Chapter – 2 (implementation – Part 2 – NN & Replay implementation)

**Artificial Intelligence**

**AI-app (Deep-Q): Self Driving Car** (part 2)

Class - NN architecture: fn - forward

Class - Experience Replay : fn – push; fn - sample

**NN architecture**

**2.7 step-4 NN architecture class (part 1): \_\_init\_\_**

Let's create the architecture of the neural network: the Heart of our AI and it will return the action to play at each time t.

* We'll make a class for the NN. The **\_\_init\_\_** function will contain the architecture and we'll define a forward function.
* In the **\_\_init\_\_** function we'll define the architecture of the NN:
* The input layer (composed of five input neurons because we have *five dimensions* for the *encoded vector of input-states*).
* There will be one or more hidden layers.
* The output layer (contain the possible actions that we can play at each time).
* The ***forward*** function will activate the neurons in the NN.
* It will activate the signals and we'll use a Rectified Activation Function because we're dealing with a purely nonlinear problem and this rectified function breaks the linearity.
* This forward function will return the **Q**-values as the ***outputs*** of the NN.
* Since we need only one **Q**-value for each action, we'll use ***Soft-max*** for that.

# *Creating the architecture of the Neural Network*

**class** Network(nn.Module):

**def** **\_\_init\_\_**(self, input\_size, nb\_action):

**super**(Network, self).**\_\_init\_\_**()

        self.input\_size = input\_size

        self.nb\_action = nb\_action

        self.fc1 = **nn.Linear**(input\_size, 30)

        self.fc2 = **nn.Linear**(30, nb\_action)

**def** **forward**(self, state):

        x = **F.relu**(self**.fc1**(state))

        q\_values = self**.fc2**(x)

**return** q\_values

* Notice, we're using *inheritance* under nn.Module. by using **super**(Network, self).**\_\_init\_\_**(). So our class **Network** becomes the *child class* of **nn.Module**.
* **\_\_init\_\_:** we have three arguments.
* **self:** refers to the object that will be created from this class that we're about to make.
* **input\_size:** number of input neurons. It is 5, because our input vectors have five dimensions: *three* *signals*, *plus orientation*, *minus orientation* these are vectors of encoded values that describe one single state of the environment. These five values are enough to describe a state of the environment.
* We could have more values, e.g. we can use more signals like ***360-degree signals*** at the ***top*** of the ***Google cars***, but for our model *3-signals (left, front & right) are enough*.
* The orientations: *+orientation* and *–orientation* will keep *track* of the *goal* that you were *trying* to *reach*.
* **nb\_action:** number of output neurons (output the actions, from which we'll choose best one using soft-max). We have three possible actions: going ***left***, going ***straight*** or going ***right***.
* We use **super**(Network, self).**\_\_init\_\_**() to inherit from **nn.module**. Notice we specify our class Network. It's just a python trick that's allows us to use all the tools from nn.module.
* Input layer: The next step is to specify the input layer.

self.input\_size = input\_size

* This variable will contain the ***number of input neurons***.
* Don't get confused with input\_size in the argument of the **\_\_init\_\_** function, actually self.input\_size is a variable of our class ***Network*** and at the right input\_size is from the argument of the **\_\_init\_\_** function. Here we're storing the value passed through the ***argument*** to a ***variable*** with ***same name***.
* Similarly we create self.nb\_action = nb\_action. This is going to be a variable for the *number of* output neurons.
* Full connections: There are *another two variables* we want to define for the class, the full connections. These are the *full connections* between the *different layers* of our neural network.
* Since we want to make a neural network composed of *only one* Hidden-layer, there will be two full connections.
* The first full connection between the *input layer* and the *hidden layer*.
* The second full connection between the *hidden layer* and the *output layer*.
* First full connection: we call it fc1,

self.fc1 = **nn.Linear**(input\_size, 30)

* We'll use ***nn.Linear()***, it'll create the full connection between the ***neurons of input layer*** to the ***neurons of the hidden layer***.
* The full connection means: *all the neurons* of the ***input layer*** will all be connected to *all the neurons* of the ***hidden layer***.

