LEARNING TO NAVIGATE THE WEB

Anonymous authors
Paper under double-blind review

ABSTRACT

Learning in environments with large state and action spaces as well as sparse rewards can hinder the ability of a Reinforcement Learning (RL) agent to learn through trial-and-error. For instance, the problem of following natural language instructions on the Web (such as booking a flight ticket) leads to RL settings where input vocabulary and number of actionable elements on a page can grow very large. Even though recent approaches improve the success rate on relatively simpler environments with the help of human demonstrations to guide the exploration, they still fail in environments where the set of possible instructions can reach millions. We approach the aforementioned problems from a different perspective and propose a meta-trainer that can generate unbounded amount of experience for an agent to learn from. Instead of learning from a complicated instruction with a large vocabulary, we decompose it into multiple sub-instructions and schedule a curriculum in which an agent is tasked with gradually increasing subset of these relatively easier sub-instructions. We train DQN, deep reinforcement learning agent, with Q-value function approximated with a novel QWeb neural network architecture on these smaller, synthetic instructions. We evaluate the ability of our agent to generalize to new instructions on World of Bits benchmark, on forms with 100 elements, supporting 14 million possible instructions. The QWeb agent outperforms the baseline without using any human demonstration achieving 100% success rate on several difficult environments.

1 INTRODUCTION

We study the problem of training reinforcement learning agents to navigate the Web (*navigator agent*) by following certain instructions, such as *book a flight ticket* or *interact with a social media web site*, that require learning through large state and action spaces with sparse and delayed rewards. In a typical web environment, an agent might need to carefully navigate through a large number of web elements to follow highly dynamic instructions formulated from large vocabularies. For example, in the case of an instruction "Book a flight from WTK to LON on 21-Oct-2016", the agent needs to fill out the origin and destination drop downs with the correct airport codes, select a date, hit submit button, and select the cheapest flight among all the options. Note the difficulty of the task: The agent can fill-out the first three fields in any order. The options for selection are numerous, among all possible airport / date combination only one is correct. The form can only be submitted once all the three fields are filled in. At that point the environment / web page changes, and flight selection becomes possible. Then the agent can select and book a flight. Reaching the true objective in these tasks through trial-and-error is cumbersome, and reinforcement learning with the sparse reward results in the majority of the episodes generating no signal at all. The problem is exacerbated when learning from large set of instructions where visiting each option could be infeasible. As an example, in the flight-booking environment the number of possible instructions / tasks can grow to more than 14 millions, with more than 1700 vocabulary words and approximately 100 web elements at each episode.

A common remedy for these problems is guiding the exploration towards more valuable states by learning from human demonstrations and using pretrained word embeddings. Previous work (Liu et al. (2018); Shi et al. (2017)) has shown that the success rate of an agent on Web navigation tasks (Miniwob (Shi et al. (2017))) can be improved via human demonstrations and pretrained word embeddings; however, they indeed use separate demonstrations for each environment and as the complexity of an environment increases, these methods fail to generate any successful episode (such

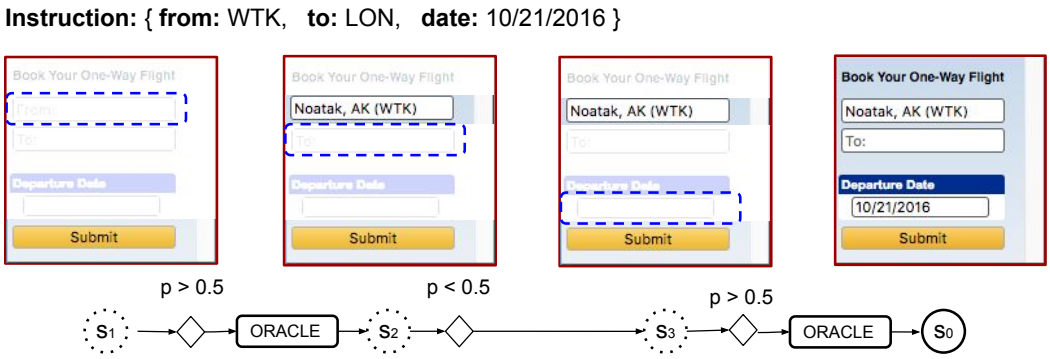

Figure 1: Curriculum learning with warm-starting an episode. Final state is used as the initial state for training the navigator agent.

as flight booking and social media interaction environments). But in environments with large state and action spaces, gathering the human demonstrations does not scale, as the training needs large number of human demonstrations for each environment.

In this work, we tackle the aforementioned problems by augmenting each environment with fine-grained signal rewards that can generate unbounded amount of rich experience for the agent to harness. Instead of guiding the exploration via human demonstrations, we develop *curriculum learning* methods where we decompose an instruction into multiple sub-instructions and the agent is assigned with an easier task of solving only a subset of these sub-instructions. We employ two different methods to implement curriculum learning: warm-starting an episode and simulating sub-goals.

