CS234 Notes - Lecture 6
CNNs and Deep Q Learning

Tian Tan, Emma Brunskill

March 20, 2018

7 Value-Based Deep Reinforcement Learning

In this section, we introduce three popular value-based deep reinforcement learning (RL) algorithms: **Deep Q-Network (DQN)** [1], **Double DQN** [2] and **Dueling DQN** [3]. All the three neural architectures are able to learn successful policies directly from *high-dimensional inputs*, e.g. preprocessed pixels from video games, by using *end-to-end* reinforcement learning, and they all achieved a level of performance that is comparable to a professional human games tester across a set of 49 names on Atari 2600 [4].

Convolutional Neural Networks (CNNS) [5] are used in these architectures for feature extraction from pixel inputs. Understanding the mechanisms behind feature extraction via CNNs can help better understand how DQN works. The Stanford CS231N course website contains wonderful examples and introduction to CNNs. Here, we direct the reader to the following link for more details on CNNs: `http://cs231n.github.io/convolutional-networks/`. The remaining of this section will focus on generalization in RL and value-based deep RL algorithms.

7.1 Recap: Action-Value Function Approximation

In the previous lecture, we use parameterized function approximators to represent the action-value function (a.k.s. Q-function). If we denote the set of parameters as $\mathbf{w}$, the Q-function in this *approximation setting* is represented as $\hat{q}(s, a, \mathbf{w})$.

Let's first assume we have access to an oracle $q(s, a)$, the approximate Q-function can be learned by minimizing the mean-squared error between the true action-value function $q(s, a)$ and its approximated estimates,

$$J(\mathbf{w}) = \mathbb{E}[(q(s, a) - \hat{q}(s, a, \mathbf{w}))^2] \tag{1}$$

We can use *stochastic gradient descent (SGD)* to find a local minimum of J by sampling gradients w.r.t. parameters $\mathbf{w}$ and updating $\mathbf{w}$ as follows:

$$\Delta(\mathbf{w}) = -\frac{1}{2}\alpha\nabla_{\mathbf{w}}J(\mathbf{w}) = \alpha\mathbb{E}[(q(s, a) - \hat{q}(s, a, \mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(s, a, \mathbf{w})] \tag{2}$$

where α is the learning rate. In general, the true action-value function $q(s, a)$ is *unknown*, so we substitute the $q(s, a)$ in Equation (2) with an approximate *learning target*.

In Monte Carlo methods, we use an unbiased return G_t as the substitute target for episodic MDPs:

$$\Delta(\mathbf{w}) = \alpha(G_t - \hat{q}(s, a, \mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(s, a, \mathbf{w}) \tag{3}$$

