CS 394R Final Project: Autonomous Web Navigation with Reinforcement Learning

Sagnik Majumder, sm72878
sagnik@cs.utexas.edu

Chinmoy Samant, cs59688
chinmoy@cs.utexas.edu

Abstract

*Autonomous web navigation has great practical relevance and the potential to be deployed as a feature in personal assistants in the future. In autonomous web navigation, an agent is trained to navigate an interactive web environment to complete a certain query-based task. The query can be a structured query or a natural language query. An example query can be '**Forward** the email **from*** Bob ***to*** Alice.' *We study such tasks in the light of reinforcement learning. Specifically, we look at the problem from the angle of neurally-controlled constrained exploration around expert demonstrations and reward shaping. Our experiments indicate that such techniques offer a distinct advantage over orthogonal approaches like behavioral cloning by speeding up learning and improving performance. We upload a short video of our work to https://youtu.be/6O6_ejGlgfM*[1] *and create a repository of the code (along with attributions) and the slides at https://github.com/SAGNIKMJR/autoweb*[2].

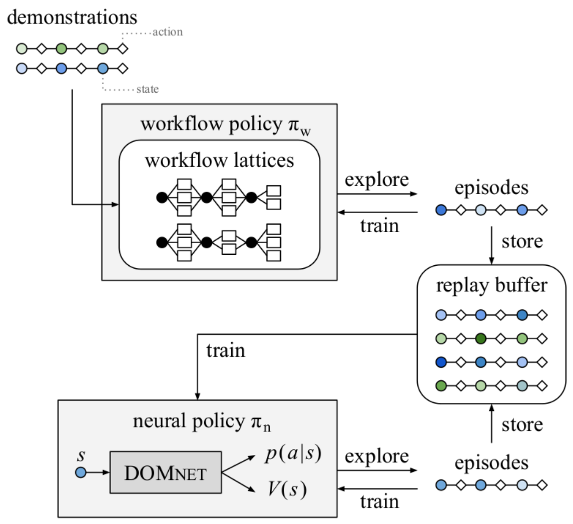
1. Introduction

Our goal is to train an RL agent to complete basic web tasks like forwarding or deleting an email, or entering time on a webpage. We face such tasks very often in our daily lives while using the internet. Although such tasks are very basic and mundane for a human, they can be pretty complex for an RL agent. An HTML webpage has numerous constituent elements that make it very high dimensional for an RL agent to navigate. Moreover, rewards in such a setting are highly sparse and usually the agent receives a reward only after it completes a task perfectly. It might occur to readers that web navigation would be easier and more tractable with hand-engineered agents. Such agents can just parse a webpage and an instruction/query, and carry out some pre-modeled actions to fulfill a user task. However, webpages can be very highly varied in structure and designing a model for every single webpage that exists on the net, can be heavily cumbersome and expensive. So, it makes sense to train an RL agent that learns to find optimal navigation trajectories in a webpage on its own.

An HTML page can be represented in a structured way using the document object module (DOM) tree format. The leaf nodes in the DOM tree are elements of interest in web navigation because an agent can be made to complete a certain task by carrying out certain atomic elements on the leaf nodes of the DOM tree. The query/instruction in a task can either be structured or unstructured. Representing the query as a regular expression (regex) or a dictionary of key-value pairs renders structure to the query. On the other hand, the query can be an unstructured natural language prompt. We expect a performant web agent to be able to deal with both types of queries. However, extracting information to build a regex or a dictionary of key-value pairs can be solved separately from navigation using a natural language parser. Thus, a good RL-based web navigator should be a combination of a natural language parser model and an RL model. The natural language parser mines the relevant keywords from a natural language utterance and builds a structured dictionary out of that. Further, there is an embedding model that embeds the dictionary key-value pairs in some feature space to represent the goal of the task. For structured queries, the embedding model can directly run on the queries without the need for a parser model. The embedding model also embeds the entire DOM tree in a similar feature space. The RL agent uses the joint embedding of the query and the DOM tree to decide actions at every step in its trajectory. From a utility point of view, it is intuitive to give the agent a positive reward on successful completion of the entire task and a zero reward otherwise.

From the above discussion, it is obvious that the problem of autonomous web-navigation is interesting in its own right not just as a practical and suitable use-case for RL algorithms. It can also be used as a benchmark setting for testing RL algorithms which are designed to provide efficiency and robustness in the face of high-dimensional environments and sparse rewards.

