A Details of QA-DQN

Notations

In this section, we use *game step* t to denote one round of interaction between an agent with the QAit environment. We use o_t to denote text observation at game step t, and q to denote question text. We use L to refer to a linear transformation. Brackets $[\cdot;\cdot]$ denote vector concatenation.

A.1 Encoder

We use a transformer-based text encoder, which consists of an embedding layer, two stacks of transformer blocks (denoted as encoder transformer blocks and aggregation transformer blocks), and an attention layer.

In the embedding layer, we aggregate both word- and character-level information to produce a vector for each token in text. Specifically, word embeddings are initialized by the 300-dimensional fastText (Mikolov et al., 2018) word vectors trained on Common Crawl (600B tokens), they are fixed during training. Character level embedding vectors are initialized with 32-dimensional random vectors. A convolutional layer with 64 kernels of size 5 is then used to aggregate the sequence of characters. We use a max pooling layer on the character dimension, then a multi-layer perceptron (MLP) of output size 64 is used to aggregate the concatenation of word- and character-level representations. Highway network (Srivastava et al., 2015) is applied on top of this MLP. The resulting vectors are used as input to the encoding transformer blocks.

Each encoding transformer block consists of a stack of convolutional layers, a self-attention layer, and an MLP. In which, each convolutional layer has 64 filters, each kernel's size is 7, there are 2 such convolutional layers that share weights. In the self-attention layer, we use a block hidden size of 64, as well as a single head attention mechanism. Layernorm and dropout are applied after each component inside the block. We add positional encoding into each block's input. We use one layer of such an encoding block.

At a game step t, the encoder processes text observation o_t and question q, context aware encoding $h_{o_t} \in \mathbb{R}^{L^{o_t} \times H_1}$ and $h_q \in \mathbb{R}^{L^q \times H_1}$ are generated, where L^{o_t} and L^q denote number of tokens in o_t and q respectively, H_1 is 64. Following (Yu et al., 2018), we use an context-query attention layer to aggregate the two representations h_{o_t} and h_q.

Specifically, the attention layer first uses two MLPs to convert both h_{o_t} and h_q into the same space, the resulting tensors are denoted as $h'_{o_t} \in \mathbb{R}^{L^{o_t} \times H_2}$ and $h'_q \in \mathbb{R}^{L^q \times H_2}$, in which H_2 is 64.

Then, a tri-linear similarity function is used to compute the similarities between each pair of h'_{o_t} and h'_q items:

$$S = W[h'_{o_t}; h'_q; h'_{o_t} \odot h'_q], \tag{1}$$

where $\odot$ indicates element-wise multiplication, W is trainable parameters of size 64.

Softmax of the resulting similarity matrix S along both dimensions are computed, this produces S^A and S^B. Information in the two representations are then aggregated by:

$$\begin{aligned} h_{oq} &= [h'_{o_t}; P; h'_{o_t} \odot P; h'_{o_t} \odot Q], \\ P &= S_q h'^{\top}_q, \\ Q &= S_q S^{\top}_{o_t} h'^{\top}_{o_t}, \end{aligned} \tag{2}$$

where h_{oq} is aggregated observation representation.

On top of the attention layer, a stack of aggregation transformer blocks is used to further map the observation representations to action representations and answer representations. The structure of aggregation transformer blocks are the same as the encoder transformer blocks, except the kernel size of convolutional layer is 5, and the number of blocks is 3.

Let $M_t \in \mathbb{R}^{L^{o_t} \times H_3}$ denote the output of the stack of aggregation transformer blocks, where H_3 is 64.

