Chapter – 2 (implementation – Part 3 – DQN implementation)

**Artificial Intelligence**

**AI-app (Deep-Q): Self Driving Car** (part 3)

Class - DQN architecture:

select\_action; learn;

update; score;

save; load

**Deep-Q Learning class (the brain)**

**2.12 step-9 DQN class (part 1): \_\_init\_\_**

In this part, we'll implement the Deep-Q-learning model. We'll implement this in 8-steps.

* Basically we are about to *implement the whole process* of the Deep-Q-learning algorithm. Also we're going to use the classes/functions what we've created before (i.e. the architecture of the neural network, the replay-memory to integrate into the whole ***Deep-Q-learning*** process).
* This whole *Deep-Q-learning algorithm* is going to fit into one class. It's the last class that we're making to implement artificial intelligence. It will contain different functions.
* **\_\_init\_\_():** It will create and initialize all the variables attached to our future DQN objects which will represent the model itself
* **select\_action()** and **learn():** To select the right action at each time and learn the process.
* **update()** and **score():**Update function will update the rewards and score function will keep track of how the learning is going, if the *exploration is going well* and if it can move on to *exploitation*.
* **save()** and **load():** save function to save the brain of the car and load function will load the brain.
* **\_\_init\_\_()**: We first create our **Dqn()** class. In the **\_\_init\_\_** function we are going to introduce the variables attached to our object.
* We'll basically ***create*** and ***initialize*** all the variables that are needed to implement the Deep-Q-Learning network.

**class** Dqn():

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

        self.gamma = gamma

        self.reward\_window = []

        self.model = **Network**(input\_size, nb\_action)

        self.memory = **ReplayMemory**(100000)

        self.optimizer = **optim.Adam**(self**.model.parameters**(), lr = 0.001)

        self.last\_state = **torch.Tensor**(input\_size).**unsqueeze**(0)

        self.last\_action = 0

        self.last\_reward = 0

* We'll create an object of our *Deep-Neural-Network-structure* Network, we'll name this variable self.model.
* We will create another variable for the memory we call it self.memory, and we'll use our ReplayMemory class to create that object.
* Then we'll use ***Adam-optimizer*** to perform Stochastic Gradient Descent to *update the weights* according to how much they will *contribute* to the *error* when the *AI* is making a *mistake*.
* We'll also create some variables for the last\_state, last\_action, last\_reward.

The variables are from the Deep-Q-Learning algorithm.

* Arguments: In this **\_\_init\_\_** function we will put a couple of arguments. Whenever we create our Deep-Q-learning model by creating an object of the **Dqn** class, we need to specify following arguments.
* **self:** referring to our object itself.
* **input\_size** & **nb\_action:** Since the Network class takes input\_size (number of dimensions in the vectors that are encoding the states, i.e. the input states) and nb\_action (the number of possible actions the car can make i.e left/straight/right) as argument in this \_\_init\_\_ function. Those are needed when we creating an object of the Network class.
* **gamma:** It is the delay coefficient from the Q-learning equation.
* Since we'll also use a object of the ReplayMemory class to create the memory object, to get our *memory* of the *transitions*.
* Note that, this class also needs an argument: the **capacity**, but since we'll use this only once, we'll not specify it in the **\_\_init\_\_** argument, we'll set it directly.
* **self.gamma:** It's the delay coefficient. We get the value from the arguments.

self.gamma = gamma

* **self.reward\_window:** We make an empty list. The second argument is going to be the reward window.
* It's going to be the ***sliding window*** of the ***mean*** of the last ***100 rewords*** which will used to evaluate "the evolution of the AI performance".

self.reward\_window = []

* How to measure the performance: We'll have the mean of the reword into this reward-window (which will slide over time). We'll observe that the ***mean*** of the last ***100 rewords*** increasing with time or not.
* We will append the mean of the reward to this empty-list.
* **self.model:** It's the neural network, the heart of our AI model. We'll use the input\_size & nb\_action as the arguments.

self.model = **Network**(input\_size, nb\_action)

* **self.memory:** It’s the memory. It's going to be an object of the ReplayMemory class. We'll use 100000 as the capacity. i.e. one hundred thousand transitions into memory. We'll sample from this memory to get a small number of random transitions and that on which the model will learn.
* **optimizer:** optimizer is another variable of our future **Dqn** object and we use **torch.optim** to perform *stochastic-gradient-descent*. tortured Upton and it contains some optimizers
* Which optimizer?:
* **rmsprop** good for RNN for unsupervised deep learning.
* **Adam** is good for self-driving car, and we'll use it.

Notice **Adam** has capital-A, it means it's a class. We're actually creating an object of **Adam** class.

* Arguments:
* self.model.parameters(): Connects the Adam-optimizer to our NN 'self.model'. Basically we're taking all the parameters of our model object.
* lr: specify a learning rate. We'll take 0.001. If we set the learning rate too large, the AI won't learn properly.
* We want to give our AI some time to explore, learn from its *mistakes* (we'll *punish* it when it's making some *mistakes* like going onto some *sand* or getting *too close to a wall*).
* We want a way to the neural network to update correctly.

        self.gamma = gamma

        self.reward\_window = []

        self.model = **Network**(input\_size, nb\_action)

        self.memory = **ReplayMemory**(100000)

        self.optimizer = **optim.Adam**(self**.model.parameters**(), lr = 0.001)

        self.last\_state = **torch.Tensor**(input\_size).**unsqueeze**(0)

        self.last\_action = 0

        self.last\_reward = 0

* The three variables: last\_state, last\_action & last\_reward are used to composing our transition-events.
* **self.last\_state:** We'll initialize it.
* Remember, it is a vector of **5** dimensions (the three signals from 3-sensors: left straight and right and plus-orientation and minus-orientation) *encoding one state* of the *environment*.
* But for PyTorch it needs to be more than a vector, it needs to be a Torch-Tensor also it needs one more dimension (a fake-dimension) that corresponds to the batch.
* And that's because the last\_state will be the input of the neural network and for PyTorch or TensorFlow the *input vectors* cannot be a *simple vector by itself*, it has to be in a batch.
* The ***network*** can only accept a ***batch of input observations*** and therefore not only will create a tensor for input\_state vectors but also we will create this fake dimension corresponding to the batch.
* To create a **Tensor** object, we need to put an argument which will specify the size of tensor

self.last\_state = **torch.Tensor**(input\_size).**unsqueeze**(0)

* "**torch.Tensor**(input\_size)" to specify the number of elements of the Tensor, we'll use input\_size because the input\_size is exactly the *number of dimensions* of our input state vectors (Tensors).
* To create a fake dimension to represent batch, we'll use " **unsqueeze**(0)".
* The dimension for the batch needs to be the first dimension. That's why we used 0, indicates the first dimension. Other dimensions will correspond to the five elements of our input states: the *three signals*, *plus\_orientation* and *minus\_orientation*.
* **self.last\_action, self.last\_reward:** We initialized self.last\_action = 0 and self.last\_reward = 0.
* last\_action will be any index of the list **[0, 20, -20]**, which we defined at the beginning the action2rotation, 0, 20 degree, -20 degree.
* last\_reward is a float number between **1** and **-1** and we initialized it to **0**.