[1] Youtube video ink - https://youtu.be/6O6_ejGlgfM
[2] Github repository link - https://github.com/SAGNIKMJR/autoweb



```
Preprocessing:
  for all demonstrations $d$ do
     Induce workflow lattice from $d$
Every iteration:
  Observe an initial environment state
  $\pi_w$ samples a workflow from a lattice
  Roll out an episode $e$ from the workflow
  Use $e$ to update $\pi_w$
  if $e$ gets reward $+1$ then
     Add $e$ to replay buffer
Periodically:
  if replay buffer size > threshold then
     Sample episodes from replay buffer
     Update $\pi_n$ with sampled episodes
  Observe an initial environment state
  $\pi_n$ rolls out episode $e$
  Update $\pi_n$ and critic $V$ with $e$
  if $e$ gets reward $+1$ then
     Add $e$ to replay buffer
```

Figure 1. **Workflow guided exploration (WGE)**. After extracting a workflow lattice from a demonstration, the workflow policy π_w performs exploration by sampling episodes from workflows. Successful episodes are saved to a replay buffer, which are further used to train the neural policy π_n. The pseudocode on the right succinctly illustrates the steps in training the workflow policy and the neurl policy. [4]

2. Related Work

Autonomous web navigation has been studied far less in RL than other tasks. To the best of our knowledge, our work closely follows [4] and [9], both of which leverage human demonstrations for solving autonomous web navigation. [9] also introduce a new platform, mini world of bits (MiniWoB) for benchmarking algorithms for autonomous web navigation. [4] further augments the MiniWoB dataset by including new tasks like stochastic environments and variation in natural language. They name this new dataset mini world of bits ++ (MiniWoB++).

Naive application of RL in MiniWoB-like environments where the state-action space is very high-dimensional and the rewards are sparse, would most likely fail. A common remedy to this problem is to provide the agent with sufficient expert demonstrations and train it through behavioral cloning [8, 14]. However, it is not viable to supply demonstrations with a broad enough support for efficiently covering the entire environment. [9] tries to overcome this issue through model pre-training for 'warm starts' but observes that it doesn't give tangible improvement over RL. To mitigate the problem of compounding errors that is common in direct behavioral cloning, they also do policy fine-tuning and optimize the expected reward using a policy gradient method [11]. They use the asynchronous advantageous actor-critic (A2C) [5] version of policy gradients in particular.

[4] further improves the results of behavioral cloning by constraining exploration around expert demonstrations through the use of workflows [1]. For the task of forwarding an email from one person to another, a workflow step specifies the high-level actions of typing into the "to" field, clicking on the 'send' button etc. However, it doesn't specify which email to click on or what to type. Workflows practically restrict the possible actions for a policy given a state and thus ensure that the agent can converge faster without overfitting. Workflow guided exploration (WGE) is similar to hierarchical RL [12] in that the policies at multiple levels of abstractions are trained. However, it's different in the sense that the higher-level policy (workflow policy) in WGE is state-independent. Also related to WGE is the idea of learning to learn (meta-learning) [13]. Workflows attempt to leverage the meta-knowledge (general structure) of a task to do exploration in a controlled manner. This allows the agent to visit only those states which are actually useful to learn the task.

They train the workflow policy using REINFORCE [10, 15]. The neural policy is trained using off-policy updates by sampling episodes from a replay buffer and the synchronous version of actor-critic (A2C) [5].

