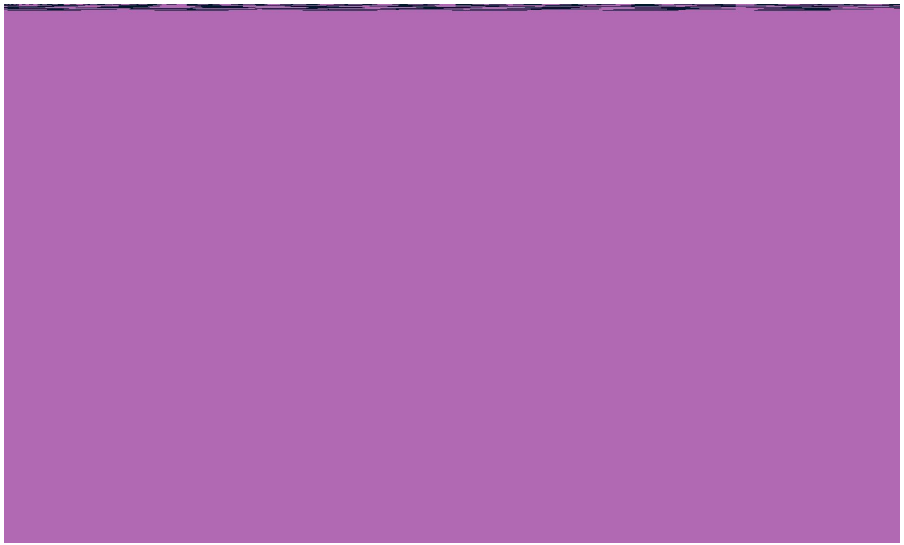
Deep Reinforcement Learning

Aditya Das Analytics Club, IIT-M
Nihal John George Computer Vision Club, IIT-M

Overview of Deep learning

The basic task of deep learning is to make an approximate function that would map some given inputs to some outputs as accurately as possible.

For this task we use a neural network, which is basically a bunch of matrix multiplication operations that move the data forward and give us the output

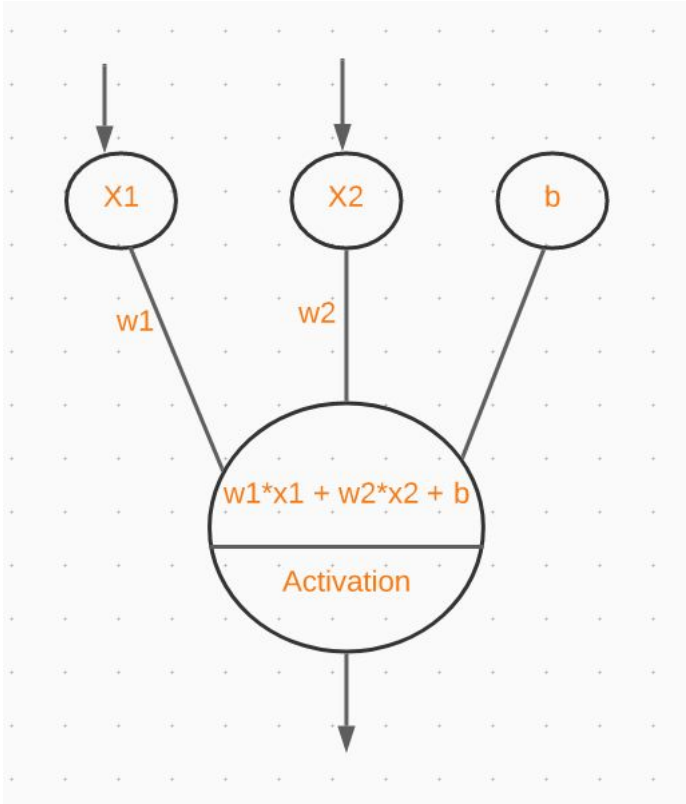


Forward Propagation

The data moves forward through a process called forward propagation.

The neural network has some parameters called weights and biases. The input to the model is multiplied with the weights, the biases are added and the sum is activated.

This transforms the input data to better and better forms.