We warm-start an episode by placing the agent closer to the goal state where only a small set of the sub-instructions is needed to successfully finish the episode. This process is illustrated in Figure 1 for the flight-booking environment where the environment is initialized with the final state (s_0) and navigator agent is tasked only with selecting correct destination airport. We also simulate sub-goals by constraining an episode to a subset of the elements in the environment where only the corresponding sub-instructions are needed to successfully finish an episode. Final state of this process is used as the new goal and the agent receives a positive reward if it can successfully reach this sub-goal. Instead of altering the initial or goal state of an environment, we alternatively practice with augmenting the environment with a potential-based reward (Ng et al. (1999)) by measuring the alignment between a given state and the goal state.

Finally, we combine the above approaches and propose a novel *meta-training* approach where a high level meta-agent trains the navigator agent by generating (instruction, goal) pairs and supplementing by fine-grained reward signals. We first train an *instruction generator* agent that learns to generate instructions from a given goal state. Next, we execute a *rule-based randomized policy* that visits each element and performs an action with arguments randomly picked from a given knowledge source. By generating an instruction from the final state, we set up the environment and the navigator agent is trained by the meta-trainer. One can think of a range of signals to train the navigator such as supervised pretraining using behavioral cloning or generating curriculum from the episodes generated by meta-trainer, using meta-trainer as off-policy learning, augmenting environment with meta-trainer reward, etc. In this work, we use reward augmentation where the meta-trainer generates a potential-based reward by comparing a given state with the simulated goal as described above.

We test the performance of our approaches on a set of Miniwob and Miniwob++ tasks (Liu et al. (2018)). We show that each of our approaches improve upon a strong baseline and outperform previous state-of-the-art.

While we focus on the Web navigation, the methods presented here, automated curriculum generation with attention-equipped DQN, might be of interest to the larger task planning community working to solve goal-oriented tasks in large discrete state and action Makrov Decision Processes.

2 RELATED WORK

Our work is closely related to previous works on training reinforcement learning policies for navigation tasks. Shi et al. (2017) and Liu et al. (2018) both work on web navigation tasks and aim to leverage human demonstrations to guide the exploration towards more valuable states where the focus of our work is investigating the potential of curriculum learning approaches and building a novel framework for meta-training of our deep Q networks. Compared to our biLSTM encoder for encoding DOM trees, Liu et al. (2018) encodes them by extracting spatial and hierarchical features. Diagonal to using pretrained word embeddings as in (Liu et al. (2018)), we used shallow encodings to enhance learning semantic relationships between DOM elements and instructions. Shah et al. (2018) also utilizes an attention-based DQN for navigating in home environments with visual inputs. They jointly encode visual representation of the environment and natural language instruction using attention mechanisms and generate Q values for a set of atomic actions for navigating in a 3D environment.

Modifying the reward function (Ng et al. (1999); Abbeel & Ng (2004)) is one practice that encourages the agent to get more fine-grained signals using potentials which also motivated our augmented rewards for Web navigation tasks. Curriculum learning methods (Bengio et al. (2009); Zaremba & Sutskever (2014); Graves et al. (2017)) are studied to divide a complex task into multiple small sub-tasks that are easier to solve. Our warm-start curriculum learning is closely related to (Florensa et al. (2017)) where a robot is placed closer to a goal state in a 3D environment and the start state is updated based on robot's performance.

Meta-learning is used to exploit past experiences to continuously learn on new tasks (Andrychowicz et al. (2016); Duan et al. (2016); Wang et al. (2016)). Frans et al. (2017) introduced a meta-learning approach where task-specific policies are trained in a multi-task setup and a set of primitives are shared between tasks. Our meta-training framework differs from previous works where we generate instruction and goal pairs to set the environment and provide fine-grained reward signals for a low level navigator agent to effectively train.

3 SETUP

We are interested in training a RL agent using Q learning that learns a value function $Q(s, a)$ to map a given state s to values over the possible set of actions a. At each time step, the agent observes a state s_t, takes an action a_t, and observes a new state s_{t+1} and a reward $r_t = r(s_{t+1}, a_t)$. Our aim is to maximize the sum of discounted rewards $\sum_t \gamma^t r_t$ by rolling out episodes as suggested by $Q(s, a)$ and accumulating the reward. We particularly focus on the case where the reward is sparse and only available at the end of an episode. More specifically, for only a small fraction of episodes that are successful, the reward is $+1$; in other cases it is -1. This setup combined with large state and action spaces make it difficult to train a Q learning model that can successfully navigate in a Web environment.

