Chapter 2

**AI: Deep Q-Learning**

**Self Driving car**

Learning

Acting

Experience Replay

Action Selection Policies

**2.1 Overview**

* In this part we are going get started into the World of AI and build our own self driving car. We will implement a Deep Q-Learning model to build an AI for a Self Driving Car.
* This is going to be a modeled version of a car (so it won't be driving on the streets of real cities) but still - it will learn how to drive itself.
* The keyword here is learn, because the ***car*** will ***not be given any rules*** on how to operate in the ***environment*** before hand - it will have to figure everything out on it's own. And to achieve that we will be using Deep Q-Learning.
* Deep Q-Learning is the result of combining Q-Learning with an Artificial Neural Network (ANN). The states of the environment are encoded by a vector which is passed as input into the Neural Network (NN).
* Then the *Neural Network* will try to predict which *action* should be played, by returning as *outputs* a *Q-value* for *each* of the *possible* *actions*.

Eventually, the best action to play is chosen by either taking the one that has the highest Q-value, or by overlaying a Softmax function.

* OBJECTIVES:

1. Deep Q-Learning intuition: (Learning)

* We'll understand how the NN actually learn, and how they update their weights based on states of the environment.
* How the whole concept of LEARNING works
* How we're going to take the temporal difference and apply them into Deep Q-Learning.

1. Deep Q-Learning intuition: (Acting)

* Then we're going to talk about how Deep Q-Learning algorithm actually decide what action to take in what states.

1. Experience Replay

* We're going to talk about Experience Replay a very important addition on top of Deep Q-Learning.
* It enables Deep Q-Learning to learn to work properly.

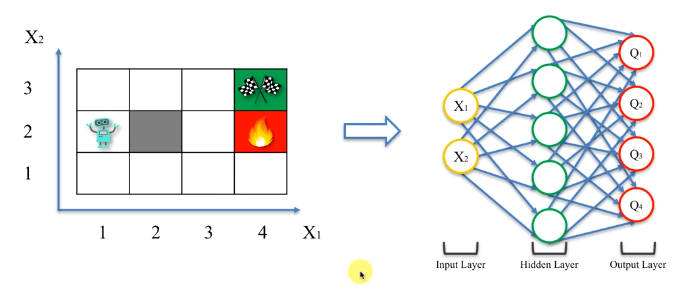
1. Action Selection Policies

* Next we're going to talk about Action Selection Policies.
* How Deep Q-Learning agents are able to combine **exploration** with **exploitation**.
* So once they found "a good approach" they can use that approach but also they need to explore so that they don't get stuck in a local maximum.

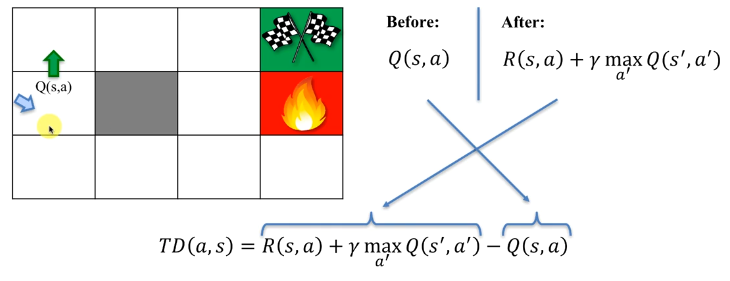
**2.2 Deep Q-Learning intuition – Part 1 (Learning)**

|  |  |
| --- | --- |
| * Previously we discussed about Basic Q-learning, * How the agent will look at the state, * Take an action get a reward. * Enter into a new state and based on that feedback. * The loop will until the agent will understand what are the best actions to take.   We understood how plans in a non-stochastic environments work and how policies work in stochastic environments and this image is an example of a policy. |  |

* Vectorize the Environment: In our environment we're going to add axis, **x** and **y** axis or **x1** and **x2**.
* Now every single state can be described by a coordinate (x1, x2). However, this is a very simplified version of an environment of describing States.



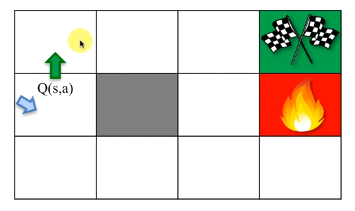
* We can feed those State Coordinates, into a NEURAL NETWORK and then it will process those information and it **spits out four values** (outputs the **4 Q-values**).
* Those ***four values*** are actually going to be our ***Q-values***. These values dictate which action we need to take.
* The key point is, we no longer look at the maze for Q-learning perspective. We're now taking the ***states*** of the ***maze*** and ***feeding*** them into a ***DEEP NEURAL NETWORK*** in order to get these ***Q-values***.
* Why NN? : The **simple Q-learning** will no longer work in more complex environments for instance the "self-driving cars" or "playing a video-game" or "robots walking around and performing actions".
* We know that Deep Learning is by far superior to any type of machine learning or a simple Q-learning.
* We're leveraging the power of Deep Learning here, we're feeding in the information about the environment as a vector of values **(x1, x2)** into a deep neural network and then we use the output values to perform the actions.
* How Deep Q-learning works?: Until this point we've seen the **Temporal Difference Learning**, recall the following concept:



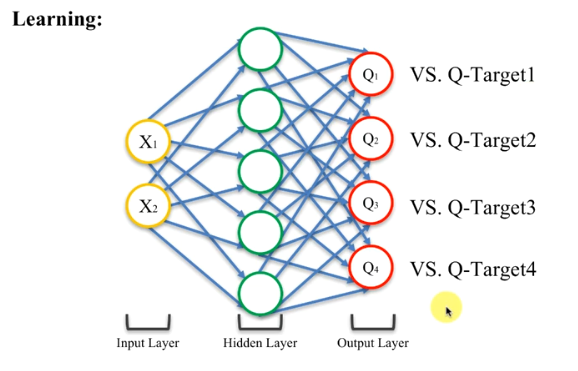
* Simple – Q learning (review): In the Simple-Q-learning case, the agent had a Before Q-Value (learned by moving through the maze), and
* He decided to take an action to go up.
* After the action, he gets a reward for that action plus the value (with decay or discount factor ) of the current state he's in.
* So this new total value **(reward + newStateValue)** is kind of the Empirical Q-value that he has just received for taking that action.
* We call this After-value.
* The **Temporal Difference** is the difference of the **Before-value** and **After-Value**.
* And then you use your learning rate Alpha to adjust your New Q-value.



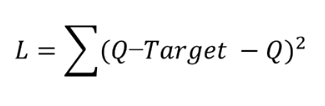
* **Deep Q-learning:** In Deep Q-Learning the neural network will **predict** four values or it ***might predict more values*** of more possible actions in a given state.
* For simplicity's sake we consider here only four actions up/right/left/down.
* In a Deep Q-Learning Situation it is important to understand that, there is no before-value or after-value.
* The NN will predict four of these values and it will compare not to what will happen after (after-value) but the exact value (the value which was calculated in the previous step).



* There is no before-value or after-value: The value of the current-square was calculated from the previous-square.
* For example, say previously the agent calculated this current value then the agents stored this value for the future and now the future has come.
* So it's going to compare the predicted Q-value to this stored value. There is no before-value or after-value.
* We're taking the predicted Q-value from the Neural Network and we comparing it to stored value (which he had from the previous time when he was in this square: *like the previous time he actually performed this action*).
* What happens in the Neural Network: In our ***NN*** we're feeding the states of the environment as ***vector***, which going through the hidden layers and outputs ***Q1, Q2, Q3, Q4***.
* In that specific state these are the Q-values that the ***NN*** is predicting for possible actions.
* Then we're comparing these predicted values to the target-values (stored value from the previous time for the same action).



* This is the part where the Neural Network or the AGENT is learning through deep learning how to take better actions.
* The key point here is that, we're still applying Q-learning.
* In Simple Q-Learning you learn through Temporal Differences.
* In Deep Learning, the Neural Networks learn through adjusting the weights.
* So our goal is: How do we adapt that concept of Temporal Difference of Simple Q-Learning to the Neural Networks.
* Calculating LOSS: Now we're comparing these predicted values to the target-values to calculate a **LOSS** (take the differences of corresponding "target-value" and "predicted Q-value", square the each difference and take the sum) which is given below:

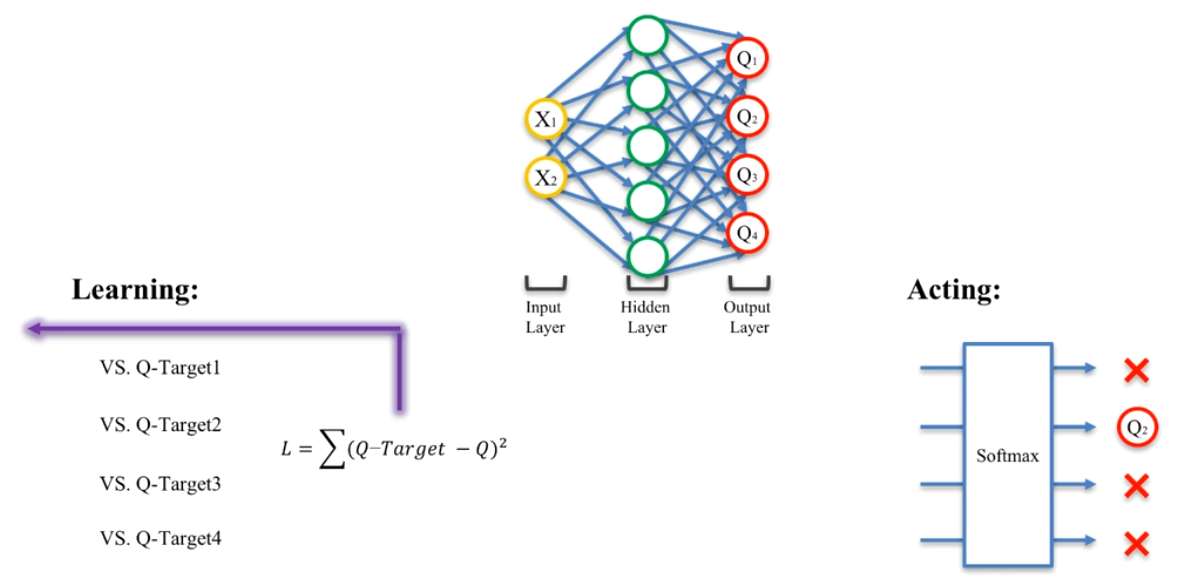


* Ideally we want this **loss-L** to be **0**, that means **Q-target** and **Q-value** (predicted values) to be the same. We did similar thing to the Temporal Difference (***Before-value*** and ***After-value*** become the same).
* **Loss = 0**, means the agent is predicting correctly, he knows environment so well that he can predict what's going on and therefore his policy is going to be very good.
* How do we do that?: We're going to take this Loss and back propagate through the Network to update the weight so that in the next iteration the Loss can reduce. Through this iterative process Loss converges to **0**. The agent learns the environment.
* Through the iterative process of Loss-calculation > Back-Propagation > Weight-update, the agent is learning and becoming more and more *descriptive* of the *environment* and therefore the *agent* is able to *navigate* the *environment*.
* *Descriptive environment basically means that:* when we put the states of the environment into the NN we get closer and closer to the actual Q-values.
* Summery:
* The **empirical Q-values** (or Target-values) are learned from **Trial-Error** process.
* Then the **NN** is constructed to **predict Q-values** that are close to the **Target-values**.

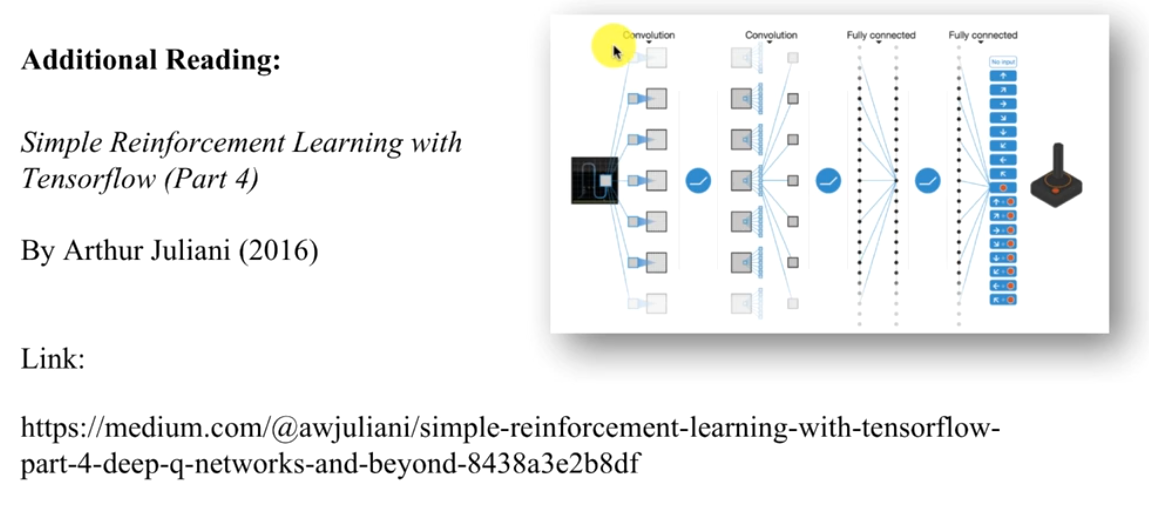
**2.3 Deep Q-Learning intuition – Part 2 (ACTION selection policy)**

Now we're going to move on to the acting part.

* Let's say our agent is learned and he actually has to take an action. He has to decide which action he's going to take (move up/left/right/down).
* Note that the **Q-values have not changed**. We just used these values to compare them with predicted Q-values. To calculate the error, we've updated the weights and the agent learned the environment, but the Q-values don't change in that whole process. They're fixed. We know what they are.
* The Networks is updated and output those same values that we had.
* To choose an action, we're going to **pass** those **Q-values** through a Soft-Max function. A Soft-Max Function basically **selects** the **best action possible**.
* After taking the selected action, all of this process happens again for the next state (in our case and the next square of the maze).
* That's how we implement a Reinforcement Learning problem into a Neural Network through a Vector describing the State that we're in.



