10-703 Deep Reinforcement Learning and Control
Assignment 3
Spring 2018

Jagjeet Singh (jagjeets), Satyen Rajpal (satyenr)

March 26, 2018

Due April 9, 2018, 11:59pm (EST)

Instructions

You may work in teams of **2** on this assignment. Only one person should submit the writeup and code on Gradescope. Additionally, the same person who submitted the writeup and code to Gradescope must upload the code to Autolab. Make sure you mark your partner as a collaborator on Gradescope (you do not need to do this in Autolab) and that both names are listed in the writeup. Writeups should be typeset in LaTeXand submitted as PDF. All code, including auxiliary scripts used for testing, should be submitted with a README.

Please limit your writeup to 8 pages or less (excluding the provided instructions).

We've provided some code templates (using Keras) that you can use if you wish. Abiding to the function signatures defined in these templates is not mandatory; you can write your code from scratch if you wish. You can use any deep learning package of your choice (e.g., Keras, Tensorflow, PyTorch). However, if you choose not to use Keras, then you'll need to figure out how to load the policy network architecture (`LunarLander-v2-config.json`) and the expert's weights (`LunarLander-v2-weights.h5`).

You should not need the cluster or a GPU for this assignment. The models are small enough that you can train on a laptop CPU.

It is expected that all of the work you submit is your own. Submitting a classmate's code or code which copied from online and claiming it is your own is not allowed. Anyone who does this will be violating University policy, and risks failure of the assignment, course and possibly disciplinary action taken by the university.

Introduction

In this assignment, you will implement different RL algorithms and evaluate them on the LunarLander-v2 environment (Figure 1). This environment is considered solved if the agent can achieve an average score of at least 200.

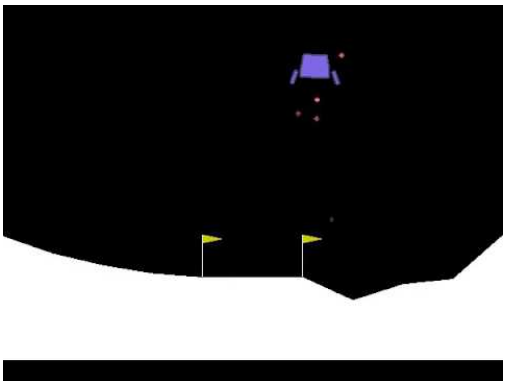


Figure 1: LunarLander-v2: https://gym.openai.com/envs/LunarLander-v2/

Installation instructions (Linux)

We've provided Python packages that you may need in `requirements.txt`. To install these packages using pip and virtualenv, run the following commands:

```
apt-get install swig
virtualenv env
source env/bin/activate
pip install -U -r requirements.txt
```

If your installation is successful, then you should be able to run the provided template code:

```
python imitation.py
python reinforce.py
python a2c.py
```

Note: You will need to install `swig` and `box2d` in order to install `gym[box2d]`, which contains the `LunarLander-v2` environment. You can install `box2d` by running

```
pip install git+https://github.com/pybox2d/pybox2d
```

If you simply do `pip install box2d`, you may get an error because the pip package for `box2d` depends on an older version of `swig`.[1] For additional installation instructions, see https://github.com/openai/gym.

[1] https://github.com/openai/gym/issues/100

Problem 1: Imitation Learning (30 pts)

In this section, you will implement behavior cloning using supervised imitation learning from an expert policy, and test on the LunarLander-v2 environment. Please write your code in `imitation.py`; the template code provided inside is there to give you an idea on how you can structure your code, but is not mandatory to use.