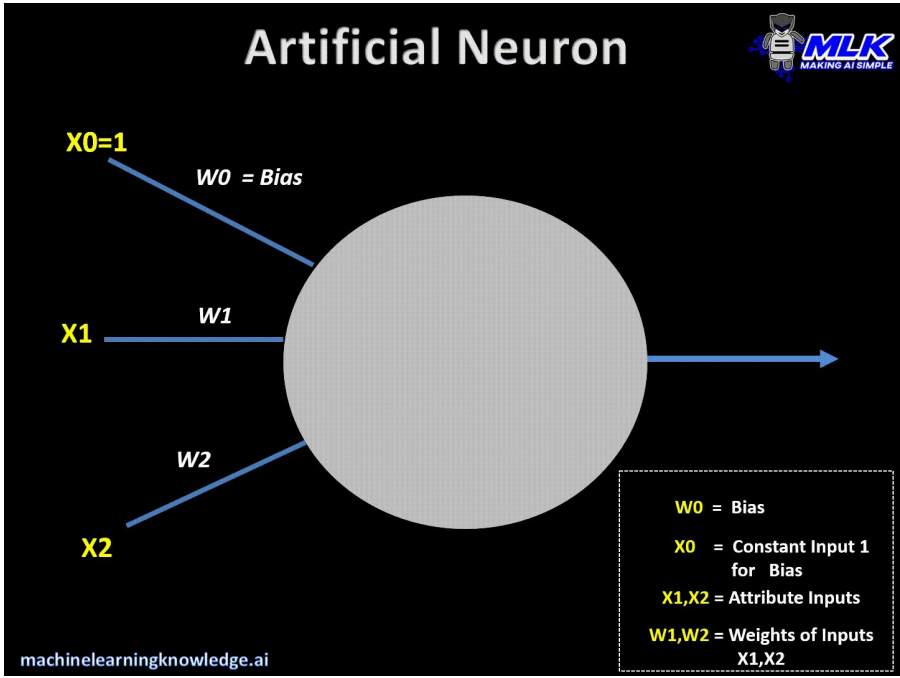


Activation Function

Neural networks are used to find complex nonlinear functions. For it to be able to do so, every neuron must be activated with some function.

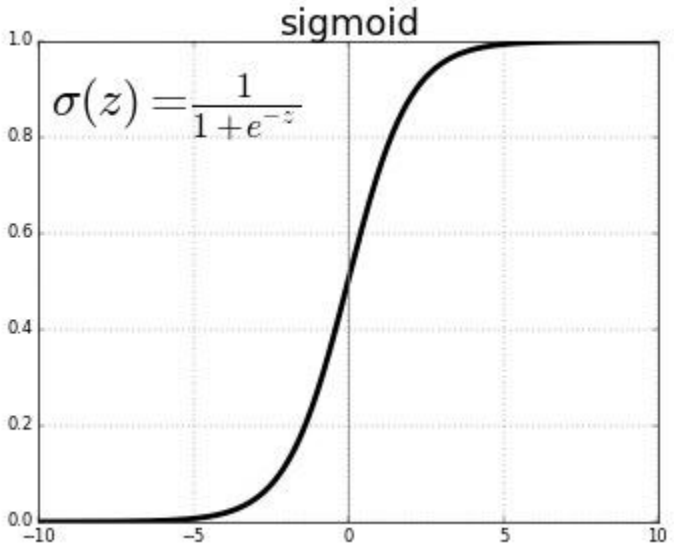
Some common activation functions are

- ReLU
- Sigmoid
- Linear



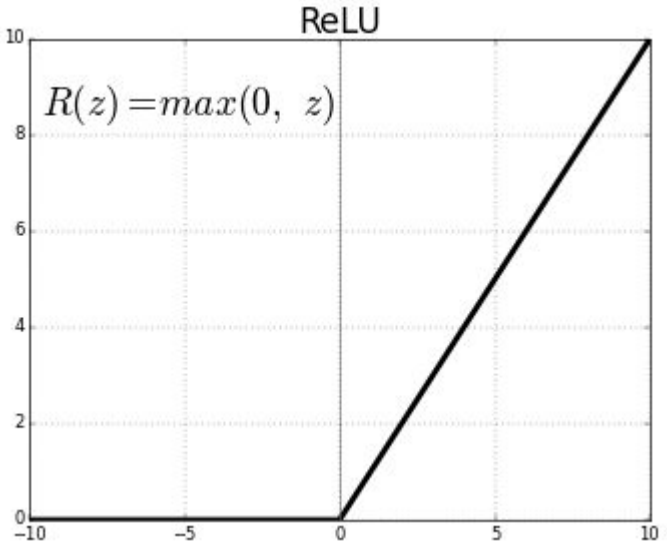
Activation Function

Sigmoid activation function takes in an input value and maps it to a value between 0 and 1. Such an activation function is used when you need the output to be a binary value. For example if you want to predict based on the input if a customer is going to commit a fraud(1) or not (0).