A.2 Command Generator

The command generator takes the hidden representations M_t as input, it estimates Q-values for all action, modifier, and object words, respectively. It consists of a shared Multi-layer Perceptron (MLP) and three MLPs for each of the components:

$$\begin{aligned} R_t &= \text{ReLU}(L_{\text{shared}}(\text{mean}(M_t)), \\ Q_{t,action} &= L_{\text{action}}(R_t), \\ Q_{t,modifier} &= L_{\text{modifier}}(R_t), \\ Q_{t,object} &= L_{\text{object}}(R_t). \end{aligned} \tag{3}$$

In which, the output size of L_{shared} is 64; the dimensionalities of the other 3 MLPs are depending on the number of the amount of action, modifier

and object words available, respectively. The overall Q-value is the sum of the three components:

$$Q_t = Q_{t,action} + Q_{t,modifier} + Q_{t,object}. \quad (4)$$

A.3 Question Answerer

Similar to (Yu et al., 2018), we append an extra stacks of aggregation transformer blocks on top of the aggregation transformer blocks to compute answer positions:

$$\begin{aligned} U &= \text{ReLU}(L_0[M_t; M'_t]). \\ \beta &= \text{softmax}(L_1(U)). \end{aligned} \quad (5)$$

In which $M'_t \in \mathbb{R}^{L^{o_t} \times H_3}$ is output of the extra transformer stack, L_0, L_1 are trainable parameters with output size 64 and 1, respectively.

For location questions, the agent outputs β as the probability distribution of each word in observation o_t being the answer of the question.

For binary classification questions, we apply an MLP, which takes weighted sum of matching representations as input, to compute a probability distribution $p(y)$ over both possible answers:

$$\begin{aligned} D &= \sum_i (\beta^i \cdot M'_t), \\ p(y) &= \text{softmax}(L_4(\tanh(L_3(D)))). \end{aligned} \quad (6)$$

Output size of L_3 and L_4 are 64 and 2, respectively.

A.4 Deep Q-Learning

In a text-based game, an agent takes an action a[4] in state s by consulting a state-action value function $Q(s, a)$, this value function is as a measure of the action's expected long-term reward. Q-Learning helps the agent to learn an optimal $Q(s, a)$ value function. The agent starts from a random Q-function, it gradually updates its Q-values by interacting with environment, and obtaining rewards. Following Mnih et al. (2015), the Q-value function is approximated with a deep neural network.

We make use of a replay buffer. During playing the game, we cache all transitions into the replay buffer without updating the parameters. We periodically sample a random batch of transitions from the replay buffer. In each transition, we update the parameters θ to reduce the discrepancy between the predicted value of current state $Q(s_t, a_t)$ and the expected Q-value given the reward r_t and the value of next state $\max_a Q(s_{t+1}, a)$.

We minimize the temporal difference (TD) error, δ:

$$\delta = Q(s_t, a_t) - (r_t + \gamma \max_a Q(s_{t+1}, a)), \quad (7)$$

in which, γ indicates the discount factor. Following the common practice, we use the Huber loss to minimize the TD error. For a randomly sampled batch with batch size B, we minimize:

$$\begin{aligned} \mathcal{L} &= \frac{1}{|B|} \sum \mathcal{L}(\delta), \\ \text{where } \mathcal{L}(\delta) &= \begin{cases} \frac{1}{2}\delta^2 & \text{for } |\delta| \leq 1, \\ |\delta| - \frac{1}{2} & \text{otherwise.} \end{cases} \end{aligned} \quad (8)$$

As described in Section 3.3.1, we design the sufficient information bonus to teach an agent to stop as soon as it has gathered enough information to answer the question. Therefore we assign this reward at the game step where the agent generates **`wait`** command (or it is forced to stop).

It is worth mentioning that for attribute type questions (considerably the most difficult question type in QAit, where the training signal is very sparse), we provide extra rewards to help QA-DQN to learn.

Specifically, we take a reward similar to as used in location questions: 1.0 if the agent has observed the object mentioned in the question. we also use a reward similar to as used in existence questions: the agent is rewarded by the coverage of its exploration. The two extra rewards are finally added onto the sufficient information bonus for attribute question, both with coefficient of 0.1.