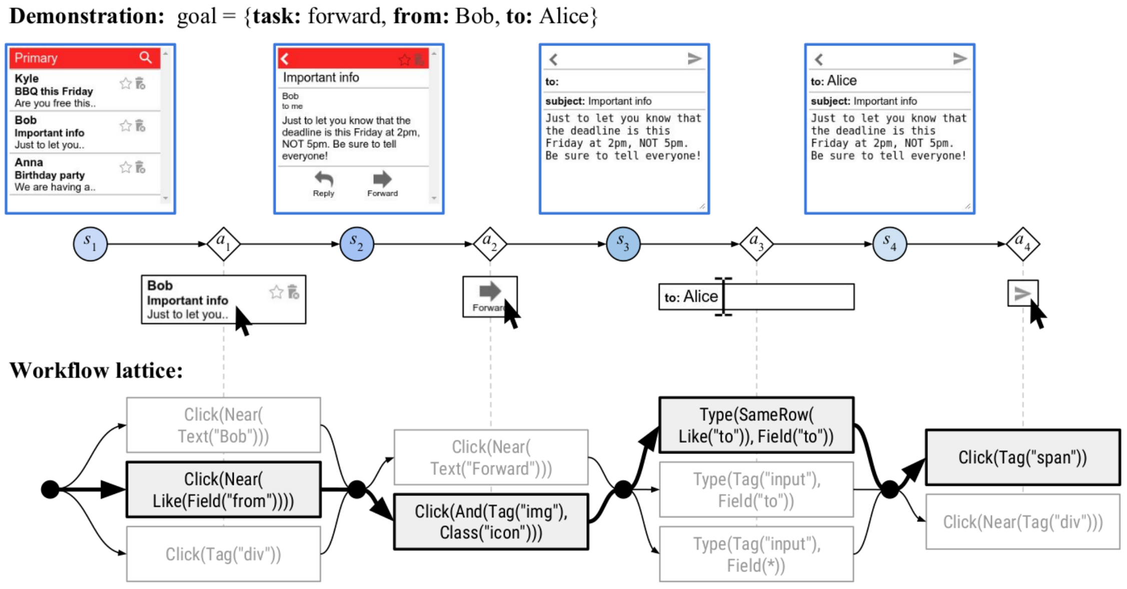

Demonstration: goal = {task: forward, from: Bob, to: Alice}
Primary
Kyle
BBQ this Friday
Are you free this..
Bob
Important info
Just to let you..
Anna
Birthday party
We are having a..
Important info
Bob
to me
Just to let you know that the deadline is this Friday at 2pm, NOT 5pm. Be sure to tell everyone!
Reply
Forward
to:
subject: Important info
Just to let you know that the deadline is this Friday at 2pm, NOT 5pm. Be sure to tell everyone!
to: Alice
s_1
a_1
s_2
a_2
s_3
a_3
s_4
a_4
Forward
to: Alice
Workflow lattice:
Click(Near(Text("Bob")))
Click(Near(Like(Field("from"))))
Click(Tag("div"))
Click(Near(Text("Forward")))
Click(And(Tag("img"), Class("icon")))
Type(SameRow(Like("to")), Field("to"))
Type(Tag("input"), Field("to"))
Type(Tag("input"), Field(*))
Click(Tag("span"))
Click(Near(Tag("div")))

Figure 2. **Workflow lattice**. From each demonstration, the workflow policy builds a workflow lattice that is consistent with the actual actions taken in a demonstration. Next, the workflow policy samples a workflow (the highlighted path in the lattice) and further samples actual actions for a task in an efficient manner. [4]

3. Preliminaries and problem formulation

The goal in a typical RL task is to learn a policy that maximizes a weighted sum of rewards in expectation. The policy, written as $\pi(a_t|s_t)$, takes in a state (s_t) as input and gives a probability distribution of actions (a_t) given that state. Here, t refers to the time step in the episode. The weighted sum of rewards, also called as the return, is given by $\sum_t \gamma^t r_{t+1}$ where r is the reward at $t+1$ and the weighting factor γ is used to decide the relative importance of near and distant rewards. For complex environments like the one in web navigation, it is almost impossible to correctly model the environment and the standard learning paradigm in such cases is model-free RL. A common technique in model-free RL is to sample trajectories of the form $(s_1, a_1, \ldots, s_T, a_T)$ from the environment and train an agent through trial-and-error.

The DOM tree representation of the webpage at a time-step and the structured (dictionary) form of the goal g together constitute the state of the environment. We would like to remind the readers that the goal in a task is the query/instruction itself. The environment transitions into a new state depending on the agent's action. Actions in our environment are only of 2 types: `Click(e)` and `Type(e,t)`. `Click(e)` clicks on a leaf element `e` of the DOM tree and `Type(e,t)` enters text `t` into a leaf element. Such atomic actions are sufficient to complete any task from the MiniWoB(++) benchmark. Figure 2 shows a sample episode for the task of forwarding an email from the MiniWoB(++) dataset. The goal of the task is represented using a dictionary of the form {**task:** forward, **from:** Bob, **to:** Alice}. The agent receives a reward of +1 only if it completes the task perfectly i.e. it picks out the right email, correctly enters the recipient's email and clicks on the forward/send button within the allotted time. It receives a reward of -1 otherwise. Episodes can be very long depending on the allotted time to complete the task and how well the agent is trained. An under-trained agent might visit different leaf elements and perform various actions in the course of task completion. The long length of the episodes leads to sparse rewards.