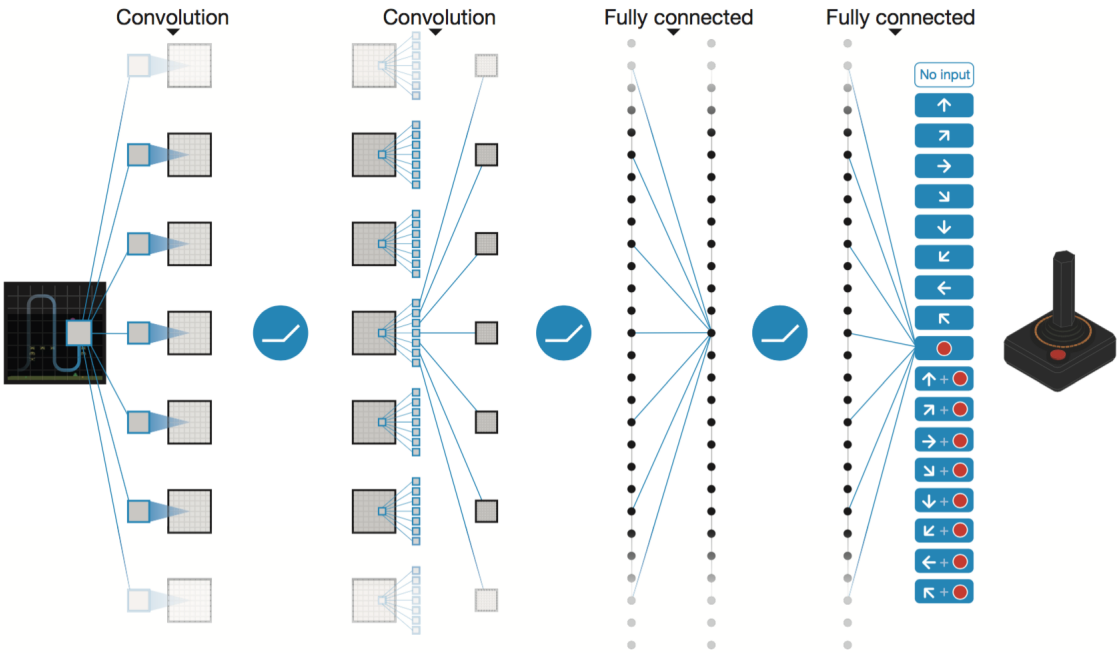

Figure 1: **Illustration of the Deep Q-network**: the input to the network consists of an $84 \times 84 \times 4$ preprocessed image, followed by three convolutional layers and two fully connected layers with a single output for each valid action. Each hidden layer is followed by a rectifier nonlinearity (ReLU) [6].

For SARSA, we instead use *bootstrapping* and present a TD (biased) target $r + \gamma\hat{q}(s', a', \mathbf{w})$, which leverages the current function approximation value,

$$\Delta(\mathbf{w}) = \alpha(r + \gamma\hat{q}(s', a', \mathbf{w}) - \hat{q}(s, a, \mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(s, a, \mathbf{w}) \tag{4}$$

where a' is the action taken at the next state s' and γ is a discount factor. For Q-learning, we use a TD target $r + \gamma \max_{a'} \hat{q}(s', a', \mathbf{w})$ and update $\mathbf{w}$ as follows:

$$\Delta(\mathbf{w}) = \alpha(r + \gamma \max_{a'} \hat{q}(s', a', \mathbf{w}) - \hat{q}(s, a, \mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(s, a, \mathbf{w}) \tag{5}$$

In subsequent sections, we will introduce how to approximate $\hat{q}(s, a, \mathbf{w})$ by using a deep neural network and learn neural network parameters $\mathbf{w}$ via end-to-end training.

7.2 Generalization: Deep Q-Network (DQN) [1]

The performance of linear function approximators highly depends on the quality of features. In general, handcrafting an appropriate set of features can be difficult and time-consuming. To scale up to making decisions in really *large domains* (e.g. huge state space) and enable automatic feature extraction, deep neural networks (DNNs) are used as function approximators.

7.2.1 DQN Architecture

An illustration of the DQN architecture is shown in Figure 1. The network takes preprocessed pixel image from Atari game environment (see 7.2.2 for preprocessing) as inputs, and outputs a vector containing Q-values for each valid action. The preprocessed pixel input is a summary of the game state s, and a single output unit represents the $\hat{q}$ function for a single action a. Collectively, the

$\hat{q}$ function can be denoted as $\hat{q}(s, \mathbf{w}) \in \mathbb{R}^{|A|}$. For simplicity, we will still use notation $\hat{q}(s, a, \mathbf{w})$ to represent the estimated action-value for a (s, a) pair in the following paragraphs.

Details of the architecture: the input consists of an $84 \times 84 \times 4$ image. The first convolutional layer has 32 filters of size 8×8 with stride 4 and convolves with the input image, followed by a rectifier nonlinearity (ReLU) [6]. The second hidden layer convolves 64 filters of 4×4 with stride 2, again followed by a rectifier nonlinearity. This is followed by a third convolutional layer that has 64 filters of 3×3 with stride 1, followed by a ReLU. The final hidden layer is a fully-connected layer with 512 rectifier (ReLU) units. The output layer is a fully-connected *linear* layer.

