Deep Q-Learning Agent

There are many more tips and tricks that researchers have discovered to make DQN training more stable and efficien. However, the basis that allows DeepMind to successfully train a DQN on a set of 49 Atari games and demonstrate the efficiency of this approach applied to complicated environments.
The original paper (without target network) was published at the end of 2013 (*Playing Atari with Deep Reinforcement Learning* 1312.5602v1, *Mnih and others.*), and they used seven games for testing. Later, at the beginning of 2015, a revised version of the article, with 49 different games, was published in *Nature* (*Human-Level Control Through Deep Reinforcement Learning* doi:10.1038/nature14236, *Mnih and others.*)
The algorithm for DQN from the preceding papers has the following steps:

1. Initialize parameters for $Q(s, a)$ and $\hat{Q}(s, a)$ with random weights, $\epsilon \leftarrow 1.0$, and empty replay buffer
2. With probability ϵ, select a random action a, otherwise $a = \arg\max_a Q_{s,a}$
3. Execute action a in an emulator and observe reward r and the next state s'
4. Store transition (s, a, r, s') in the replay buffer
5. Sample a random minibatch of transitions from the replay buffer
6. For every transition in the buffer, calculate target $y = r$ if the episode has ended at this step or $y = r + \gamma \max_{a' \in A} \hat{Q}_{s',a'}$ otherwise
7. Calculate loss: $\mathcal{L} = (Q_{s,a} - y)^2$
8. Update $Q(s, a)$ using the SGD algorithm by minimizing the loss in respect to model parameters
9. Every N steps copy weights from Q to $\hat{Q}$
10. Repeat from step 2 until converged

Before we jump into the code, some introduction is needed. Our examples are becoming increasingly challenging and complex, which is not surprising, as the complexity of problems we're trying to tackle is also growing. The examples are as simple and concise as possible, but some of the code may be difficult to understand at first.
Another thing to note is performance. Our previous examples for FrozenLake, or Q-learning Atari, were not demanding from a performance perspective, as observations were small, neural network parameters were tiny, and shaving off extra milliseconds in the training loop wasn't important. However, from now on, that's
not the case anymore. One single observation from the Atari environment is 100k values, which has to be rescaled, converted to floats, and stored in the replay buffer. One extra copy of this data array can cost you training speed, which is not seconds and minutes anymore, but could be hours even on the fastest GPU available. The neural network training loop could also be a bottleneck. Of course, RL models are not such huge monsters as state-of-the-art ImageNet models, but even the DQN model from 2015 has more than 1.5M parameters, which is a lot for a GPU to crunch. So, to make a long story short: performance matters, especially when you're experimenting with hyperparameters and need to wait not for a single model to train, but for dozens of them.

PyTorch is quite expressive, so more-or-less efficient processing code could look much less cryptic than optimized TensorFlow graphs, but there is still lots of opportunity for doing things slowly and making mistakes. For example, a naive

version of DQN loss computation, which loops over every batch sample, is about two times slower than a parallel version. However, a single extra copy of the data batch can make the speed of the same code 13 times slower, which is quite significant.
This example has been split into three modules due to its length, logical structure, and reusability. The modules are as follows:

I. Deep Q Learning Agent/lib/wrappers.py: These are Atari environment wrappers mostly taken from the OpenAI Baselines project
II. Deep Q Learning Agent/lib/dqn_model.py: This is the DQN neural net layer, with the same architecture as the DeepMind DQN from the *Nature* paper
III. Deep Q Learning Agent/02_dqn_pong.py: This is the main module with the training loop, loss function calculation, and experience replay buffer.

I. Wrappers

Tackling Atari games with RL is quite demanding from a resource perspective.To make things faster, several transformations are applied to the Atari platform interaction, which are described in DeepMind's paper. Some of these transformations influence only performance, but some address Atari platform features that make learning long and unstable. Transformations are usually implemented as OpenAI Gym wrappers of various kinds. The full list is quite lengthy and there are several implementations of the same wrappers in various sources. My personal favorite is in the OpenAI repository called **baselines**,which is a set of RL methods and algorithms implemented in TensorFlow and applied to popular benchmarks, to establish the common ground for comparing methods. The repository is available from https://github.com/openai/baselines,
and wrappers are available in this file:
https://github.com/openai/baselines/blob/master/baselines/common/atari_wrappers.py
The full list of Atari transformations used by RL researchers includes:Converting individual lives in the game into separate episodes. In general, an episode contains all the steps from the beginning of the game until the"Game over" screen appears?, which can last for thousands of game steps (observations and actions). Usually, in arcade games, the player is given several lives, which provide several attempts in the game. This transformation splits a full episode into individual small episodes for every life that a player has. Not all games support