We have provided you with an expert, which you will use to generate training data that you can use with the Keras `fit` method. The expert model's network architecture is given in `LunarLander-v2-config.json` and the expert weights in `LunarLander-v2-weights.h5`. You can load the expert model using the following code snippet:

```
import keras

model_config_path = 'LunarLander-v2-config.json'
with open(model_config_path, 'r') as f:
    expert = keras.models.model_from_json(f.read())

model_weights_path = 'LunarLander-v2-weights.h5f'
expert.load_weights(model_weights_path)
```

Tasks and questions:

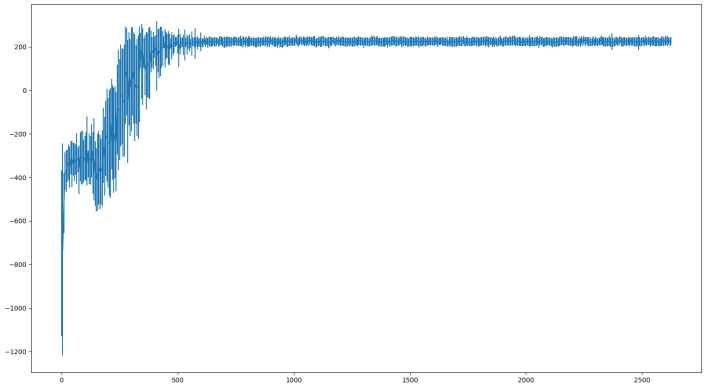
1. Use the provided expert model to generate training datasets consisting of 1, 10, 50, and 100 expert episodes. You will need to collect states and a one-hot encoding of the expert's selected actions.

2. Use each of the datasets to train a cloned behavior using supervised learning. For the cloned model, use the same network architecture provided in `LunarLander-v2-config.json`. When cloning the behavior, we recommend using the Keras `fit` method. You should compile the model with cross-entropy as the loss function, Adam as the optimizer, and include 'accuracy' as a metric.

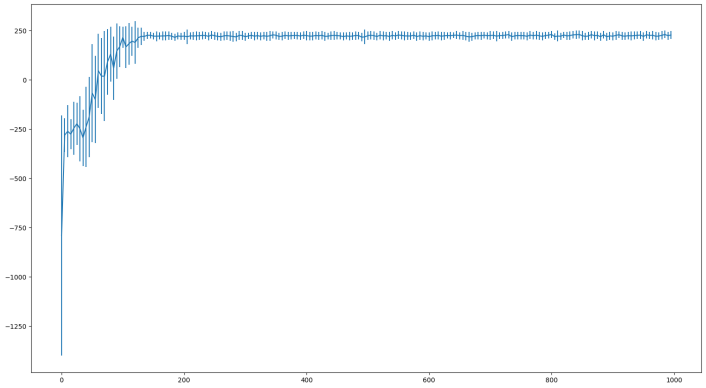
   For each cloned model, record the final training accuracy after training for at least 50 epochs. Training accuracy is the fraction of training datapoints where the cloned policy successfully replicates the expert policy. Make sure you include any hyperparameters in your report.

3. Run the expert policy and each of your cloned models on `LunarLander-v2`, and record the mean/std of the total reward over 50 episodes. How does the amount of training data affect the cloned policy? How do the cloned policies' performance compare with that of the expert policy?