4. Workflow guided exploration

It is infeasible to provide the agent with demonstrations that exhaustively include all episodes possible in MiniWoB(++) tasks. Thus, direct behavior cloning from expert demonstrations might lead to the agent overfitting heavily. [4] proposes workflow guided exploration (WGE) for doing bounded exploration around demonstrations. This notion of bounded exploration is called workflow [1]. Workflows are more high-level than actual episodes and are environment-blind. The action at every step in a workflow policy doesn't depend on the state of the environment. Rather, it's just a rough layout for actions that can be followed blindly. The high level knowledge or meta-knowledge contained in workflows samples trajectories which provide much richer information for the actual policy to learn from. A sample workflow for email forwarding can look like the highlighted path in figure 2.

WGE can be split into the following simple sub-steps [4]:

- Extraction of a workflow lattice (figure 2) that is consistent with the actual actions taken in a sample episode
- Train a workflow policy using an off-the-shelf RL algorithm to sample workflows and subsequently sample actual actions for a task efficiently
- Add the reward-earning trajectories to a replay buffer that a neural policy (the main policy) can learn from

4.1. Demonstrations to workflow lattice

A workflow can be formally described as a sequence of steps $z_{1:T}$. Each step $z_t(s_t, a_t)$ takes in a state-action pair (s_t, a_t) as input and returns a set of possible actions. The set of possible actions is similar to the actual action a_t taken in the demonstration and conditioned on the state s_t. For example, `Click(Near(Like(Field("from"))))` in figure 2 refers to the generic step of clicking near any tab similar to the 'from' tab.

A workflow lattice (figure 2) refers to a collection of all possible workflows given a certain demonstration. Technically, it is the cross-product $Z_1 \times \ldots \times Z_T$ of the sets of all workflow steps at every time-step of an episode (demonstration). Here, Z_t is the set of workflow steps at t.

Noisy demonstrations that have redundant and repetitive actions induce unnecessary sparsity and noise which hamper learning. To deal with such noise, we add *shortcut actions* for skipping redundant actions (e.g., an expert demonstrator accidentally clicks on the background) and merging repetitive actions into one (e.g., collapsing two consecutive type actions in the same field into a single type action) [4].

4.2. Workflow exploration policy

The workflow exploration policy interacts with the environment as follows. It first samples a demonstration which exactly matches the structured form (dictionary form) of the goal. Thus, it only samples demonstrations which complete a task perfectly. We control the level of perfectness of demonstrations to be sampled using the *minimum reward for demo* hyperparameter which we set to 1 in all our experiments. Fortunately, MiniWoB(++) is clean enough to contain perfect demonstrations for every task that we consider in this work.

Having sampled a perfect demonstration d, it next samples the workflow step z_t as per the following [4]:

$$z_t \sim \pi_w(z|d,t) \propto exp(\psi_{z,t,d}) \tag{1}$$

where $\psi_{z,t,d}$ is a scalar similarity metric that needs to be learned using some RL algorithm. Subsequently, the workflow policy takes an actual action that is sampled from a uniform distribution conditioned on $z_t(s_t)$ [4].

$$a_t \sim p(a|z_t, s_t) = \frac{1}{|z_t(s_t)|} \tag{2}$$

where $|z_t(s_t)|$ is the cardinality of the set of all actions possible given the workflow step at t. We reiterate that the our workflow policy is *environment-blind* because $\pi_w(z|d,t)$ doesn't depend on s_t at all. This drastically reduces the number of parameters in the workflow policy neural net and results in fast learning without overfitting. The overall probability of an episode $p(e = s_1, a_1, \ldots, s_T, a_T|d)$ can be written as

$$p(e|d) = \prod_{t=1}^{T} p(s_t|s_{t-1}, a_{t-1}) \sum_z p(a_t|z, s_t)\pi_w(z|d,t) \tag{3}$$

where $p(s_t|s_{t-1}, a_{t-1})$ gives the state-transition probability.