this feature (for example, Pongdoesn't), but for the supported environments, it usually helps to speed up convergence as our episodes become shorter.
In the beginning of the game, performing a random amount (up to 30) of no-op actions. This should stabilize training, but there is no proper explanation why it is the case.
Making an action decision every K steps, where K is usually 4 or 3. On intermediate frames, the chosen action is simply repeated. This allows training to speed up significantly, as processing every frame with a neural network is quite a demanding operation, but the difference between consequent frames is usually minor.
Taking the maximum of every pixel in the last two frames and using it as an observation. Some Atari games have a flickering effect, which is due to the platform's limitation (Atari has a limited amount of sprites that can be shown on a single frame). For a human eye, such quick changes are not visible, but they can confuse neural networks.
Pressing **FIRE** in the beginning of the game. Some games (including Pong and Breakout) require a user to press the **FIRE** button to start the game. In theory, it's possible for a neural network to learn to press **FIRE** itself, but it will require much more episodes to be played. So, we press **FIRE** in the wrapper.
Scaling every frame down from 210 × 160, with three color frames, into a single-color 84 × 84 image. Different approaches are possible. For example,the DeepMind paper describes this transformation as taking the Y-color channel from the YCbCr color space and then rescaling the full image to an 84 × 84 resolution. Some other researchers do grayscale transformation,cropping non-relevant parts of the image and then scaling down. In the Baselines repository (and in the following example code), the latter approach is used.
Stacking several (usually four) subsequent frames together to give the network the information about the dynamics of the game's objects.

Clipping the reward to *-1*, *0*, and *1* values. The obtained score can vary wildly among the games. For example, in Pong you get a score of 1 for every ball that your opponent passes behind you. However, in some games, like KungFu, you get a reward of 100 for every enemy killed. This spread in reward values makes our loss have completely different scales between the games, which makes it harder to find common hyperparameters for a set of games. To fix this, reward just gets clipped to the range [*-1...1*].
Converting observations from unsigned bytes to float32 values. The screen obtained from the emulator is encoded as a tensor of bytes with values from 0 to 255, which is not the best representation for a neural network. So, we need to convert the image into floats and rescale the values
to the range [*0.0…1.0*].
In our example on Pong, we don't need some of the above wrappers, such as converting lives into separate episodes and reward clipping, so those wrappers aren't included in the example code. However, you should be aware of them, just in case you decide to experiment with other games. Sometimes, when the DQN is not converging, the problem is not in the code but in the wrongly wrapped environment. I've spend several days debugging convergence issues caused by missing the **FIRE** button press at the beginning of a game!
Let's take a look at the implementation of individual wrappers from
Deep Q Learning Agent /lib/wrappers.py:

```
import cv2
import gym
import gym.spaces
import numpy as np
import collections
class FireResetEnv(gym.Wrapper):
def __init__(self, env=None):
super(FireResetEnv, self).__init__(env)
assert env.unwrapped.get_action_meanings()[1] == 'FIRE'
assert len(env.unwrapped.get_action_meanings()) >= 3
def step(self, action):
return self.env.step.action()
def reset(self):
self.env.reset()
obs, _, done, _ = self.env.step(1)
if done:
self.env.reset()
obs, _, done, _ = self.env.step(2)
```

```
if done:
self.env.reset()
return obs
```

The preceding wrapper presses the **FIRE** button in environments that require them for the game to start. In addition to pressing **FIRE**, this wrapper checks for several corner cases that are present in some games.

```
class MaxAndSkipEnv(gym.Wrapper):
def __init__(self, env=None, skip=4):
"""Return only every 'skip'-th frame"""
super(MaxAndSkipEnv, self).__init__(env)
# most recent raw observations (for max pooling across time
steps)
self._obs_buffer = collections.deque(maxlen=2)
self._skip = skip
def step(self, action):
total_reward = 0.0
done = None
for _ in range(self._skip):
obs, reward, done, info = self.env.step(action)
self._obs_buffer.append(obs)
total_reward += reward
if done:
break
max_frame = np.max(np.stack(self._obs_buffer), axis=0)
return max_frame, total_reward, done, info
def _reset(self):
self._obs_buffer.clear()
obs = self.env.reset()
self._obs_buffer.append(obs)
return obs
```

This wrapper combines the repetition of actions during K frames and pixels from two consecutive frames.

```
class ProcessFrame84(gym.ObservationWrapper):
def __init__(self, env=None):
super(ProcessFrame84, self).__init__(env)
self.observation_space = gym.spaces.Box(low=0, high=255,
shape=(84, 84, 1), dtype=np.uint8)
def observation(self, obs):
return ProcessFrame84.process(obs)
@staticmethod
def process(frame):
if frame.size == 210 * 160 * 3:
img = np.reshape(frame, [210, 160,
3]).astype(np.float32)
elif frame.size == 250 * 160 * 3:
img = np.reshape(frame, [250, 160,
```

```
3]).astype(np.float32)
else:
assert False, "Unknown resolution."
img = img[:, :, 0] * 0.299 + img[:, :, 1] * 0.587 + img[:,
:, 2] * 0.114
resized_screen = cv2.resize(img, (84, 110),
interpolation=cv2.INTER_AREA)
x_t = resized_screen[18:102, :]
x_t = np.reshape(x_t, [84, 84, 1])
return x_t.astype(np.uint8)
```