B Implementation Details

During training with vanilla DQN, we use a replay memory of size 500,000. We use ϵ-greedy, where the value of ϵ anneals from 1.0 to 0.1 within 100,000 episodes. We start updating parameters after 1,000 episodes of playing. We update our network after every 20 game steps. During updating, we use a mini-batch of size 64. We use *Adam* (Kingma and Ba, 2014) as the step rule for optimization, The learning rate is set to 0.00025.

When our agent is trained with Rainbow algorithm, we follow Hessel et al. (2017) on most of the hyper-parameter settings. The four MLPs L_{shared}, L_{action}, L_{modifier} and L_{object} as described

[4] In our case, a is a triplet contains {action, modifier, object} as described in Section 2.4.

in Eqn. 3 are Noisy Nets layers (Fortunato et al., 2017) when the agent is trained in Rainbow setting. Detailed hyper-parameter setting of our Rainbow agent are shown in Table 6.

Parameter	Value
Exploration ϵ	0
Noisy Nets σ_0	0.5
Target Network Period	1000 episodes
Multi-step returns n	$n \sim \text{Uniform}[1, 3]$
Distributional atoms	51
Distributional min/max values	[-10, 10]

Table 6: Hyper-parameter setup for rainbow agent.

The model is implemented using PyTorch (Paszke et al., 2017).

C Supported Text Commands

All supported text commands are listed in Table 7.

D Heuristic Conditions for Attribute Questions

Here, we derived some heuristic conditions to determine when an agent has gathered enough information to answer a given attribute question. Those conditions are used as part of the reward shaping for our proposed agent (Section 3.3.1). In Table 8, for each attribute we list all the commands for which their outcome (pass or fail) gives enough information to answer the question correctly. Also, in order for a command's outcome to be informative, each command needs to be executed while some state conditions hold. For example, to determine if an object is indeed a **heat_source**, the agent needs to try to cook something that is cookable and uncooked while standing next to the given object.

E Full results

We provide full results of our agents on **fixed map** games in Table 9, and provide full results of our agents on **random map** games in Table 10. To help investigating the generalizability of the sufficient information bonus we used in our proposed agent, we also report the rewards during both training and test phases. Note during test phase, we do not update parameters with the rewards.

Command	Description
look	describe the current location
inventory	display the player's inventory
go ⟨**dir**⟩	move the player to north, east, south, or west
examine ...	examine something more closely
open ...	open a door or a container
close ...	close a door or a container
eat ...	eat edible object
drink ...	drink drinkable object
drop ...	drop an object on the floor
take ...	take an object from the floor, a container, or a supporter
put ...	put an object onto a supporter (supporter must be present at the location)
insert ...	insert an object into a container (container must be present at the location)
cook ...	cook an object (heat source must be present at the location)
slice ...	slice cuttable object (a sharp object must be in the player's inventory)
chop ...	chop cuttable object (a sharp object must be in the player's inventory
dice ...	dice cuttable object (a sharp object must be in the player's inventory)
wait	stop interaction

Table 7: Supported command list.

Attribute	Command	State	Pass	Fail	Explanation
sharp	**cut** *cuttable*	holding (*cuttable*) & uncut (*cuttable*) & holding (object)	1	1	Trying to cut something cuttable that hasn't been cut yet while holding the object.
	take object	reachable(object)	0	1	Sharp objects should be portable.
cuttable	**cut** object	holding (object) & holding (*sharp*)	1	1	Trying to cut the object while holding something sharp.
	take object	reachable (object)	0	1	Cuttable object should be portable.
edible	**eat** object	holding (object)	1	1	Trying to eat the object.
	take object	reachable (object)	0	1	Edible objects should be portable.
drinkable	**drink** object	holding (object)	1	1	Trying to drink the object.
	take object	reachable (object)	0	1	Drinkable objects should be portable.
holder	**–**	on (*portable*, object)	1	0	Observing object(s) on a supporter.
		in (*portable*, object)	1	0	Observing object(s) inside a container.
	take object	reachable (object)	1	0	Holder objects should not be portable.
portable	**–**	holding (object)	1	0	Holding the object means it is portable.
	take object	reachable (object)	1	1	Portable objects can be taken.
heat_source	**cook** *cookable*	holding (*cookable*) & uncooked (*cookable*) & reachable (object)	1	1	Trying to cook something cookable that hasn't been cooked yet while being next to the object.
	take object	reachable (object)	1	0	Heat source objects should not be portable.
cookable	**cook** object	holding (object) & reachable (*heat_source*)	1	1	Trying to cook the object while being next to a heat source.
	take object	reachable(object)	0	1	Cookable objects should be portable.
openable	**open** object	reachable (object) & closed (object)	1	1	Trying to open the closed object.
	close object	reachable (object) & open (object)	1	1	Trying to close the open object.