Next we do the most important thing for AI: deciding which action to play at each time t by creating the **select\_action()** method.

**2.13 step-10 DQN class (part 2): select\_action()**

Now, we're going to make the function that will select the right action at each time. Basically we're going to implement the part that will make the car do the *right move at each time*, i.e. going ***left***, ***straight***, ***right*** to reach the ***goal*** and to avoid the obstacles (i.e. sand).

* This **select\_action** will take two arguments.
* **self:** referring to our object itself.
* **state:** since the actions come from the output of the neural network (the Q-values for each of the three possible actions) and therefore the ***action*** depends on the ***input state***.
* We name it **"state"** it’s the ***input state***, because the *input* of the *neural networks* are the *input states* that are encoded by a vector of *five dimensions* (three signals, +ve\_orientation and –ve\_orientation).
* We're literally going to take the output of the neural network, which directly depends on the input of the neural network.
* We are going to feed the input state into the neural network (using previously defined **Network** class) then we're going to get the outputs the **Q-values** for each of the three possible actions and then using the **soft-Max** method we're going to get the **final action** to play.
* The idea of the soft-Max: The idea of the soft-Max is that we try to get the ***best action*** *to play at each time*, but at the same time we will be ***exploring*** *the different* *actions*.
* soft-Max consists of generating a ***distribution of probabilities*** for each of the Q-values: Q-State-Action.
* We have one Q-value for each action: left, straight or right. But this Q-value also depends on the input state (Q function is a function of the state and the action).
* So since we have one input state which we named state and we have ***three Q-values***  (Q\_state\_action\_1, Q\_state\_action\_2, Q\_state\_action\_3) for three possible actions.
* We are going to generate a distribution of probabilities w.r.to these three ***Q-values***. Each *Q-value* will have a *probability* i.e. *three probabilities* for *three Q-values* and *all the three probabilities will sum up to 1*.
* We're going to do all this with soft-Max and *soft-Max will attribute a large probability* to the highest Q-value.
* argmax: An alternative to soft-Max is a simple argmax: it directly takes the maximum of the Q-values and not exploring the other actions.
* But due to the probabilities of Soft-max we can explore somewhere else using a temperature parameter.
* That's why for ***Deep-Q-learning*** Soft-max is the best option rather than argmax. Now let's focus on implementation:

**def** **select\_action**(self, state):

        probs = **F.softmax**(self**.model**(**Variable**(state, volatile = **True**))\*100) # *T=100*

        action = **probs.multinomial**()

**return** action.data[0,0]

* **probs:** It contains the probability. Since ***Soft-max*** returns the *probabilities* of each of the three Q-values for the three possible actions.
* We'll get the **softmax()** function from **torch.nn.functional** (functional module contains most of the actions to implement a neural network) module.
* Now we apply the Q-values, we'll use our ***defined Neural-Network***, which we initialized at ***self.model*** inside **\_\_init\_\_**.
* Next we apply the input state to this ***model*** as the argument
* We need to transform the **input state** using **Variable**, because currently it's Torch-tensor, later we'll use ***self.last\_state*** as an argument of ***select\_action()*** function and this ***state*** argument will be ***self.last\_state*** . However ***state*** being a Torch-tensor the model will accept it.
* But now we can improve the algorithm.
* We wrapped our ***most tensors*** into Torch-Variable to contain a gradient, but since ***state*** is our input sate and there will be no differentiation.
* So we only wrap state by torch-Variable but we ***won't include any gradients*** to the graph of all the computations of the nn.module, hence we'll apply volatile = True. It'll save some memory and improve performance.
* Temperature Parameter (T=100): This temperature parameter is the parameter that would allow us to ***modulate*** how the ***neural network will be sure*** of which action it should decide to play.
* This parameter will be a positive number and:
* The closer it is to zero ,the neural network will be less sure when playing an action
* Higher the temperature parameter, neural network will be more sure when playing an action
* To *add this parameter*, we simply multiply the outputs "self**.model**(**Variable**(state, volatile = **True**))" *(which are the Q-values)* by this ***temperature*** parameter.
* Now let's consider an example with temperature = 7, i.e., "self**.model**(**Variable**(state, volatile = **True**))\*7" where temperature T =7.
* We're going to try some other values but I just want to start with a small one because you're going to see that with a ***small temperature value***, our car will still behave like some kind of an insect.
* By increasing the temperature parameter, our car will look more like a car and the self-driving will be much better.
* That makes sense because the higher is this temperature parameter, the higher will be the probability of the winning Q-value.
* For example if we have **softmax** of the Q-values **(1, 2, 3)** resulting **[0.04, 0.11, 0.85]**

Softmax((1, 2, 3)) = [0.04, 0.11, 0.85]

* Right now **temperature = 1**, by taking a high temperature for example **temperature = 2** and apply it in the ***softmax*** we'd get following result:

Softmax((1, 2, 3)\*2) = [0, 0.02, 0.98]

After applying the **softmax**: lower probabilities like (0.04, 0.11) will decrease and higher probability 0.85 will be increased.

Since (1, 2, 3)\*2 = (2, 4, 6) i.e. the difference between 1 & 3 was 2 but now the difference between 2 & 6 is 4. More-over we're applying softmax over (2, 4, 6) to the *total probability* = 1. That's how we'd get [0, 0.02, 0.98] result after applying softmax.

* **0** for the first Q-value : **2** because this had a very low probability, for the second probability **0.02** because this was still a low probability.
* But the ***third probability*** was the ***largest***, increasing the temperature this probability will be even larger make sure this is the right Q-value corresponding to the action we must play and therefore this is going to be something like **0.98**.
* By increasing the **temperature** parameter, we are now even more sure that the ***third action*** should be the ***action to play*** because the probability for the Q-value of this action is *not only the largest one but also very high*.