The goal of this wrapper is to convert input observations from the emulator, which normally has a resolution of 210 × 160 pixels with RGB color channels, to a grayscale 84 × 84 image. It does this using a colorimetric grayscale conversion (which is closer to human color perception than a simple averaging of color channels), resizing the image and cropping the top and bottom parts of the result.

```
class BufferWrapper(gym.ObservationWrapper):
def __init__(self, env, n_steps, dtype=np.float32):
super(BufferWrapper, self).__init__(env)
self.dtype = dtype
old_space = env.observation_space
self.observation_space =
gym.spaces.Box(old_space.low.repeat(n_steps, axis=0),
old_space.high.repeat(n_steps, axis=0), dtype=dtype)
def reset(self):
self.buffer = np.zeros_like(self.observation_space.low,
dtype=self.dtype)
return self.observation(self.env.reset())
def observation(self, observation):
self.buffer[:-1] = self.buffer[1:]
self.buffer[-1] = observation
return self.buffer
```

This class creates a stack of subsequent frames along the first dimension and returns them as an observation. The purpose is to give the network an idea about the dynamics of the objects, such as the speed and direction of the ball in Pong or how enemies are moving. This is very important information, which it is not possible to obtain from a single image.

```
class ImageToPyTorch(gym.ObservationWrapper):
def __init__(self, env):
super(ImageToPyTorch, self).__init__(env)
old_shape = self.observation_space.shape
self.observation_space = gym.spaces.Box(low=0.0, high=1.0,
shape=(old_shape[-1], old_shape[0], old_shape[1]),
dtype=np.float32)
def observation(self, observation):
return np.moveaxis(observation, 2, 0)
```

This simple wrapper changes the shape of the observation from HWC to the CHW format required by PyTorch. The input shape of the tensor has a color channel as the last dimension, but PyTorch's convolution layers assume the color channel to be the first dimension.

```
class ScaledFloatFrame(gym.ObservationWrapper):
def observation(self, obs):
return np.array(obs).astype(np.float32) / 255.0
```

The final wrapper we have in the library converts observation data from bytes to floats and scales every pixel's value to the range [*0.0...1.0*].

```
def make_env(env_name):
env = gym.make(env_name)
env = MaxAndSkipEnv(env)
env = FireResetEnv(env)
env = ProcessFrame84(env)
env = ImageToPyTorch(env)
env = BufferWrapper(env, 4)
return ScaledFloatFrame(env)
```

At the end of the file is a simple function that creates an environment by its name and applies all the required wrappers to it. That's it for wrappers, so let's look at our model.

II. DQN model

The model published in *Nature* has three convolution layers followed by two fully connected layers. All layers are separated by ReLU nonlinearities. The output of the model is Q-values for every action available in the environment,without nonlinearity applied (as Q-values can have any value). The approach to have all Q-values calculated with one pass through the network helps us to increase speed significantly in comparison to treating *Q(s, a)* literally and feeding observations and actions to the network to obtain the value of the action.

The code of the model is in Deep Q Learning Agent /lib/dqn_model.py:

```
import torch
import torch.nn as nn
import numpy as np
class DQN(nn.Module):
def __init__(self, input_shape, n_actions):
super(DQN, self).__init__()
self.conv = nn.Sequential(
nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4),
nn.ReLU(),
```

```
nn.Conv2d(32, 64, kernel_size=4, stride=2),
nn.ReLU(),
nn.Conv2d(64, 64, kernel_size=3, stride=1),
nn.ReLU()
)
conv_out_size = self._get_conv_out(input_shape)
self.fc = nn.Sequential(
nn.Linear(conv_out_size, 512),
nn.ReLU(),
nn.Linear(512, n_actions)
)
```

To be able to write our network in the generic way, it was implemented in two parts: convolution and sequential. PyTorch doesn't have a 'flatter' layer which could transform a 3D tensor into a 1D vector of numbers, required to feed convolution output to the fully connected layer. This problem is solved in the `forward()` function, where we can reshape our batch of 3D tensors into a batch of 1D vectors.

Another small problem is that we don't know the exact number of values in the output from the convolution layer produced with input of the given shape.