Arguments:

* input\_size: number of neurons of the first layer. It will b e 5 since we've 5 inputs.
* 30: number of neurons of the next layer. It depends on parameter-tuning and experimentation, turns out 30 is the best number. With this number we will get some pretty good results
* Bias = True: its True by default, so we're not gonna change it.
* Second full connection: It is the full connection between the hidden layer and the output layer. We call it fc2,

self.fc2 = **nn.Linear**(30, nb\_action)

Arguments:

* 30: number of neurons of the previous layer.
* nb\_action: number of neurons of the next layer (output layer). It will be 3 since we've 3 outputs.
* we're not gonna change Bias.
* Finally our **\_\_init\_\_** function is ready. It'll initialize our object whenever we create an object from our **Network** class. Each object will have the NN architecture defined inside **\_\_init\_\_()**.
* Each object will correspond to a neural network of five input neurons, 30 hidden neurons and three output neurons.
* Feel free to change the architecture of the neural network.
* You can not only change the number of neurons in the ***hidden layer*** but also you can add some more layers so that maybe you get an even better car.

Next we'll define the ***forward function*** of the **Network** class which will be used to activate the neurons in the ***neural network*** using the ***rectifier activation*** ***function*** and eventually return the ***Q-values*** as output.

**2.8 step-5 NN architecture class (part 2): fn - forward**

Now we build the forward function, which will activate the neurons. i.e. the function will perform forward propagation. This function is going to take two arguments.

* **self**: indicates the object itself.
* **state**: state is the input of our neural networks.
* This forward function is not only going to activate the neurons but also **return** the **Q-values** for each possible action depending on the input ***states***.

**def** **forward**(self, state):

        x = **F.relu**(self**.fc1**(state))

        q\_values = self**.fc2**(x)

**return** q\_values

* Activate the hidden neurons: We'll going to call the hidden neurons by the variable **x**. To activate them, we'll use **torch.nn.funtional** (**F** in short) and from it we'll use **relu()**, as a rectifier-linear-unit as an activation-function.
* We'll use our input neurons "**fc1**" by putting **state** as the input. Because, we want to go from *input-neurons* to *hidden-neurons*.
* Now we have the *hidden neurons*, let's return the output neurons. The *output neurons* correspond to our *actions*. But we are not returning the ***actions*** ***directly***, we'll return the Q-values first. Then we'll get the final action by applying the Soft-Max functions over the Q-values.
* Here we're building a ***Deep Q-learning Model*** that combines a *deep-learning model* to *Q-learning*. We use Q-learning here to get the Q-values for each of our actions.

That's why we name the output neurons as q\_values.

        q\_values = self**.fc2**(x)

**return** q\_values

We directly take our full connection **fc2**, then we input the hidden-neurons **x** that we got in the previous line. Finally we return those q\_values, from our *forward-function*. It will return the q\_values for each possible action to go left/straight/right.

**Experience Replay**

**2.9 step-6 Experience Replay class (part 1): \_\_init\_\_()**

Now we are going to implement experience replay. We'll make a new class: **ReplayMemory** and that will implement experience replay exactly as we saw in the intuition lectures.

* What is experience replay?: We know all this artificial intelligence is based on Markov decision processes and *Markov decision* *processes* consist of looking at a ***series of events***. e.g. going from one state to the next state .
* But if the next state it is very correlated to the current state. The network would not be doing very well.
* In deep learning that's exactly the same as where we learned our time series with only one timestep. It was not learning anything because one timestep was not sufficient enough for a model to learn to understand ***long term correlations***.
* Same thing goes with the experience replay.
* How does it work: Instead of only considering the current-state (i.e. one state at time t), we're going to consider more in the past. It's like our LSTMs and therefore our series of events will not be and , this will be for example the ***100 states*** in the past i.e .
* In other words we put 100 last transitions into the memory and that's why we have a long-term memory as opposed to a short-term/ instant memory and that makes the whole process work much better.
* Once we create this ***memory*** of the ***last 100 events***, we'll sample it (i.e. we'll take some random batches of these transitions) to make our next update (i.e. our next move by selecting the next action).
* Therefore in this ReplayMemory class that we're implementing for experience replay, we will make three functions.

