10-703 Deep Reinforcement Learning and Control
Assignment 2
Spring 2017

March 1, 2017

Due March 17, 2017

Instructions

You have around 15 days from the release of the assignment until it is due. Refer to Gradescope for the exact time due. You may work with a partner on this assignment. **We HIGHLY recommend you work with a partner this assignment**. Only one person should submit the writeup and code on Gradescope. Make sure you mark your partner as a collaborator on Gradescope and that both names are listed in the writeup. Writeups should be typeset in Latex and submitted as PDF. All code, including auxiliary scripts used for testing should be submitted with a README.

Introduction

In this assignment, you will implement Q-learning using deep learning function approximators in OpenAI Gym. You will work with the Atari game environments, and learn how to train a Q-network directly from pixel inputs. The goal is to understand and implement some of the techniques that were found to be important in practice to stabilize training and achieve better performance. As a side effect, we also expect you to get comfortable using Tensorflow or Keras to experiment with different architectures and different hyperparameter configurations and gain insight into the importance of these choices on final performance and learning behavior.

The Atari environments in OpenAI Gym return observations as a $210 \times 160 \times 3$ tensor of unsigned 8-bit integers. The last dimension corresponds to the RGB channels. Remember to convert the tensor of unsigned integers to floats when feeding it to a Tensorflow computational graph, otherwise you may observe strange behavior.

Before starting your implementation, make sure that you have Tensorflow and the Atari environments correctly installed. For this assignment you may choose either Enduro (`Enduro-v0`) or Space Invaders (`SpaceInvaders-v0`). In spite of this, the observations from other Atari

games have the same size so you can try to train your models in the other games too. Note that while the size of the observations is the same across games, the number of available actions may be different from game to game.

Q-learning and variants

Due to the state space complexity of Atari environments, we represent Q-functions using a class of parametrized function approximators $\mathcal{Q} = \{Q_w \mid w \in \mathbb{R}^p\}$, where p is the number of parameters. Remember that in the *tabular setting*, given a 4-tuple of sampled experience (s, a, r, s'), the vanilla Q-learning update is

$$Q(s,a) := Q(s,a) + \alpha \left(r + \gamma \max_{a' \in A} Q(s', a') - Q(s,a) \right), \tag{1}$$

where $\alpha \in \mathbb{R}$ is the learning rate. In the *function approximation setting*, the update is similar:

$$w := w + \alpha \left(r + \gamma \max_{a' \in A} Q_w(s', a') - Q_w(s,a) \right) \nabla_w Q_w(s,a). \tag{2}$$

Q-learning can be seem as a pseudo stochastic gradient descent step on

$$\ell(w) = \mathbb{E}_{s,a,r,s'} \left(r + \gamma \max_{a' \in A} Q_w(s', a') - Q_w(s,a) \right)^2,$$

where the dependency of $\max_{a' \in A} Q_w(s', a')$ on w is ignored, i.e., it is treated as a fixed target.

Many of the methods that you will implement in this homework are variants of update (2), namely in the way the targets are constructed and maintained. For example, the *deep Q-learning implementation* described in [1, 2] maintains two Q-networks: the online network, which plays the same role of the Q_w terms $Q_w(s,a)$ and $\nabla_w Q_w(s,a)$ in update (2), and the target network, which is used in the target in update (2). The update in this case is

$$w := w + \alpha \left(r + \gamma \max_{a' \in A} Q_{w^-}(s', a') - Q_w(s,a) \right) \nabla_w Q_w(s,a). \tag{3}$$

The target Q-network is assigned every so often to be equal to the online Q-network, and is kept frozen until the next assignment. This helps the stability of the learning procedure, as with deep learning function approximators, updating the target Q-network with every update to the online Q-network proves too unstable.

Double Q-learning [3] also maintains two Q-networks, but they do not play a fixed role as online and target networks as in [1, 2]. Let us call the networks Q_{w_1} and Q_{w_2}; at each update step, we flip a fair coin and either do

$$w_1 := w_1 + \alpha \left(r + \gamma Q_{w_2}(s', \arg\max_{a' \in A} Q_{w_1}(s', a')) - Q_{w_1}(s,a) \right) \nabla_{w_1} Q_{w_1}(s,a) \tag{4}$$

or

$$w_2 := w_2 + \alpha \left(r + \gamma Q_{w_1}(s', \arg\max_{a' \in A} Q_{w_2}(s', a')) - Q_{w_2}(s, a) \right) \nabla_{w_2} Q_{w_2}(s, a).$$

As at each update the role of Q_{w_1} and Q_{w_2} is determined stochastically with probability 0.5, these networks play a symmetric role. This helps with the over-optimism of the targets in update (2).