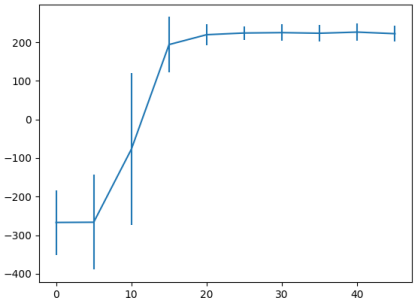
Solution



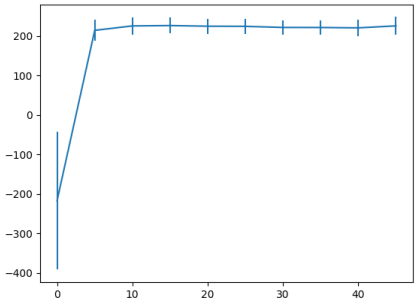
: N=1



: N=10



: N=50



: N=100

Figure 2: Mean Rewards and Standard deviation for different values of N

Solution

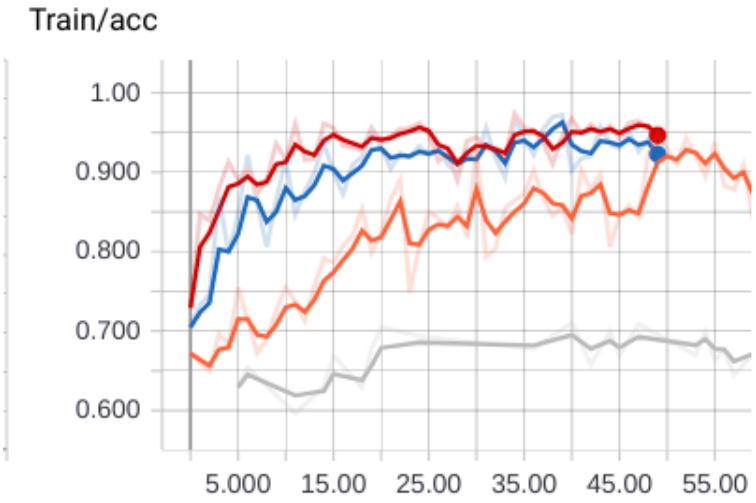


Figure 3: Accuracy for different cloned models, N=1 (Gray), N=10 (Orange), N=50 (Blue), N=100 (Red)

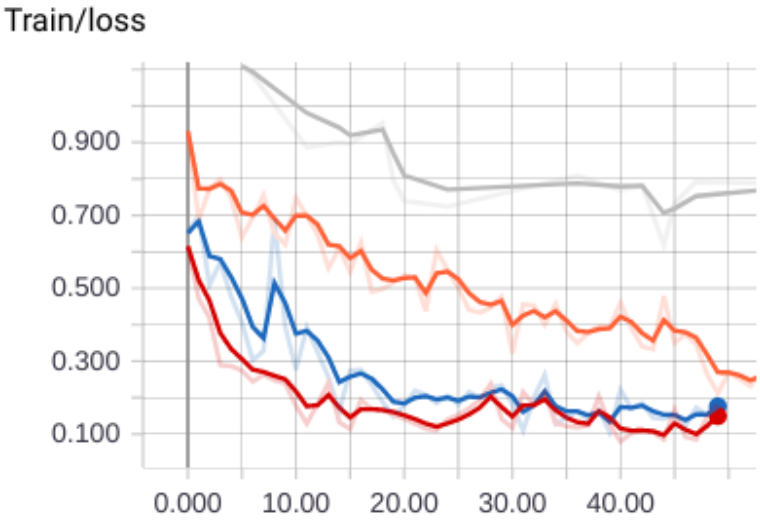


Figure 4: Training Loss for different cloned models, N=1 (Gray), N=10 (Orange), N=50 (Blue), N=100 (Red)

Solution

Setting	Epochs	Mean Reward	Std	Final Training Accuracy (%)
N=1	2625	226.56	18.68	96.00
N=10	1000	228.177	20.667	96.83
N=50	50	221.171	21.018	90.5
N=100	50	225.387	22.545	93.3
Expert	-	226.75	17.86	93.3

Note: The training accuracies mentioned above are as of last epoch. Best accuracy for each cloned model was above 98%

Inferences:

- Number of expert episodes severely affected the performance. For N=100, the model was able to consistently achieve > 200 reward within 10 epochs. For N=1, it took more than 600 epochs to do the same. For N=50, it took 25 epochs and for N=10, it took 200 epochs.
- The final accuracy for all the cloned models (provided they are trained for enough epochs) was in the same range
- The final mean reward and standard deviation for all the cloned models was around 225 and it was almost the same as expert episode.

Problem 2: REINFORCE (30 pts)

In this section, you will implement episodic REINFORCE, a policy-gradient learning algorithm. Please write your code in `reinforce.py`; the template code provided inside is there to give you an idea on how you can structure your code, but is not mandatory to use.