* There's two parts of the process that happen:
* Learning:
* Compare each of the predicted ***Q-values*** with the target and
* Then we back propagate the loss through the network to update the weights so that our network is learning as we go through this maze or the environment.
* Action:
* To select an action we pass the values through a soft-max function (action selection policy).
* Then we perform that action and then this whole process starts again for the next state.
* For another epoch the agent plays the whole game again. Until it learns the environment or it's in his favor.
* One point to note, this process happens for every single time the agent in a new state, not just for every single game that he plays but for every single state.
* The learning and the acting both happens as well.
* Additional reading: If you'd like to get some additional information on Deep-Q-Learning, there are Arthur Juliani's series of blog posts: **"Simple Reinforcement Learning With Tensorflow Part 4"** you will find the part that's relevant to what we discussed here.
* Here he also discussed the **Convolutional-Deep-Q Learning**.
* Convolution: the agent is looking at the image and therefore he has to process an image to take and action.



**2.4 Experience Replay**

Here we're still going to talk about the learning part, we'll discuss about a very important feature for Deep-Q-Learning called Experience Replay.

* Here is our network we've got that Loss, which is Back Propagated through network.



* So above is an example of a Self-Driving Car that is driving along this road and it has to learn how to navigate this road. The yellow line represents the Sand-Border of the road.
* x1, x2 represents the states of this environment, there are also couple of parameters for instance: angle of the car and reading from the sensor. So there's going to be more parameters than that to describe the state.
* But nevertheless it's going to be a ***vector of values***, which we ***feed*** to the ***Neural Network*** and then on the output you're going to have some ***Q values***.

Again there'll be a difference depending on the environment, They can be a different number of possible actions. But for simplicity sake leave it at four, so that we can understand better what's going on here.

* So the question is: How often do we trigger this neural network? How often does this neural net go through update?
* **In summery following things will happen:**
* Every time the car moves, it ends up in a new state. Then the Network is triggered, data goes through the network, outputs Q-value, error calculated and back propagated through the network then weights are updated.
* Then the car selects which action need to take, it makes a move, ends up in a new state and everything starts over again.

This happens every time the car is in the new state.

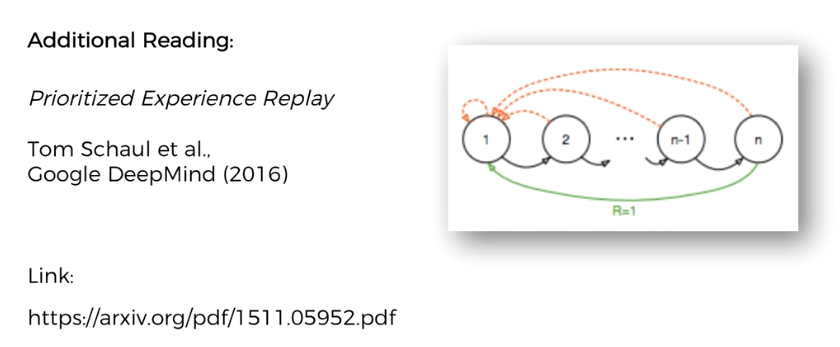
* **Problems of CORRELATED or INTERDEPENDENT STATES:** *Experience Replay* is not just something that we use in this specific problem. It is something that you will see *everywhere* in *artificial* *intelligence* *algorithms*, because it is so powerful and it's so important.
* In this car problem or in this environment it's goal reaching the finish line of the road, without crossing the sand-border. The road is not straight, it has two curves (turns).
* Its reward is based on how close it is to the sand-border. So that the car moves along this sand-border without crossing it.
* Notice that, every *single time* it moves *forward* along the sand-border, *nothing is changing*.
* We know the car is moving, but in terms of the surrounding environment not many things are changing. It's still that same sand-border.
* If you are sitting in the car you probably seen the situation like a monotonous environment, that you're just seeing kind of the same thing is just passing by.
* Now every single time we're putting that new state into our Neural Network.
* On those inputs, there are *some inputs* that *might be changing*, for example car's GPS.
* But a lot of other inputs for example the *sensors for sand-border* on the car, they're not changing. For most of the new state (moving forward) the inputs are pretty much the same.

|  |  |
| --- | --- |
| * So if we **keep inputting** the **same values** in our **NN**. The car will learn very well about how to drive along this wall which is on its right. * The *network* will *trained* (update its weights) *only* for that kind of movement. * It will have this false perception that it's actually doing very well even though it only learns how to drive along the sand-border. * The **problem arise** when the **turning point comes** all of a sudden, there's a curve and the car doesn't know what to do. And it completely doesn't fit in with this neural network. |  |

