ECE 471 Fall 2022

Mini Project 3: Reinforcement Learning for Performance-aware Serverless Resource Management

Broad Problem Statement

Reinforcement learning (RL) has become an active area in machine learning research in the past decades and has recently been combined with deep learning techniques to be used in applications such as video games (e.g., Atari) [1], Computer Go [2], cooling data centers [3], and so on. RL deals with agents that learn to make better decisions directly from experience interacting with the environment. Inspired by those advances in applying deep RL in AI tasks, a series of works [4-10] focuses on building RL frameworks that learn to manage or control systems directly from experience. Resource management problem is one such example which in systems and networking often manifests as difficult online decision-making tasks. Appropriate solutions depend on the understanding of the workload and system environment, and traditional heuristics-based solutions typically require recurring human expert effort to test heuristics or tune parameters. RL approaches are especially well-suited to resource management systems because (1) decisions made by these systems are often highly repetitive, thus generating an abundance of training data for RL algorithms; (2) RL can model complex systems and decision-making policies as deep neural networks analogous to the models used for game-playing agents.

In this project, you will learn how RL algorithms are designed and implemented, and how the resource management problem can be tackled by a learned RL agent. Specifically, you'll be exploring various new reinforcement learning algorithms in this MP and are free to use Python packages unless otherwise specified (see "Useful Python Libraries" for reference). If you do use any additional libraries, please identify them in the README.md file or provide a requirements.txt file in the repository that you submit so that we can still run your code.

Concepts you will learn and apply:

- Reinforcement learning (RL)
- RL environment (for policy training and policy serving)
- Policy-gradient, Value-based, Actor-critic RL training algorithms
- Deep RL neural network structure design and implementation
- RL reward function design
- RL training progression visualization
- RL model checkpointing

Background - Serverless Computing

Serverless Function-as-a-Service (FaaS) is an emerging cloud computing paradigm that frees customers (i.e., application developers) from infrastructure management tasks such as resource provisioning and scaling. In traditional cloud computing paradigms such as Infrastructure-as-a-Service (IaaS), customers deploy their applications in virtual machines and are charged by the virtual machine lifetime. In serverless FaaS, customers provide their applications (in terms of ***functions***[1]) and are charged only based on function execution time, hence serverless FaaS is becoming a popular cloud computing paradigm for the deployment of bursty services (e.g., social media and machine learning model serving) as cloud functions. The functions are deployed as containers. There are both commercial serverless computing providers (e.g., AWS Lambda, Azure Functions, Google Cloud Functions, and IBM Cloud Functions) and open-source serverless platforms such as OpenWhisk, OpenFaaS, and KNative.

All serverless platforms obey similar architecture and workflow. A serverless platform runs functions in response to invocations (i.e., requests) from end-users or clients. It consists of a central controller and a group of invokers (workers). For example, Fig. 1 shows the architecture of OpenWhisk [11], a production-grade serverless FaaS platform based on Docker containers. The controller creates function containers, allocates CPU and memory for each function container, and assigns the containers to invokers. When requests arrive through the API gateway, the controller distributes the requests to invokers. An invoker executes the function after it receives a request and the execution results are written to a data store.