1. **\_\_init\_\_()** : We will define the *variables* that will be *attached* to the *future instances (objects)* of the class. These variables will be:

* the memory of the 100 transitions to 100 events.
* the capacity (i.e. 100 number, you can try a longer memory by increasing the capacity).

1. **push():** To make sure that the memory doesn't ever contain more than ***100 transitions***, we'll use the ***capacity*** by applying IF-condition.
2. **sample():** To sample some ***transitions*** in the memory of the ***last 100 transitions***.

# *Implementing Experience Replay*

**class** ReplayMemory(object):

**def** **\_\_init\_\_**(self, capacity):

        self.capacity = capacity

        self.memory = []

**def** **push**(self, event):

        self**.memory.append**(event)

**if** **len**(self.memory) **>** self.capacity:

**del** self.memory[0]

**def** **sample**(self, batch\_size):

        samples = **zip**(\***random.sample**(self.memory, batch\_size))

**return** **map**(**lambda** x: **Variable**(**torch.cat**(x, 0)), samples)

* **\_\_init\_\_**(self, capacity): The variable capacity is to try some other experience-replays some of the memories. capacity will simply be **100** because we're going to make experience-replay with the 100 last transitions.

**def** **\_\_init\_\_**(self, capacity):

        self.capacity = capacity

        self.memory = []

* self.capacity: maximum number of transitions we want to have in our memory of events. It stores the value coming from the argument capacity.
* Memory: is just a simple list, to contain the last 100 events (last 100 transitions) and we initialize as an empty list.

Our memory is initialized!! Of course at the *beginning* of the experiment/exploration the *memory* will be an empty list and then we will put the transitions *each time* we reach a future state.

* That's exactly what we will do with the **push** function.
* We'll make this **push** function to append the events in this memory list and then we'll use the ***capacity*** to make sure that this memory list always contains **100** events and never more.

**2.10 step-7 Experience Replay class (part 2): push()**

* The push function will do two tasks.
* First it will append a new transition or a new event in the *memory* and
* Then it will make sure that the memory has always **100** transitions. In fact this will be much more than 100. But anyway this value will be the ***capacity***.

**def** **push**(self, event):

        self**.memory.append**(event)

**if** **len**(self.memory) **>** self.capacity:

**del** self.memory[0]

* This function will have two arguments.
* **self:** to attach to the object itself.
* **event:** will be our input and we will append this input into memory. Since this **push()** function will be used to append a new event into memory.
* We can actually call it event or transition. This event/transition that we're adding to the memory is a tuple of four elements.
* The first one is the last state that is .
* The second one is the new state that is .
* The third one is the last action that is , the action that was displayed.
* The fourth one is the last reword obtained that is .
* Let's say capacity is now 100,000. To make sure that the memory always contains **capacity** elements i.e. 100000 transitions (events).
* We'll set the limit using an IF-condition, if we go over the limit we'll delete the first transition of the memory.

**del** self.memory[0]

will remove the first element (oldest transition) from the list.

Next we'll define **sample()** function which will take some random samples from this memory of the last ***capacity*** *number of* *elements* and doing this will improve the deep Q-learning process.

**2.11 step-8 Experience Replay class (part 3): sample()**

**sample()** is used to get some random samples from our memory. This function takes two arguments as input.

* **self:** to attach to the object itself.
* **batch\_size:** batch\_size is a fixed size to get the random samples.

**def** **sample**(self, batch\_size):

        samples = **zip**(\***random.sample**(self.memory, batch\_size))

**return** **map**(**lambda** x: **Variable**(**torch.cat**(x, 0)), samples)

samples = **zip**(\***random.sample**(self.memory, batch\_size))

* ***samples*** variable contain the samples of the memory. To get these samples,
* First of all we have to take our memory **self.memory**, because we are getting the samples from our memory.
* We'll need the **batch\_size** because there will be **batch\_size** elements in our samples.
* Also we'll use a python trick cales "zip" to get a good format of these samples.
* **random.sample()** is from random library, it selects **batch\_size** number of elements from **self.memory**.
* **zip(\*):** It is just like reshape function. It reshapes our list. Eg: since each event or transition will be a tuple of *4-elements*