In this work, we further make the following assumption where we are given an instruction $I = [F = (K, V)]$ as a list of fields F where each field is represented as a key-value pair (K, V) (ex. *from*: "San Francisco", *to*: "LA", *date*: "12/04/2018"). At each time step, the state of the environment s_t consists of the instruction I and a representation of the web page as a hierarchy of DOM elements D_t. Each DOM element is represented as a list of named attributes such as *tag*, *value*, *name*, *text*, *id*, *class*. We also assume that the reward of the environment is computed by comparing the final state of an episode (D_N) with the final goal state $G(I)$. Following Liu et al. (2018), we constrain the action space to `Click(e)` and `Type(e, y)` actions where e is a leaf node in the DOM tree and y is a value from the instruction. We represent these composite actions using a hierarchy of atomic actions defined by the dependency graph in Figure 2. Following this layout, we define our composite Q value

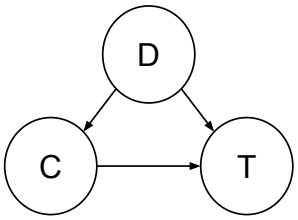

Figure 2: Action dependency graph for hierarchical Q learning. D denotes a DOM element, C denotes a click or type action, and T denotes a type sequence.

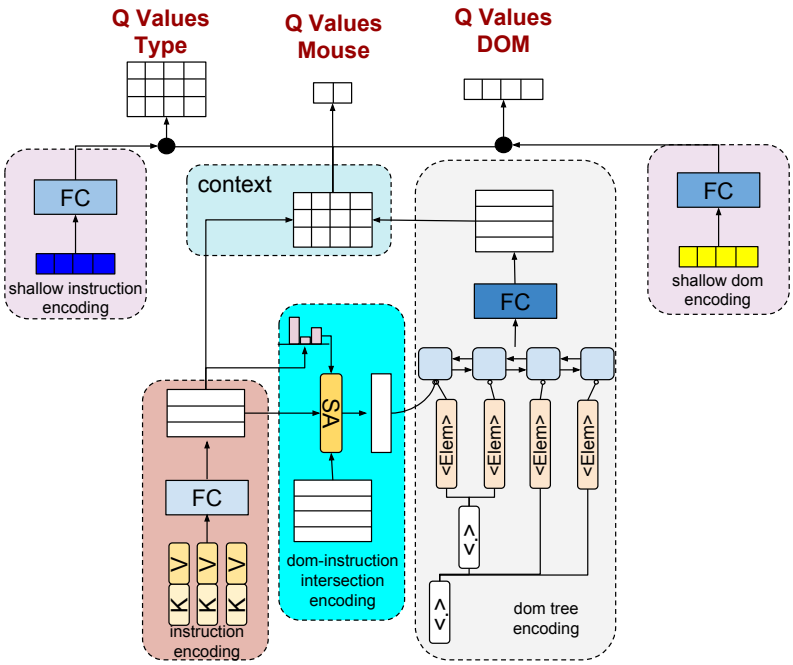

Figure 3: QWeb network: boxes indicate fully connected layers (FC) with ReLU (for instruction encoding) or tanh (for shallow encoding) activation. (K, V) indicates embeddings of key and value pairs on the instruction; <*Elem*> shows the leaf DOM element embeddings. SA denotes a self-attention mechanism that generates a distribution over the instruction fields. Black circles indicate the gating mechanism to join Q values generated by shallow and deep encodings.

function that we use in this work :

$$Q(s,a) = Q(s,a_D) + Q(s,a_C|a_D) + Q(s,a_T|a_D,[a_C == \text{"}type\text{"}]) \quad (1)$$

where $a = (a_D, a_C, a_T)$, a_D denotes a DOM element selection, $a_C|a_D$ denotes a click or type action given the DOM element, and $a_T|a_D,[a_C == \text{"}type\text{"}]$ denotes typing a sequence action given the DOM element. When executing the policy (during exploration or during testing), the agent first picks a DOM element with the highest $Q(s,a_D)$; then decides to `Type` or `Click` on the chosen DOM element based on $Q(s,a_C|a_D)$; and for a type action, the agent selects a value from the instruction using $Q(s,a_T|a_D,[a_C == \text{"}type\text{"}])$.

4 Guided Q Learning for Web Navigation

We now describe our proposed models for handling large state and action spaces with sparse rewards. We first describe our deep Q network, called QWeb, for generating Q values for a given observation ($s_t = (I, D_t)$) and for each atomic action a_D, a_C, a_T. Next, we explain how we extend this network with shallow dom and instruction encoding layers to mitigate the problem of learning a large number of input vocabulary. Finally, we delve into our reward augmentation and curriculum learning approaches to solve the aforementioned problems.

4.1 Deep Q Network for Web Navigation (QWeb)

QWeb is composed of three different layers linked in a hierarchical structure where each layer encodes a different component of a given state (Figure3): (i) Encoding user instruction, (ii) Encoding the overlapping words between attributes of the DOM elements and instruction, and (iii) Encoding DOM tree. Given an instruction $I = [F = (K, V)]$, QWeb first encodes each field F into a fixed length vector by learning an embedding for each K and V. The sequence of overlapping words

between DOM element attributes and instruction fields is encoded into a single vector to condition each element on contextually similar fields. Finally, the DOM tree is encoded by linearizing the tree structure (Vinyals et al. (2015)) and running a bidirectional LSTM (biLSTM) network on top of the DOM elements sequence. Output of the LSTM network and encoding of the instruction fields are used to generate Q values for each atomic action.