Policy gradient methods directly optimize the policy $\pi(A \mid S, \theta)$, which is parameterized by θ. The REINFORCE algorithm proceeds as follows. We keep running episodes. After each episode ends, for each time step t during that episode, we alter the parameter θ with the REINFORCE update. This update is proportional to the product of the return G_t experienced from time step t until the end of the episode and the gradient of $\ln \pi(A_t \mid S_t, \theta)$. See Fig. 1 for details.

Algorithm 1 REINFORCE

procedure REINFORCE
    Start with policy model π_θ
    repeat:
        Generate an episode $S_0, A_0, r_0, \ldots, S_{T-1}, A_{T-1}, r_{T-1}$ *following* $\pi_\theta(\cdot)$
        for t *from* $T-1$ *to* 0:
            $G_t = \sum_{k=t}^{T-1} \gamma^{k-t} r_k$
        $L(\theta) = \frac{1}{T} \sum_{t=0}^{T-1} G_t \log \pi_\theta(A_t|S_t)$
        Optimize π_θ *using* $\nabla L(\theta)$
end procedure

For the policy model $\pi(A \mid S, \theta)$, use the network config provided in `LunarLander-v2-config.json`. It already has a softmax output so you shouldn't have to modify the config. As shown in the template code, you can load the model by doing:

```
with open('LunarLander-v2-config.json', 'r') as f:
    model = keras.models.model_from_json(f.read())
```

You can choose which optimizer and hyperparameters to use, so long as they work for learning on `LunarLander-v2`. We recommend using Adam as the optimizer. It will automatically adjust the learning rate based on the statistics of the gradients it's observing. Think of it like a fancier SGD with momentum. Both Tensorflow and Keras provide versions of Adam.

Downscale the rewards by a factor of 1e-2 (i.e., divide by 100) when training (but not when plotting the learning curve). When you implement A2C in the next section, this will help with the optimization since the initial weights of the Critic are far away from being able to predict a large range such as $[-200, 200]$.

To train the policy model, you need to take the gradient of the log of the policy. This is simple to do with Tensorflow: take the output tensor of the Keras model, call `tf.log` on that tensor, and use `tf.gradients` to get the gradients of the network parameters with respect to this log. You will also have to scale by the returns from your sampled policy runs (i.e., scale by G).

Train your implementation on the `LunarLander-v2` environment until convergence[2], and answer the following questions:

1. Describe your implementation, including the optimizer and any hyperparameters you used (learning rate, γ, etc.).
2. Plot the learning curve: Every k episodes, freeze the current cloned policy and run 100 test episodes, recording the mean/std of the cumulative reward. Plot the mean cumulative reward μ on the y-axis with $\pm\sigma$ standard deviation as error-bars vs. the number of training episodes.

   Hint: You can use matplotlib's `plt.errorbar()` function. https://matplotlib.org/api/_as_gen/matplotlib.pyplot.errorbar.html
3. Discuss, in detail, how the learned policy compares to our provided expert and your cloned models.

Solution

1. Implementation setup-
   - Learning Rate-5e-4
   - Optimizer- Adam
   - gamma-1
   - clipped probabilities by [1e-7, 1-e-7]
   - Number of episodes- 45k
2. Learning Curve-

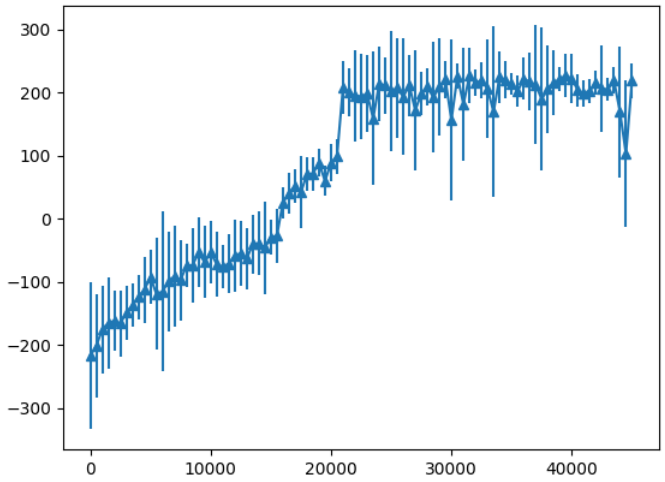


Figure 5: Mean and std. deviation for 100 test episodes every 500 train episodes

3. Discussion- Since the cloned model was directly supervised by the expert, its variance is lower and mean reward higher when compared to the policy learned through REINFORCE. The learned policy allows for exploration while the cloned