7.2.2 Preprocessing Raw Pixels

The raw Atari 2600 frames are of size $(210 \times 160 \times 3)$, where the last dimension is corresponding to the RGB channels. The preprocessing step adopted in [1] aims at reducing the input dimensionality and dealing with some artifacts of the game emulator. We summarize the preprocessing as follows:

- single frame encoding: to encode a single frame, the maximum value for each pixel color value over the frame being encoded and the previous frame is returned. In other words, we return a pixel-wise max-pooling of the 2 consecutive raw pixel frames.
- dimensionality reduction: extract the Y channel, also known as luminance, from the *encoded* RGB frame and rescale it to $(84 \times 84 \times 1)$.

The above preprocessing is applied to the 4 most recent raw RGB frames and the encoded frames are stacked together to produce the input (of shape $(84 \times 84 \times 4)$) to the Q-Network. Stacking together the recent frames as game state is also a way to transform the game environment into a (almost) Markovian world.

7.2.3 Training Algorithm for DQN

The use of large deep neural network function approximators for learning action-value functions has often been avoided in the past since theoretical performance guarantees are impossible, and learning and training tend to be very unstable. In order to use large nonlinear function approximators and scale online Q-learning, DQN introduced two major changes: the use of *experience replay*, and a separate *target network*. The full algorithm is presented in Algorithm 1. Essentially, the Q-network is learned by minimizing the following mean squared error:

$$J(\mathbf{w}) = \mathbb{E}_{(s_t, a_t, r_t, s_{t+1})}[(y_t^{DQN} - \hat{q}(s_t, a_t, \mathbf{w}))^2] \tag{6}$$

where y_t^{DQN} is the one-step ahead learning target:

$$y_t^{DQN} = r_t + \gamma \max_{a'} \hat{q}(s_{t+1}, a', \mathbf{w}^-) \tag{7}$$

where $\mathbf{w}^-$ represents the parameters of the target network, and the parameters $\mathbf{w}$ of the online network are updated by sampling gradients from minibatches of past transition tuples (s_t, a_t, r_t, s_{t+1}).
(*Note*: although the learning target is computed from the target network with $\mathbf{w}^-$, the targets y_t^{DQN} are considered to be fixed when making updates to $\mathbf{w}$.)

Experience replay:

The agent's experiences (or transitions) at each time step $e_t = (s_t, a_t, r_t, s_{t+1})$ are stored in a fixed-sized dataset (or *replay buffer*) $D_t = \{e_1, ..., e_t\}$. The replay buffer is used to store the most recent $k = 1$ million experiences (see Figure 2 for an illustration of replay buffer). The Q-network is updated by SGD with sampled gradients from minibatch data. Each transition sample in the minibatch is

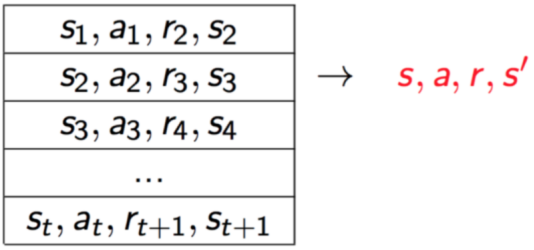

Figure 2: **Illustration of replay buffer:** the transition (s, a, r, s') is uniformly sampled from the replay buffer for updating Q-network.

sampled uniformly at random from the pool of stored experiences, $(s, a, r, s') \sim U(D)$. This approach has the following ***advantages*** over standard online Q-learning:

- Greater data efficiency: each step of experience can be potentially used for many updates, which improves data efficiency.
- Remove sample correlations: randomizing the transition experiences breaks the correlations between consecutive samples and therefore reduces the variance of updates and stabilizes the learning.
- Avoiding oscillations or divergence: the behavior distribution is averaged over many of its previous states and transitions, smoothing out learning and avoiding oscillations or divergence in the parameters. (Note that when using experience replay, it is required to use off-policy method, e.g. Q-learning, because the current parameters are different from those used to generate the samples).

limitation of experience replay: the replay buffer does not differentiate important transitions or informative transitions and it always overwrites with the recent transitions due to fixed buffer size. Similarly, the uniform sampling from the buffer gives equal importance to all stored experiences. A more sophisticated replay strategy, Prioritized Replay, has been proposed in [7], which replays important transitions more frequently, and therefore the agent learns more efficiently.

Target network:

To further improve the stability of learning and deal with *non-stationary learning targets*, a separate target network is used for generating the targets y_j (line 12 in Algorithm 1) in the Q-learning update. More specifically, every C updates/steps the target network $\hat{q}(s, a, \mathbf{w}^-)$ is updated by copying the parameters' values ($\mathbf{w}^- = \mathbf{w}$) from the online network $\hat{q}(s, a, \mathbf{w})$, and the target network remains unchanged and generates targets y_j for the following C updates. This modification makes the algorithm more stable compared to standard online Q-learning, and $C = 10000$ in the original DQN.

7.2.4 Training Details

In the original DQN paper [1], a different network (or agent) was trained on each game with the same architecture, learning algorithm and hyperparameters. The authors clipped all positive rewards from the game environment at $+1$ and all negative rewards at -1, which makes it possible to use the same learning rate across all different games. For games where there is a life counter (e.g. *Breakout*), the emulator also returns the number of lives left in the game, which was then used to mark the end of an episode during training by explicitly setting future rewards to zeros. They also used a simple frame-skipping technique (or *action repeat*): the agent selects actions on every 4-th frame instead of every frame, and its last action is repeated on skipped frames. This reduces the frequency of decisions

Algorithm 1 deep Q-learning

1: Initialize replay memory D with a fixed capacity
2: Initialize action-value function $\hat{q}$ with random weights $\mathbf{w}$
3: Initialize target action-value function $\hat{q}$ with weights $\mathbf{w}^- = \mathbf{w}$
4: **for** episode $m = 1, \ldots, M$ **do**
5: Observe initial frame x_1 and preprocess frame to get state s_1
6: **for** time step $t = 1, \ldots, T$ **do**
7: Select action $a_t = \begin{cases} \text{random action} & \text{with probability } \epsilon \\ \arg\max_a \hat{q}(s_t, a, \mathbf{w}) & \text{otherwise} \end{cases}$
8: Execute action a_t in emulator and observe reward r_t and image x_{t+1}
9: Preprocess s_t, x_{t+1} to get s_{t+1}, and store transition (s_t, a_t, r_t, s_{t+1}) in D
10: Sample uniformly a random minibatch of N transitions (s_j, a_j, r_j, s_{j+1}) from D
11: Set $y_j = r_j$ if episode ends at step $j + 1$;
12: otherwise set $y_j = r_j + \gamma \max_{a'} \hat{q}(s_{j+1}, a', \mathbf{w}^-)$ ← wrt $\mathbf{w}^-$
13: Perform a stochastic gradient descent step on $J(\mathbf{w}) = \frac{1}{N}\sum_{j=1}^{N}(y_j - \hat{q}(s_j, a_j, \mathbf{w}))^2$ w.r.t. parameters $\mathbf{w}$
14: Every C steps reset $\mathbf{w}^- = \mathbf{w}$

without impacting the performance too much and enables the agent to play roughly 4 times more games during training.

RMSProp (see https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf) was used in [1] for training DQN with minibatches of size 32. During training, they applied ϵ-greedy policy with ϵ linearly annealed from 1.0 to 0.1 over the first million steps, and fixed at 0.1 afterwards. The replay buffer was used to store the most recent 1 million transitions. For evaluation at test time, they used ϵ-greedy policy with $\epsilon = 0.05$. why use ϵ during testing?