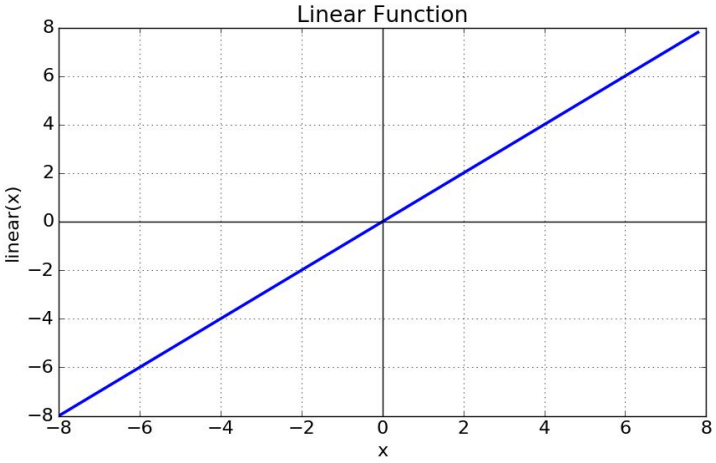
Activation Function

The **ReLU** or the **Rectified Linear Unit** activation function is as given here. For any value < 0 it returns 0, else it returns the input value. This activation function is used in all our hidden layers because it is so simple to learn with and allows the network to be non linear.



Activation Function

The **Linear activation** function is basically the identity function. It returns the same value as the input. You would want to use such a function when you want your output to be a continuous value. For example by how much value, the price of gold will change in the next 5 years. Now this value can be negative, so you can use ReLU, and its not necessary to be between 0 and 1, so you can’t use sigmoid. Hence the linear activation



BackPropagation

To tell the model if it is doing a good job, we define a cost function which is basically the error between the actual value and the model predictions. The objective of this model is to minimize this cost function.

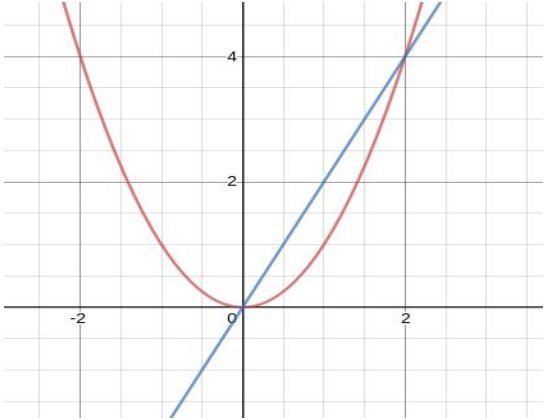
Changing the weights of the model changes the output and hence the cost. So the weights are changed in a way to reduce the cost function.

The derivative of the cost wrt the weights tells us how to change the weights to reduce the cost

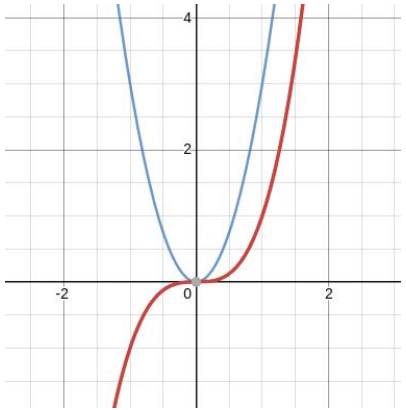
Backpropagation

Suppose my cost function is x^2 and I want to know how I should change x(my model parameter) to reduce this cost.

What if it was x^3? It's the same procedure



Now if your cost function has multiple parameters, like $x^{31}+y^{27}$, you would differentiate wrt to each variable and change it accordingly



Backpropagation

So Once we know in which direction to change each parameter, we follow the give equation to change each parameter iteratively, until we have the best possible set of parameters.