Table 8: Heuristic conditions for determining whether the agent has enough information to answer a given attribute question. We use "object" to refer to the object mentioned in the question. Words in italics represents placeholder that can be replaced by any object from the environment that has the appropriate attribute (e.g. carrot could be used as a *cuttable*). The columns Pass and Fail represent how much reward the agent will receive given the corresponding command's outcome (resp. success or failure). NB: **cut** can mean any of the following commands: **slice**, **dice**, or **chop**

Model	Location		Existence		Attribute	
	Train	Test	Train	Test	Train	Test
Human	–	1.000	–	1.000	–	1.000
Random	–	0.027	–	0.497	–	0.496
1 game						
DQN	0.972(0.972)	0.122(0.160)	1.000(0.881)	0.628(0.124)	1.000(0.049)	0.500(0.035)
DDQN	0.960(0.960)	0.156(0.178)	1.000(0.647)	0.624(0.148)	1.000(0.023)	0.498(0.033)
Rainbow	0.562(0.562)	0.164(0.178)	1.000(0.187)	0.616(0.083)	1.000(0.049)	0.516(0.039)
2 games						
DQN	0.698(0.698)	0.168(0.182)	0.948(0.700)	0.574(0.136)	1.000(0.011)	0.510(0.028)
DDQN	0.702(0.702)	0.172(0.178)	0.882(0.571)	0.550(0.109)	1.000(0.098)	0.508(0.036)
Rainbow	0.734(0.734)	0.160(0.168)	0.878(0.287)	0.616(0.085)	1.000(0.030)	0.524(0.022)
10 games						
DQN	0.654(0.654)	0.180(0.188)	0.822(0.390)	0.568(0.156)	1.000(0.055)	0.518(0.030)
DDQN	0.608(0.608)	0.188(0.208)	0.842(0.479)	0.566(0.128)	1.000(0.064)	0.516(0.036)
Rainbow	0.616(0.616)	0.156(0.170)	0.768(0.266)	0.590(0.131)	0.998(0.059)	0.520(0.023)
100 games						
DQN	0.498(0.498)	0.194(0.206)	0.756(0.139)	0.614(0.160)	0.838(0.019)	0.498(0.014)
DDQN	0.456(0.458)	0.168(0.196)	0.768(0.134)	0.650(0.216)	0.878(0.020)	0.528(0.017)
Rainbow	0.340(0.340)	0.156(0.160)	0.762(0.129)	0.602(0.207)	0.924(0.044)	0.524(0.022)
500 games						
DQN	0.430(0.430)	0.224(0.244)	0.742(0.136)	0.674(0.279)	0.700(0.015)	**0.534**(0.014)
DDQN	0.406(0.406)	0.218(0.228)	0.734(0.173)	0.626(0.213)	0.714(0.021)	0.508(0.026)
Rainbow	0.358(0.358)	0.190(0.196)	0.768(0.187)	0.656(0.207)	0.736(0.032)	0.496(0.029)
unlimited games						
DQN	0.300(0.300)	0.216(0.216)	0.752(0.119)	0.662(0.246)	0.562(0.034)	0.514(0.016)
DDQN	0.318(0.318)	0.258(0.258)	0.744(0.168)	0.628(0.134)	0.572(0.027)	0.480(0.024)
Rainbow	0.316(0.330)	**0.280**(0.280)	0.734(0.157)	**0.692**(0.157)	0.566(0.017)	0.514(0.014)

Table 9: Agent performance on **fixed map** games. Accuracies in percentage are shown in black. We also investigate the sufficient information bonus used in our agent proposed in Section 3.3.1, which are shown in blue.