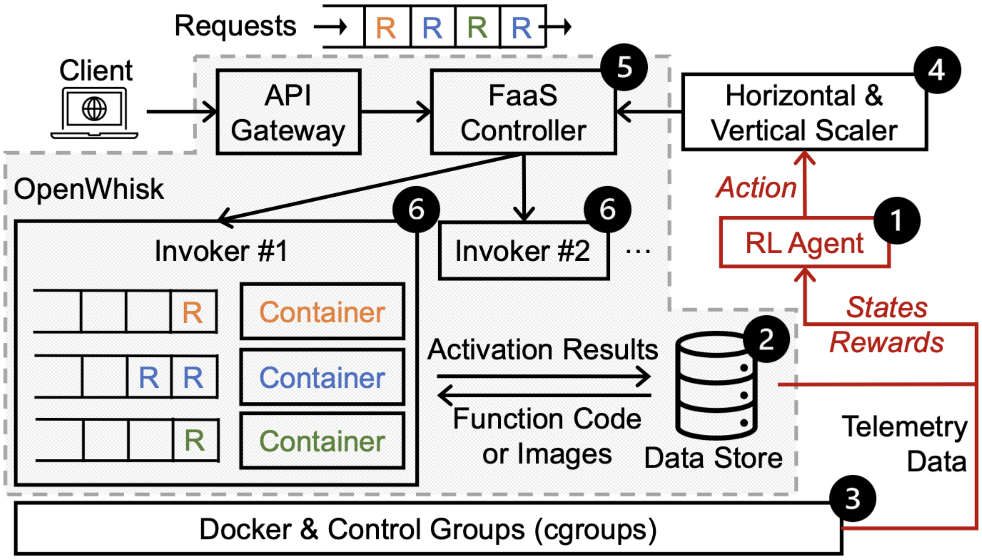

Fig. 1: Overview of the architecture of OpenWhisk [11].

[1] A function is a program or a script that runs in a serverless FaaS platform such as AWS Lambda.

Serverless FaaS workloads, like most cloud data-center services, have service-level objectives (SLOs) defined by each tenant that codify the expected performance. The most common type is a latency SLO, which specifies the acceptable latencies for function requests in the serverless computing context. For example, a latency SLO might specify that 99% of requests have latencies smaller than 100 ms. If a service fails to meet its SLOs, the service provider may risk severe penalties or financial loss. In this project, we focus on resource management to meet per-function SLOs while keeping the utilization[2] at a high level, since low utilization efficiency is undesirable for the cloud provider.

Background - Serverless Resource Management

The resource management problem in serverless computing is basically divided into two dimensions: horizontal scaling and vertical (container) resizing. Horizontal scaling means that the number of containers for a function is dynamically adjusted to adapt to workload variations. Vertical resizing means that the size of the function containers is dynamically updated to fulfill performance and utilization requirements. There have been several heuristics-based autoscaling approaches proposed for serverless functions. For example, horizontal scaling in Kubernetes is a threshold-based method. When the CPU utilization is greater than 60%, a new container will be created. Such methods are static and it is hard to come up with an optimal utilization threshold that works for all workload cases.

Lately, RL-based resource management approaches have gained significant momentum toward achieving application SLOs. RL is well-suited for learning resource management policies, as it provides a tight feedback loop for exploring the action space and generating optimal policies without relying on inaccurate assumptions (i.e., heuristics or rules). Since resource management decisions made for each function are highly repetitive (i.e., a decision is needed to be made every few seconds for bursty workloads), an abundance of data is generated for training such RL algorithms. RL allows direct learning from the actual workload and operating conditions to understand how the allocation of resources affects application performance. It has been shown that RL with neural networks can express complex system-application environment dynamics and decision-making policies. For instance, FIRM [4] is an RL-based resource management framework that first identifies the microservices that cause SLO violations and then mitigates those violations via dynamic resource reprovisioning. The single-agent RL example in SIMPPO [6] shows the potential to manage both vertical and horizontal scaling of a function using RL.

In this project, we will study the application of RL in the case of serverless function resource management (including both horizontal and vertical autoscaling) to maintain application SLOs and resource utilization efficiency. In particular, you will (1) learn how serverless resource management is formulated as an RL problem; (2) explore the provided RL environment to get familiar with the problem formulation (i.e., Task 1); (3) design and implement three RL algorithms mentioned in our lecture (i.e., Task 2); (4) design and evaluate different reward

[2] Utilization is defined as the ratio between resource usage and resource limit. For example, if a container's memory usage is 128 MB while its resource allocation of memory capacity is 256 MB, then the memory utilization is 50%.

functions (i.e., Task 3); and (5) learn to virtualize and checkpoint the RL model training process (i.e., Task 4). Now, let's begin!

RL Formulation and the Environment for Policy Training and Serving (Inference)

In the task of serverless resource management, a number of containers are created for a function to serve a burst of function requests. The container size is determined by the CPU (in terms of cpu.shares) and memory capacity (MB) allocation. The function requests typically arrive in a stream and to adapt to the increase and decrease of arrival requests, the resource manager will add or remove function containers (i.e., "horizontal scaling"). The resource manager can also resize the function containers by adjusting the CPU and memory limit (i.e., "vertical scaling"), mostly for keeping the resource utilization high. The goal of resource management in serverless computing is to (1) maintain performance (at SLO latency); and (2) keep resource utilization at a high level.