In this homework, we will also ask you to implement the *dueling deep Q-network* described in [4]. This amounts to a slightly different Q-network architecture from the one in [1, 2]. Most models will be trained using *experience replay* [1, 2], meaning that the 4-tuples (s, a, r, s') will be sampled from the replay buffer rather than coming directly from the online experience of the agent.

Guidelines on hyperparameters

In this assignment you will implement improvements to the simple update Q-learning formula that make learning more stable and the trained model more performant. We briefly comment on the meaning of each hyperparameter and some reasonable values for them.

- Discount factor γ: 0.99.
- Learning rate α: 0.0001; typically a schedule is used where learning rates get increasingly smaller.
- Exploration probability ϵ in ϵ-greedy: Recommended that you use 0.05 when testing your policy (the small randomness prevents the agent from getting stuck). When training you should stat at a big epsilon (near 1) and decay to some small random value (e.g. .1) using some decay schedule. The original paper decays from 1 to .1 over a 1 million training updates. We recommend you start with something similar.
- Number of training sampled interactions with the environment: 5000000; depending on the convergence of the learning process, fewer or more interactions may be necessary; look at the average reward achieved in the last few episodes to test if performance has plateaued; it is usually a good idea to consider reducing the learning rate or the exploration probability if performance plateaus.
- Number of frames to feed to the Q-network: 4; as a single frame may not be a good representation of the current state of the MDP (e.g., in space invaders, from a single frame you cannot tell if the spaceships are moving left or right), multiple frames are fed to the Q-network to compute the Q-values; note that in this case, the state effectively is a list of last few frames.
- Input image resizing: $84 \times 84 \times 1$: using models of moderate size in the original $210 \times 160 \times 3$ image is computationally expensive, therefore we resize the original image to make it more manageable. The image becomes 1 channel because it is converted to grayscale. You can use the Python Image Library to do the conversions: `from PIL import Image`.

- Replay buffer size: 1000000; this hyperparameter is used only for experience replay; how many of the last
- Target Q-network reset interval: 10000; this hyperparameter only matters when we are maintaining both an online and target network; after this number of updates to the online Q-network, the target Q-network is set to be the same as the online Q-network.
- Batch size: 32; typically, rather doing the update as in (2), we use a small batch of sampled experiences from the replay buffer; this provides better hardware utilization.
- Learning Rate: 0.00025. You can try increasing this to converge faster, but you may end up with worse results. You can also try decreasing this value, but it will be a lot slower to train.

In addition to the hyperparameters you also have the choice of which optimizer and which loss function to use. We recommend you use Adam as the optimizer. It will automatically adjust the learning rate based on the statistics of the gradients its observing. Think of it like a fancier SGD with momentum. Both Tensorflow and Keras provide versions of Adam.

For the loss function, you can use Mean Squared Error, but you will likely have to clip the gradients to get stable learning. Instead, you can use the Mean Huber Loss. Huber loss uses quadratic loss within some range around zero, and absolute error loss outside of that. This puts a cap on the maximum gradient magnitude. Refer to the source code template for more information. **DO NOT** just clip the loss function and ask TF for the gradient with respect to the clipped loss. You will end up killing your gradients and not learning anything.

The implementations of the methods in this homework have multiple hyperparameters. These hyperparameters (and others) are part of the experimental setup described in [1, 2]. For the most part, we strongly suggest you to follow the experimental setup described in each of the papers. [1, 2] was published first; your choice of hyperparameters and the experimental setup should follow closely their setup. [3, 4] follow for the most part the setup of [1, 2]. We recommend you to read all these papers. We give pointers for the most relevant portions for you to read in a future section.

Guidelines on implementation

This homework requires a significant implementation effort. It is hard to read through the papers once and know immediately what you will need to be implement. We suggest you to think about the different components (e.g., image preprocessor, replay buffer, Tensorflow or Keras model definition, model updater, model runner, exploration schedule, learning rate schedule, ...) that you will need to implement for each of the different methods that we ask you about, and then read through the papers having these components in mind. By this we mean that you should try to divide and implement small components with well-defined functionalities rather than try to implement everything at once. Much of the code and experimental setup is shared between the different methods so identifying well-defined reusable components will save you trouble. We provide some code templates that you can use if you wish. Contrary to the previous assignment, abiding to the function signatures

defined in these templates is not mandatory – you can write your code from scratch if you wish.

Please note, that while this assignment has a lot of pieces to implement, most of the algorithms you will use in your project will be using the same pieces. Feel free to reuse any code you write for your homeworks in your class projects.