We train the workflow policy using REINFORCE with a baseline to keep the variance of gradients in check. The equation for a gradient step in the algorithm is

$$\sum_t (G_t - v_{d,t})\nabla_\psi log \sum_z p(a_t|z, s_t)\pi_w(z|d,t) \tag{4}$$

where $v_{d,t}$ is the baseline value. We use a constant baseline of 0.1 in our implementation.

5. Neural policy

We train the neural policy in both on-policy and off-policy fashion by using a synchronous version of advantage actor-critic (A2C). For off-policy training, we maintain a replay buffer and sample old episodes from it. In addition to appending perfect episodes from WGE, we also add perfect trajectories discovered during on-policy exploration by the neural policy, to the replay buffer.

We use the DOMNet architecture, proposed by [4], for our neural policy. DOMNet is designed to leverage both spatial and hierarchical features in a DOM tree. The architecture is pretty complex and has two sub-modules: an embedder, and an attention module that uses the embeddings to produce a state-action mapping $\pi_n(a|s)$ and a

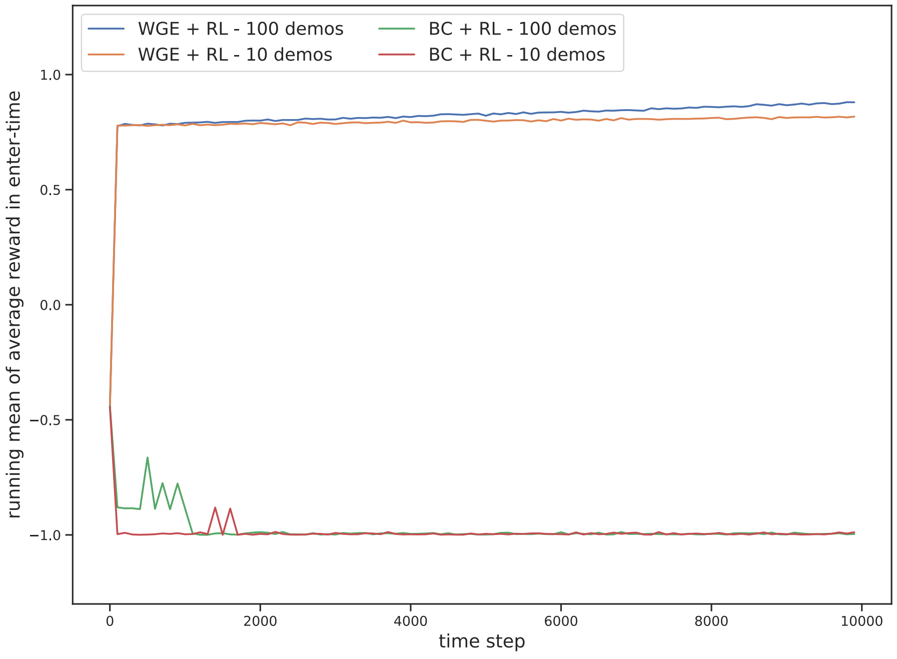


Figure 3. **Performance comparison of WGE with BC on varying the number of demonstrations.** It is evident that BC fails to train even after increasing the number of demonstrations. Besides, it looks like using more demonstrations offers some advantage asymptotically in WGE. The performance metric considered is the running mean of the average reward as a function of time. The average is taken over the rewards from 500 episodes and the running mean is computed over this quantity every 100 training steps.

value function $V(s)$ which acts as the critic in A2C updates.

The DOM embedder captures similarities in DOM elements and facilitate information sharing in their embeddings. For example, the DOM elements which are related/similar (e.g., a checkbox and its label) should have highly correlated embeddings. For unstructured natural language text data in queries or fields, DOMnet directly uses pre-trained GloVe [7] embeddings. For every element e, the embedding for the query/instruction is computed by summing the vectors of all words that appear in both e and the query. For further details about the embedder and the attention sub-module, we ask the readers to refer to [4]. Figure 1 illustrates the pipeline for WGE and the main steps involved in sampling trajectories and training the workflow and neural policies.