We can then model serverless resource management as a sequential decision-making problem that can be solved by the RL framework (illustrated in Fig. 1). At each step in the sequence, the RL agent monitors system and application conditions from both the OpenWhisk data store and the Linux cgroups. Measurements include function-level performance statistics (i.e., 99%-percentile latencies on function execution time) and system-level resource utilization statistics (e.g., CPU utilization of function containers). These measured telemetry data are pre-processed and used to define a state, which is then mapped to a resource management decision by the RL agent which includes both horizontal and vertical scaling.

Environment. In this project, we provide the RL environment for training and evaluating the agent with the RL algorithms that you will design and implement. The agent will then learn what to do by interacting with the environment (training) and use the learned policy to do inference. The environment is essentially an API to the OpenWhisk serverless platform specifically designed to train and test any RL algorithm. The main purpose is to control the vertical and horizontal scaling of a serverless function. There are two main functions supported by the environment object:

- reset(): to reset the environment to an initial state
    - Input: N/A
    - Output: state
- step(action): to execute an action and get the new transition
    - Input: action
    - Output: (state, reward)

State: A state vector is a list consisting of variables that are related to the function itself. All variables are of range [0, 1] to facilitate RL training.

state = [cpu_util, slo_preservation, cpu_shares, cpu_shares_others, num_containers, arrival_rate_change]

- cpu_util is a floating-point number (type “double” in Python) representing the average CPU utilization in range [0, 1]
- slo_preservation is a floating-point number (type “double” in Python) calculated by min(1, slo_latency / actual_latency) in range (0, 1]
- cpu_shares is an integer denoting the total CPU shares of all function containers, assuming each container has the same CPU shares (normalized)
- cpu_shares_others is an integer denoting the total CPU shares of all the other containers on the same node[3] (normalized)
- num_containers is an integer denoting the total number of function containers (normalized)
- arrival_rate is an integer indicating the number of requests coming per second

Action: An action variable is a dictionary consisting of two types of actions (vertical and horizontal scaling):

```
{
    'vertical': cpu_shares_to_change,
    'horizontal': num_containers_to_change
}
```

The step size for CPU shares to change is 128 (the cpu.shares is added or reduced by 128 at each time in an action) and the step size for the number of containers to change is 1 (the number of containers is increased or decreased by 1 at each time in an action).

Reward: A reward variable is a floating point number (type “double” in Python) denoting the RL reward of the state and action pair. The reward is an incentive mechanism that tells the agent what is correct and what is wrong. The goal of agents in RL is to maximize the total expected rewards per episode. [Note that you will design your own reward function in Task 3.]

Useful Python Libraries

You may find the following Python libraries for implementing RL algorithms (including neural networks), visualizing the RL training processes, and checkpointing. You may also use other basic Python libraries but please do not directly import existing libraries that implement any RL algorithms. We know that there are a lot of open-sourced RL algorithms online. : -) If you are not sure about any additional libraries that you want to use, please ask one of our TAs to confirm.

- matplotlib
- numpy
- pandas

[3] Since all function containers on a node share the CPU cores and cpu.shares is a weight that decides how much CPU time percentage can a container use. If the cpu.shares for a container is 512 and the total cpu.shares of the other containers is 1536 (on a 2-core node), the actual CPU time percentage for the function container is 2 * 512 / (512+1536) = 0.5 CPU.

- torch
- os

Task 1: RL Environment Exploration (20pts)

In this task, you will initialize an RL environment and get familiar with basic RL environment functionalities for training and inference.

An RL environment is the agent’s world in which it lives and interacts. The agent can interact with the environment by performing some action. When an agent performs an action in the environment, the environment returns a new state of the environment making the agent move to this new state. The environment also sends a reward to the agent which is a scalar value that acts as feedback for the agent whether its action was good or bad.