[2] `LunarLander-v2` is considered solved if your implementation can attain an average score of at least 200.

model has direct access to the sequence of best possible actions,given a starting state. In other words, the cloned model learned an overfit policy. This means that if the cloned agent is introduced to a state outside of its explored state space, there is a high probability that the episode from then on would result in a failure.
Since the model is learned via exploration, it naturally converges later when compared to the cloned model.

Problem 3: Advantage-Actor Critic (40 pts)

In this section, you will implement N-step Advantage Actor Critic (A2C). Please write your code in `a2c.py`; the template code provided inside is there to give you an idea on how you can structure your code, but is not mandatory to use.

Algorithm 2 N-step Advantage Actor-Critic

procedure N-Step Advantage Actor-Critic
  Start with policy model π_θ *and value model* V_ω
  repeat:
    Generate an episode $S_0, A_0, r_0, \ldots, S_{T-1}, A_{T-1}, r_{T-1}$ *following* $\pi_\theta(\cdot)$
    for t *from* $T-1$ *to* 0:
      $V_{end} = 0$ if $(t+N \geq T)$ *else* $V_\omega(s_{t+N})$
      $R_t = \gamma^N V_{end} + \sum_{k=0}^{N-1} \gamma^k \left(r_{t+k} \text{ if } (t+k < T) \text{ else } 0\right)$
    $L(\theta) = \frac{1}{T} \sum_{t=0}^{T-1} (R_t - V_\omega(S_t)) \log \pi_\theta(A_t|S_t)$
    $L(\omega) = \frac{1}{T} \sum_{t=0}^{T-1} (R_t - V_\omega(S_t))^2$
    Optimize π_θ *using* $\nabla L(\theta)$
    Optimize V_ω *using* $\nabla L(\omega)$
end procedure

N-step A2C provides a balance between bootstraping using the value function and using the full Monte-Carlo return, using an N-step trace as the learning signal. See Algorithm 2 for details. N-step A2C includes both REINFORCE with baseline ($N = \infty$) and the 1-step A2C covered in lecture ($N = 1$) as special cases and is therefore a more general algorithm.

The Critic updates the state-value parameters ω, and the Actor updates the policy parameters θ in the direction suggested by the N-step trace.

As in Problem 2, use the network architecture for the policy model $\pi(A \mid S, \theta)$ provided in `LunarLander-v2-config.json`. Play around with the network architecture of the Critic's state-value approximator to find one that works for `LunarLander-v2`. You can choose which optimizer and hyperparameters to use, so long as they work for learning on `LunarLander-v2`. Downscale the rewards by a factor of 1e-2 during training (but not when plotting the learning curves); this will help with the optimization since the initial weights of the Critic are far away from being able to predict a large range such as $[-200, 200]$.

Answer the following questions:

1. Describe your implementation, including the optimizer, the critic's network architecture, and any hyperparameters you used (learning rate, γ, etc.).
2. Train your implementation on the `LunarLander-v2` environment several times with N varying as [1, 20, 50, 100] (it's alright if the N=1 case is hard to get working). Plot the learning curves for each setting of N in the same fashion as Problem 2.
3. Discuss (in max 500 words) how A2C compares with REINFORCE and how A2C's performance varies with N. Which algorithm and N setting learns faster, and why do you think this is the case?