4.1.1 ENCODING USER INSTRUCTIONS

We represent an instruction with a list of vectors where each vector corresponds to a different instruction field. A field is encoded by encoding its corresponding key and value and transforming the combined encodings via a FC layer with ReLU activation. Let $E_K^f(i,j)$ ($E_V^f(i,j)$) denote the embedding of the j-th word in the key (value) of i-th field. We represent a key or value as the average of these embeddings over the corresponding words, i.e., $E_K^f(i) = \sum_j E_K^f(i,j)$ represents the encoding of a key. Encoding of a field is then computed as follows : $E^f(i) = FC([E_K(i), E_V(i)])$ where $[.]$ denotes vector concatenation.

4.1.2 ENCODING DOM-INSTRUCTION INTERSECTION

For each field in the instruction and each attribute of a DOM element, we generate the sequence of overlapping words. By encoding these sequences in parallel, we generate instruction-aware DOM element encodings. We average the word embeddings over each sequence and each attribute to compute the embedding of a DOM element conditioned on each instruction field. Using a self-attention mechanism, we compute a probability distribution over instruction fields and reduce this instruction-aware embedding into a single DOM element encoding. Let $E(f, D_t(i))$ denote the embedding of a DOM element conditioned on a field f where $D_t(i)$ is the i-th DOM element. Conditional embedding of $D_t(i)$ is the weighted average of these embeddings, i.e., $E_C = \sum_f p_f * E(f, D_t(i)$ where self-attention probabilities are computed as $p_f = softmax_i(u * E^f)$ with u being a trainable vector.

4.1.3 ENCODING DOM TREES

We represent an attribute by averaging its word embeddings. Each DOM element is encoded as the average of its attribute embeddings. Given conditional DOM element encodings, we concatenate these with DOM element embeddings to generate a single vector for each DOM element. We run a bidirectional LSTM (biLSTM) network on top of the list of DOM element embeddings to encode the DOM tree. Each output vector of the biLSTM is then transformed through a FC layer with tanh to generate DOM element representations.

4.1.4 GENERATING Q VALUES

Given encodings for each field in the instruction and each DOM element in the DOM tree, we compute the pairwise similarities between each field and each DOM element to generate a context matrix M. Rows and columns of M show the posterior values for each field and each DOM element in the current state, respectively. By transforming through a FC layer and summing over the rows of M, we generate Q values for each DOM element, i.e., $Q(s_t, a_t^D)$. We use the rows of M as the Q values for typing a field from the instruction to a DOM element, i.e., $Q(s_t, a_t^T) = M$. Finally, Q values for *click* or *type* actions on a DOM element are generated by transforming the rows of M into 2 dimensional vectors using another FC layer, i.e., $Q(s_t, a_t^C)$. Final Q value for a composite action a_t is then computed by summing these Q values : $Q(s_t, a_t) = Q(s_t, a_t^D) + Q(s_t, a_t^T) + Q(s_t, a_t^C)$.

4.1.5 INCORPORATING SHALLOW ENCODINGS

In a scenario where the reward is sparse and input vocabulary is large, such as in flight-booking environments with hundreds of airports, it is difficult to learn a good semantic similarity using only word embeddings. We augment our deep Q network with shallow instruction and DOM tree encodings to alleviate this problem. A joint shallow encoding matrix of fields and elements is generated by computing word-based similarities (such as jaccard similarity, binary indicators such as subset or superset) between each instruction field and each DOM element attribute. We also append shallow encodings of siblings of each DOM element to explicitly incorporate DOM hiearchy. We sum over columns and rows of the shallow encoding matrix to generate a shallow input vector for

DOM elements and instruction fields, respectively. These vectors are transformed using a FC layer with tanh and scaled via a trainable variable to generate a single value for a DOM element and a single value for an instruction field. Using a gating mechanism between deep Q values and shallow Q values, we compute final Q values as follows:

$$\hat{Q}(s,a) = Q_{deep}(s_t, a_t^D)(1-\sigma(u)) + Q_{shallow}(s, a_t^D)(\sigma(u)) \tag{2}$$

$$\hat{Q}(s,a) = Q_{deep}(s_t, a_t^T)(1-\sigma(v)) + Q_{shallowT}(s, a_t^T)(\sigma(v)) \tag{3}$$

where u and v are scalar variables learned during training.

4.2 Reward Augmentation

Following Ng et al. (1999), we use potential based rewards for augmenting the environment reward function. Since the environment reward is computed by evaluating if the final state is exactly equal to the goal state, we compute the potential by the number of matching DOM elements between a given state and the goal state; normalized by the number of DOM elements in the goal state. Potential based reward is then computed as the scaled difference between two potentials for the next state and current state:

$$R_{potential} = \gamma(Potential(s_{t+1}, g) - Potential(s_t, g)) \tag{4}$$

where g is the goal state. For example, in the flight-booking environment, there are 3 DOM elements that the reward is computed from. Let's assume that at current time step the agent correctly enters the date. In this case, the potential for the current state will increase by $1/3$ compared to the potential of the previous state and the agent will receive a positive reward. Keep in mind that not all actions generate a non-zero potential difference. In our example, if the action of the agent is to click on the date picker, not to choose a specific date, the potential will remain unchanged.