* Since we're updating the NN every single state, and we have lots of consecutive stuff. There are so many consecutive states that are somehow correlated or interdependent.
* We don't want that interdependency to BIAS our neural network.
* We don't want the car to just learn how to drive along like a straight line or along curved line.
* Experience Replay: Any kind of correlated or interdependent states that come after another can really mess up our neural network. If you just going to let the agent learn from that.
* That's where Experience Replay comes in.
* In experience replay, we don't input those correlated or interdependent states into the Neural Network.
* They are actually saved into memory of the agent.
* At some point once it reaches a certain threshold the agent decides to learn.
* It has this batch of experiences and its going to learn from that.
* It randomly selects a **uniformly distributed sample** from that batch of experiences that it has in its memory. Where basically all experiences are considered to be equal. Then it goes through them and learns from them.
* Each experience is characterized by following four elements
* Previous state: **State** it **was** in
* Action: **The action** that it took
* Next state: **The state** it **ended up** in and
* Reward: **The reward** it achieved through that action in that specific state.
* It takes all those experiences and then it passes them through the network and it learns.
* And that way it **breaks** the **pattern** of that **bias** which comes from the **sequential nature** of the **experiences**, if you were to put them through the network one after the other.
* Another benefit of Experience Replay:

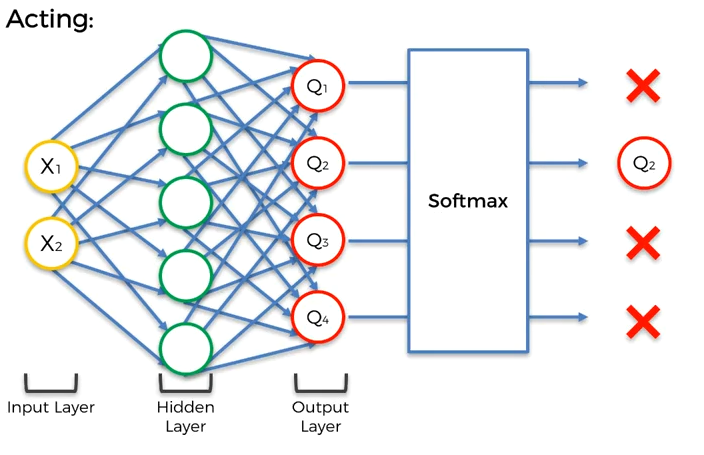
Another benefit of Experience Replay is that sometimes in an **environment** you might have very **valuable rare experiences**.

* For instance: In case of self-driving car, sudden U-turns or sharp corners.
* Those are the opportunities in the whole environment to learn from a sharp right corner or U-turns.
* And those are very important experience because the agent might get really good at driving along straight lines and soft corners, but then it'll keep messing up this sharp (right/left) corners (or sharp-U-turns) because it doesn't have that much opportunity to learn from it.
* Therefore it will learn everything else very quickly but it'll take a *long time to learn* these rare experiences.
* The role of Experience Replay for RARE EXPERIENCES: If we're just doing a Simple Neural Network and putting these rare-values there, they'll go through the NN and forgotten about that rare-experience. Because rare experiences doesn't appear in sequential nature.
* The NN instantly forget about the rare experiences when it moves on to the next state. So those sharp-corners as soon as it goes through a network they're gone and you don't have any memory of that valuable experience.
* But when we apply the Experience Replay, the values appears in batches.
* We're putting these experiences into batches. It acts as a **rolling window**.
* When the agent takes a uniform distribution from its batch of experiences and then this rolling-window moves, the sharp-corner (rare-experience) might come up *several times* in its learning process because it is in that *batch*.
* The older experiences removed out from the rolling-window and newer experiences are added, but the RARE-EXPERIENCES stays in that window for several time, and it *won't disappear suddenly*. And that's the benefit of Experience Replay in case of RARE-EXPERIENCES.
* That’s how the NN remembers the Rare-Experiences and learns form it properly.
* The Rare- experience stays in the batch for quite some time and the car or agent can learn from that experience several times.
* The final advantage is: *Experience Replay* gives you an opportunity to learn from more experiences.
* A batch here act as a rolling window and therefore even if your environment is limited to experience, then experience replay approach can help you learn faster.
* Additional Reading: There's an interesting article published by Google Deep Mind in 2016 is called Prioritized Experience Replay.
* It talks about why are we using a uniform distribution to select our experiences from the experience batch.
* Why don't we find a better way to select our *experiences* and *prioritize* some of the *experiences* which we feel that are *important*.
* How we can change the *uniform-distribution* to a *different approach*.



**2.5 Action Selection Policies**

Here we'll discuss about **Action The Selection Policies**. Previously we talked about Learning-part, now it's time to discuss the Acting-part.