If ls = [(1, 2, 3, 4), (4, 5, 6, 7), (0 ,-1, -2, -4)] then

zip(\*ls) = [(1, 4, 0), (2, 5, -1), (3, 6, -2), (4, 7, -4)]

* Why we had to do it: Since each event in the memory have the form (state, action, reward). But for our algorithm we don't want this format we actually want our samples to have the following format:
* A format ***composed of three samples:*** One sample for the states, one sample for the actions and one sample for the reward.
* Then we'll wrap these batches/samples into **PyTorch** variables, variables that contain both a tensor and a gradient.
* To summarize, we creating one batch for each of the (state, action, reward) and then we're going to put each of these batches separately into some **PyTorch** variables which contain both a tensor and a gradient, so that eventually we'll be able to differentiate each of them.
* Return the samples: As we just explained, we *cannot return the samples directly*, we want to put the samples into a **PyTorch** variable.

**return** **map**(**lambda** x: **Variable**(**torch.cat**(x, 0)), samples)

* To do this for each of the samples, we're going to use the **map** function. This **map** function will do the mapping from the samples to Torch-Variables that will contain a tensor and a gradient.
* **map** function takes several arguments:
* The first argument **lambda** x: **Variable**(**torch.cat**(x, 0)) is a function. This function is going to be the function that will convert the samples into some Torch-Variables. **Variable**(**torch.cat**(x, 0) converts the sample into Torch-Variable (it converts the *torch-tensor* into a *tensor-with-gradient*). Here **x** represents the samples.
* There is one *last technical thing* that we need to implement. *For each batch which is contained in the sample* (eg. the batch of actions and the other actions) *we have to concatenate it with respect to the first dimension*.
* The first dimension corresponds to the States. It's just for everything to be well aligned.
* That is in each row the state, the action and the reward corresponds to the same time t.
* So that eventually we get a list of batches all *well-aligned* and each batch is a PyTorch variable.
* We need to use the **cat()** function from the *Torch library*.
* This ***Lambda-function*** will take the ***samples***, *concatenate them w.r.to the first dimension* then it ***convert*** the tensors into some ***Torch-Variables*** that contains both a tensor and a gradient so that later when we apply stochastic gradient descent, we will be able to *differentiate to update the weights*.
* The second argument will be the samples, on which we want to apply the Lambda function. We will apply this *Lambda function* on ***all the samples*** so that eventually we obtain a ***list of batches*** where each batch is a PyTorch variable.

The next and final class will be the whole Deep-Q-learning model. In that class we will have of course our ***network*** we'll also add the ***experience replay*** and then all the rest of the ***Deep-Q-learning algorithm***. We're going to make about ***10 functions***.

**All code at once (Before Deep-Q-Learning class)**

# *-------------    Deep Q-learning (Dqn)    -------------*

# *AI for Self Driving Car*

# *Importing the libraries*

**import** numpy **as** np

**import** random

**import** os

**import** torch

**import** torch.nn **as** nn

**import** torch.nn.functional **as** F

**import** torch.optim **as** optim

**import** torch.autograd **as** autograd

**from** torch.autograd **import** Variable

# *Creating the architecture of the Neural Network*

**class** Network(nn.Module):

**def** **\_\_init\_\_**(self, input\_size, nb\_action):

**super**(Network, self).**\_\_init\_\_**()

        self.input\_size = input\_size

        self.nb\_action = nb\_action

        self.fc1 = **nn.Linear**(input\_size, 30)

        self.fc2 = **nn.Linear**(30, nb\_action)

**def** **forward**(self, state):

        x = **F.relu**(self**.fc1**(state))

        q\_values = self**.fc2**(x)

**return** q\_values

# *Implementing Experience Replay*

**class** ReplayMemory(object):

**def** **\_\_init\_\_**(self, capacity):

        self.capacity = capacity

        self.memory = []

**def** **push**(self, event):

        self**.memory.append**(event)

**if** **len**(self.memory) **>** self.capacity:

**del** self.memory[0]

**def** **sample**(self, batch\_size):

        samples = **zip**(\***random.sample**(self.memory, batch\_size))

**return** **map**(**lambda** x: **Variable**(**torch.cat**(x, 0)), samples)