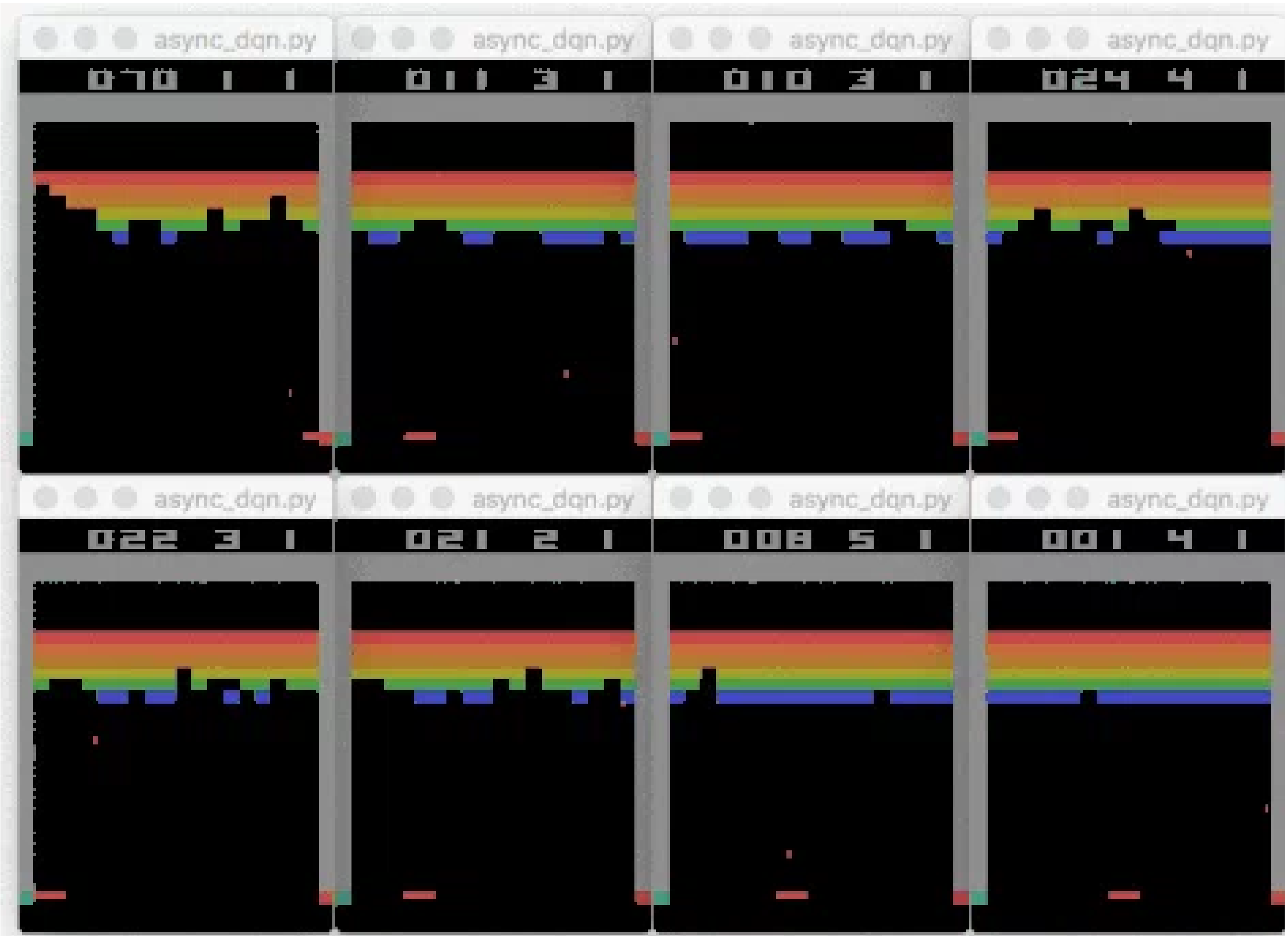
Deep Q-Network with Pytorch

Unnat Singh · Follow
13 min read · Mar 18, 2019

Listen Share More



DQN

- A deep neural network that acts as a function approximator.
- **Input:** Current state vector of the agent.

- **Output:** On the output side, unlike a traditional reinforcement learning setup where only one Q value is produced at a time, The Q network is designed to produce a Q value for every possible state-actions in a single forward pass.
- Training such a network requires a lot of data, but even then, it is not guaranteed to converge on the optimal value function. In fact, there are situations where the network weights can oscillate or diverge, due to the **high correlation between action and states.**
- This can result in a very unstable and ineffective policy we can solve this by:
- **Experience Replay**
- **Fixed Q-Target**

Experience Replay

- The idea of experience replay and its application to training the neural network isn't new.
- It was originally proposed to make more efficient use of observed experiences.
- Consider the basic online Q-Learning algorithm where we interact with the environment and at each time step, we obtained a state action reward next state tuple,

$$(S_t, A_t, R_{t+1}, S_{t+1})$$

- we learn from it and discard it.
- Moving on the next tuple in the following time step.
- We could possibly learn more from these experienced tuples if we store them somewhere.
- Moreover, some states are pretty rare to come by and some action can be pretty costly, so it would be nice to **recall** such experiences.
- That is exactly what a replay buffer allows us to do.

Replay Buffer

- We store each experience tuple in this buffer as we are interacting with the environment and then sample a small batch of tuples from it in order to learn.