* Once you input the vector of values describing the "state agent is currently in", and the environment. Then the Network outputs the ***Q-values***.
* The question is: How do we understand which one we need to use? How do we choose the best Q-value?
* Here we're dealing with Predicted Q-values from the NN.
* In the simple Q-learning algorithm, we selected the ***best Q-value*** by selecting the ***highest*** ***value***.
* With the ***highest Q-value*** we just take that ***action*** because it brings us the ***highest value***, which recursively calculated through ***reward***, ***Discount-factor*** & ***next state-value***.
* **So why wouldn't we take the best predicted Q-value in case of Deep-Q-Learning?**
* The selection policies: But it's not as simple as that, we have to use Soft Max function and this is the part of actual selection policies.
* In reality we don't have to have just a Soft Max function.We can have different action selection policies. For example we've got

1. (Epsilon) greedy
2. (Epsilon) soft and
3. Soft Max

* Of course there are other action selection policies, but these are most commonly used.
* For instance the MOST BASIC ONE is: Just select the best, the one with the highest Q-value.
* **EXPLORATION vs EXPLOITATION:** This is the reason why do we have different types of action selection policies.
* This is the Core of Reinforcement Learning because your agent when it's operating in an *environment* it might *predict* certain *Q-values* which might be GOOD or it might turn out BAD and the agent will be forced to *explore* the *environment*.
* For instance in our case, it predicted that ***Q2*** is the best one and then it takes ***Q2-action*** and it gets a very ***negative reward***. Then the ***environment*** is ***forcing*** the agent to go and explore because now he's going to learn that *"Oh!! Actually I thought Q2 was going to be very good but it turned out very bad!"*
* So the networks gonna update itself so that next time he's in that state he's going to take different action (however it might need several penalty to take different action).
* **Stuck in a Local Maximum:** So sometimes the environment forces the agent to explore, to take different actions. But sometimes the agent might get it find itself stuck in a local maximum.
* For example, it followed through it's initial exploration and in a state it found a good action and it stuck there, it doesn't explore more to get a better action.
* Say in our maze problem, it's gone through UP, Left, Right of a specific state and it found that going Right is a *good* *action*, and it stuck there, but in reality going Down is ***far better action*** than going Right.
* Now it's kind of biased towards this action of going Right and thinks it's a good and it is going to keep taking this action and keep getting a good reward.
* It thinks, going Right is the *best action* because it *hasn't explored down from that specific state* that it's in.
* So we want to come up with an action selection policy that allows our agent not to get stuck in a local maximum.
* **Exploitation Part:** However it's important to keep doing the good actions. We won't exploit what we've found.
* **Exploration:** But at the same time we still want to explore (we never want to stop exploring, as like in our life we won't stop learning).
* So our agent wants to keep learning. And that's where these **action selection policies** comes in.
* Action Selection policies: Now let's discuss the *three common action selection policies*.

1. **Epsilon () greedy:** It will select the action with the ***best Q-value*** for the times.

* So it selects the Highest-Q-value all the time except for Epsilon () percent of the time.
* For instance if you set **Epsilon ()** to **10%** then 0.1 of the time **(i.e 10% of the time)** a **random action** is going to be selected. Random action will be taken uniformly, i.e. it's going to be absolutely random action.
* And **90% of the time** the *best action* based on the *highest value* will be *selected*.
* So here is **10% Exploration** and **90% Exploitation**.
* Why it's called Greedy? Because we're *greedily selecting* the *good action* except for that little *epsilon* **()**.

The lower the *epsilon* **()** the more *greedily* you're selecting that *good action*. You're leaving the less chances for exploration.

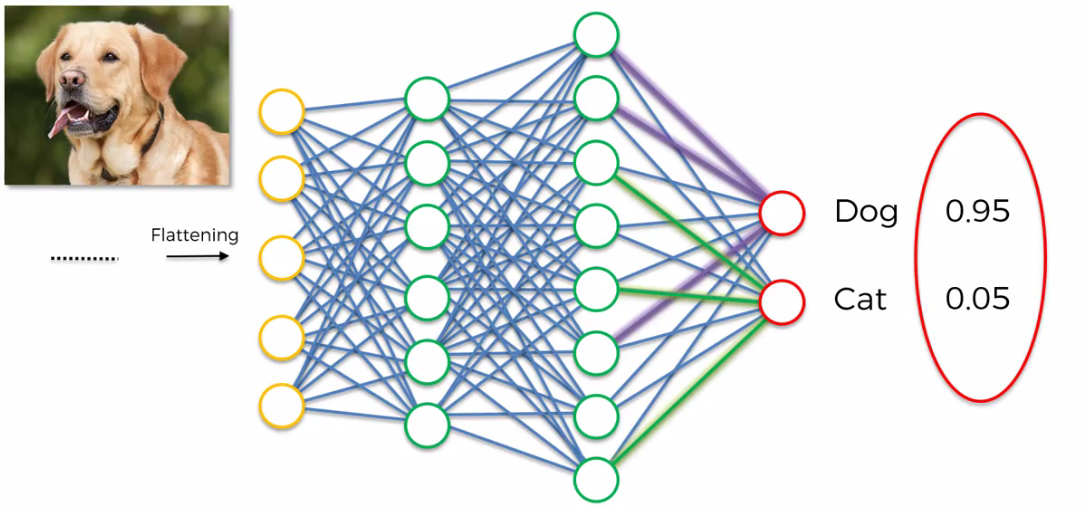
1. **Epsilon (ε) Soft :** **(Epsilon) Soft** is the opposite of **(Epsilon) Greedy**.