6. Reward shaping

Similar to the technique used in [6], as an extension to our baseline approach we augment the reward formulation to reduce sparsity and assist learning. We replace the discrete reward of +1 or 0 at the end of an episode with a time-dependent potential function based reward [2]. The potential function (*Potential(s,g)*) is the count of the number of matching DOM elements between a given state (s) and the goal (g); normalized by the number of DOM elements in the goal. We take the difference of the potential value between a state and the next state in an episode to compute the potential-based reward for that action:

$$R_{potential} = \gamma(Potential(s_{t+1}, g) - Potential(s_t, g)) \tag{5}$$

For example, in the email-forwarding task, there are 4 DOM elements that need to be acted on to complete the goal: i) clicking on the correct *from* field, ii) clicking on the *forward* button, iii) correctly entering the recepient's email address in the *to* field, and iv) clicking on the *send* button. The potential of the state where one of these steps is complete is 0.25 and so on and so forth.

7. Tasks

We broadly focus on the MiniWoB++ dataset in this work because it's open-source. MiniWoB++ contains a suite of 105+ interactive web tasks of varying nature and

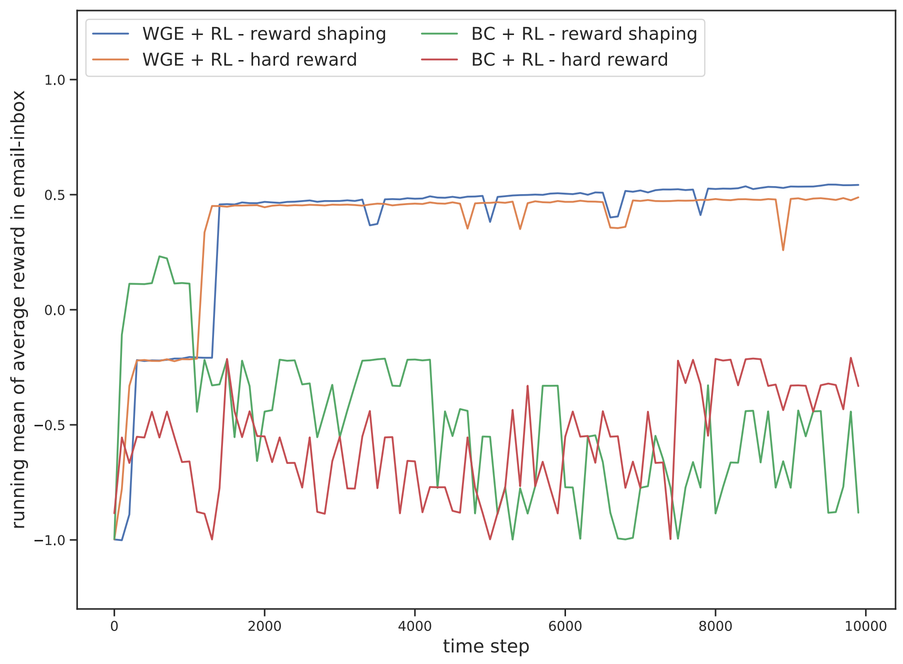


Figure 4. **Effect of reward shaping on the running mean of the average reward with time.** For WGE, reward shaping looks to improve the overall performance of the agent by a little. The trends in BC are not really informative. The average is taken over the rewards from 500 episodes and the running mean is computed over this quantity every 100 training steps.

complexity. The webpage in every certain task is 160 px $\times$ 210 px. The DOM representation of a webpage is the environment that an RL agent tries to navigate and the associated instruction/query defines the goal. Interaction between python and the environment takes place using the Chrome web-driver and the Selenium web-interfacing toolkit.

We specifically look at the tasks: *email-inbox* and *enter-time*. The *email-inbox* task expects the agent to navigate through an email inbox and perform several basic tasks like forwarding an email, deleting or starring an email etc. It can also be categorized on the basis of the type of query: structured or unstructured/natural-language. For the *enter-time* task, the agent has to learn to correctly enter time on a web-page.

8. Experiments and results

Our work is mainly aimed at demonstrating the effectiveness of workflow guided exploration (WGE) over vanilla behavioral cloning (BC) in terms of performance and demonstration-efficiency. We also hypothesize that reward augmentation helps the agent in learning better and faster by reducing sparsity of reward in MiniWoB(++)-type tasks. Further, we wish to show that it is possible for an agent to complete a task given an unstructured natural language query if it's provided with a model to build a structured form of the query. To better illustrate our goals, we conduct a set of experiments with the *email-inbox* and *enter-time* tasks from MiniWoB++.