- As a result, we are able to learn from individual tuples multiple times, recall rare occurrences, and in general make better use of our experience.

Another Problem that replays buffer solves:

- This what DQN takes advantage of:
- If you think about the experiences being obtained, we realize that every action A_t affects the next state S_t+1 in some way, which means that a sequence of experienced tuples can be highly correlated.
- A naive Q-Learning approach that learns from each of these experiences in sequential order runs the risk of getting swayed by the effect of this correlation.
- With experience replay, can sample from this buffer at random.
- It doesn't have to be in the same sequence as we stored the tuples.
- This helps break the correlation and ultimately prevents action values from oscillating or diverging catastrophically.

Example to show why we need to break the correlation between subsequent experience tuple

Tennis Example:

- Practising forehand, learning to play tennis.
- More confident with forehand shot than backhand.
- I hit the ball straight, the ball comes straight back to my forehand.
- Now, if I were an online Q-Learning agent learning to play, this is what I might pick up.
- When the ball comes to my right, I should hit with my forehand less certainly at first but with increasing confidence as I repeatedly hit the ball.
- I'm learning to play forehand pretty well **but not exploring the rest of the state space.**
- This could be addressed by Epsilon-Greedy policy action randomly with small chances.

- So I try different combinations of states and actions and sometimes I make mistakes, but I eventually figure out the best overall policy.
- Use a forehand shot when the ball comes to my right and a backhand when it comes to my left.
- This works fine with simplified state space with just two discrete states.

Continuous state-space — > Problem

- But when we consider a continuous state space things can fall apart. Let's s how
- First, the ball can actually come anywhere between the extreme left and extreme right.
- If I discretized this range into buckets I will have too many buckets (too many possibilities).
- What if I end up learning a policy with holes in it. For example states or situation that we may not have visited during practice.
- Instead, it makes more sense to use a function approximator like a linear combination of (RBF kernels or a Q-network) that can generalize my learning across space.
- Now, every time the ball comes to my right and I successfully hit a forehand shot, my value function changes slightly.
- What happens when I learn while (processing each experience tuple in order)
- For instance, if my forehand shots are fairly straight, I likely get back the ball around the same spot.
- This procedure a state very similar to the previous one, so I use my forehand again and if it is successful it reinforces my belief that the forehand is a good choice.
- I can easily get trapped in this cycle.
- Ultimately, if I don't see too many examples of the ball coming to my left for a while, the probability of the forehand shot become greater than the backhand

across the entire state space.

- My policy would then be to choose forehand regardless of where I see the ball coming.

Fix it

- The first thing I should do is stop learning while practising.
- This time is the best spend in trying out different shots playing little randomly and thus exploring the state space.
- It becomes important to remember my interactions, what shot was well in th given situations, etc.
- When I take a break or when I am back home or resting, that's a good time to recall this experience and learn from them.
- The main advantages are that now I have a more comprehensive set of examples.
- I can call random experience tuple from the buffer and learn different shot in a different region.
- After this, with this learned experience, I will again play and collect more experience tuple and learn from them in batches.
- **Experience replay** can help us to learn a more robust policy, one that is not affected by the inherent correlation present in the sequence of observed experience tuples.

Summary

When the agent interacts with the environment, the sequence of experienced tuples can be highly correlated. The naive Q-Learning algorithm that learns from each of these experience tuples in sequential order runs the risk of getting swayed by the effect of this correlation. By instead keeping track of the replay buffer and using experience replay to sample from the buffer at random, we can prevent action values from oscillating or diverging.

The replay buffer contains a collection of experience tuples [current state, action, reward, next state]. These tuples are gradually added to the buffer as we are interacting with the environment.

The act of sampling a small batch of tuples from the replay buffer in order to learn is known as experience replay. In addition to breaking harmful correlations, experience replay allows us to learn more from individual tuples multiple times, recall rare occurrences, and in general make better use of our experience.

Fixed Q-Targets

- Experience replay helps us address one type of correlation. That is between consecutive experience tuples.
- There is another kind of correlation that Q-Learning is susceptible to:
- The main idea of introducing fixed Q targets is that both labels and predicted values are functions of the same weights.
- All the Q values are intrinsically tied together through the function parameters.
- Doesn't experience replay take care of this problem?
- Well, it addresses a similar but slightly different issue.
- There we broke correlation effects between consecutive tuples by sampling them randomly out of order.
- Here, the correlation between the target and the parameters we are changing.

Q-Learning Update

$$\Delta w = \alpha(R + \gamma max_a q`(S`, a, w^-) - q`(S, A, w))dwq`(S, A, w)$$

- TD error: $(R + \gamma max_a q`(S`, a, w^-) - q`(S, A, w))$
- TD target: $R + \gamma max_a q`(S`, a, w^-)$
- Old value: $q`(S, A, w)$

where w^- are the weights of a sperate target network that are not changed during learning step, and $(S, A, R, S`)$ is an experience tuple.

$$J(w) = E_\pi[(q_\pi(S, A) - q`(S, A, w))^2]$$
$$dJ(w) = -2(q_\pi(S, A) - q`(S, A, w))dq`(S, A, w)$$
$$dw = -\alpha\tfrac{1}{2}dJ(w)$$
$$= \alpha(q_\pi(S, A) - q`(S, A, w))dq`(S, A, w)$$
$$dw = \alpha(R + \gamma max_a q`(S`, a, w) - q`(S, A, w))dq`(S, A, w)$$

Fixed Target

$$dw = \alpha(R + \gamma max_a q`(S`, a, w^-) - q`(S, A, w))dq`(S, A, w)$$

- The fixed parameters indicated by a w minus are basically a copy of w that we don't change during the learning step.
- In practice, we copy w into w minus, use to generate targets while changing w for a certain number of learning steps.
- Then, we update w minus with the latest w, again, learn for a number of steps and so on.
- This decouples the target from the parameters, makes the learning algorith much more stable, and less likely to diverge or fall into oscillations.