7.3 Reducing Bias: Double Deep Q-Network (DDQN) [2]

like using double Q-learning to eliminate maximization bias.

The max operator in DQN (line 12 of Algorithm 1), uses the same network values both to select and to evaluate an action. This setting makes it more likely to select overestimated values and resulting in overoptimistic target value estimates. Van Hasselt et al. also showed in [2] that the DQN algorithm suffers from substantial overestimations in some games in the Atari 2600. To prevent overestimation and reduce bias, we can *decouple* the *action selection* from *action evaluation*.

Recall in Double Q-learning, two action-value functions are maintained and learned by randomly assigning transitions to update one of the two functions, resulting in two different sets of function parameters, denoted here as $\mathbf{w}$ and $\mathbf{w}'$. For computing targets, one function is used to select the greedy action and the other to evaluate its value:

$$y_t^{DoubleQ} = r_t + \gamma \hat{q}(s_{t+1}, \arg\max_{a'} \hat{q}(s_{t+1}, a', \mathbf{w}), \mathbf{w}') \tag{8}$$

Note that the action selection ($argmax$) is due to the function parameters $\mathbf{w}$, while the action value is evaluated by the other set of parameters $\mathbf{w}'$.

The idea of reducing overestimations by decoupling action selection and action evaluation in computing targets can also be extended to deep Q-learning. The target network in DQN architecture provides a natural candidate for the second action-value function, without introducing additional networks. Similarly, the greedy action is generated according to the online network with parameters $\mathbf{w}$, but its value is estimated by the target network with parameters $\mathbf{w}^-$. The resulting algorithm is referred as *Double DQN* [2], which just replaces line 12 in Algorithm 1 by the following update target:

$$y_t^{DoubleDQN} = r_t + \gamma \hat{q}(s_{t+1}, \arg\max_{a'} \hat{q}(s_{t+1}, a', \mathbf{w}), \mathbf{w}^-) \tag{9}$$

In double DQN, $\mathbf{w}^-$ not only serves the purpose of stabilizing the updates, but also as the second sets of weights used for evaluation.