4.3 Curriculum Learning

We perform curriculum learning by decomposing an instruction into multiple sub-instructions and assigning the agent with an easier task of solving only a subset of these sub-instructions. We practiced two different curriculum learning strategies to train QWeb: (i) Warm-starting an episode and (ii) Simulating sub-goals.

Warm-Start. We warm-start an episode by placing the agent closer to the goal state where the agent only needs to learn to perform a small number of sub-instructions to successfully finish the episode. We independently visit each DOM element with a certain probability p and probe an *Oracle* policy to perform a correct action on the selected element. The environment for the QWeb is initialized with the final state of the Warm-Start process and the original goal of the environment is kept the same. This process is also illustrated in Figure 1 for the flight-booking environment. In this example scenario, QWeb starts from the partially filled web form (origin and departure dates are already entered) and only tasked with learning to correctly enter the destination airport. At the beginning of the training, we start p with a large number (such as 0.85) and gradually decay towards 0.0 over a predefined number of steps. After this limit, the initial state of the environment will revert to the original state of plain DOM tree.

Goal Simulation. We simulate simpler but related sub-goals for the agent by constraining an episode to a subset of the elements in the environment where only the corresponding sub-instructions are needed to successfully finish an episode. We randomly pick a subset of elements of size K and probe the *Oracle* to perform correct set of actions on this subset. The goal of the current episode for QWeb will be the final state of Goal Simulation process and the initial state of the environment will remain unchanged. QWeb will receive a positive reward if it can successfully reach to this sub-goal. At the beginning of the training, we start K with 1 and gradually increase it towards the maximum number of elements in the DOM tree over a number of steps. After this limit, the environment will revert to the original environment as above.

5 Meta-Trainer for Training QWeb

We combine the curriculum learning and reward augmentation under a more general unified framework and propose a novel meta-training approach where a high level meta-agent trains the navigator

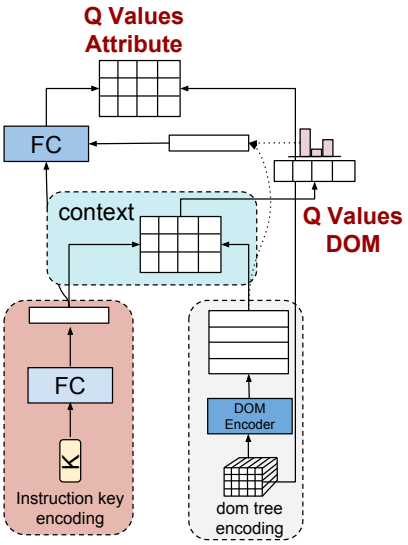

Figure 4: INET Model: Instruction generation from goal states. Only shown with attribute pointing.

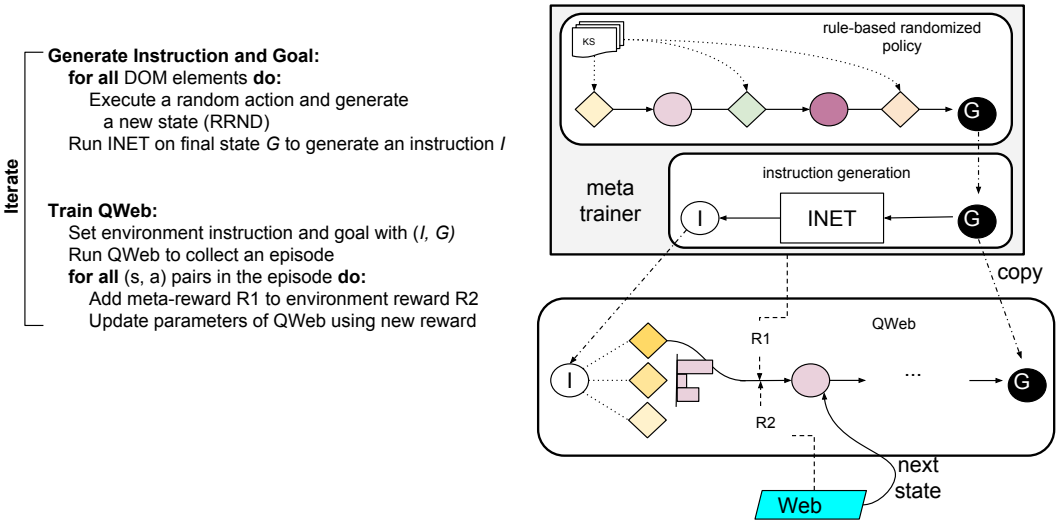

Figure 5: MetaQWeb: Meta trainer model for training QWeb agent.

agent by generating (instruction, goal) pairs and supplementing by fine-grained signals (Figure 5). We first train an instruction generator agent (INET) that given a goal state and an instruction, it learns to generate correct value for each field in the instruction by pointing to attributes of DOM elements or by pointing to words from a predefined knowledge source (Figure 4). We build a meta-trainer framework that executes the following high-level actions for each individual episode to train QWeb: (i) Run a rule-based randomized policy (RRND) that visits each DOM element and executes an action with a random parameter from a pre-given Knowledge Source (KS), (ii) Run INET to generate an instruction from the output state generated by RRND, (iii) Set the instruction and goal of the environment with the generated instruction and goal for the current episode, and (iv) Generate a fine-grained signal when probed by the QWeb agent.