* Basically you're selecting ***random action*** for of the time and ***good action*** for of the time.
* So if you epsilon is **0.1** or **10%** then only **10%** of the time you're taking good/optimal action and **90%** of the time you're selecting a random action.
* It means **90% Exploration** and **10% Exploitation**

1. **Soft Max:** It's a more advanced version of the Epsilon () Greedy algorithm. Although, they both have advantages in different situations.

* We're going to use soft-max in our coding and practical application.
* We've discussed this Soft-Max in CNN part of *"Introduction of Machine-learning and Deep-learning (Tutorials on Machine-learning and Deep-learning): in* section 9.4.3", now lets revisit this topic below.

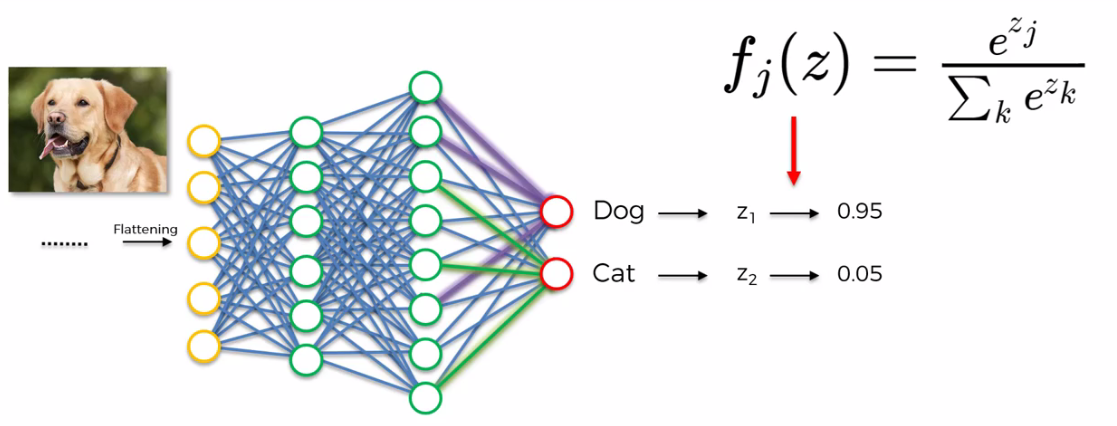
**2.6 SoftMax function (CNN 9.4.3 MLDL-1)**



* Here is the CNN that we built to Recognize Cat/Dog in a given image. Now notice the output ***probabilities*** ***0.05 for Cat*** and ***0.95 for Dog***.
* Now the question is: How the ***sum*** of these two values is ***1***. (in case of more than two class Dog, Cat & Bird this Sum of output probabilities is always 1).
* Because as far as we know from ANN there is nothing to say that these two *output-neurons* are connected between each other.

So how would they know about the value that other one is holding? And how would they know to add their probability-values up to 1.

* The answer is : They wouldn't in the classic version of our ANN.
* SoftMax: The only way that they would know about those values if we use a *special function* called the SoftMax function.
* So normally the Dog and the Cat Neurons would have any kind of real values that their sum don’t have to be 1. Say these values are and **.** And .
* Then we would apply the SoftMax function upon and so that the sum of the output becomes 1.



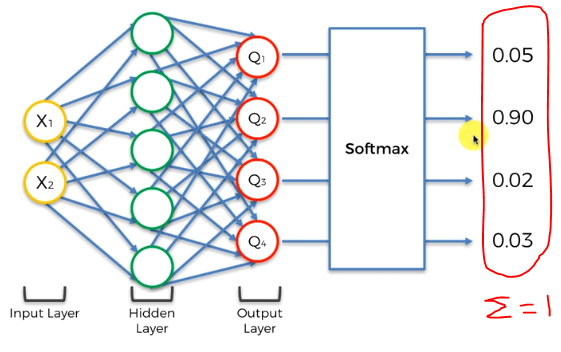
* Here the SoftMax function or the *Normalized Exponential Function* is a Generalization of the Logistic Function that Squashes a k-dimensional vector of arbitrary real values to a *k-dimensional vector of real values in the range of [0, 1]* that add up to 1. i.e. for example ***(23, 45, 79, 45)*** becomes something like ***(0.1, 0.25, 0.4, 0.25)***, it ***normalize*** the ***vector*** in a way so that the sum of the ***elements*** of the vector becomes ***1***.
* The *elements* of the *vector* in our case the *values* returned by *Cat-Neuron* and *Dog-neuron*. SoftMax function brings these values to be in range ***[0, 1]*** and make sure that they add up to ***1***.
* Why SoftMax function is used in CNN: It makes sense to introduce the SoftMax function into CNN because it is strange that a probability of being a dog is 95% and also probability of being a cat is 65%.