To test and compare results of different RL algorithms, we need testbed environments. OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms. The gym library is a collection of environments that makes no assumptions about the structure of your agent. Gym comes with a diverse suite of environments, ranging from classic video games such as Atari 2600 and continuous control tasks. Gym environment is quite simple to initialize and use. See below as an example.

```
import gym
env = gym.make("LunarLander-v2", render_mode="human")
observation, info = env.reset(seed=42)
for _ in range(1000):
  action = policy(observation)  # User-defined policy function
  observation, reward, terminated, truncated, info = env.step(action)
```

To use any environment, you need to check its documentation. For example, you may find the documentation of the Atari environment here. In this project, we will provide the RL environment similar to the gym environment. You can check the serverless_env.py code or the documentation we give, and follow similar code to initialize and use the environment.

Task 1.1: Basic RL Environment Functions (10pts)

- Create and initialize an RL environment
- Reset the environment and print the initial state
- Perform one RL step (i.e., calling the step function step()) by providing a vertical scaling-up action (i.e., adding 128 cpu.shares)
- Print the received current state (ignore the reward for now)
- Perform one RL step by providing a horizontal scaling-out action (i.e., adding one container)
- Print the received current state (ignore the reward for now)

Task 1.2: Random Exploration (10pts)

- Create your own policy (could be a random action generator) and perform 10 RL steps, which is called one trajectory (of length 10) in RL
- Print the utilization and the SLO preservation along the trajectory you get

Task 2: RL Algorithm Implementation (30pts)

In this task, you will implement three RL algorithms from each category: value-based, policy-based, and actor-critic.

Task 2.1: Value-based (Optional)

In a value-based approach, a random parameterized value function is created initially (usually represented as a neural network). Then the value function (neural network) parameters are updated based on the rewards collected by interacting with the environment. This process is repeated until it finds the optimal value function. The intuition here is the policy that follows the optimal value function will be the optimal policy. Here, the policy is *implicitly* updated through the value function. Q-learning [12] is one example of value-based RL algorithms.

Implement the Q-learning algorithm using the provided RL environment for testing.

Reference Tutorials:

- https://icaps18.icaps-conference.org/fileadmin/alg/conferences/icaps18/summerschool/lectures/Lecture5-rl-intro.pdf
- https://lilianweng.github.io/posts/2018-02-19-rl-overview/

Task 2.2: Policy-based (Optional)

In a policy-based RL algorithm, a random parametrized policy (usually represented as a neural network as well) is created and initialized with random model parameters. It consists of a step to find the value function of that policy (i.e., the evaluation step). Then it finds the new policy from the value function computed in the improvement step. The process repeats until it finds the optimal policy. In policy-based approaches, the policy is updated directly. REINFORCE [13] is one example of a value-based RL algorithm.

Implement the REINFORCE algorithm using the provided RL environment for testing.

Reference Tutorials:

- http://www.cs.toronto.edu/~tingwuwang/REINFORCE.pdf
- https://lilianweng.github.io/posts/2018-04-08-policy-gradient/

Task 2.3: Actor-Critic (30pts)

In the actor-critic-based RL algorithm, there are two main components: the policy model and the value function. Actor-critic is like a combination of policy-based and value-based approaches. It makes a lot of sense to learn the value function in addition to the policy, since knowing the value function can assist the policy update, such as by reducing gradient variance in policy-based approaches, and that is exactly what the actor-critic method does. Actor-critic methods consist of two models:

- Critic updates the value function neural network parameters.
- Actor updates the policy parameters for the policy neural network, in the direction suggested by the critic.

Proximal Policy Optimization (PPO) [14] is one example of the actor-critic method. PPO is the default RL algorithm at OpenAI which performs comparably or better than state-of-the-art approaches while being much simpler to tune.

Implement the REINFORCE algorithm using the provided RL environment for testing. You don't need to implement it from scratch. Please refer to the ppo.py file for the skeleton code.

The provided code base consists of three main classes:

- ActorNetwork
- CriticNetwork
- PPO

The ActorNetwork performs the task of learning what action to take under a particular observed state of the environment. In our case, it takes the measurements from the serverless platform as input and gives a particular action like scaling up or scaling out as output. You will need to implement a neural network of three fully-connected layers (connected with ReLU) and a softmax layer directing to the output.