However, we need to pass this number to the first fully connected layer constructor. One possible solution would be to hard-code this number, which is a function of input shape (for 84 × 84 input, the output from the convolution layer will have 3136 values), but it's not the best way, as our code becomes less robust to input shape change. The better solution would be to have a simple function (`_get_conv_out()`) that accepts the input shape and applies the convolution layer to a fake tensor of such a shape. The result of the function will be equal to the number of parameters returned by this application. It will be fast, as this call will be done once on model creation but will allow us to have generic code:

```
def _get_conv_out(self, shape):
o = self.conv(torch.zeros(1, *shape))
return int(np.prod(o.size()))
def forward(self, x):
conv_out = self.conv(x).view(x.size()[0], -1)
return self.fc(conv_out)
```

The final piece of the model is the `forward()` function, which accepts the 4D input tensor (the first dimension is batch size, the second is the *color* channel,which is our stack of subsequent

frames, while the third and fourth are image dimensions). The application of transformations is done in two steps: first we apply the convolution layer to the input and then we obtain a 4D tensor on output. This result is flattened to have two dimensions: a batch size and all the parameters returned by the convolution for this batch entry as one long vector of numbers. This is done by the `view()` function of the tensors, which lets one single dimension be a `-1` argument as a *wildcard* for the rest of the parameters.
For example, if we have a tensor `T` of shape `(2,3,4)`, which is a 3D tensor of 24 elements, we can reshape it into a 2D tensor with six rows and four columns using `T.view(6,4)`. This operation doesn't create a new memory object or move the data in memory, it just changes the higher-level shape of the tensor.
The same result could be obtained by `T.view(-1,4)` or `T.view(6,-1)`, which is very convenient when your tensor has a batch size in the first dimension. Finally, we pass this flattened 2D tensor to our fully connected layers to obtain Q-values for every batch input.

III. Training

The third module contains the experience replay buffer, the agent, the loss function calculation, and the training loop itself. Before going into the code,something needs to be said about the training hyperparameters. DeepMind's *Nature* paper contained a table with all the details about hyperparameters used to train its model on *a*// 49 Atari games used for evaluation. DeepMind kept all those parameters the same for all games (but trained individual models for every game), and it was the team's intention to show that the method is robust enough to solve lots of games with varying complexity, action space, reward structure,and other details using one single model architecture and hyperparameters.However, our goal here is much more modest: we want to solve just the Pong game.
Pong is quite simple and straightforward in comparison to other games in the Atari test set, so the hyperparameters in the paper are overkill for our task. For example, to get the best result on all 49 games, DeepMind used a million observations replay

buffer, which requires approximately 20 GB of RAM to keep and lots of samples from the environment to populate. The epsilon decay schedule that was used is also not the best for a single Pong game. In the training, DeepMind linearly decayed epsilon from 1.0 to 0.1 during the first million frames obtained from the environment. However, my own experiments have shown that for Pong, it's enough to decay epsilon over the first 100k frames and then keep it stable. The replay buffer can also be much smaller: 10k transitions will be enough. In the following example, I've used my parameters.

These differ from the parameters in the paper but allow us to solve Pong about ten times faster. On a GeForce GTX 1080 Ti, the following version converges to a mean score of 19.5 in one to two hours, but with DeepMind's hyperparameters it will require at least a day.

This speed up, of course, is fine-tuning for one particular environment and can break convergence on other games. You're free to play with the options and other games from the Atari set.

```
from lib import wrappers
from lib import dqn_model
import argparse
import time
import numpy as np
import collections
import torch
import torch.nn as nn
import torch.optim as optim
from tensorboardX import SummaryWriter
```

First, we import required modules and define hyperparameters.

```
DEFAULT_ENV_NAME = "PongNoFrameskip-v4"
MEAN_REWARD_BOUND = 19.5
```

These two values set the default environment to train on and the reward boundary for the last 100 episodes to stop training. They are just defaults; you can redefine them using the command line.

```
GAMMA = 0.99
BATCH_SIZE = 32
REPLAY_SIZE = 10000
REPLAY_START_SIZE = 10000
LEARNING_RATE = 1e-4
SYNC_TARGET_FRAMES = 1000
```

These parameters define the following:

Our gamma value used for Bellman approximation

The batch size sampled from the replay buffer (BATCH_SIZE)

The maximum capacity of the buffer (REPLAY_SIZE)
The count of frames we wait for before starting training to populate the replay buffer (REPLAY_START_SIZE)
The learning rate used in the Adam optimizer, which is used in this example How frequently we sync model weights from the training model to the target model, which is used to get the value of the next state in the Bellman approximation.

```
EPSILON_DECAY_LAST_FRAME = 10**5
EPSILON_START = 1.0
EPSILON_FINAL = 0.02
```