So that's what this **temperature** parameter is all about. It's about the certainty of which direction we should decide to play.

* **action:** The principle of the soft-max method is not only to generate a probability distribution (first step) for each of the Q-values but also take a random draw from this distribution (second step of the soft-max method) to get our final action.

action = **probs.multinomial**()

**return** action.data[0,0]

* Of course we'll have a high chance to get the action that corresponds to the ***Q-value*** that has the highest probability because that's exactly how the distribution works.
* Since probs is our *probabilities of each of the* ***Q-values***, we simply make a random draw from this distribution. To get a random draw we're going to use multinomial() function.
* Lastly we return the action. There is a little trick here. Since probs.multinomial() returns a ***Pytorch-Variable*** with a *fake batch* *(fake dimension corresponding to the batch)*, therefore to get the right result (actions) we need to specify the index.

The next function we gonna create is the learn() function. That's where we will train the whole Neural Network with *forward propagation* and *back propagation* using ***stochastic-gradient-descent***. Basically we will implement the whole training of the Deep Q-learning model i.e. the heart of our AI.

**2.14 step-11 DQN class (part 3): learn()**

In this function we'll implement the Deep Neural Network that is inside our AI. We'll going to do the *forward propagation* and *back propagation*. We already did following steps in Deep-Learning.

* First we're going to get our output & target.
* We'll compare the output & target to compute the loss error
* then we're going to back propagate this loss error into the new network.
* Using ***stochastic-gradient-descent*** we'll update the *weights* according to how much they *contributed* to the *loss error*.
* Arguments: **learn()** function is going to take several arguments.

**def** **learn**(self, batch\_state, batch\_next\_state, batch\_reward, batch\_action)

**self**: refer to the object itself of the **Dqn()** class.

**batch\_state**: It refers to the current state.

**batch\_next**\_state: refers to the next state

**batch\_reward**: refers to reward

**batch\_action**: refers to action

* These arguments are for the transitions of the Markov-decision-process. The *Markov-decision-process* is the *base* of the *Deep-Q-Learning*.
* We take all of them into some batches, because we don't consider the transitions by a series of the tuple: current\_state, next\_state, current\_reward and current\_action.
* Since we created some ***sample batches*** in the sample() *function definition*.
* And so now our transitions are in the form of: *first batch* for the current-state, a *second batch* for the next state, a *batch* for the reward and a *batch* for the action.
* Also they're all well aligned w.r.to time because we also made a ***concatenation*** in the sample() function definition (**torch.cat** in the lambda w.r.to the *first dimension* i.e. 0 index).
* So the point is: now we have these transition of batches, one batch for each of the state, next state, reward and action and we do all this because we're using this experience replay trick so that our Deep-Q-Neural-Network can learn something.
* Remember, if you only had a transition's by themselves (no batches), it would be some instant learning. Or, very short memory learning and therefore the *model wouldn't learn anything*.
* So we have to take these batches from the memory, which become our ***transitions*** and then eventually we will get the ***different outputs*** for ***each*** of the ***states*** of the ***input batch states*** and we will do this for the ***batch\_states*** and for the ***batch\_next\_states*** because we will need *both* to *compute* the loss.

**def** **learn**(self, batch\_state, batch\_next\_state, batch\_reward, batch\_action):

        outputs = self**.model**(batch\_state).**gather**(1, **batch\_action.unsqueeze**(1)).**squeeze**(1)

        next\_outputs = self**.model**(batch\_next\_state).**detach**().**max**(1)[0]

        target = self.gamma\*next\_outputs + batch\_reward

        td\_loss = **F.smooth\_l1\_loss**(outputs, target)

        self**.optimizer.zero\_grad**()

**td\_loss.backward**(retain\_variables = **True**)

        self**.optimizer.step**()

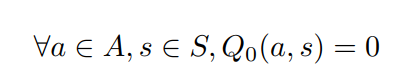
* outputs of the batch\_state: We call this variable outputs and then we're going to use our self**.model()**, with batch\_states as an argument because we want to get our model outputs of the *input states* of the batch\_states.
* Since our model is actually expecting a *batch of input states* we input batch\_states for the ***input*** of the model.
* That's how we ***initialized the states*** that are going into the *network* with *Torch-tensor* with a *fake-dimension* (for the batch). Then we get the outputs from the model.
* Another technical trick: If we only do self**.model**(batch\_state), we'll get the outputs of all the possible actions (0, 1 and 2) but that's not what we want. We are only interested in the actions that were chosen, the actions that were *decided* by the *network* to *play at each time*.
* To get these we have to use the gather() function.
* Arguments of gather():
* **1**: Because we only want the action that was chosen
* **batch\_action.unsqueeze(1))**: We only want the action that is played (the *action that is* *chosen*) and gather(1, batch\_action.unsqueeze(1)) will gather each time the best action to play for each of the input\_states of the batch\_state.
* Be careful, batch\_state has ***fake dimension*** corresponding to the batch and batch\_action doesn't have it. That's why we have to use unsqueeze(1) on batch\_action. Because we haven't use any unsqueeze for actionsto create any fake dimension.
* Now the batch\_action has the exact same dimension as the batch\_state.
* Note: **0** represents fake dimension of the state (unsqueeze(0)) and **1** represents fake dimension of the actions (unsqueeze(1)).
* Finally the last thing we need to do is to kill this fake batch with a squeeze.

outputs = self**.model**(batch\_state).**gather**(1, **batch\_action.unsqueeze**(1)).**squeeze**(1)

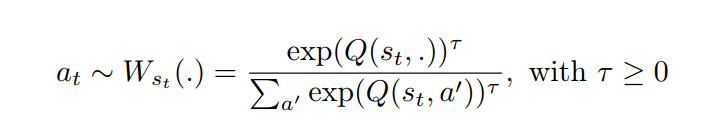
* Why do we need to do that? Because now we are *out of the neural network*.
* We have our *outputs* but *we don't want them in a batch*. We want them in a simple Tensor (a simple vector of output).
* The batches used when we work in the ***Neural Network*** because the ***Neural Network*** *is expecting* the format of ***Tensors* *into a batch***.
* In the next ***balance equation*** of the ***Deep Q-learning*** we won't need them into a ***batch***. Hence we're *killing* the *fake dimension* to get back the *simple form* of our *outputs*.
* We want to kill the fake dimension corresponding to the batch of the action, since this fake-dimension has **index 1** we used **squeeze**(1).