Summary

- In Q-Learning, **we update a guess with a guess**, and this can potentially lead to harmful correlations. To avoid this, we can update the parameters **w** in the network to get the current Q value for the current state and action and w- to get the target q value for the next state and action.

DQN — Implementation

Model Architecture

```python
import torch
import torch.nn as nn
import torch.nn.functional as F


class QNetwork(nn.Module):
    """ Actor (Policy) Model."""
    def __init__(self, state_size,action_size, seed, fc1_unit=64,
                 fc2_unit = 64):
        """
        Initialize parameters and build model.
        Params
        =======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_unit (int): Number of nodes in first hidden layer
            fc2_unit (int): Number of nodes in second hidden layer
        """
        super(QNetwork,self).__init__() ## calls __init__ method of nn.Module class
        self.seed = torch.manmual_seed(seed)
        self.fc1= nn.Linear(state_size,fc1_unit)
        seed.fc2 = nn.Linear(fc1_unit,fc2_unit)
        seed.fc3 = nn.Linear(fc2_unit,action_size)

    def forward(self,x):
        # x = state
        """
        Build a network that maps state -> action values.
        """
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

ModelArchitecture.py hosted with ❤ by GitHub view raw

DQN Agent

```python
import numpy as np
import random
from collections import namedtuple, deque

##Importing the model (function approximator for Q-table)
from model import QNetwork

import torch
import torch.nn.functional as F
import torch.optim as optim

BUFFER_SIZE = int(1e5)  #replay buffer size
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.99            # discount factor
TAU = 1e-3              # for soft update of target parameters
LR = 5e-4               # learning rate
UPDATE_EVERY = 4        # how often to update the network

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

class Agent():
    """Interacts with and learns form environment."""

    def __init__(self, state_size, action_size, seed):
        """Initialize an Agent object.

        Params
        =======
            state_size (int): dimension of each state
            action_size (int): dimension of each action
            seed (int): random seed
        """

        self.state_size = state_size
        self.action_size = action_size
        self.seed = random.seed(seed)


        #Q- Network
        self.qnetwork_local = QNetwork(state_size, action_size, seed).to(device)
        self.qnetwork_target = QNetwork(state_size, action_size, seed).to(device)

        self.optimizer = optim.Adam(self.qnetwork_local.parameters(),lr=LR)

        # Replay memory
        self.memory = ReplayBuffer(action_size, BUFFER_SIZE,BATCH_SIZE,seed)
        # Initialize time step (for updating every UPDATE_EVERY steps)
        self.t_step = 0
```

```
        self.t_step = 0

    def step(self, state, action, reward, next_step, done):
        # Save experience in replay memory
        self.memory.add(state, action, reward, next_step, done)

        # Learn every UPDATE_EVERY time steps.
        self.t_step = (self.t_step+1)% UPDATE_EVERY
        if self.t_step == 0:
            # If enough samples are available in memory, get radom subset and learn

            if len(self.memory)>BATCH_SIZE:
                experience = self.memory.sample()
                self.learn(experience, GAMMA)
    def act(self, state, eps = 0):
        """Returns action for given state as per current policy

        Params
        =======
            state (array_like): current state
            eps (float): epsilon, for epsilon-greedy action selection

        """
        state = torch.from_numpy(state).float().unsqueeze(0).to(device)
        self.qnetwork_local.eval()
        with torch.no_grad():
            action_values = self.qnetwork_local(state)
        self.qnetwork_local.train()

        #Epsilon -greedy action selction
        if random.random() > eps:
            return np.argmax(action_values.cpu().data.numpy())
        else:
            return random.choice(np.arange(self.action_size))

    def learn(self, experiences, gamma):
        """Update value parameters using given batch of experience tuples.

        Params
        =======

            experiences (Tuple[torch.Variable]): tuple of (s, a, r, s', done) tuples

            gamma (float): discount factor
        """
        states, actions, rewards, next_state, dones = experiences
        ## TODO: compute and minimize the loss
        criterion = torch.nn.MSELoss()
```

```python
        # Local model is one which we need to train so it's in training mode
        self.qnetwork_local.train()
        # Target model is one with which we need to get our target so it's in evaluati
        # So that when we do a forward pass with target model it does not calculate gr
        # We will update target model weights with soft_update function
        self.qnetwork_target.eval()
        #shape of output from the model (batch_size,action_dim) = (64,4)
        predicted_targets = self.qnetwork_local(states).gather(1,actions)

        with torch.no_grad():
            labels_next = self.qnetwork_target(next_states).detach().max(1)[0].unsque

        # .detach() ->  Returns a new Tensor, detached from the current graph.
        labels = rewards + (gamma* labels_next*(1-dones))

        loss = criterion(predicted_targets,labels).to(device)
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        # ------------------- update target network ------------------- #
        self.soft_update(self.qnetwork_local,self.qnetwork_target,TAU)

    def soft_update(self, local_model, target_model, tau):
        """Soft update model parameters.
        θ_target = τ*θ_local + (1 - τ)*θ_target

        Params
        =======
            local model (PyTorch model): weights will be copied from
            target model (PyTorch model): weights will be copied to
            tau (float): interpolation parameter

        """
        for target_param, local_param in zip(target_model.parameters(),
                                           local_model.parameters()):
            target_param.data.copy_(tau*local_param.data + (1-tau)*target_param.data)

class ReplayBuffer:
    """Fixed -size buffe to store experience tuples."""

    def __init__(self, action_size, buffer_size, batch_size, seed):
        """Initialize a ReplayBuffer object.