The last batch of hyperparameters is related to the epsilon decay schedule. To achieve proper exploration, at early stages of training, we start with epsilon=1.0, which causes all actions to be selected randomly. Then, during first 100,000 frames, epsilon is linearly decayed to 0.02, which corresponds to the random action taken in 2% of steps. A similar scheme was used in the original DeepMind paper, but the duration of decay was 10 times longer (so, epsilon = 0.02 is reached after a million frames).
The next chunk of the code defines our experience replay buffer, the purpose of which is to keep the last transitions obtained from the environment (tuples of the observation, action, reward, done flag, and the next state). Each time we do a step in the environment, we push the transition into the buffer, keeping only a fixed number of steps, in our case 10k transitions. For training, we randomly sample the batch of transitions from the replay buffer, which allows us to break the correlation between subsequent steps in the environment.

```
Experience = collections.namedtuple('Experience', field_names=
['state', 'action', 'reward', 'done', 'new_state'])
class ExperienceBuffer:
def __init__(self, capacity):
self.buffer = collections.deque(maxlen=capacity)
def __len__(self):
return len(self.buffer)
def append(self, experience):
self.buffer.append(experience)
def sample(self, batch_size):
indices = np.random.choice(len(self.buffer), batch_size,
replace=False)
states, actions, rewards, dones, next_states = zip(*
[self.buffer[idx] for idx in indices])
return np.array(states), np.array(actions),
np.array(rewards, dtype=np.float32), \
```

```
np.array(dones, dtype=np.uint8),
np.array(next_states)
```

Most of the experience replay buffer code is quite straightforward: it basically exploits the capability of the deque class to maintain the given number of entries in the buffer. In the sample() method, we create a list of random indices and then repack the sampled entries into NumPy arrays for more convenient loss calculation.

The next class we need to have is an Agent, which interacts with the environment and saves the result of the interaction into the experience replay buffer that we've just seen:

```
class Agent:
def __init__(self, env, exp_buffer):
self.env = env
self.exp_buffer = exp_buffer
self._reset()
def _reset(self):
self.state = env.reset()
self.total_reward = 0.0
```

During the agent's initialization, we need to store references to the environment and experience replay buffer, tracking the current observation and the total reward accumulated so far.

```
def play_step(self, net, epsilon=0.0, device="cpu"):
done_reward = None
if np.random.random() < epsilon:
action = env.action_space.sample()
else:
state_a = np.array([self.state], copy=False)
state_v = torch.tensor(state_a).to(device)
q_vals_v = net(state_v)
_, act_v = torch.max(q_vals_v, dim=1)
action = int(act_v.item())
```

The main method of the agent is to perform a step in the environment and store its result in the buffer. To do this, we need to select the action first. With the probability epsilon (passed as an argument) we take the random action, otherwise we use the past model to obtain the Q-values for all possible actions and choose the best.

```
new_state, reward, is_done, _ = self.env.step(action)
self.total_reward += reward
new_state = new_state
exp = Experience(self.state, action, reward, is_done,
new_state)
self.exp_buffer.append(exp)
self.state = new_state
if is_done:
```

```
done_reward = self.total_reward
self._reset()
return done_reward
```

As the action has been chosen, we pass it to the environment to get the next observation and reward, store the data in the experience buffer and the handle the end-of-episode situation. The result of the function is the total accumulated reward if we've reached the end of the episode with this step, or None if not.

Now it is time for the last function in the training module, which calculates the loss for the sampled batch. This function is written in a form to maximally exploit GPU parallelism by processing all batch samples with vector operations, which makes it harder to understand when compared with a naive loop over the batch. Yet this optimization pays off: the parallel version is more than two times faster than an explicit loop over the batch.

As a reminder, here is the loss expression we need to calculate:

$$\mathcal{L} = (Q_{s,a} - y)^2$$

for steps which aren't at the end of the episode, or

$\mathcal{L} = (Q_{s,a} - r)^2$ for final steps.

```
def calc_loss(batch, net, tgt_net, device="cpu"):
states, actions, rewards, dones, next_states = batch
```

In arguments, we pass our batch as a tuple of arrays (repacked by the sample() method in the experience buffer), our network that we're training and the target network, which is periodically synced with the trained one. The first model (passed as the argument net) is used to calculate gradients, while the second model in the tgt_net argument is used to calculate values for the next states and this calculation shouldn't affect gradients. To achieve this, we're using the detach() function of the PyTorch tensor to prevent gradients from flowing into the target network's graph.

```
states_v = torch.tensor(states).to(device)
next_states_v = torch.tensor(next_states).to(device)
actions_v = torch.tensor(actions).to(device)
rewards_v = torch.tensor(rewards).to(device)
done_mask = torch.ByteTensor(dones).to(device)
```

The preceding code is simple and straightforward: we wrap individual NumPy arrays with batch data in PyTorch tensors and copy them to GPU if the CUDA device was specified in arguments.

```
state_action_values = net(states_v).gather(1,
actions_v.unsqueeze(-1)).squeeze(-1)
```

In the line above, we pass observations to the first model and extract the specific Q-values for the taken actions using the gather() tensor operation. The first argument to the gather() call is a dimension index that we want to perform gathering on (in our case it is equal to 1, which corresponds to actions). The second argument is a tensor of indices of elements to be chosen. Extra unsqueeze() and squeeze() calls are required to fulfill the requirements of the gather functions to the index argument and to get rid of extra dimensions that we created (the index should have the same number of dimensions as the data we're processing). In the following image, you can see an illustration of what gather does on the example case, with a batch of six entries and four actions.