Now we have our outputs.

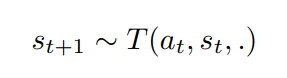
* **next\_outputs**: Why do we need the next\_outputs? To understand this we need to go back to the Deep Learning Algorithm lets go back to our Intuition Handbook. So following is the whole Deep Q-learning process:
* First we initialize the Q-values: At the beginning we were initializing all the Q-values to 0



* At each time **t** we select the action with softmax (we did this with the select\_action() function)



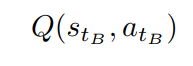
* Here is the transitions:



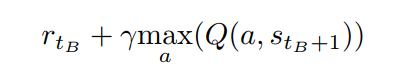
* Then we append the transition



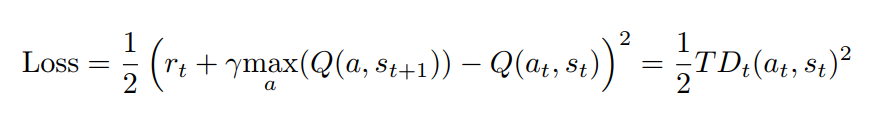
* we get the prediction

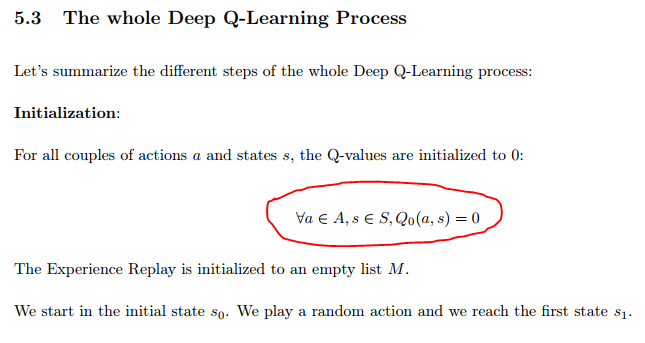


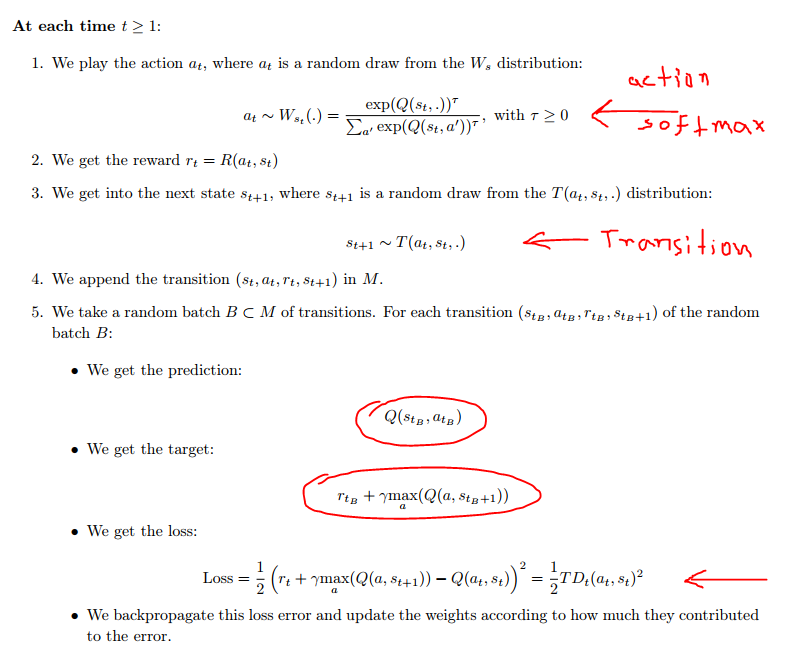
* we get the target



* we compute the loss







* To compute the target we need the next\_output. Since, the target is equal to Gamma times the next\_output plus the reward.

next\_outputs = self**.model**(batch\_next\_state).**detach**().**max**(1)[0]

* The next\_output is going to be the result of our neural network when the batch\_next\_state is entering it as *input*. So we used self**.model**(batch\_next\_state).
* model: is our neural network
* batch\_next\_state: is the input of the neural network
* Let's go back to ***Deep-Q-learning algorithm***, you can see that the next output is the maximum of the ***Q-values*** for the next state with respect to all the actions a.
* To get the next\_output we need to get the maximum of the ***Q-values***.
* We first detach all the outputs of the model (because we have several states in this batch\_next\_state, the batch of all the *next states* in *all the transitions* taken from the ***random sample*** of our memory).

self**.model**(batch\_next\_state).**detach**()

* Then we're taking the ***maximum*** of these ***Q-values*** w.r.to the ***action***. Since the ***action*** is represented by **index 1**, we have to specify this index in **max()**.

self**.model**(batch\_next\_state).**detach**().**max**(1)

* To specify that we're taking the ***Q-values*** of the next state , we have to use **index 0**, because the *index zero* corresponds to the ***states***.

self**.model**(batch\_next\_state).**detach**().**max**(1)[0]

That way we get the maximum of the Q-values of the next states represented by **index 0** according to all the actions that are represented by the **index 1**.

The balance equation: The *balance equation* is at the heart of the Deep-Q-learning algorithm.

Next we'll use this next\_output to compute the target.

|  |  |
| --- | --- |
| * Target: The target is equal to reward + gamma\*next\_output. The next output is the maximum of the ***Q-values*** of the next state according to the actions |  |

* Following is the target (in one sample of the memory):

target = self.gamma\*next\_outputs + batch\_reward

* self.gamma was initialized in **Dqn**-class definition at the beginning.
* Loss: Now we have our outputs and our targets and therefore we can compute the **loss** representing the *error* of the *prediction*.

td\_loss = **F.smooth\_l1\_loss**(outputs, target)

* We call it td\_loss because ***td*** is for the temporal difference. This is going to be **Huber-loss** because it improves **Q-learning**, we will choose it for our artificial intelligence.
* To get this **Huber-loss** we'll use the Functional module "F" (we renamed), and we'll use **smooth\_l1\_loss**() function to get this loss.
* The arguments we need to input are the predictions (output of the neural network) and the targets (the thing we're trying to get), which are outputs and target respectively.
* Back-Propagation: Now we have the loss error, we can back propagators this error back into the ***network*** to ***update*** the ***weights*** with stochastic gradient descent.
* We take our self**.optimizer** which is the **Adam** optimizer and we initialized in **\_\_init\_\_** it is already fitted with the parameters of our model. We already chose a learning rate of 0.1 percent and our optimizer is ready but now we need to apply it on the loss error to perform stochastic gradient descent to update weights.
* But when we're working with PyTorch, we need to re-initialize the *optimizer* at each iteration of the loop. We must *re-initialize the optimizer* from one interaction to the other in the loop of the *stochastic gradient descent* .
* To re-initialize it at each iteration of the loop, we use zero\_grad() method.

self**.optimizer.zero\_grad**()

* After this re-initialization, we can perform backward propagation with our optimizer.