        Params
        ======
            action_size (int): dimension of each action
            buffer_size (int): maximum size of buffer
```

```
            buffer_size (int): maximum size of buffer
            batch_size (int): size of each training batch
            seed (int): random seed
        """

        self.action_size = action_size
        self.memory = deque(maxlen=buffer_size)
        self.batch_size = batch_size
        self.experiences = namedtuple("Experience", field_names=["state",
                                                               "action",
                                                               "reward",
                                                               "next_state",
                                                               "done"])
        self.seed = random.seed(seed)

    def add(self,state, action, reward, next_state,done):
        """Add a new experience to memory."""
        e = self.experiences(state,action,reward,next_state,done)
        self.memory.append(e)

    def sample(self):
        """Randomly sample a batch of experiences from memory"""
        experiences = random.sample(self.memory,k=self.batch_size)

        states = torch.from_numpy(np.vstack([e.state for e in experiences if e is not
        actions = torch.from_numpy(np.vstack([e.action for e in experiences if e is no
        rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if e is no
        next_states = torch.from_numpy(np.vstack([e.next_state for e in experiences if
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is not No

        return (states,actions,rewards,next_states,dones)
    def __len__(self):
        """Return the current size of internal memory."""
        return len(self.memory)
```

Train the Agent with DQN

Run the code below to train the agent from scratch.

```python
agent = Agent(state_size=8,action_size=4,seed=0)

def dqn(n_episodes= 200, max_t = 1000, eps_start=1.0, eps_end = 0.01,
       eps_decay=0.996):
    """Deep Q-Learning

    Params
    ======
        n_episodes (int): maximum number of training epsiodes
        max_t (int): maximum number of timesteps per episode
        eps_start (float): starting value of epsilon, for epsilon-greedy action sel
        eps_end (float): minimum value of epsilon
        eps_decay (float): mutiplicative factor (per episode) for decreasing epsi

    """
    scores = [] # list containing score from each episode
    scores_window = deque(maxlen=100) # last 100 scores
    eps = eps_start
    for i_episode in range(1, n_episodes+1):
        state = env.reset()
        score = 0
        for t in range(max_t):
            action = agent.act(state,eps)
            next_state,reward,done,_ = env.step(action)
            agent.step(state,action,reward,next_state,done)
            ## above step decides whether we will train(learn) the network
            ## actor (local_qnetwork) or we will fill the replay buffer
            ## if len replay buffer is equal to the batch size then we will
            ## train the network or otherwise we will add experience tuple in our
            ## replay buffer.
            state = next_state
            score += reward
            if done:
                break
            scores_window.append(score) ## save the most recent score
            scores.append(score) ## sae the most recent score
            eps = max(eps*eps_decay,eps_end)## decrease the epsilon
            print('\rEpisode {}\tAverage Score {:.2f}'.format(i_episode,np.mean(scores_
            if i_episode %100==0:
                print('\rEpisode {}\tAverage Score {:.2f}'.format(i_episode,np.mean(sc

            if np.mean(scores_window)>=200.0:
                print('\nEnvironment solve in {:d} epsiodes!\tAverage score: {:.2f}'.f

                torch.save(agent.qnetwork_local.state_dict(),'checkpoint.pth')
                break
    return scores
```

```
scores= dqn()

#plot the scores
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(len(scores)),scores)
plt.ylabel('Score')
plt.xlabel('Epsiode #')
plt.show()
```

Watch a Smart Agent!

In the next code cell, we will load the trained weights from file to watch a smart agent!

```
#load the weights from file
agent = Agent(state_size=8,action_size=4,seed=0)
agent.qnetwork_local.load_state_dict(torch.load('checkpoint.pth'))

for i in range(3):
    state = env.reset()
    img = plt.imshow(env.render(mode='rgb_array'))
    for j in range(200):
        action = agent.act(state)
        img.set_data(env.render(mode='rbg_array'))
        plt.axix('off')
        display.display(plt.gcf())
        display.clear_output(wait=True)
        state,reward,done,_ = env.step(action)
        if done:
            break