The CriticNetwork learns to evaluate if the action taken by the Actor leads our environment to a better state or not, and gives its feedback to the Actor. It outputs a real number indicating a rating (Q-value) of the action taken in the previous state. By comparing this rating obtained from the Critic, the Actor can compare its current policy with a new policy and decide how it wants to improve itself to take better actions. The structure of the Critic neural net is almost the same as the Actor. The only major difference being, the final layer of Critic outputs a real number (so no softmax layer is needed).

The PPO class initializes the Actor and Critic networks, initializes an optimizer, updates the network parameters through a for-loop training by interacting with the environment, and collects the rewards per episode for backward propagation. What you need to do here is to complete the code which does backward propagation through the optimizer given the total loss.

Reference Tutorials:

- https://courses.grainger.illinois.edu/ece448/sp2022/slides/lec36.pdf
- https://lilianweng.github.io/posts/2018-04-08-policy-gradient/

Task 3 & 4 (50pts)

To be released later

Deadlines

Check Point 1:

- Task 1
- Task 2

Due 11/16

Submission Requirements

Pack your entire working directory into a zip file and submit it to canvas. We should be able to execute main.py in the directory without any modification. You should also write a README.md file describing how each task is completed. Remember to document your code well either by comments or in the README.md file. If you use any extra non-standard library other than those given to you in requirements.txt, make sure to mention them in README.md and include the library in requirements.txt.

Academic Integrity

When completing this project, please ensure that your submissions reflect your own work and ideas. Academic integrity is taken seriously, and any issues will be dealt with strictly to ensure a fair course for all students. All submissions will be tested at the end of the MP to check for the plagiarism of both answers and code.

References

[1] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing Atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.
[2] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershevlvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. Nature, 2016.
[3] J. Gao and R. Evans. DeepMind AI reduces google data center cooling bill by 40%. https://deepmind.com/blog/deepmind-ai-reducesgoogle-data-centre-cooling-bill-40/.
[4] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, Ravishankar K. Iyer. FIRM: An Intelligent Fine-Grained Resource Management Framework for SLO-Oriented Microservices. In Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020).
[5] Haoran Qiu, Weichao Mao, Archit Patke, Chen Wang, Hubertus Franke, Zbigniew T. Kalbarczyk, Tamer Başar, Ravishankar K. Iyer. SIMPPO: A Scalable and Incremental Online Learning Framework for Serverless Resource Management. To Appear in Proceedings of the 13th ACM Symposium on Cloud Computing (SoCC 2022).
[6] Inductive-bias-driven Reinforcement Learning for Efficient Schedules in Heterogeneous Clusters. Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. ICML 2020.

[7] Mao, H., Alizadeh, M., Menache, I., & Kandula, S. (2016, November). Resource management with deep reinforcement learning. In Proceedings of the 15th ACM workshop on hot topics in networks (pp. 50-56).
[8] Mao, H., Negi, P., Narayan, A., Wang, H., Yang, J., Wang, H., ... & Alizadeh, D. (2019). Park: An open platform for learning-augmented computer systems. Advances in Neural Information Processing Systems, 32.
[9] Mao, H., Schwarzkopf, M., Venkatakrishnan, S. B., Meng, Z., & Alizadeh, M. (2019). Learning scheduling algorithms for data processing clusters. In Proceedings of the ACM special interest group on data communication (pp. 270-288).
[10] Jay, N., Rotman, N., Godfrey, B., Schapira, M., & Tamar, A. (2019, May). A deep reinforcement learning perspective on internet congestion control. In International Conference on Machine Learning (pp. 3050-3059). PMLR.
[11] Apache OpenWhisk. https://github.com/apache/openwhisk.
[12] Christopher JCH Watkins and Peter Dayan. Q-learning. Machine learning, 8(3-4):279–292, 1992.
[13] Sutton, R. S., McAllester, D., Singh, S., & Mansour, Y. (1999). Policy gradient methods for reinforcement learning with function approximation. Advances in Neural Information Processing Systems (NeuIPS 1999).
[14] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 (2017).