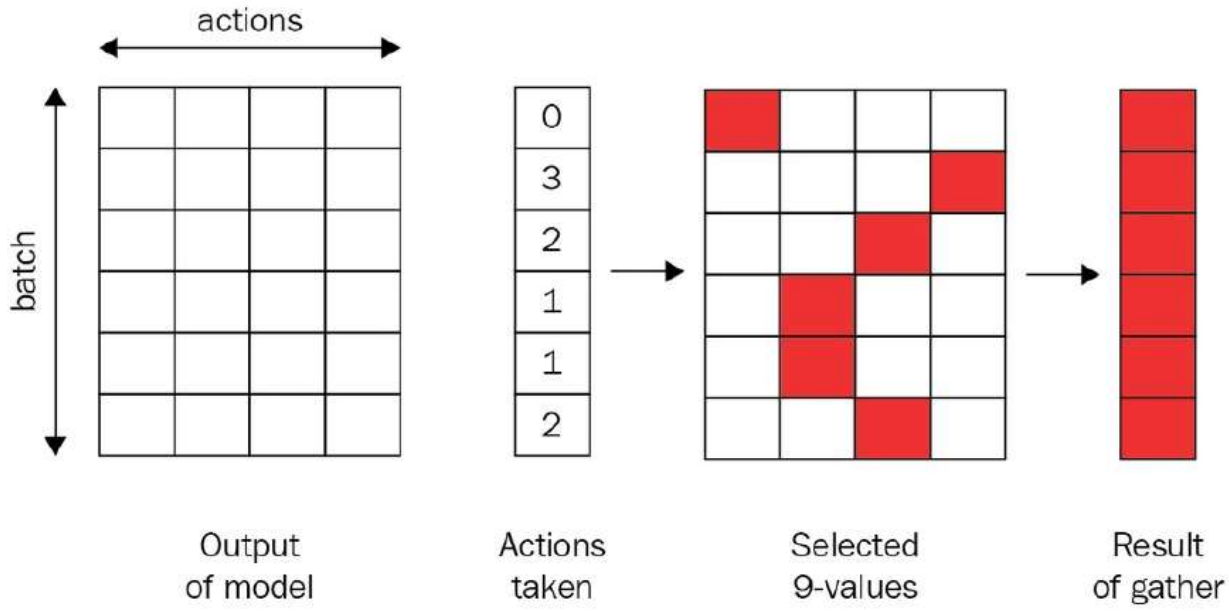

Figure : Transformation of tensors during a DQN loss calculation

Keep in mind that the result of gather() applied to tensors is a differentiable operation, which will keep all gradients with respect to the final loss value.

```
next_state_values = tgt_net(next_states_v).max(1)[0]
```

On in the above line, we apply the target network to our next state observations and calculate the maximum Q-value along the same *action* dimension 1.

Function `max()` returns both maximum values and indices of those values (so it calculates both max and argmax), which is very convenient. However, in this case, we're interested only in values, so we take the first entry of the result.

```
next_state_values[done_mask] = 0.0
```

Here we make one simple, but very important, point: if transition in the batch is from the last step in the episode, then our value of the action doesn't have a discounted reward of the next state, as there is no next state to gather reward from. This may look minor, but this is very important in practice: without this, training will not converge.

```
next_state_values = next_state_value.detach()
```

In this line, we detach the value from its computation graph to prevent gradients from flowing into the neural network used to calculate Q approximation for next states. This is important, as without this our backpropagation of the loss will start to affect both predictions for the current state and the next state. However, we don't want to touch predictions for the next state, as they're used in the Bellman equation to calculate reference Q-values. To block gradients from flowing into this branch of the graph, we're using the `detach()` method of the tensor, which returns the tensor without connection to its calculation history. In previous versions of PyTorch, we used a volatile attribute of the `Variable` class, which was obsoleted with the 0.4.0 release.

```
expected_state_action_values = next_state_values * GAMMA +
rewards_v
return nn.MSELoss()(state_action_values,
expected_state_action_values)
```

Finally, we calculate the Bellman approximation value and the mean squared error loss. This ends our loss function calculation, and the rest of the code is our training loop.

```
if __name__ == "__main__":
parser = argparse.ArgumentParser()
parser.add_argument("--cuda", default=False,
action="store_true", help="Enable cuda")
parser.add_argument("--env", default=DEFAULT_ENV_NAME,
help="Name of the environment, default=" +
DEFAULT_ENV_NAME)
parser.add_argument("--reward", type=float,
default=MEAN_REWARD_BOUND,
```

```
help="Mean reward boundary for stop of
training, default=%.2f" % MEAN_REWARD_BOUND)
args = parser.parse_args()
device = torch.device("cuda" if args.cuda else "cpu")
```

To begin with, we create a parser of command-line arguments. Our script allows us to enable CUDA and train on environments that are different from the default.

```
env = wrappers.make_env(args.env)
net = dqn_model.DQN(env.observation_space.shape,
env.action_space.n).to(device)
tgt_net = dqn_model.DQN(env.observation_space.shape,
env.action_space.n).to(device)
```

Here we create our environment with all required wrappers applied, the neural network we're going to train, and our target network with the same architecture.