All our experiments share some common training and inference hyperparameters. We train our workflow and neural policy for a combined total of 10000 iterations with Adam optimizers [3] and initial learning rates of 0.8 and $1e^{-3}$ respectively. In our WGE experiments, we start sampling a batch of 64 episodes without replacement from the replay buffer every step for training the workflow policy and every 10 steps for training the neural policy after an initial 100 steps of random exploration. We use the sampled trajectories to do gradient updates to our workflow and neural policy for 30 gradient steps. Our exploration schedule is highly skewed towards workflow-driven exploration: the workflow policy is used to roll out trajectories every training step while the neural policy

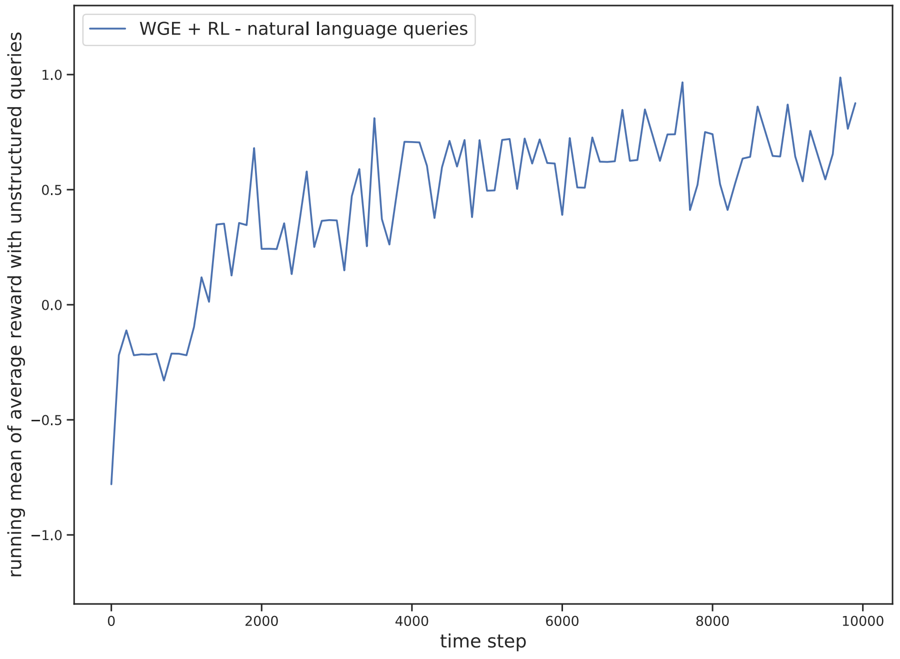


Figure 5. **Running mean of the average reward for the *email-inbox* task with natural language queries.** The agent is able to extract structured information from unstructured text using the natural language parser and use that to complete the task perfectly upon sufficient training. The average is taken over the rewards from 500 episodes and the running mean is computed over this quantity every 100 training steps.

generates trajectories every 10^{th} step. While our neural policy is a highly sophisticated DOMNet, our workflow policy is a single-layered network with softmax outputs. For exact architectural hyperparameters of DOMNet, we request readers to refer to [4]. We initialize V of critic to 0.0 and set α to 0.1. On the other hand, for BC there is no replay step and the neural policy tries to directly learn from demonstrations. Our choice of hyperparameters roughly follows [4]; we set the values of a few intuitively. We couldn't complete more than a single run of our experiments due to time and computational constraints. Due to a bug in the selenium web-driver interface, we were unable to run the experiments headless on a GPU and had to resort to running them on our own laptops.

Our metric of evaluation is the running average of reward at test time. We use the neural policy every 100 training steps to roll out 500 episodes and measure the mean of the reward from those episodes.

8.1. Enter-time

We use the *enter-time* task to mainly analyse how the performance and training speed of WGE and BC change with number of available demonstrations. We set the number of demonstrations to 10 and 100, and report the behavior of mean reward and success-rate for BC and WGE in both cases. Figure 3 shows that WGE gives a far superior performance over BC irrespective of the number of demonstrations used. The running mean of the average reward for WGE experiments crosses 0.9 while the same for BC experiments deteriorates gradually and reaches -1. Moreover, using more demonstrations seems to help WGE a little as seen in the difference in asymptotic performance between WGE with 10 demonstrations and WGE with 100 demonstrations.