This is a challenging assignment. **Please start early!**

Guidelines on references

We recommend you to read all the papers mentioned in the references. There is a significant overlap between different papers, so in reality you should only need certain sections to implement what we ask of you. We provide pointers for relevant sections for this assignment for your convenience

The work in [1] contains the description of the experimental setup. Read paragraph 3 of section 4 for a description of the replay memory; read Algorithm 1; read paragraphs 1 and 3 of section 4.1 for preprocessing and model architecture respectively; read section 5 for the rest of the experimental setup (e.g., reward truncation, optimization algorithm, exploration schedule, and other hyperparameters). The methods section in [2], may clarify a few details so it may be worth to read selectively if questions remain after reading [1].

In [3], read "Double Q-learning" for the definition of the double Q-learning target; read paragraph 3 of "Empirical results" for some brief comments on the experimental setup followed. In [4], look at equation 11 and read around three paragraphs up and down for how to set up the dueling architecture; read paragraph 2 of section 4.2 for comments on the experimental setup and model architecture. It may be worth to skim additional sections of all these papers.

Questions

Try to keep your implementation modular. If you write it properly then once you have the basic DQN working it will only be a little bit of work to get double DQN and dueling networks working.

1. **[5pts]** Show that update (1) and update (2) are the same when the functions in $\mathcal{Q}$ are of the form $Q_w(s,a) = w^T\phi(s,a)$, with $w \in \mathbb{R}^{|S||A|}$ and $\phi : S \times A \to \mathbb{R}^{|S||A|}$, where the feature function ϕ is of the form $\phi(s,a)_{s',a'} = \mathbb{1}[s' = s, a' = a]$, where $\mathbb{1}$ denotes the indicator function which evaluates to 1 if the condition evaluates to true and evaluates to 0 if the condition evaluates to false. Note that the coordinates in the vector space $\mathbb{R}^{|S||A|}$ can be seem as being indexed by pairs (s', a'), where $s' \in S, a' \in A$.

2. **[5pts]** Implement a linear Q-network (no experience replay or target fixing). Use the the experimental setup of [1, 2] to the extent possible. Use the preprocessed state image pixels as your "features". If you want you can try to hand code some more advanced features for your particular game, but it is not necessary. If you can get this running you will have understood the basics of the assignment.

3. [**10pts**] Implement a linear Q-network with experience replay and target fixing. Use the experimental setup of [1, 2] to the extent possible. Again, you may not see great performance. Use this as an opportunity to work out any bugs with your replay memory.

4. [**5pts**] Implement a linear double Q-network. Use the the experimental setup of [1, 2] to the extent possible. Again, you may not see great performance.

5. [**35pts**] Implement the deep Q-network as described in [1, 2]. This should converge. You should run for as many iterations as necessary to show that your agent is converging. We recommend you run for at least a million. This will take some time to run! You may use the cluster to train this agent once you have worked out the bugs.

6. [**20pts**] Implement the double deep Q-network as described in [3].

7. [**20pts**] Implement the dueling deep Q-network as described in [4].

5pts out of the total of **100pts** are reserved for overall report quality. We recommend you to follow closely the experimental setup described in the papers. Even if you fully replicate the experimental from the paper, we expect you to summarize it briefly in the report once. After that you can simply describe differences from it and mention that you used the same experimental setup otherwise if that was the case.

For each of the models, we want you to generate a *performance plot across time*. To do this, you should periodically run (e.g., every 10000 or 100000 updates to the Q-network) the policy induced by the current Q-network for 20 episodes and average the total reward achieved. Note that in this case we are interested in total reward without discounting or truncation. During evaluation, you should use a tinier ϵ value in your ϵ-greedy policy. We recommend 0.05. The small amount of randomness is to prevent the agent from getting stuck. Also briefly comment on the training behavior and whether you find something unexpected in the results obtained.

Additionally, for each of the models, we want you to generate a *video capture* of an episode played by your trained Q-network at different points of the training process (0/3, 1/3, 2/3, and 3/3 through the training process) of either Enduro or Space Invaders. An episode is defined as the interval between the moment you start the game until the moment you lose all lives. You can use the `Monitor` wrapper to generate both the performance curves (although only for ϵ-greedy in this case) and the video captures. Look at the OpenAI Gym tutorial for more details on how to use it. We recommend that you periodically checkpoint your network files and then reload them after training to generate the evaluation curves. It is recommended you do regular checkpointing anyways in order to ensure no work is lost if your program crashes.

Finally, construct a *table* with the average total reward per episode in 100 episodes achieved by your fully trained model. Also show the information about the standard deviation, i.e., each entry should have the format mean $\pm$ std. There should be an entry per model. Briefly comment on the results of this table.

You should submit your report, video captures, and code through Gradescope. Your code should be reasonably well-commented in key places of your implementation.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[3] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. 2016.

[4] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.