Model	Location		Existence		Attribute	
	Train	Test	Train	Test	Train	Test
Human	–	1.000	–	1.000	–	0.750
Random	–	0.034	–	0.500	–	0.499
2 games						
DQN	0.990(0.990)	0.148(0.162)	1.000(0.779)	0.638(0.157)	1.000(0.039)	0.534(0.033)
DDQN	0.978(0.978)	0.146(0.152)	1.000(0.727)	0.602(0.158)	1.000(0.043)	**0.544**(0.032)
Rainbow	0.916(0.916)	0.178(0.178)	0.972(0.314)	0.602(0.136)	1.000(0.025)	0.512(0.021)
10 games						
DQN	0.818(0.818)	0.156(0.160)	0.898(0.607)	0.566(0.142)	1.000(0.056)	0.518(0.036)
DDQN	0.794(0.794)	0.142(0.154)	0.868(0.575)	0.606(0.153)	1.000(0.037)	0.500(0.033)
Rainbow	0.670(0.670)	0.144(0.170)	0.828(0.468)	0.586(0.128)	1.000(0.071)	0.530(0.018)
100 games						
DQN	0.550(0.550)	0.184(0.204)	0.758(0.230)	0.668(0.181)	0.878(0.021)	0.524(0.017)
DDQN	0.524(0.524)	0.188(0.204)	0.754(0.365)	0.662(0.205)	0.890(0.025)	**0.544**(0.019)
Rainbow	0.442(0.442)	0.174(0.184)	0.754(0.285)	0.654(0.190)	0.878(0.044)	0.504(0.032)
500 games						
DQN	0.430(0.430)	0.204(0.216)	0.752(0.162)	0.678(0.214)	0.678(0.019)	0.530(0.017)
DDQN	0.458(0.458)	0.222(0.246)	0.754(0.158)	0.656(0.188)	0.716(0.024)	0.486(0.023)
Rainbow	0.370(0.370)	0.172(0.178)	0.748(0.275)	0.678(0.191)	0.636(0.020)	0.494(0.017)
unlimited games						
DQN	0.316(0.316)	0.188(0.188)	0.728(0.213)	0.668(0.218)	0.812(0.055)	0.506(0.018)
DDQN	0.326(0.326)	0.206(0.206)	0.740(0.246)	**0.694**(0.196)	0.580(0.023)	0.482(0.017)
Rainbow	0.340(0.340)	**0.258**(0.258)	0.728(0.210)	0.686(0.193)	0.564(0.018)	0.470(0.017)

Table 10: Agent performance on **random map** games. Accuracies in percentage are shown in black. We also investigate the sufficient information bonus used in our agent proposed in Section 3.3.1, which are shown in blue.