Therefore it's much better when you use SoftMax function to CNN

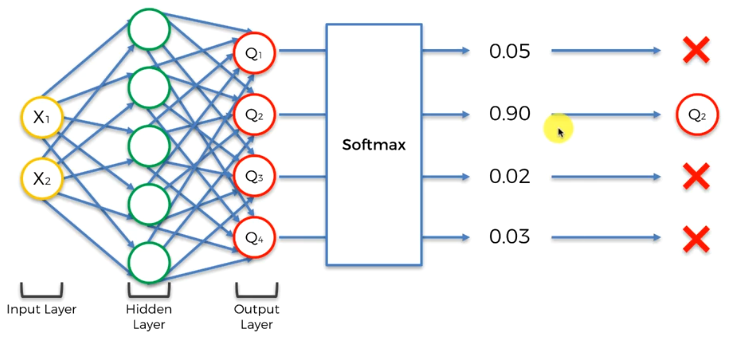
* **How Soft-Max function applied in deep Q-learning:** Hence, in the case of deep Q-Learning Soft-max is very beneficial for us.

|  |  |
| --- | --- |
| * What happens is: we get these Q-values (***Q1, Q2, Q3, Q4***) we select the best. In reality those values can be any numbers, but after ***Soft-Max*** is applied, these values in the range between ***0*** and ***1*** and there sum will be equal to 1. * We also know that probabilities always add up to ***1***. So after applying ***Soft-Max***, those Q-values (***Q1, Q2, Q3, Q4***) become actual probabilities. |  |

* Of course, before applying Soft-Max, these Q-values are the Predicted values (i.e. probabilities), but their sum can be greater than 1 (and there is no relation between them), and that reason we cannot apply them properly as probabilities. Soft-Max function just resolved this problem (and it establishes relation between these probabilities).



* And therefore that's how we come up with Q2 (the best Q-value). Then the action is selected.



* But if you look at it closely this isn't a strict 100% or 0%. The most natural way to apply the SOFT MAX in order to preserve **exploration** in the algorithm is to "*use these exact probabilities as how often we're going to be taking that action* ".
* Basically SOFT MAX makes it very easy for us to combine **exploitation** and **exploration**. So these probabilities (after Soft-Max applied) are actually present the DISTRIBUTION of these actions that we're taking.
* So that the best action will always have the high probability because it has highest Q value and therefore here we're going to use these as our distribution
* 90% of the time we're going to be taking Q2
* 5% of the time we'll take Q1
* 2% of the time Q3
* 3% of the time Q4.
* And the beauty here is that, as those **SOFT-MAX** applied **Q-values** will be updated, as the agent goes through the network more and more. After many iterations the agent becomes more familiar with the environment and therefore these probabilities change/update according to reward (therefore Q2 won't always be a best option, it may change).
* So Q1, Q2, Q3, Q4 will change.
* Here every action has a chance to play in this process as long as we have enough iterations, agent goes through lots and lots of times through the states that they're in.
* And they learn from the experience.
* **The benefit of Soft-Max:** We're not just selecting randomly like Epsilon (ε) greedy algorithm (for example it selects Q1 , Q3, Q4 randomly 10% of the time).
* In Soft-Max we're selecting them based on their Soft-Max values, which makes it much more logical selection. We're doing it based on the Q-values that we've explored.
* **Additional Reading:** So this is called "Adaptive (Epsilon) - Greedy Exploration In Reinforcement Learning Based On Value Differences", it's the 2010 article by Michel Tokic.
* Here a different type of algorithm is introduced called epsilon greedy VDBE - Boltzman algorithm.
* Here he compares to the (Epsilon) - Greedy and soft-Max.
* The main idea behind the adjusted (Epsilon) - Greedy is to ***adjust*** the ***value of epsilon*** depending on the ***state*** the ***agent*** is in.
* If the agent is very certain about the state in then (Epsilon) should be smaller, so there should be less exploration.
* If the agent is uncertain then (Epsilon) should be higher and there be more exploration.
* This article will definitely help you reinforce your knowledge about action selection policies and also you can see in which direction people actually think when they want to improve artificial intelligence.