This step is called gradient descent

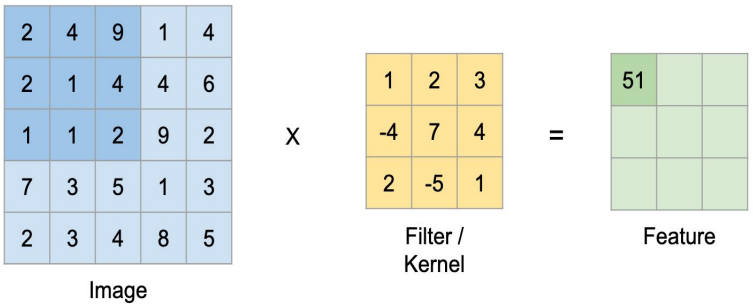
$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

Convolutional Neural Networks

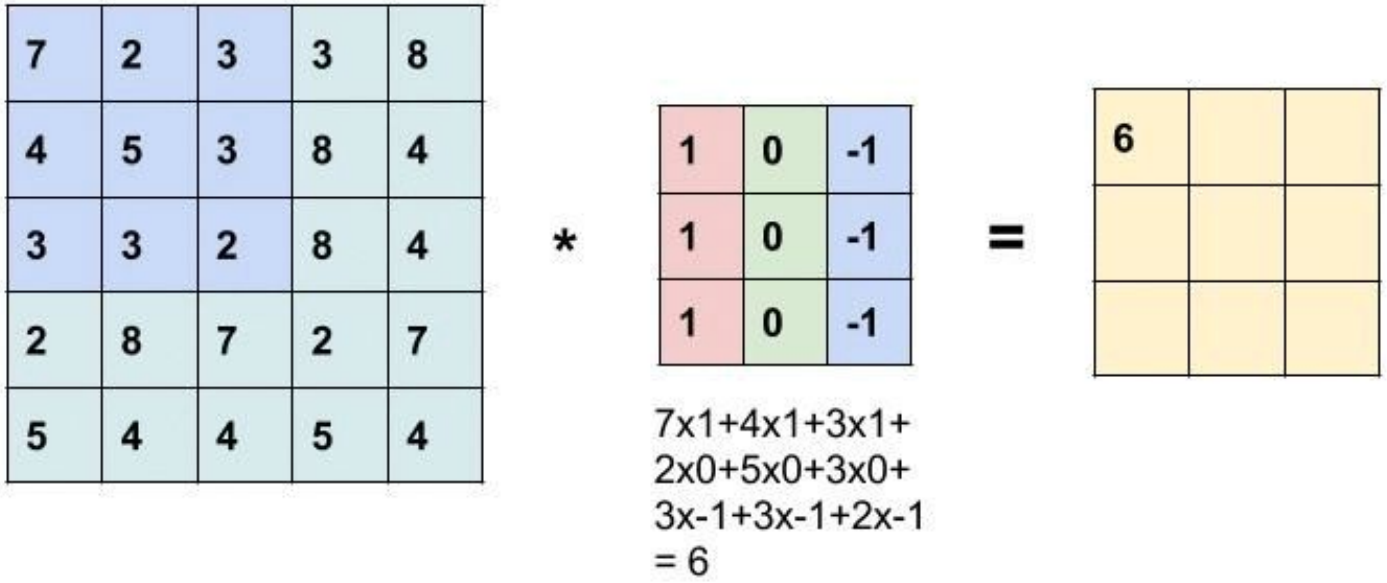
There exists a special kind of neural networks which can process and extract information from images. These are called Convolutional Neural networks or CNNs.

Lets understand what a convolution operation is first. You have an input image and you have a filter. The convolution is basically overlapping the filter with the image and computing a weighted sum.

You keep sliding this filter over the image and computing this sum for every location until the whole image is covered



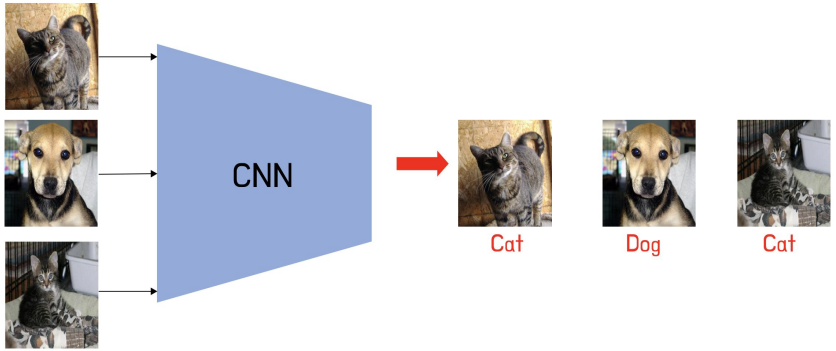
Visualisation of Convolution



Convolutional Neural Networks

These filters are basically your model parameters in this case and you update them to minimize your cost function. But what are these filters actually learning?