5.1 LEARNING INSTRUCTIONS FROM GOAL STATES

Instruction Generation Environment We build a reinforcement learning environment for instruction generation task. We denote a *state* by a single *key* from the set of possible keys predefined for each environment. At each time step, the environment generates the next state by randomly picking from the set of keys not visited before. *Actions* are composite actions as in web navigation tasks : *selecting a DOM element* and *copying the value of an attribute from the selected element or from a global knowledge source*. After each action, agent receives a positive *reward* (+1) if the value of the corresponding key is correct; otherwise a negative reward (-1).

Q Network for Instruction Generation (INET) Similar to QWeb, INET generates a vector representation for each DOM element by encoding the DOM tree using a biLSTM network. INET encodes the state (key) using a similar encoding layer as in QWeb instruction encoding and generates a single vector representing the state. A context is then generated from DOM element encodings and state encoding which gives a score over the DOM elements conditioned on the given state. We use these scores as Q values for DOM elements, i.e., $Q^I_D(s_t, a_t)$. Next, we generate a distribution over DOM elements by transforming the generated Q values into a probability distribution using a softmax operation and reduce the DOM element encodings into a single DOM encoding. After concatenating each DOM element encoding with the key encoding and transforming through a FC layer with tanh, we attend over DOM element attributes and generate Q values for each attribute, i.e., $Q^I_A(s_t, a_t)$. Final Q values are computed by combining these Q values : $Q^I(s_t, a_t) = Q^I_D(s_t, a_t) + Q^I_A(s_t, a_t)$.

5.2 META-TRAINING OF THE QWEB (METAQWEB)

We first design a simple rule-based randomized policy (RRND) that iteratively visits each DOM element in the current state and takes an action. If the action is `Click(e)`, the agent clicks on the element and the process continues. If the DOM element is part of a group, that their values depend on the state of other elements in the group (such as radio buttons), RRND clicks on one of them randomly and ignores the others. However, if the action is `Type(e, t)`, the type sequence is randomly selected from a given knowledge source (KS). Consider the flight-booking environment as example, if the visited element is a text box, RRND randomly picks an airport from a list of available airports and types into the text box. RRND stops after each DOM element is visited and final DOM tree (D) is generated. Using pretrained INET model, we generate an instruction I from D and set the environment for the current episode using (I, D) pair. After QWeb takes an action and observes a new state, new state is sent to the meta-trainer to gather a fine-grained reward signal ($R1$) which is added to the environment reward to compute the final reward, i.e., $R = R1 + R2$.

Discussion. There are numerous possibilities to utilize MetaQWeb to provide various fine-grained signals such as generating supervised episodes and performing behavioral cloning, scheduling a curriculum from the episodes generated by meta-trainer, using MetaQWeb as a behavior policy for off-policy learning, generating augmented rewards for assessing the performance of QWeb, etc. In this work, we practice generating augmented rewards using MetaQWeb and leave the other cases as future work.

6 EXPERIMENTAL RESULTS

In this section, we evaluate QWeb and MetaQWeb by comparing to previous state-of-the-art approaches on web navigation tasks.

6.1 EXPERIMENTAL SETUP

We evaluate our approaches on a number of environments from Miniwob Shi et al. (2017) and Miniwob++ Liu et al. (2018) benchmark tasks. We use a set of tasks that require various combinations of clicking and typing to set a baseline for the QWeb including a difficult environment, *social-media-all*, where previous approaches fail to generate any succesful episodes. We then conduct extensive experiments on a more difficult environment, *book-flight-form* (a clone of the original book-flight environment with only the initial web form is used), that require learning through a large number of states and actions. Each task consists of a structured instruction (also presented in natural language)