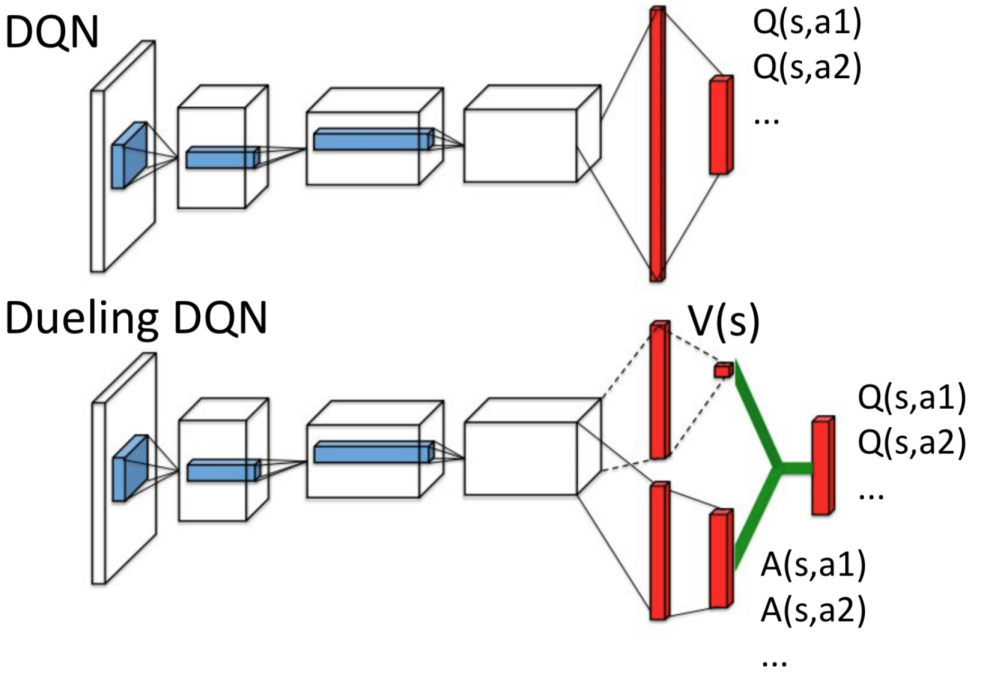

Figure 3: Single stream Deep Q-network (top) and the dueling Q-network (bottom). The dueling network has two streams to separately estimate (scalar) state-value $V(s)$ and the advantages $A(s,a)$ for each action; the green output module implements equation (13) to combine the two streams. Both networks output Q-values for each action.

The update to the target network stays unchanged from DQN, and remains a periodic copy of the online network $\mathbf{w}$. The rest of the DQN algorithm remains intact.

7.4 Decoupling Value and Advantage: Dueling Network [3]

7.4.1 The Dueling Network Architecture

Before we delve into dueling architecture, let's first introduce an important quantity, the *advantage function*, which relates the value and Q functions (assume following a policy π):

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s) \tag{10}$$

Recall $V^\pi(s) = \mathbb{E}_{a\sim\pi(s)}[Q^\pi(s,a)]$, thus we have $\mathbb{E}_{a\sim\pi(s)}[A^\pi(s,a)] = 0$. Intuitively, the advantage function subtracts the value of the state from the Q function to get a relative measure of the importance of each action. current action

Like in DQN, the dueling network is also a DNN function approximator for learning the Q-function. Differently, it approximates the Q-function by *decoupling* the value function and the advantage function. Figure 3 illustrates the dueling network architecture and the DQN for comparison.

The lower layers of the dueling network are convolutional as in the DQN. However, instead of using a single stream of fully connected layers for Q-value estimates, the dueling network uses two streams of fully connected layers. One stream is used to provide value function estimate given a state, while the other stream is for estimating advantage function for each valid action. Finally, the two streams

are combined in a way to produce and approximate the Q-function. As in DQN, the output of the network is a vector of Q-values, one for each action.

Note that since the inputs and the final outputs (combined two streams) of the dueling network are the same as that of the original DQN, the training algorithm (Algorithm 1) introduced above for DQN and for Double DQN can also be applied here to train the dueling architecture. The separated two-stream design is based on the following observations or intuitions from the authors:

- For many states, it is unnecessary to estimate the value of each possible action choice. In some states, the action selection can be of great importance, but in many other states the choice of action has no repercussion on what happens next. On the other hand, the state value estimation is of significant importance for every state for a bootstrapping based algorithm like Q-learning.
- Features required to determine the value function may be different than those used to accurately estimate action benefits.

Combing the two streams of fully connected layers for Q-value estimate is not a trivial task. This aggregating module (shown in *green* lines in Figure 3), in fact, requires very thoughtful design, which we will see in the next subsection.

7.4.2 Q-value Estimation

From the definition of advantage function (10), we have $Q^\pi(s,a) = A^\pi(s,a) + V^\pi(s)$, and $\mathbb{E}_{a\sim\pi(s)}[A^\pi(s,a)] = 0$. Furthermore, for a deterministic policy (commonly used in value-based deep RL), $a^* = \arg\max_{a'\in A} Q(s,a')$, it follows that $Q(s,a^*) = V(s)$ and hence $A(s,a^*) = 0$. The greedily selected action has zero advantage in this case.