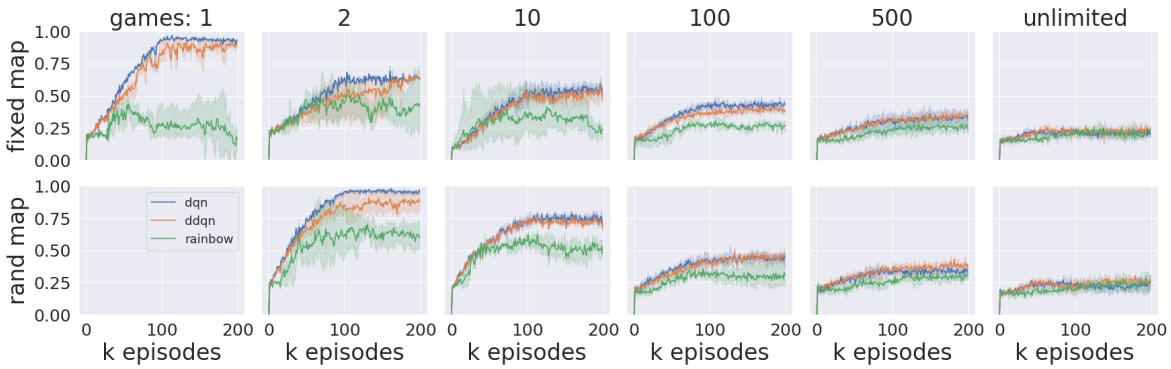

Figure 4: Training accuracy over episodes on **location** questions. Upper row: **fixed map**, 1/2/10/100/500/unlimited games; Lower row: **random map**, 2/10/100/500/unlimited games.

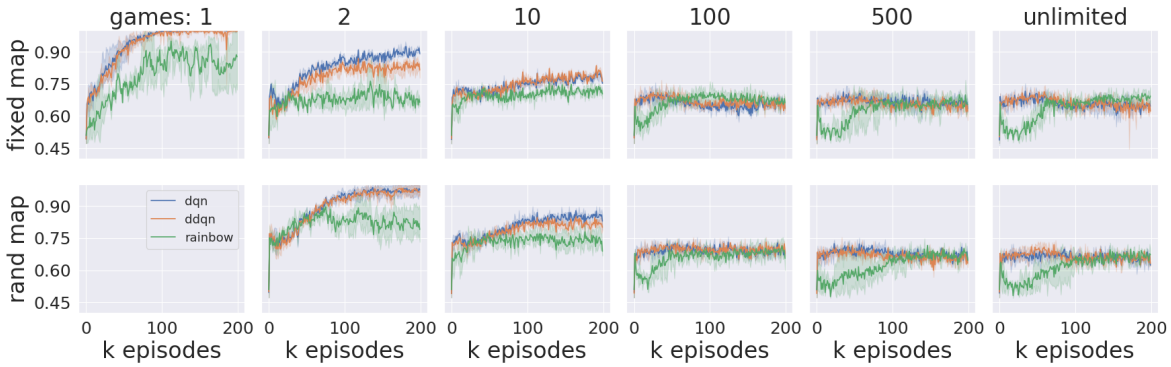

Figure 5: Training accuracy over episodes on **existence** questions. Upper row: **fixed map**, 1/2/10/100/500/unlimited games; Lower row: **random map**, 2/10/100/500/unlimited games.

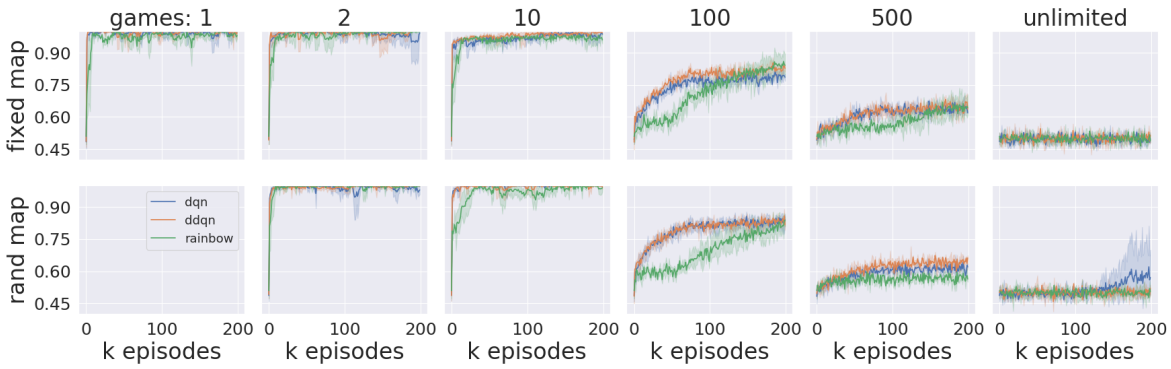

Figure 6: Training accuracy over episodes on **attribute** questions. Upper row: **fixed map**, 1/2/10/100/500/unlimited games; Lower row: **random map**, 2/10/100/500/unlimited games.