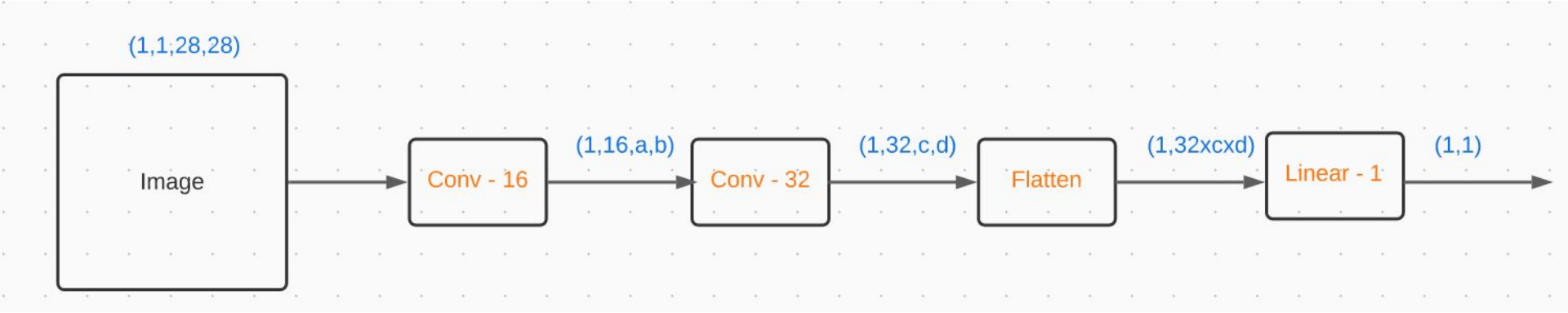
Suppose you are training a network to distinguish dog and cat images, the the filters could for example be the nose of the dog that it convolves and searches for. If there is a nose in the picture, the corresponding sum calculated will be very high and that conveys some information that perhaps this is a dog



Convolutional Neural Networks

A typical CNN consists of a bunch of convolutional layers, which returns 2D feature maps. These maps are then flattened and fed into a bunch of linear layers that give your final output.

After this its the same procedure, compute loss and perform gradient descent.

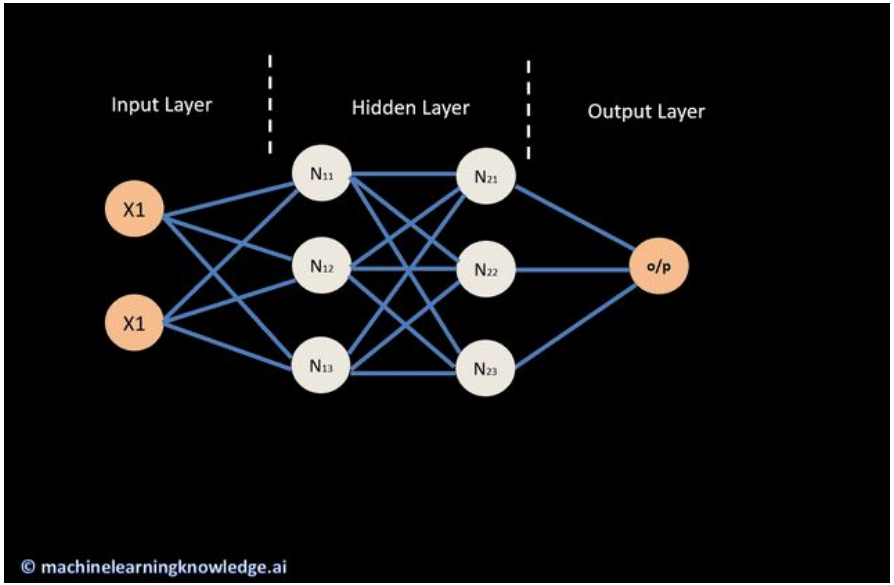


Recap

To sum up, the data is input to the model, it gets forward propagated and the model makes a prediction.

Based on this prediction and the actual values, a cost is calculated.

The cost function is then used to update the model parameters such that the cost is reduced



Let's put this knowledge to use

Now instead of making a table to store the value of every state-action pair, we will construct a neural network that takes its input as the state and the output is the q values of all the actions in that state. So basically it returns one row of the qtable of the corresponding state.