**td\_loss.backward**(retain\_variables = **True**)

* We'll take our **td\_loss** (object of **F.smooth\_l1\_loss**) and to back propagated into the network we need to use the backward() function and inside this function we set retain\_variables = **True** it improve the training performance (it will free some memory and we need to free the memory because we are going to go several times on the loss).
* update the weights: Finally the last step of this **learn()** function is to *update the weights* according to the ***back propagation*** (i.e. according to how much the weights contributed to the error).

self**.optimizer.step**()

* To do this we take our self**.optimizer** again (which was initialized in re-initialized) and we use the step() function.
* **td\_loss.backward**(retain\_variables = **True**) *back-propagates* the *error* into the *neural network* and
* self**.optimizer.step**() uses the optimizer to update the weights

self**.optimizer.zero\_grad**()

**td\_loss.backward**(retain\_variables = **True**)

        self**.optimizer.step**()

Finally, we have a learning neural network.

* Remember, PyTorch can be tricky sometimes with these un- squeeze and squeeze but in the end you will get a very functional neural network and therefore Deep-Q-learning model and eventually a great artificial intelligence.
* Next we'll implement the update() function that will obviously update when the AI will discover the new state.
* When it will discover the ***new state***, it will ***receive*** the ***reward*** depending on the action that it just played and this new state.

**2.15 step-12 DQN class (part 4.1): update()**

The update() function will update everything as soon as the AI-agent reaches a new state. When it reaches new state we need to update

* **action**: The *last action* becomes the *new action* that was just played.
* **state**: the *last state* becomes the *new state*
* **reward**: the *last reword* that becomes the *new reword* we get when we play the action.

Above is the logical path that happens *right after selecting* an *action*.

* **transitions**: We need to *update all the elements* of the *transitions*.
* Of course we'll get ***new transitions***, we have to ***append*** this *new transitions* to the ***memory***.
* **reward window**: And finally we will also update our *reward window* to keep an eye on the evolution of how the training and the exploration is going.
* *But the most important thing is:* we can finally make a connection between the AI *(ai.py file)* that we're implementing right now to our map *(map.py file)*.
* Remember, in the **map.py** there is also an update() function into the game class and that's where we were actually making the game with the car and defining how the *car should be punished* when it's making a mistake.
* And in this function of **map.py**, there is a line

action = **brain.update**(last\_reward, last\_signal)

* And actually this is exactly what we're about to make in this Dqn class of ai.py file.
* We are about to make this **update()** function inside **Dqn** *class* that will take the last reword and the last signal to get the next action to play.
* It not only ***updates*** all the *different elements* of the ***transition*** but also it'll be *playing the* ***action*** that we should play when getting the *last reword* and the *last signal*.
* So, in this update() function we will use the select\_action() function that we implemented before, we will integrate the select\_action() function to select the *right action to play* *besides making all the* ***updates***.
* Connect the AI and the Game: Now it's time to make the connection with the map.py right now.
* We'll make the connection between our AI and the game (the game that we make in the Game class in map.py).
* It's the action = **brain.update**(last\_reward, last\_signal) line in map.py file that makes the connection between our AI and the game. Notice the brain is the object of our **Dqn** class.
* Now we're going to implement this **update**(last\_reward, last\_signal) function in the Dqn class in our ***ai.py*** file.
* Arguments: Function takes 2 arguments- ***reward*** and ***new\_signal***

**def** **update**(self, reward, new\_signal)

* **self**: refers to the object itself of the **Dqn()** class.
* **reward**: We've changed this name to ' reward ' because we've already used the name 'last\_reward' in our \_\_init\_\_() function of the **Dqn** class we don't want to confuse with this one. Note that, this argument will be 'last\_reward' but in the map.py file inside the **Game** class.
* **new\_signal**: same for last\_signal, we rename it to new\_signal to specify that you know after the update we reaching a ***new state*** and therefore getting a ***new signal***. This argument also will be 'last\_signal' in the map.py file inside the **Game** class.
* The 'last\_reward' and last\_signal are going to be the input of the update function in the map.py file's **Game** class these are updated when Car is going onto some sand or getting too close to one edge of the map.
* But in ai.py file's Dqn class's update() function, we're just giving another name for the argument. We want to not confuse it with last\_reword defined in the \_\_init\_\_ function here.
* In this update function we'll do two things:
* Update all the **elements** of our **transition** and
* **select** the **action**.
* Updating new\_state: What do we need to update first? We want to make update when reaching a new state.
* So the first thing we'll be updating is obviously the new\_state that *we're reaching*.
* How can we get this new state? That depends on the *new signal* that the *sensors* just detected.
* And as a reminder *the* ***state*** *is the* ***signal*** *itself* composed of the three signals of the sensors signal\_1, signal\_2, signal\_3, +ve orientation and -ve orientation. That's our state.
* So be sure to understand that the ***signal*** *is the* ***state***. But right now it is a simple list of five elements.
* And since this is going to be the *input* to the ***Neural Network***, we have to *convert* it into a ***torch-Tensor***.
* Also it's better to make sure that all the elements of the torch-Tensor are floats, so we've used a type conversion.
* Lastly we create the fake dimension correspond to the batch and we used unsqueeze() function with **index 0**.

new\_state = **torch.Tensor**(new\_signal).**float**().**unsqueeze**(0)

Now we have our *new states* composed of the *three signals* of the *three sensors* and *+ve orientation* and *-ve orientation*

Of course this ***new states*** will depend on the ***new signal*** we are getting with the update() from map.py, *(the three signals are the density of sand detected around the sensors)*.