env.close()
```


Deep Q-Learning PipeLine

1. **Qnetwork** → Actor (Policy) model.

- Basically maps state space to actions space, it's a neural network that works as Q-table, its input dimension is equal to dimensions of state space and output dimension is equal to action space dimensions.
- We basically keep two neural networks because while training our labels and predicted values are both functions of neural network weights. To decouple the

label from weights we keep two sets of neural networks weights(two networks with the same architecture) fixed Q-targets.

2. **dqn_agent** → it's a class with many methods and it helps the agent (dqn_agent) to interact and learn from the environment.

3. **Replay Buffer** → Fixed-size buffer to store experience tuples.

Different methods of dqn_Agent

1. **__init__** method: We initialize the state_size, action and random seed.

- then we initialize two different q-network (qnetwork_local and qnetwork_target) one for mapping predictions and the other for mapping targets.

- then we declare an optimizer and we only define this for parameters of qnetwork_local and later we will do a soft update and update the parameters for qnetwork_target using the parameters of qnetwork_local.

- then we initialize the Replay buffer.

- then we initialize t_step, which decides after how many steps our agent should learn from experience.

2. **step(self, state, action, reward, next_state, done)**

- this method decides whether we will train(learn) the network actor (local_qnetwork) and fill the replay buffer or we will only fill the replay buffer.

- we will only learn from the experiences if the length of replay buffer is greater than batch_size and t_step is multiple of a number(of our choice, say after this many steps we want our agent to learn (for e.g. 40 iterations)).

3. **learn(self, experience, gamma)**

- this step is equivalent to the step in qlearning where we update the qtable (state-action value) for a state (S) after taking corresponding action (A)

$$Q[s,a] += \alpha(R + \gamma \times max_a Q[nextState] - Q[s,a])$$

- But instead of using the above equation, in DQN we use the neural network to map state-space which is continuous so we have a non-linear function approximator for mapping the state space and then we do backpropagation on our neural network to get the new update for qvalues.

- And our target is:

$$max_a Q[nextState, A, w^-] \times \gamma + R$$

- where Q[nextState, A, w_minus] is the output from qnetwork_target, the dimension of this [batchSize, dimensions of action space] so according to thi we define the architecture of our neural network, we do the following in Pytorch to get the target/labels.

```
labels_next = self.qnetwork_target(next_State).detach().max(1)[0].unsqueeze(1)
```

targetDqn.py hosted with ❤ by GitHub view raw

Pytorch operations we have to use unsqueeze(1) method.

- The states which we get from replay buffer has dimensions (batch_size, state_dimension) and one important thing to note here is along batch_size we have the different state at random order because of Replay Buffer (we have broken the **correlation of sequence**)

- And this implementation (1- done)*labelsnext makes sure that there is no next state after terminating state.

- After passing this state from qnetwork_local our output's dimension will be (batch_size, actionSpace dimensions) so in the experience tuple (state, action, reward, next_state, done) we have action corresponding to the current state, so here we only want qvalue to that corresponding action which was there in the experience tuple and we can get that with the following command:

```
self.qnetwork_local(state).gather(1,actions)
```

pytorchgather.py hosted with ❤ by GitHub view raw

- One important thing here to note is Q-table is a table that contains all possible states in the rows and all possible actions in the columns, in a particular row (state) whichever action has the highest value, that is the preferred action in that state that's how Q-learning(Sarsamax) works, but for this to work state space and action space should be discrete but in our case, the state space is continuous and have discrete action space, so **we use a neural network to approximate the Q-table**.

- So from the above code snippet, we can get our predicted value which has a dimension of [batch_size, 1].

- Now we can compute the loss and then we can use backpropagation to upda our weights and hence is equivalent to updating of **state action value(Q-tab**

- And then we do the **soft update** the gradient of **qnetwork_target**, remember we are only training one set of weights that is of qnetwork_local, so we need a way to update the weights of qnetwork_target and with those weights, we are hoping that our target too improve after each step as we are improving our predicted value, and the main idea we are using two networks is because we want to decouple both targets and predicted value from each other as both are functions of same weights, and with fixed q-target, we are making sure that our ta**rget and predicted value are functions of a different set of weights. So our network doesn't oscillate.**

4. soft_update(local_model, target_model, tau)

- One important thing to note is that when we are passing *next-state* to the *qnetwork_target* we are not calculating the gradient for each pass because we have wrapped with *torch.no_grad()* and there is no need of calculating the gradient.

- tau decides how much weightage will be given to the **qnetwork_local** and **qnetwork_target** weights respectively.

5. act(state, eps=0)

- Returns the action for the given state as per current policy.
- First, we change our model in evaluation mode.
- then we change the state tensor from *NumPy* to *torch.tensor* and then *.unsqueeze(1)* method is used to add a dimension along the batch_size, because in Pytorch we can only pass an input when it has a dimension that addresses the *batch_size*.
- And then we pass the state and get the corresponding action and note that we have used qnetwork_local.

- And then we have an implementation of **greedy action selection** because we want to explore more random actions. So that the agent gets more experience and **eps** hyperparameter control this process.