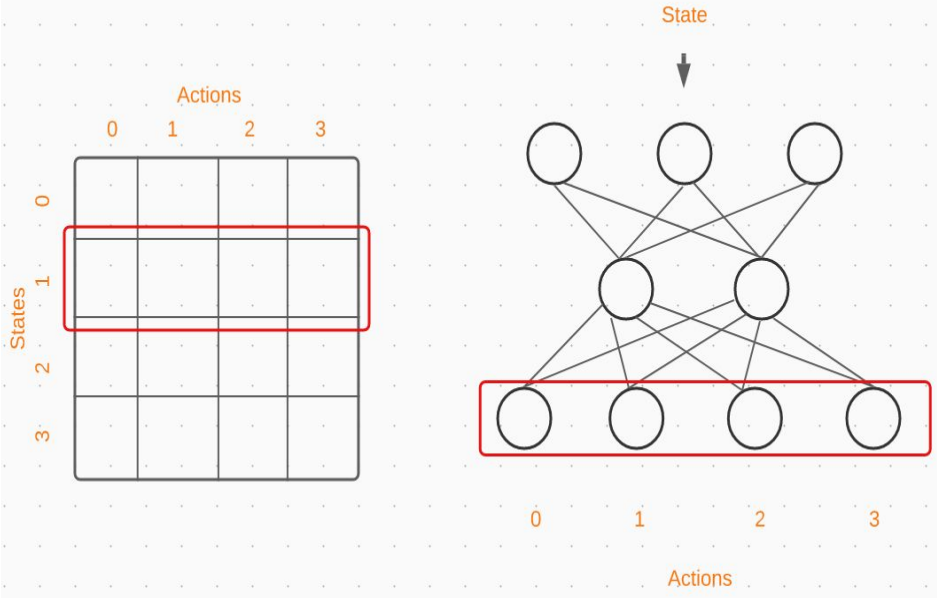
The advantage is of course we do not need such a big memory requirement, but also a neural network can tell if two states are similar or not, whereas in case of the q table, they would be stored as different states.

Let's put this knowledge to use

See the image for the difference in the two methods.

Here our cost function is the mean squared error of the expected q value from the bellman equation (target value) and the q value according to our neural network (predicted value).

When this cost comes close to 0, it means that our network is following the bellman equation, and we have found the optimal q values



Replay Buffer

Replay buffer is a very important aspect needed to stabilize training of Deep Q-Networks (Deep Learning + Q learning). You saw that after every interaction with the environment, we have the following tuple with us: (state, action, reward, next_state, done). This has all the information needed to compute the loss and update the values.

During DQN training, after every interaction, we store this tuple in a memory of fixed size. Periodically during playing the games, we sample a batch of such tuples from the memory, compute our cost function, update model parameters and continue playing. This batch of data is important to make sure the model does not fluctuate too much

Demo

Vizdoom

Policy Gradients

Now that we have Deep learning in our arsenal, we are open to a whole new class of methods called policy gradients.

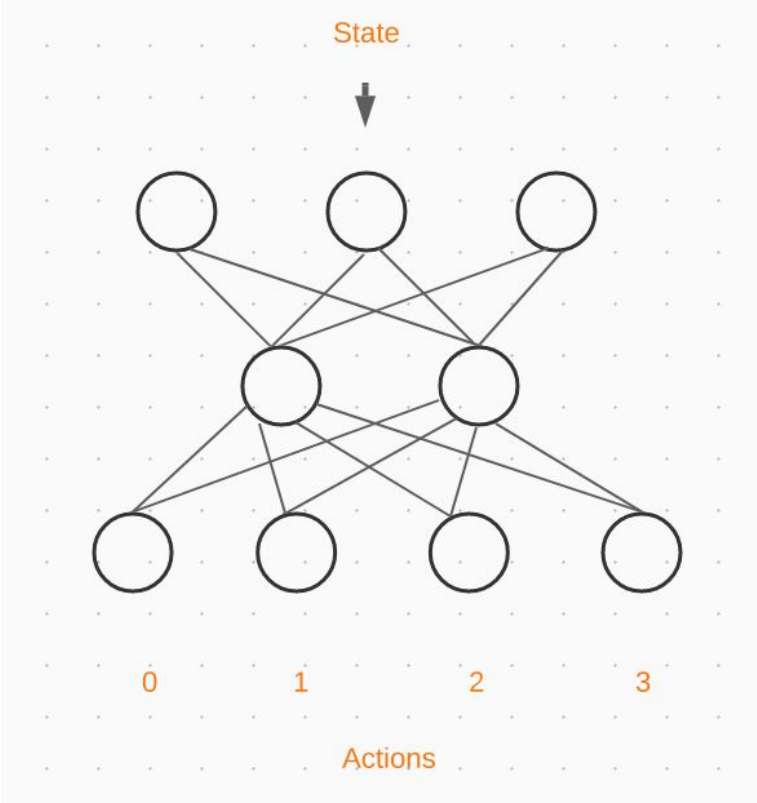
So think about it, instead of making the network output q values, why not make it output the action it thinks is the best and then we can just perform that action. That is exactly what policy gradient methods are. It learns the policy directly.