* Updating memory: Now we just got our new state, that means *we've reached the new state* and now we have to make the *next update*.
* We need to update now the memory. Why is that?
* It's because at each time **t** a transition is composed of the ***current state*** the ***next state*** the ***reward*** and the ***action*** .
* Right now we already have , and and we just got the last element of the transition .
* By getting this ***new state*** , we are getting a *brand new transition of the memory* and therefore we have to *append* this *brand new transition* to the *memory* because that's simply our ***next transition***.

So that's why we have to update the memory right now.

* Therefore we take our memory object created from the replay\_memory class and therefore we're going to take ***self.memory*** to refer to the object.
* How we're going to update that?
* Well the good news is that we already made a function to do that. It's the push() function which appends an event or a transition to the ***memory***.
* We're going to use the push() function to append our new transition that we just made to the memory
* **self.last\_stae**: It is our *last state*.
* **new\_state**: our *new transition* that we just got, the *new state* that we just reached. Notice, it is *not a variable of the* *object* that we created and initialized in this \_\_init\_\_() function, hence we don't put "self" here.
* **self.last\_action**: It is the action. Notice, we had an action, initialized to 0, but now it'll be updated with the select\_action() function.
* We need to convert it to the integer. The ***elements*** that were including in this ***transition*** should all be *Torch-tensors* (all last\_state, new\_state are torch-tensors, so the actions and reward must be torch-tensors).
* Now the question is: How can it be a torch-tensor considering that it's simply a number? We know that the ***action*** is either ***0***, ***1*** or ***2***. We can still convert this **0**, **1** or **2** into a **torch-tensor**. This is what we call a long tensor.
* The long is a type and that tensor will contain an integer because the last\_action is an *integer:* **0**, **1** or **2**.
* Note that we must be sure that there is integer inside this **LongTensor**, to do that we convert the value into an integer (type-conversion) **int**(self.last\_action).

**torch.LongTensor**([**int**(self.last\_action)])

* We put it into '[]' brackets (we have to put the number into a list and then this list will go as input into **torch.LongTensor** class) so that we get a *long tensor of one element* which will be this last\_action **0** or **1** or **2** itself.

So we used **torch.LongTensor()** class that will create an object of LongTensor but it will still contain 0, 1 or 2. Now it is consistent with the transition that only contains *tensors* because we're working with *Pytorch* and we're working with a *neural network*. So we have to work with Tensors.

The key point is that's how you convert a simple number into a tensor with **torch**.

self**.memory.push**((self.last\_state, new\_state, **torch.LongTensor**([**int**(self.last\_action)]),

**torch.Tensor**([self.last\_reward])))

* Finally the *last element* of the transition is the **last\_reword** we got, in **\_\_init\_\_** function it was also initialized to **0**.
  + But then of course is updated: **-1** is the *worst reward* if it goes *further away from the goal*, also it got some *punishment* if it goes *near sand* or *barrier*. If it gets closer to the goal a *positive reward* will be given.
  + Now we have to make another conversion, same as previous one, but now it's a float number, that's why we'll simply make a **torch.Tensor** instead of **torch.LongTensor**. Also we'll use '[]' to put the number into a list and then this list will go as input into **torch.Tensor** class.

**torch.Tensor**([self.last\_reward])

To summarize: With the new\_state we just reached and the reword we observe a *new event of transition* that we *add to the memory*. And this transition contains the last\_state and the new\_state the last\_action played and the last\_reword.

Then we are going to update the memory by push() function.

**2.16 step-13 DQN class (part 4.2): update()**

We just updated the memory after reaching the new state. And now let's take care of the next update.

* Basically we're *done with one transition* we have updated the *last element of the transition* which is the new\_state.
* Now it's time to play an action because we already got the observation of the news States. We use the select\_action() function to play the action.
* This select\_action() function is a method of the object of the **Dqn** class, since this action function takes the current state as input. Right now our current state is the new\_state,

self.last\_action = action

with this line of code we simply play the new action after reaching the new state.

* Now we played an action, we get the ***reward*** and therefore we get a ***feedback*** with the ***reward***.
* At this point, if we have *more than 100 elements* in the *memory* it would be ***time to learn***. So *after selecting an* ***action*** the AI needs to *start learning*, if it's doing the things in the right way.
* Now since *it just played* the action, we're *going to make the AI* ***learn*** from its ***actions*** in the ***last 100 events***.
* ***If-condition:*** But before we apply this **learn()** function of Dqn class, we apply an if ***condition*** to make sure that we already have reached ***more than 100 events***, because we're learning from the random samples of the memory of **10000** elements.
* The ***AI*** is ***learning*** from the *information contained* in this sample of ***100******random transitions***.

**if** **len**(self.memory.memory) **>** 100:

* Notice the trick 'self.memory.memory', first portion self.memory is the variable of Dqn class defined in \_\_init\_\_, which is an object of ReplayMemory class. The extra memory is actually the attribute from the ReplayMemory class.
* Random Samples: Before start learning we need to get this random sample of **100** transitions so we use our **sample()** function.
* Since the sample() function returns the different batches to states at time t, states at time t+1, the actions at time t and the rewards at time t.
* So we first create some variables to store these values: which are going to be the *batch of the* ***states*** *at time t*, the *batch of the* ***next******states***, the *batch of the* ***rewords*** and the *batch of the* ***actions*** and we can simply give the same names:
* These variables are returned from sample() as batches, so we applied sample from our memory object with 100 transitions self.**memory.sample**(100):

**if** **len**(self.memory.memory) **>** 100:

            batch\_state, batch\_next\_state, batch\_action, batch\_reward = self**.memory.sample**(100)

So it's going to learn from 100 transitions of the memory.

* Learning: since the learn() method is a method of our **Dqn** class, we have to use self. learn() takes the *four previous values* as four ***arguments***.

            self**.learn**(batch\_state, batch\_next\_state, batch\_reward, batch\_action)

* batch\_state, batch\_next\_state, batch\_reward, batch\_action are the *batches* sampled from the **memory** and we get 100 of them because we have 100 transitions.

There we go, now the learning will happen. It will happen from all these random batches.