- And as we know we decrease the **eps** gradually as our agent becomes smarter so we want to decrease the **exploration** and increase **exploitations**. Sounds fancy!

Deep Q-Learning Improvements

Several improvements to the original Deep Q-Learning algorithm have been suggested.

1. **Double DQN**

- Deep Q-Learning tends to overestimate action values. Double Q-Learning has been shown to work well in practice to help with this.

2. **Prioritized Experience Replay**

- Deep Q-Learning samples experience transition **uniformly** from a replay buffer.
- Prioritized experience replay is based on the idea that the agent can learn more effectively from some transition than others, and the more important transitions should be sampled with higher probability.

3. **Duelling DQN**

- Currently, in order to determine which states are (or are not) valuable, we have to estimate the corresponding action value for each action. However, by replacing the traditional Deep Q-Network (DQN) architecture with a duelling architecture, **we can assess the value of each state, without having to learn the effect of each action.**

Double DQN

- The basic idea here is while training the agent in the early stages when the agent is naive for target updating, we use the action that **maximizes** the *Q-value[next_state]*. But in the early stage, this is a noisy approximation so we tend to **overestimate the Q-value**.
- To overcome the overestimation problem we can use both the networks the local and target as we have two sets of weights, so we can cross-validate it with both sets of weights and minimize the overestimation problem.

- We select the best action using one set of parameters **w** (*qnetwork_local*), but evaluate it with the different set of parameters **w-** (*qnetwork_target*).

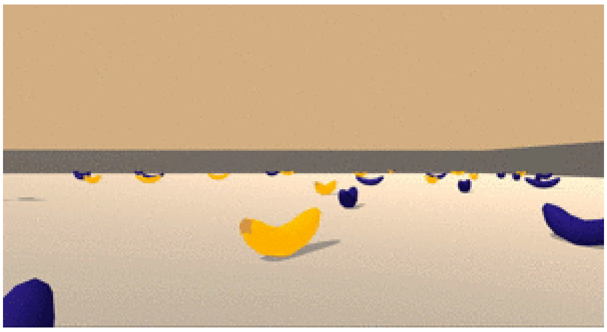
$$R + \gamma \dot{Q}[\dot{S}, (argmax_a \dot{Q}(\dot{S}, a, w)), w^-]$$

- Its basically likes having two separate function approximator that must agree on the best action.
- If **w** picks an action that is not best according to **w-**, then Q-value returned is that high.

Trained agent Example

In my Udacity Deep Reinforcement Learning nanodegree, I trained an agent to navigate in large grid world and collect bananas and it was trained using DQN algorithm.

For this project, you will train an agent to navigate (and collect bananas!) in a large, square world.



A reward of +1 is provided for collecting a yellow banana, and a reward of -1 is provided for collecting a blue banana. Thus, the goal of your agent is to collect as many yellow bananas as possible while avoiding blue bananas.