In the beginning, they'll be initialized with different random weights, but it doesn't matter much as we'll sync them every 1k frames, which roughly corresponds to one episode of Pong.

```
writer = SummaryWriter(comment="-" + args.env)
print(net)
buffer = ExperienceBuffer(REPLAY_SIZE)
agent = Agent(env, buffer)
epsilon = EPSILON_START
```

Then we create our experience replay buffer of the required size and pass it to the agent. Epsilon is initially initialized to 1.0, but will be decreased every iteration.

```
optimizer = optim.Adam(net.parameters(), lr=LEARNING_RATE)
total_rewards = []
frame_idx = 0
ts_frame = 0
ts = time.time()
best_mean_reward = None
```

The last things we do before the training loop are to create an optimizer, a buffer for full episode rewards, a counter of frames and several variables to track our speed, and the best mean reward reached. Every time our mean reward beats the record, we'll save the model in the file.

```
while True:
frame_idx += 1
epsilon = max(EPSILON_FINAL, EPSILON_START - frame_idx /
EPSILON_DECAY_LAST_FRAME)
```

At the beginning of the training loop, we count the number of iterations completed and decrease epsilon according to our schedule. Epsilon will drop linearly during the given number of frames (EPSILON_DECAY_LAST_FRAME=100k)

and then will be kept on the same level of EPSILON_FINAL=0.02.

```
reward = agent.play_step(net, epsilon, device=device)
if reward is not None:
total_rewards.append(reward)
speed = (frame_idx - ts_frame) / (time.time() - ts)
ts_frame = frame_idx
ts = time.time()
mean_reward = np.mean(total_rewards[-100:])
print("%d: done %d games, mean reward %.3f, eps %.2f,
speed %.2f f/s" % (
frame_idx, len(total_rewards), mean_reward,
epsilon,
speed
))
writer.add_scalar("epsilon", epsilon, frame_idx)
writer.add_scalar("speed", speed, frame_idx)
writer.add_scalar("reward_100", mean_reward, frame_idx)
writer.add_scalar("reward", reward, frame_idx)
```

In this block of code, we ask our agent to make a single step in the environment (using our current network and value for epsilon). This function returns a non-None result only if this step is the final step in the episode. In that case, we report our progress. Specifically, we calculate and show, both in the console and in TensorBoard, these values:

--Speed as a count of frames processed per second

--Count of episodes played

--Mean reward for the last 100 episodes

--Current value for epsilon

```
if best_mean_reward is None or best_mean_reward <
mean_reward:
torch.save(net.state_dict(), args.env + "-
best.dat")
if best_mean_reward is not None:
print("Best mean reward updated %.3f ->
%.3f, model saved" % (best_mean_reward, mean_reward))
best_mean_reward = mean_reward
if mean_reward > args.reward:
print("Solved in %d frames!" % frame_idx)
break
```

Every time our mean reward for the last 100 episodes reaches a maximum, we report this and save the model parameters. If our mean reward exceeds the specified boundary, then we stop training. For Pong, the boundary is 19.5, which means winning more than 19 games from 21 possible games.

```
if len(buffer) < REPLAY_START_SIZE:
continue
if frame_idx % SYNC_TARGET_FRAMES == 0:
tgt_net.load_state_dict(net.state_dict())
```

Here we check whether our buffer is large enough for training. In the beginning, we should wait for enough data to start, which in our case is 10k transitions. The next condition syncs parameters from our main network to the target net every SYNC_TARGET_FRAMES, which is 1k by default.

```
optimizer.zero_grad()
batch = buffer.sample(BATCH_SIZE)
loss_t = calc_loss(batch, net, tgt_net, device=device)
loss_t.backward()
optimizer.step()
```

The last piece of the training loop is very simple, but requires the most time to execute: we zero gradients, sample data batches from the experience replay buffer, calculate loss, and perform the optimization step to minimize the loss.