**if** **len**(self.memory.memory) **>** 100:

            batch\_state, batch\_next\_state, batch\_action, batch\_reward = self**.memory.sample**(100)

            self**.learn**(batch\_state, batch\_next\_state, batch\_reward, batch\_action)

* Updating state, action & reward: Now we do the *very last updates* after reaching a new state and playing an action.
* Well we got the action to play but we still didn't update that action i.e. our sefl.last\_action variable. We assign it to the value: action that we got from select\_action() function.

self.last\_action = action

* Next we update sefl.last\_state because we didn't update the state yet. We *reached the new state* but we haven't updated the last\_state yet, because the last\_state was before the *state at time* ***t***.

self.last\_state = new\_state

* But since now we reached the new state at time ***t+1***, the last\_state becomes this new state.
* Also we now update the reward it's the argument of this **update()** function which is connected to our map.py.
* Notice inside the **Game** class we have following code:

action = **brain.update**(last\_reward, last\_signal)

* last\_reward is the reword we get *right after playing the action* in this reached *new states*.

*(So if we go on to some sand this last\_reward will be that* ***-1****, if we get further from the goal we will get a slightly bad word minus* ***-0.2****, if we get closer to the goal we will get a slightly good reward* ***0.1****. And if we get too close to an edge of the map that will be a bad punishment we'll get* ***-1****.)*

* So that's the last reword we get in reality i.e. when that happens for real on the map. And this will be the argument of the **update()** function, the value is passed to the ai.py's update() from map.py.

self.last\_reward = reward

So we update sefl.last\_reward by the reward coming from map.py in reality after taking an ***action*** ending to a ***new state***.

        self.last\_action = action

        self.last\_state = new\_state

        self.last\_reward = reward

* Updating Reward Window: Since we have our last reward, we can update our **reward\_window**. We initialized **reward\_window** in **\_\_init\_\_** function of our **Dqn** class as an empty list.
* The **reward\_window** is going to keep track of how AI's training is going by taking the ***average*** of the ***last 100 reward***.
* **reward\_window** will be like a sliding window showing us how the mean of the reward is ***evolving***.
* How do we update it? We simply append our **last\_reward** to the window.

        self.reward\_window.append(reward)

* this reward\_window is going to have a ***fixed size*** sliding with time to show the *evolution of the* ***reward***.
* We give it a fixed size, i.e. the ***number of means*** of the reward we will have in this window, for example let's get the last **1000** means of the **last 100 rewards**.
* To do that, we will delete the *first element* if **reward\_window**'s length is greater than **1000**. We make sure that this **reward\_window** will never get more than 1000 elements.

**if** **len**(self.reward\_window) **>** 1000:

**del** self.reward\_window[0]

* i.e. one thousand means of the **last 100 rewards** so that we can see if the mean of the reward is increasing and the training is going well.
* Return the played action: Now this update() function not only updates the different elements of the transition and the reward-window but also it ***returns the action*** that was ***played*** when reaching this ***new state***.
* That's why we have following in the map.py

action = **brain.update**(last\_reward, last\_signal)

therefore our **update()** function in **ai.py** is supposed to return something and it is of course the action.

* So lastly we return the action from update() function that was just played when *reaching* the new state.

**def** **update**(self, reward, new\_signal):

        new\_state = **torch.Tensor**(new\_signal).**float**().**unsqueeze**(0)

        self**.memory.push**((self.last\_state, new\_state,

**torch.LongTensor**([**int**(self.last\_action)]), **torch.Tensor**([self.last\_reward])))

        action = self**.select\_action**(new\_state)

**if** **len**(self.memory.memory) **>** 100:

            batch\_state, batch\_next\_state, batch\_action, batch\_reward = self**.memory.sample**(100)

            self**.learn**(batch\_state, batch\_next\_state, batch\_reward, batch\_action)

        self.last\_action = action

        self.last\_state = new\_state

        self.last\_reward = reward

        self**.reward\_window.append**(reward)

**if** **len**(self.reward\_window) **>** 1000:

**del** self.reward\_window[0]

**return** action

After that our update() function is ready. It's going to do all the required updates and it will return the action when reaching the new state.

Next, we'll make a score() function to return the means of the rewards in the reward window. Then we will make a save() function to save the brain of the car whenever you want to *quit the application*. Also we'll make a load() function to load the brain of your car (which was previously saved) when you get *back to the application*.

Finally we'll play a demo, where we'll see if the AI works. We will see if the car reaches the goals and we will see how we can improve it and then eventually you will have to build your first AI.

**2.17 step-14 DQN class (part 5): score()**

* score() function will just compute the score on the sliding window of the reward. Basically we'll simply compute the mean of all the rewards in the reward-window.
* This function doesn’t need any arguments other than self.

**def** **score**(self):

**return** **sum**(self.reward\_window)/(**len**(self.reward\_window)+1.)

* sum(self.reward\_window)computes sum of all the reward.
* len(self.reward\_window)Gets the total number of elements.
* Since we're dividing with the length of a list, it could be 0, and dividing by 0 gives math error. To avoid that error we add an extra 1, so that the denominator will never be 0.
* We'll still get a good measure of the score.

Next we'll implement **save()** function, that'll save our model.

**2.18 step-15 DQN class (part 6): save()**

Now we'll make a function save() that will save our model (i.e. the brain of the trained car will be saved) *so that we can we use it* whenever we *quit* the application.

* Then the load() function (we'll make it in the next section) will load the last version of our model that was trained.
* This function doesn’t need any arguments other than self.
* We *do not save the whole model*, we only save the neural network 'self.model' and our optimizer 'self.optimizer'. Because we want to save just the ***last weights*** that were ***updated*** at the ***last iteration***, because whenever we want to *reuse* our *saved model*, we want to ***predict*** the ***action to play*** with the ***weights*** that were ***already trained***.
* So we need to take this last version of the weights and also we need the last version of the optimizer because it is connected to these weights.
* We will be saving these two objects 'self.model' and 'self.optimizer' in a ***Python dictionary***. To save these two objects, we're going to use *the* **save()** *function* from the **torch** module, torch.save().
* We're going to make two keys,
* one key 'state\_dict' for the 'self**.model**'
* another key 'optimizer' for the 'self**.optimizer**'
* We've used **state\_dict**() to save the parameters of our *model* and the *optimizer*. Following is our dictionary:

{

'state\_dict': self**.model.state\_dict**(),

'optimizer' : self**.optimizer.state\_dict**(),

}

That will save all the weight and our optimizer.

* Finally, we'll save all this into a file and to do this we're going to add a second argument to the torch.save() function. This argument is going to be the name of the file where we want to have our ***model*** and ***optimizer***.
* This file is a '**.pth**' type file, and we name our file "last\_brain.pth".