Now consider the dueling network architecture in Figure 3 for function approximation. Let's denote the scalar output value function from one stream of the fully-connected layers as $\hat{v}(s,\mathbf{w},\mathbf{w_v})$, and denote the vector output advantage function from the other stream as $A(s,a,\mathbf{w},\mathbf{w_A})$. We use $\mathbf{w}$ here to denote the shared parameters in the convolutional layers, and use $\mathbf{w_v}$ and $\mathbf{w_A}$ to represent parameters in the two different streams of fully-connected layers. Then, probably the most simple way to design the aggregating module is by following the definition:

$$\hat{q}(s,a,\mathbf{w},\mathbf{w_A},\mathbf{w_v}) = \hat{v}(s,\mathbf{w},\mathbf{w_v}) + A(s,a,\mathbf{w},\mathbf{w_A}) \tag{11}$$

The main problem with this simple design is that Equation (11) is unidentifiable. Given $\hat{q}$, we cannot recover $\hat{v}$ and A uniquely, e.g. adding a constant to $\hat{v}$ and subtracting the same constant from A gives the same Q-value estimates. The unidentifiable issue is mirrored by poor performance in practice.

To make Q-function identifiable, recall in the deterministic policy case discussed above, we can force the advantage function to have zero estimate at the chosen action. Then, we have

$$\hat{q}(s,a,\mathbf{w},\mathbf{w_A},\mathbf{w_v}) = \hat{v}(s,\mathbf{w},\mathbf{w_v}) + \Big(A(s,a,\mathbf{w},\mathbf{w_A}) - \max_{a'\in A} A(s,a',\mathbf{w},\mathbf{w_A})\Big) \tag{12}$$

For a deterministic policy, $a^* = \arg\max_{a'\in A}\hat{q}(s,a',\mathbf{w},\mathbf{w_A},\mathbf{w_v}) = \arg\max_{a'\in A} A(s,a',\mathbf{w},\mathbf{w_A})$, Equation (12) gives $\hat{q}(s,a^*,\mathbf{w},\mathbf{w_A},\mathbf{w_v}) = \hat{v}(s,\mathbf{w},\mathbf{w_v})$. Thus, the stream $\hat{v}$ provides an estimate of the value function, and the other stream A generates advantage estimates.

The authors in [3] also proposed an alternative aggregating module that replaces the max with a mean operator:

$$\hat{q}(s,a,\mathbf{w},\mathbf{w_A},\mathbf{w_v}) = \hat{v}(s,\mathbf{w},\mathbf{w_v}) + \Big(A(s,a,\mathbf{w},\mathbf{w_A}) - \frac{1}{|A|}\sum_{a'} A(s,a',\mathbf{w},\mathbf{w_A})\Big) \tag{13}$$

Although this design in some sense loses the original semantics of $\hat{v}$ and A, the author argued that it improves the stability of learning: the advantages only need to change as fast as the mean, instead of having to compensate any change to the advantage of the optimal action. Therefore, the aggregating module in the dueling network [3] is implemented following Equation (13). When acting, it suffices to evaluate the advantage stream to make decisions.

The advantage of the dueling network lies in its capability of approximating the value function efficiently. This advantage over single-stream Q networks grows when the number of actions is large, and the dueling network achieved state-of-the-art results on Atari games as of 2016.

References

[1] Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." Nature 518.7540 (2015): 529.

[2] Van Hasselt, Hado, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-Learning." AAAI. Vol. 16. 2016.

[3] Wang, Ziyu, et al. "Dueling network architectures for deep reinforcement learning." arXiv preprint arXiv:1511.06581 (2015).

[4] Bellemare, Marc G., et al. "The Arcade Learning Environment: An evaluation platform for general agents." J. Artif. Intell. Res.(JAIR) 47 (2013): 253-279.

[5] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012.

[6] Nair, Vinod, and Geoffrey E. Hinton. "Rectified linear units improve restricted boltzmann machines." Proceedings of the 27th international conference on machine learning (ICML-10). 2010.

[7] Schaul, Tom, et al. "Prioritized experience replay." arXiv preprint arXiv:1511.05952 (2015).