The state space has 37 dimensions and contains the agent's velocity, along with the ray-based perception of objects around the agent's forward direction. Given this information, the agent has to learn how to best select actions. Four discrete actions are available, corresponding to:

- `0` - move forward.
- `1` - move backwards.
- `2` - turn left.
- `3` - turn right.

The task is episodic, and in order to solve the environment, your agent must get an average score of +13 over 100 consecutive episodes.

Thank you! Happy Learning to everyone!!

Machine Learning | Reinforcement Learning | Deep Learning | Python | Artificial Intelligence

Follow

Written by Unnat Singh

18 Followers

More from Unnat Singh

Unnat Singh

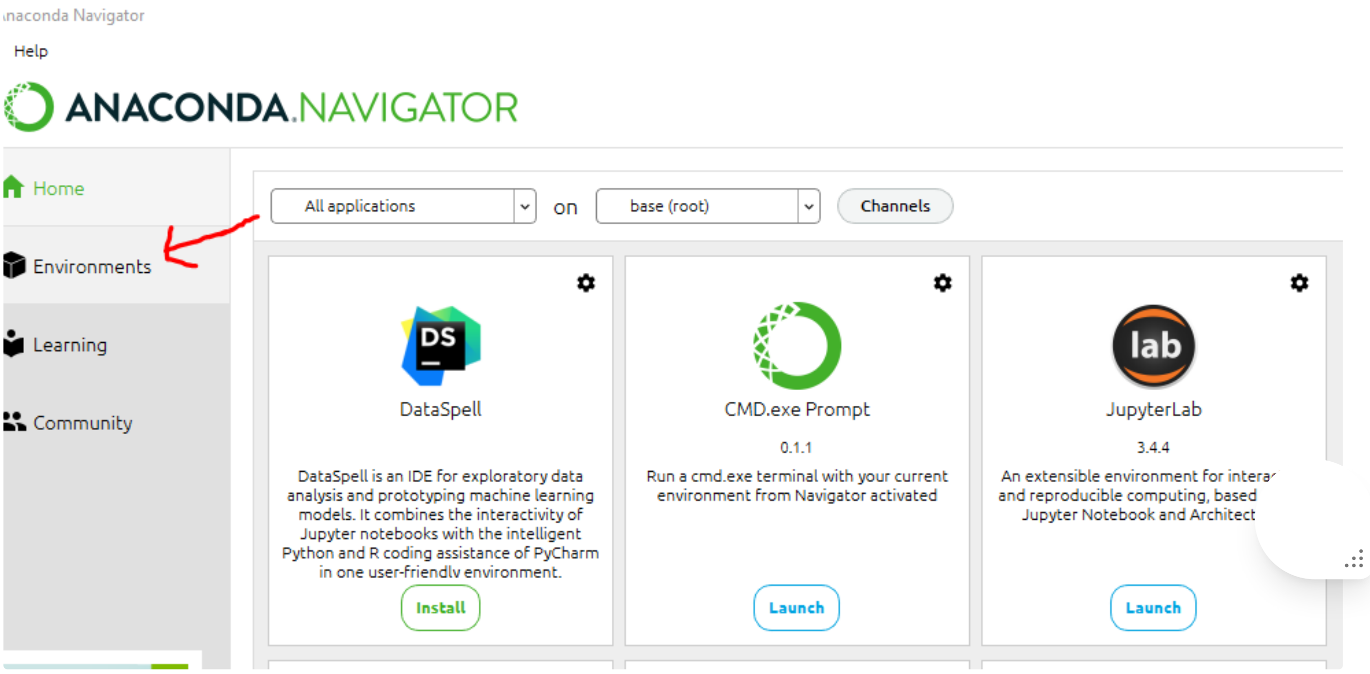
RL- Policy Gradient Method

What are Policy Gradient Methods?

12 min read · Jun 8, 2019

See all from Unnat Singh

Recommended from Medium



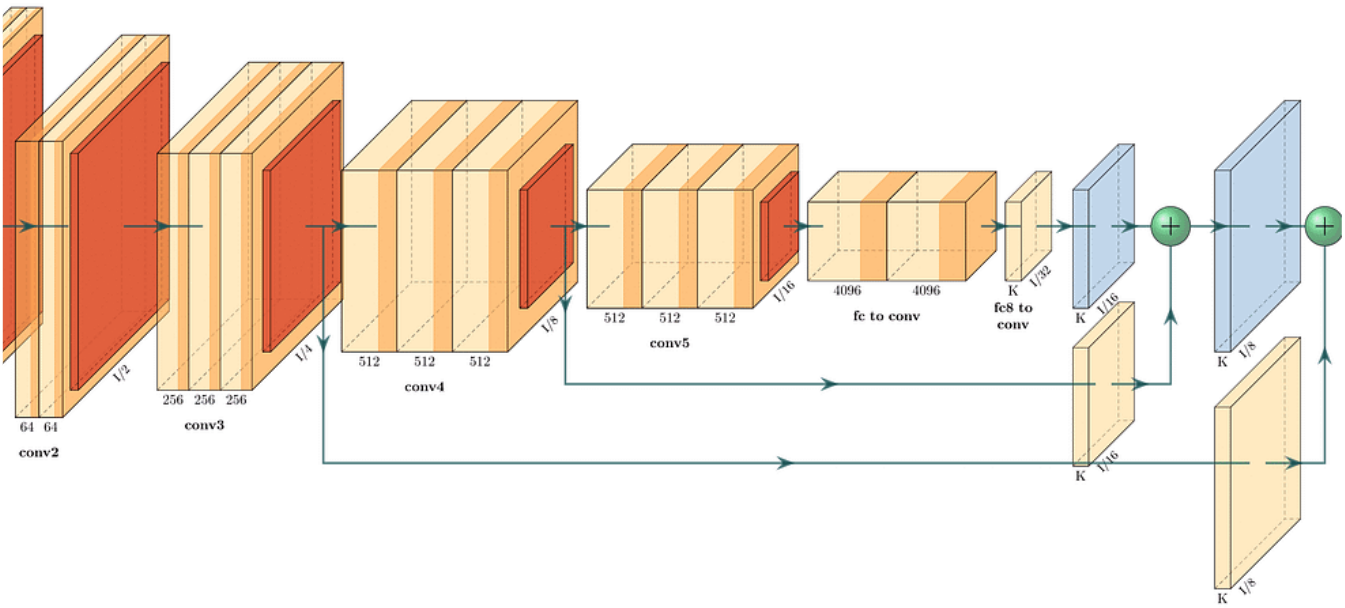
Harun Ijaz

A Step-by-Step Guide to Installing CUDA with PyTorch in Conda on Windows — Verifying via Console…

Installing CUDA using PyTorch in Conda for Windows can be a bit challenging, but with the right steps, it can be done easily. Here's a…

5 min read · Feb 14

42 3



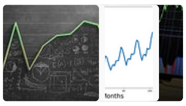
Clément Delteil in Towards AI

Creating Stunning Neural Network Visualizations with ChatGPT and PlotNeuralNet

Presenting PlotNeuralNet, a LaTeX / Python package to visualize Neural Networks

· 8 min read · Mar 8

Lists



Predictive Modeling w/ Python
18 stories · 195 saves

Practical Guides to Machine Learning
10 stories · 211 saves

Natural Language Processing
455 stories · 89 saves

ChatGPT
21 stories · 79 saves

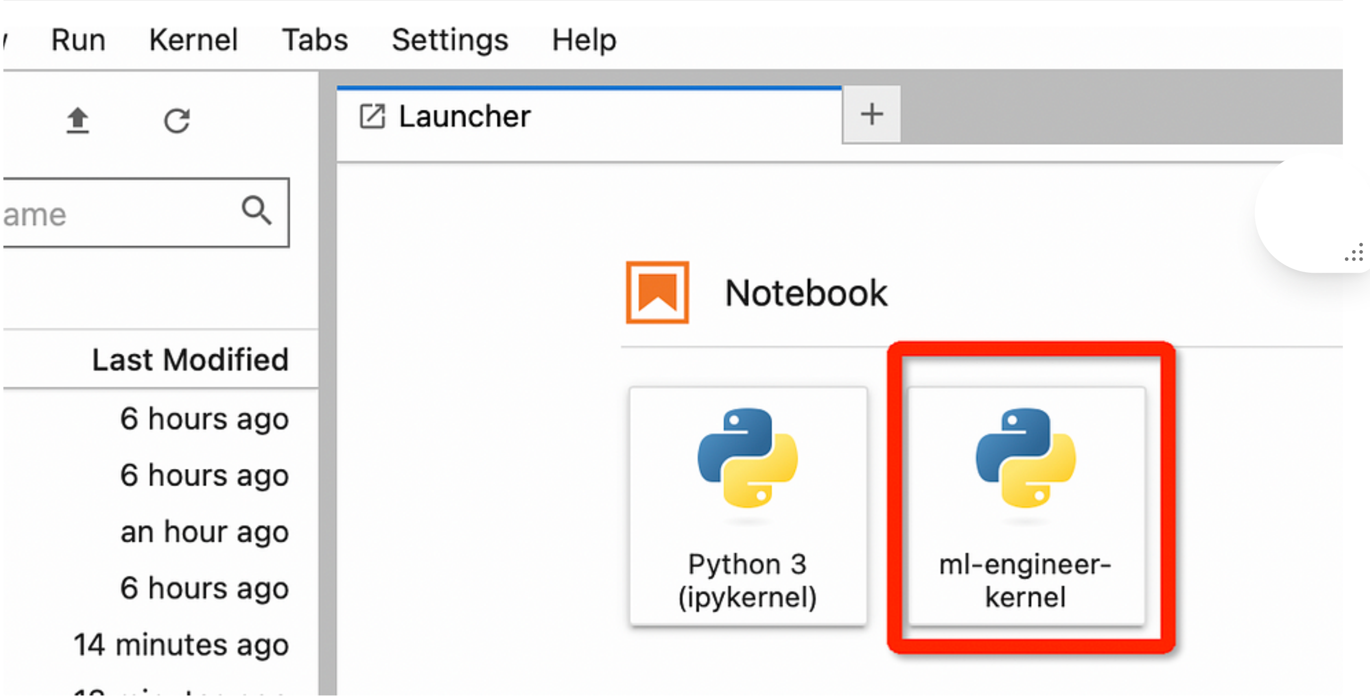
GeoSense ✅ in **AI monks.io**

Training PyTorch models on a Mac M1 and M2

PyTorch models on Apple Silicon M1 and M2

✦ · 9 min read · Mar 24

114



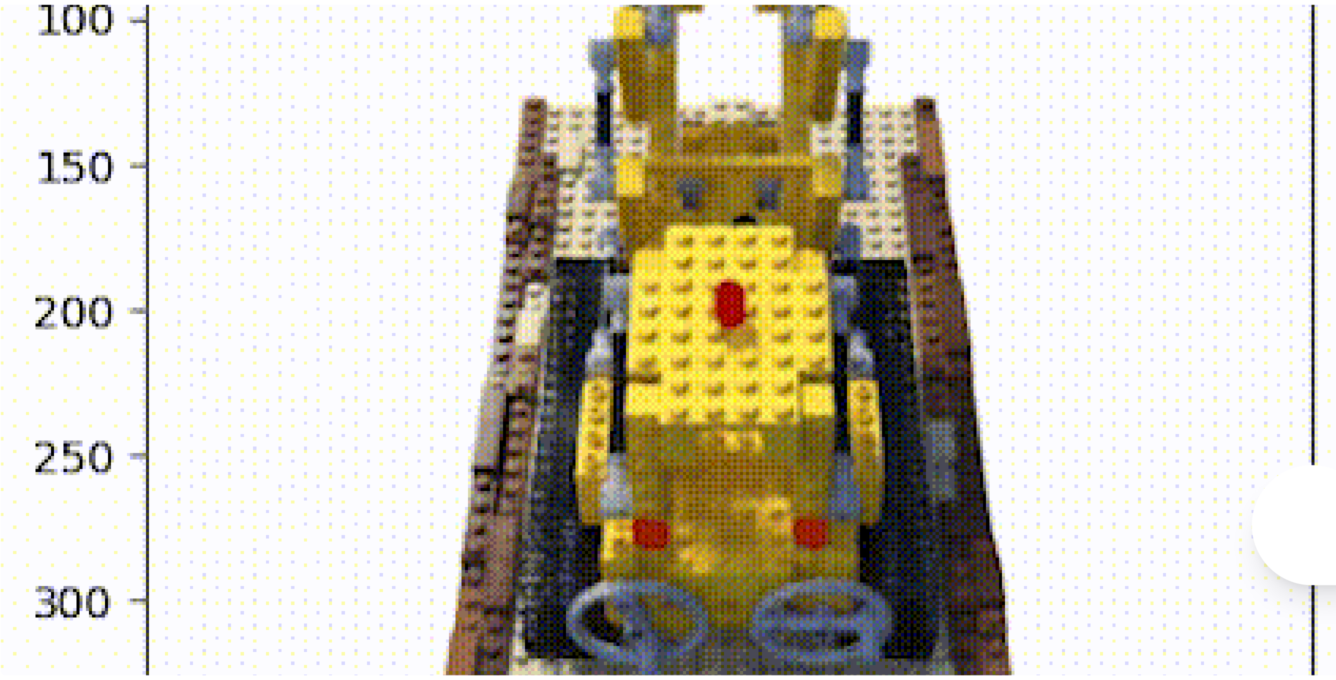
Ozgur Guler

How to run Pytorch and Tensorflow with GPU Acceleration on M2 MAC

I struggled a bit trying to get Tensoflow and PyTorch work on my M2 MAC properly...I put together this quick post to help others who might be...

2 min read · Feb 25

9



Papers in 100 Lines of Code

Neural Radiance Fields (NeRF) Tutorial in 100 lines of PyTorch code

Neural Radiance Fields (NeRF) is a hot topic in the computer vision community. It is a 3D scene representation technique that allows...

· 3 min read · Feb 12

52 1

Waleed Mousa in Artificial Intelligence in Plain English

Reinforcement Learning Made Easy: A Step-by-Step Guide to Building RL Algorithms with Python and...

Reinforcement learning (RL) is a type of machine learning that enables an agent to learn how to make optimal decisions in a given...

6 min read · Mar 5

9