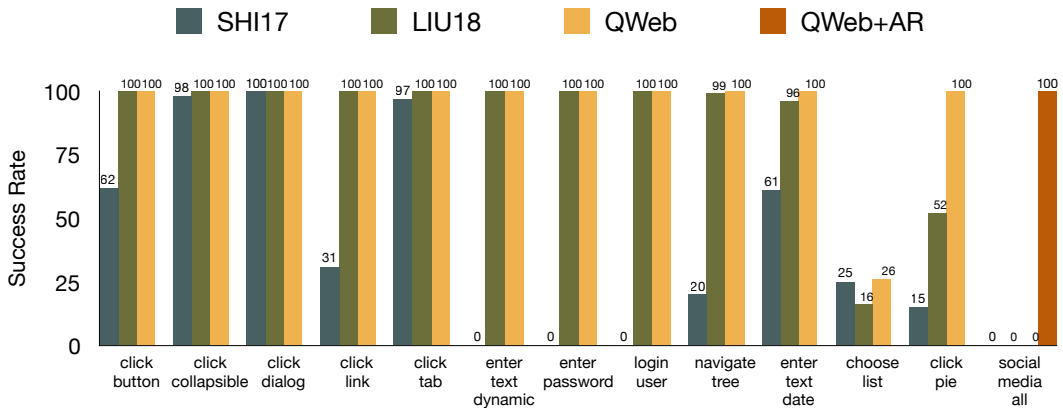

Figure 6: Performance of QWeb on a subset of Miniwob and Miniwob++ environments compared to previous state-of-the-art approaches. AR denotes augmented reward.

and a 160px x 210px environment represented as a DOM tree. All the environments return a sparse reward at the end of an episode with (+1) for successful and (-1) for failure episodes, respectively. We also use a small step penalty (-0.1) to encourage QWeb to find successful episodes using as small number of actions as possible.

As evaluation metric, we follow previous work and report *success rate* which is the percentage of successful episodes that end with a +1 reward. For instruction generation tasks, *success rate* is +1 if all the values in the instruction is correctly generated.

6.2 Previous Approaches

We compare the performance of QWeb to previous state-of-the-art approaches:

- SHI17 (Shi et al. (2017)) pre-trains with behavioral cloning using approximately 200 demonstrations for each environment and fine-tunes using RL. They mainly use raw pixels with several features to represent states.
- LIU18 (Liu et al. (2018)) uses an alternating training approach where a program-policy and a neural-policy are trained iteratively. Program-policy is trained to generate a high level workflow from human demonstrations. Neural-policy is trained using an actor-critic network by exploring states suggested by program-policy.

6.3 Performance on Miniwob Environments

We evaluate the performance of QWeb on a set of simple and difficult Miniwob environments based on the size of state and action spaces and the input vocabulary. On the simple Miniwob environments (first 11 environments in Figure 6), we show the performance of QWeb without any shallow encoding, reward augmentation or curriculum learning. Figure 6 presents the performance of QWeb compared to previous approaches. We observe that, QWeb can match the performance of previous state-of-the-art on every simple environment; setting a strong baseline for evaluating our improvements on more difficult environments. We can also loosely confirm the effectiveness of using a biLSTM encoder instead of extracting features for encoding DOM hierarchy where biLSTM encoding gives consistently competitive results on each environment.

Using shallow encoding and augmented rewards, we also evaluate on more difficult environments (*click-pie* and *social-media-all*). On *click-pie* environment, the episode length is small (1) but the agent needs to learn a relatively large vocabulary (all the letters and digits in English language) to correctly deduce the semantic similarities between instruction and DOM elements. For *social-media-all*, episode length as well as state and action spaces are larger and generating a correct action requires incorporating siblings of a DOM element. In both of these environments, QWeb successfully solves the task and outperforms the previous approaches without using any human

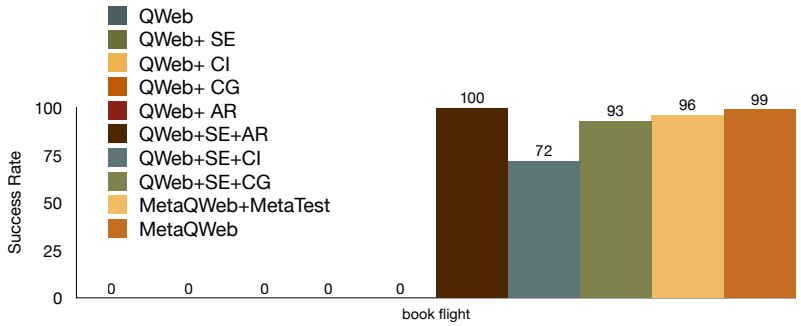

Figure 7: Performance of variants of QWeb and MetaQWeb on the book-flight-form environment. SE, CI, CG, and AR denote shallow encoding, curriculum with warm-start, curriculum with simulated sub-goals, and augmented reward, respectively.

demonstration. Our empirical results suggest that it is critical to use both shallow encoding and augmented rewards in *social-media-all* environment where without these improvements, we weren't able to train a successful agent.

6.4 Performance on Book Flight Environment

In Figure 7, we present the effectiveness of each of our improvements on *book-flight-form* environment. Without using any of the improvements or using only a single improvement, QWeb is not able to generate any successful episodes. The main reason is that without shallow encoding, QWeb is not able to learn a good semantic matching even with a dense reward; with only shallow encoding vast majority of episodes still produce -1 reward that prevents QWeb from learning. When we analyze the cause of the second case, we observe that as the training continues QWeb starts clicking submit button at first time step to get the least negative reward. When we remove the step penalty or give no reward for an unsuccessful episode, QWeb converges to one of the modes: clicking submit button at first time step or generating random sequence of actions to reach step limit where no reward is given. Using shallow encoding, both curriculum approaches offer very large improvements reaching above 90% success rate. When the reward can easily be augmented with a potential-based dense reward, we get the most benefit and completely solve the task.