Remember our objective here is the same, choose actions that will lead to a higher sum of rewards at the end of the episode. So how do we proceed?

Policy Gradients

So in this case, we build a neural network that takes in a state, and outputs an action probability distribution.

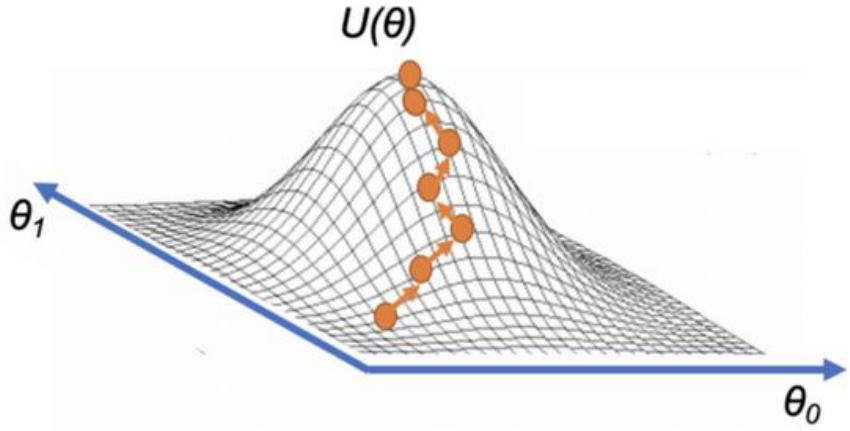
So suppose in our current state, action 2 is the best action, then we want it to output 1 at the node corresponding to action 2 and 0 in the others.



Policy gradients

So our cost function in its most fundamental explanation is the expected sum of rewards starting from that state till the end of the game.

Now this function needs to be maximised and not minimised. So we must change our weights such that this function maximises. This is called **gradient ascent**.



$$\theta \leftarrow \theta + \alpha \nabla_\theta U(\theta)$$

Exploration in Policy Gradients

In Q learning, to explore all possible states and actions, we had to choose an epsilon greedy strategy because we did not have a policy network that can output action probabilities. But now we do.

So our exploration strategy is to sample the actions according to their probabilities. Suppose our model gives us action probs as [0.1, 0.7, 0.2]. It means that action 2 is the best action. But during training, with 10% prob we will select action 1, with 70% action 2 and with 20% action 3.

So we still move ahead in the game and explore along the way which is what we want.

Policy Gradients

The picture given is the cost function for policy gradients. Here Gt is the total reward from step t onwards, $\pi(a_t|s_t)$ is the probability of performing action a_t at state s_t according to the policy network

Now the cost function for policy gradients is very unique and maybe a little confusing out of context. It has a long and quite complicated derivation behind it but we will try to understand the meaning of each term intuitively and then it will make sense

$$\log \pi_\theta(a_t|s_t)G_t$$

Actor Critic

Now you have seen the intuition behind that equation. That was the most basic policy gradient algorithm - **Reinforce**. The problem is G_t is not a good enough estimate for the goodness of that action. We can do better.

So think about it, when you start in a state, your value function will tell you the average return from that state onwards considering all possible actions. From this information, we can now deduce if the action we just performed is better or worse than the average.

For this simply replace G_t with G_t - V(s) and you'll have better information about the action. It tells us how much better it was to do said action compared to the avg of other action which is a more refined understanding of the goodness of the action.

Actor Critic

And that's what actor critic is. You have an actor network that tells you what action to perform and you have a critic network that tells you how good that action was.

In this case the critic network returns the value of that state, which we use to figure out the goodness of our action. Can you think of any more ways to measure goodness?

Ans : Q(s,a) or Q(s,a) - V(s)