8.2. Email-inbox

Through our experiments with *email-inbox*, we compare how reward augmentation (time-dependent reward) fares against hard rewards received at the end of an episode. We provide the agent with 10 demonstrations in these expeiments. It can be observed that reward shaping offers a small benefit over hard rewards for WGE. Besides, the figure also reveals similar advantages of WGE over BC as seen in figure 3 in terms of performance. However, the running mean reaches near 0.5 for *email-inbox* as opposed to 1.0 for *enter-time*. This might be an indicator of the the relative complexity of the two tasks: even for a human, *enter-time* is a much simpler task in comparison to *email-inbox*. We also show the results from training an agent for *email-inbox* using natural language queries with 16 demonstrations in figure 5. Note that the running mean of the average reward starts from a negative value but increases in a near-monotonic fashion to reach near +1 in contrast to near 0.5 for structured queries. This is a bit counter-intuitive because using natural language queries should have made it harder for the agent to learn.

9. Discussion and outlook

Workflow guided exploration is a judicious way to do exploration around demonstrations in a controlled manner. It cleverly prevents the agent from visiting too many states that show very little promise of resulting in a positive reward. Besides, as elucidated by our experiments WGE far outperforms vanilla behavioral cloning which fails clearly due to insufficiently available demonstrations, across multiple tasks. Further, we notice that reward shaping shows some promise vis-a-vis performance improvement for WGE. Despite investigating multiple facets of our approach, we fail to study the following aspects of our work and leave them out as part of future work.

It might be interesting to see how WGE responds to replacing demonstration-specific workflow lattices with task-specific lattices. Having one lattice per task in place of a lattice per demonstration might allow the workflow policy to learn even richer information about the task and the environment.

It also makes sense to try curriculum learning for 'warm start'-ing the agent [2]. Curriculum learning can initially place the agent very close to the goal and gradually loosen this constraint so that the agent can learn more efficiently.

Finally, the effects of WGE and reward shaping should be tested more thoroughly by benchmarking the algorithms on other tasks. Moreover, due to stochasticity in sampling and batching of trajectories, and initialization of neural network parameters, it would be a good idea to run all experiments multiple times and report a measure of variance in performance in addition to its mean.

10. Acknowledgements

We thank our course instructors, Prof. Peter Stone and Prof. Scott Niekum, and teaching assistants for their instruction and guidance throughout the course.

References

[1] B. Deka, Z. Huang, and R. Kumar. Erica: Interaction mining mobile apps. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, UIST '16, pages 767–776, New York, NY, USA, 2016. ACM.

[2] I. Gur, U. Rückert, A. Faust, and D. Hakkani-Tür. Learning to navigate the web. *CoRR*, abs/1812.09195, 2018.

[3] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[4] E. Z. Liu, K. Guu, P. Pasupat, T. Shi, and P. Liang. Reinforcement learning on web interfaces using workflow-guided exploration. *CoRR*, abs/1802.08802, 2018.

[5] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In M. F. Balcan and K. Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937, New York, New York, USA, 20–22 Jun 2016. PMLR.

[6] A. Y. Ng, D. Harada, and S. J. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning*, ICML '99, pages 278–287, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[7] J. Pennington, R. Socher, and C. Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, Oct. 2014. Association for Computational Linguistics.

[8] D. A. Pomerleau. Efficient training of artificial neural networks for autonomous navigation. *Neural Comput.*, 3(1):88–97, Mar. 1991.

[9] T. Shi, A. Karpathy, L. Fan, J. Hernandez, and P. Liang. World of bits: An open-domain platform for web-based agents. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 3135–3144, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[10] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[11] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In S. A. Solla, T. K. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems 12*, pages 1057–1063. MIT Press, 2000.

[12] R. S. Sutton, D. Precup, and S. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in rein-

forcement learning. *Artif. Intell.*, 112(1-2):181–211, Aug. 1999.

[13] S. Thrun and L. Pratt, editors. *Learning to Learn*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[14] M. Vecerík, T. Hester, J. Scholz, F. Wang, O. Pietquin, B. Piot, N. Heess, T. Rothörl, T. Lampe, and M. A. Riedmiller. Leveraging demonstrations for deep reinforcement learning on robotics problems with sparse rewards. *CoRR*, abs/1707.08817, 2017.

[15] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3-4):229–256, May 1992.