Before we examine the performance of MetaQWeb, we first evaluate the performance of INET on generating successful instructions to understand the effect of the noise that MetaQWeb introduces into training QWeb. We observe that INET gets 96% success rate on fully generating an instruction. When we analyze the errors, we see that the majority of them comes from incorrect DOM element prediction (75%). Furthermore, most of the errors are on date field (75%) where the value is mostly copied from an incorrect DOM element. After we train QWeb using meta-training, we evaluate its performance in two different cases : (i) We meta-test QWeb using the instruction and goal pairs generated by MetaQWeb to analyze the robustness of QWeb to noisy instructions and (ii) We test using the original environment. Figure 7 suggests that in both of the cases, MetaQWeb generates very strong results reaching very close to solving the task. When we examine the error cases for meta-test use case, we observe that 75% of the errors come from incorrect instruction generations where more than 75% of those errors are from incorrect date field. When evaluated using the original setup, the performance reaches up to 99% success rate showing the effectiveness of our meta-training framework for training a successful QWeb agent.

References

Pieter Abbeel and Andrew Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the International Conference on Machine Learning*, ICML '04, pp. 1–, New York, NY, USA, 2004. ACM. ISBN 1-58113-838-5. doi: 10.1145/1015330.1015430. URL http://doi.acm.org/10.1145/1015330.1015430.

Marcin Andrychowicz, Misha Denil, Sergio Gomez Colmenarejo, Matthew W. Hoffman, David Pfau, Tom Schaul, and Nando de Freitas. Learning to learn by gradient descent by gradient

descent. *CoRR*, abs/1606.04474, 2016. URL http://arxiv.org/abs/1606.04474.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the International Conference on Machine Learning*, ICML '09, pp. 41–48, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-516-1. doi: 10.1145/1553374.1553380. URL http://doi.acm.org/10.1145/1553374.1553380.

Yan Duan, John Schulman, Xi Chen, Peter L. Bartlett, Ilya Sutskever, and Pieter Abbeel. Rl^2: Fast reinforcement learning via slow reinforcement learning. *CoRR*, abs/1611.02779, 2016. URL http://arxiv.org/abs/1611.02779.

Carlos Florensa, David Held, Markus Wulfmeier, and Pieter Abbeel. Reverse curriculum generation for reinforcement learning. *CoRR*, abs/1707.05300, 2017. URL http://arxiv.org/abs/1707.05300.

Kevin Frans, Jonathan Ho, Xi Chen, Pieter Abbeel, and John Schulman. Meta learning shared hierarchies. *CoRR*, abs/1710.09767, 2017. URL http://arxiv.org/abs/1710.09767.

Alex Graves, Marc G. Bellemare, Jacob Menick, Rémi Munos, and Koray Kavukcuoglu. Automated curriculum learning for neural networks. *CoRR*, abs/1704.03003, 2017. URL http://arxiv.org/abs/1704.03003.

Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, and Percy Liang. Reinforcement learning on web interfaces using workflow-guided exploration. In *Proceedings of the International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=ryTp3f-0-.

Andrew Y. Ng, Daishi Harada, and Stuart J. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the International Conference on Machine Learning*, ICML '99, pp. 278–287, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. ISBN 1-55860-612-2. URL http://dl.acm.org/citation.cfm?id=645528.657613.

Pararth Shah, Marek Fiser, Aleksandra Faust, J. Chase Kew, and Dilek Hakkani-Tür. Follownet: Robot navigation by following natural language directions with deep reinforcement learning. *CoRR*, abs/1805.06150, 2018. URL http://arxiv.org/abs/1805.06150.

Tianlin Shi, Andrej Karpathy, Linxi Fan, Jonathan Hernandez, and Percy Liang. World of bits: An open-domain platform for web-based agents. In Doina Precup and Yee Whye Teh (eds.), *Proceedings of the International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 3135–3144, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

Oriol Vinyals, Lukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. Grammar as a foreign language. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, Proceedings of the Annual Conference on Neural Information Processing Systems, pp. 2773–2781, Cambridge, MA, USA, 2015. MIT Press. URL http://dl.acm.org/citation.cfm?id=2969442.2969550.

Jane X. Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z. Leibo, Rémi Munos, Charles Blundell, Dharshan Kumaran, and Matthew Botvinick. Learning to reinforcement learn. *CoRR*, abs/1611.05763, 2016. URL http://arxiv.org/abs/1611.05763.

Wojciech Zaremba and Ilya Sutskever. Learning to execute. *CoRR*, abs/1410.4615, 2014. URL http://arxiv.org/abs/1410.4615.