Running and performance

This example is demanding on resources. On Pong, it requires about 400k frames to reach a mean reward of 17 (which means winning more than 80% of games). A similar number of frames will be required to get from 17 to 19.5, as our learning progress saturates and it's hard for the model to improve the score. So, on average, a million frames are needed to train it fully. On the GTX 1080Ti, I have a speed of about 150 frames per second, which is about two hours of training. On a CPU, the speed is much slower: about nine frames per second, which will take about a day and a half. Remember that this is for Pong, which is relatively easy to solve. Other games require hundreds of millions of frames and a 100 times larger experience replay buffer. Nevertheless, for Atari you'll need resources and patienc! The following image shows a TensorBoard screenshot with training dynamics:

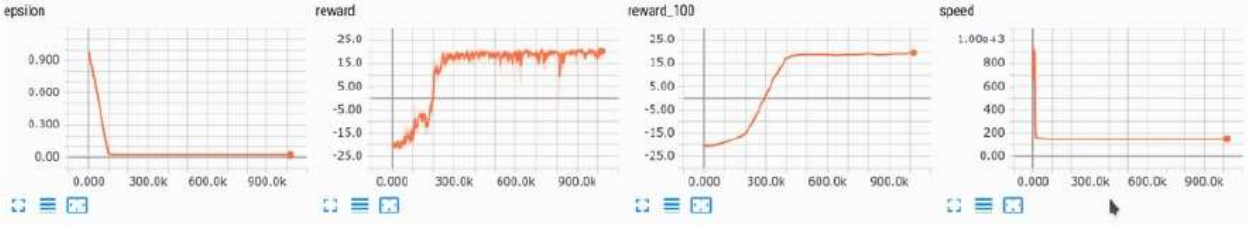

Figure : Characteristics of the training process (the X axis is the iteration number)

In the beginning of the training:

```
$ ./02_dqn_pong.py --cuda
DQN (
(conv): Sequential (
(0): Conv2d(4, 32, kernel_size=(8, 8), stride=(4, 4))
(1): ReLU ()
(2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
(3): ReLU ()
(4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
(5): ReLU ()
)
(fc): Sequential (
(0): Linear (3136 -> 512)
(1): ReLU ()
(2): Linear (512 -> 6)
)
)
1048: done 1 games, mean reward -19.000, eps 0.99, speed 83.45 f/s
1894: done 2 games, mean reward -20.000, eps 0.98, speed 913.37 f/s
2928: done 3 games, mean reward -20.000, eps 0.97, speed 932.16 f/s
3810: done 4 games, mean reward -20.250, eps 0.96, speed 923.60 f/s
4632: done 5 games, mean reward -20.400, eps 0.95, speed 921.52 f/s
5454: done 6 games, mean reward -20.500, eps 0.95, speed 918.04 f/s
6379: done 7 games, mean reward -20.429, eps 0.94, speed 906.64 f/s
7409: done 8 games, mean reward -20.500, eps 0.93, speed 903.51 f/s
8259: done 9 games, mean reward -20.556, eps 0.92, speed 905.94 f/s
9395: done 10 games, mean reward -20.500, eps 0.91, speed 898.05
f/s
10204: done 11 games, mean reward -20.545, eps 0.90, speed 374.76
f/s
10995: done 12 games, mean reward -20.583, eps 0.89, speed 160.55
f/s
11887: done 13 games, mean reward -20.538, eps 0.88, speed 160.44
f/s
12949: done 14 games, mean reward -20.571, eps 0.87, speed 160.67
f/s
```

Hundreds of games later, our DQN should start to figure out how to win one or two games out of 21. The speed has decreased due to epsilon drop: we need to use our model not only for training but also for the environment step.

```
101032: done 83 games, mean reward -19.506, eps 0.02, speed 143.06
f/s
103349: done 84 games, mean reward -19.488, eps 0.02, speed 142.99
f/s
106444: done 85 games, mean reward -19.424, eps 0.02, speed 143.15
f/s
108359: done 86 games, mean reward -19.395, eps 0.02, speed 143.18
f/s
110499: done 87 games, mean reward -19.379, eps 0.02, speed 143.01
f/s
113011: done 88 games, mean reward -19.352, eps 0.02, speed 142.98
```

f/s
115404: done 89 games, mean reward -19.326, eps 0.02, speed 143.07
f/s
117821: done 90 games, mean reward -19.300, eps 0.02, speed 143.03
f/s
121060: done 91 games, mean reward -19.220, eps 0.02, speed 143.10
f/s

Finally, after many more games, it can finally dominate and beat the (not very sophisticated) built-in Pong AI opponent:

982059: done 520 games, mean reward 19.500, eps 0.02, speed 145.14
f/s
984268: done 521 games, mean reward 19.420, eps 0.02, speed 145.39
f/s
986078: done 522 games, mean reward 19.440, eps 0.02, speed 145.24
f/s
987717: done 523 games, mean reward 19.460, eps 0.02, speed 145.06
f/s
989356: done 524 games, mean reward 19.470, eps 0.02, speed 145.07
f/s
991063: done 525 games, mean reward 19.510, eps 0.02, speed 145.31
f/s
Best mean reward updated 19.500 -> 19.510, model saved
Solved in 991063 frames!