**def** **save**(self):

**torch.save**({'state\_dict': self**.model.state\_dict**(),

                    'optimizer' : self**.optimizer.state\_dict**(),

                   }, 'last\_brain.pth')

Now we have a **save()** function that will save our model (saving the brain of our AI car) by saving the weights and the optimizer of the *Neural Network*.

Next, we'll make a load function to load the saved brain of our car.

**2.19 step-16 DQN class (part 7): load()**

Now we'll implement the **load()** function, it will load the saved brain (saved previously using save() function) of our AI car.

* This function doesn’t need any arguments other than self.
* This **self** will load what was saved using the save() function. So we will load self.model and self.optimizer. So the **self** here will be for the **model** and the **optimizer**.
* Since the model is saved in the file 'last\_brain.pth', we want to make sure that- this file exist.
* We'll apply an if-condition to make sure that 'last\_brain.pth' file exists and if it exists we will be loading what we have in the dictionary stored inside 'last\_brain.pth'.
* os.path is the path that *leads* to the *working directory folder*. isfile will return **True** if the file 'last\_brain.pth' exists and **False** if it doesn't.

**if** **os.path.isfile**('last\_brain.pth'):

**print**("=> loading checkpoint... ")

* Inside the block, we'll print "=> loading checkpoint... ".
* To load the dictionary containing the model & the optimizer, we'll use **torch.load** and our file name 'last\_brain.pth' as argument. We'll load it in the variable 'checkpoint ':

checkpoint = **torch.load**('last\_brain.pth')

* Now we loaded the model and the optimizer. Next thing to do is update our model and the optimizer separately, because actually we loaded the ***parameters***, the ***weights*** and the ***parameters of the optimizer***.
* So now we need to update our existing model: self.model and our existing optimizer: self.optimizer with the parameters & weights that are in this 'last\_brain.pth' file.
* So we simply need to make these two update separately and to do this we're going to use a method load\_state\_dict() from the torch modules (there's going to be *inheritance* which will allow us to *use this method*).
* This method will allow us to update all the parameters of our model self.model and our optimizer self.optimizer.

self**.model.load\_state\_dict**(checkpoint['state\_dict'])

* since the self.model *inherits the methods* of the torch module it allows us to use **load\_state\_dict**()method.
* Using this method we are going to update all the parameters of the model that is all the ***weights***. Here 'state\_dict'is the key of the dictionary.
* And now we need to do the same for the optimizer. And this time we're updating our optimizer self.optimizer, in this case 'optimizer' is the key

self**.optimizer.load\_state\_dict**(checkpoint['optimizer'])

* then just to finish we can print "done" **print**("done !").
* And finally we use an 'else' statement if the 'last\_brain.pth' file doesn't exist.

else:

            print("no checkpoint found...")

**def** **load**(self):

**if** **os.path.isfile**('last\_brain.pth'):

**print**("=> loading checkpoint... ")

            checkpoint = **torch.load**('last\_brain.pth')

            self**.model.load\_state\_dict**(checkpoint['state\_dict'])

            self**.optimizer.load\_state\_dict**(checkpoint['optimizer'])

**print**("done !")

**else**:

**print**("no checkpoint found...")

Finally we finished our **Dqn class**. Congratulations!! We have a functional **Dqn class**. And our artificial intelligence is ready you can probably

We just made a brain and we're going to put this brain in the car and we will see how it is clever enough to do these round trips between the airport and downtown wherever the road is. (1.1.24)

**All Dqn class code at once**

# *Implementing Deep Q Learning*

**class** Dqn():

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

        self.gamma = gamma

        self.reward\_window = []

        self.model = **Network**(input\_size, nb\_action)

        self.memory = **ReplayMemory**(100000)

        self.optimizer = **optim.Adam**(self**.model.parameters**(), lr = 0.001)

        self.last\_state = **torch.Tensor**(input\_size).**unsqueeze**(0)

        self.last\_action = 0

        self.last\_reward = 0

**def** **select\_action**(self, state):

        probs = **F.softmax**(self**.model**(**Variable**(state, volatile = **True**))\*100) # *T=100*

        action = **probs.multinomial**()

**return** action.data[0,0]

**def** **learn**(self, batch\_state, batch\_next\_state, batch\_reward, batch\_action):

        outputs = self**.model**(batch\_state).**gather**(1, **batch\_action.unsqueeze**(1)).**squeeze**(1)

        next\_outputs = self**.model**(batch\_next\_state).**detach**().**max**(1)[0]

        target = self.gamma\*next\_outputs + batch\_reward

        td\_loss = **F.smooth\_l1\_loss**(outputs, target)

        self**.optimizer.zero\_grad**()

**td\_loss.backward**(retain\_variables = **True**)

        self**.optimizer.step**()

**def** **update**(self, reward, new\_signal):

        new\_state = **torch.Tensor**(new\_signal).**float**().**unsqueeze**(0)

        self**.memory.push**((self.last\_state, new\_state, **torch.LongTensor**([**int**(self.last\_action)]), **torch.Tensor**([self.last\_reward])))

        action = self**.select\_action**(new\_state)

**if** **len**(self.memory.memory) **>** 100:

            batch\_state, batch\_next\_state, batch\_action, batch\_reward = self**.memory.sample**(100)

            self**.learn**(batch\_state, batch\_next\_state, batch\_reward, batch\_action)

        self.last\_action = action

        self.last\_state = new\_state

        self.last\_reward = reward

        self**.reward\_window.append**(reward)

**if** **len**(self.reward\_window) **>** 1000:

**del** self.reward\_window[0]

**return** action

**def** **score**(self):

**return** **sum**(self.reward\_window)/(**len**(self.reward\_window)+1.)

**def** **save**(self):

**torch.save**({'state\_dict': self**.model.state\_dict**(),

                    'optimizer' : self**.optimizer.state\_dict**(),

                   }, 'last\_brain.pth')

**def** **load**(self):

**if** **os.path.isfile**('last\_brain.pth'):

**print**("=> loading checkpoint... ")

            checkpoint = **torch.load**('last\_brain.pth')

            self**.model.load\_state\_dict**(checkpoint['state\_dict'])

            self**.optimizer.load\_state\_dict**(checkpoint['optimizer'])

**print**("done !")

**else**:

**print**("no checkpoint found...")