Solution

1. Implementation setup-
   - Actor Learning Rate-5e-4 for N={1,20,50,100}
   - Critic Learning Rate-1e-4 for N={1,20,50,100}
   - Critic Network architecture-3 fully connected layers, ReLU activations and 16 filters in each layer
   - Optimizer- Adam
   - gamma-1
   - clipped probabilities by [1e-7, 1-e-7]
   - Number of episodes- 45k
2. Learning Curves-

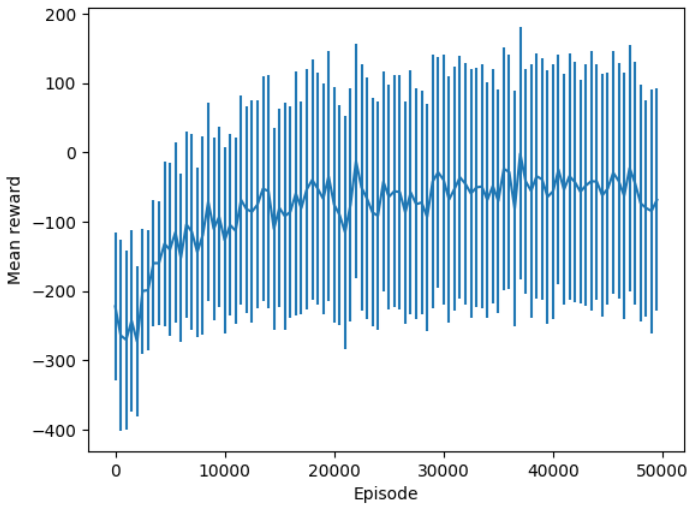


Figure 6: Mean and std. deviation for 100 test episodes every 500 train episodes for N=1

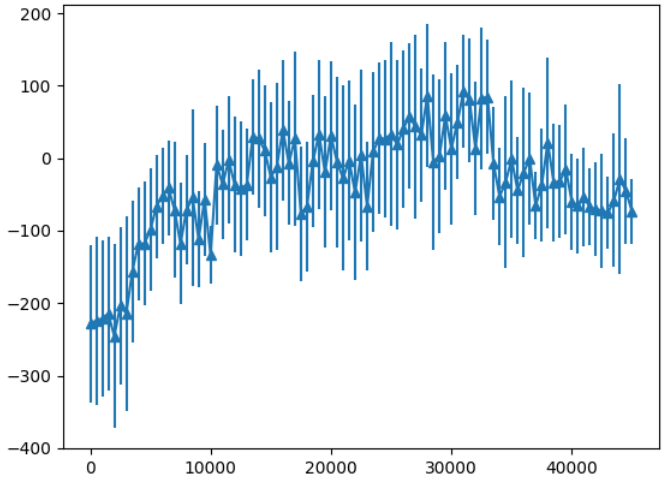


Figure 7: Mean and std. deviation for 100 test episodes every 500 train episodes for N=20

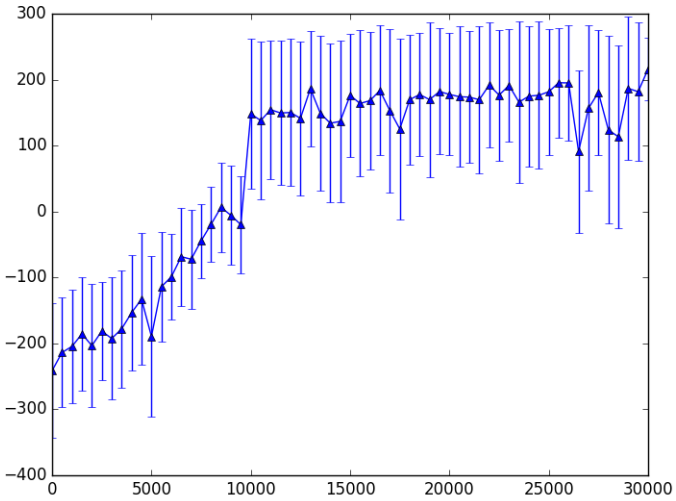


Figure 8: Mean and std. deviation for 100 test episodes every 500 train episodes for N=50

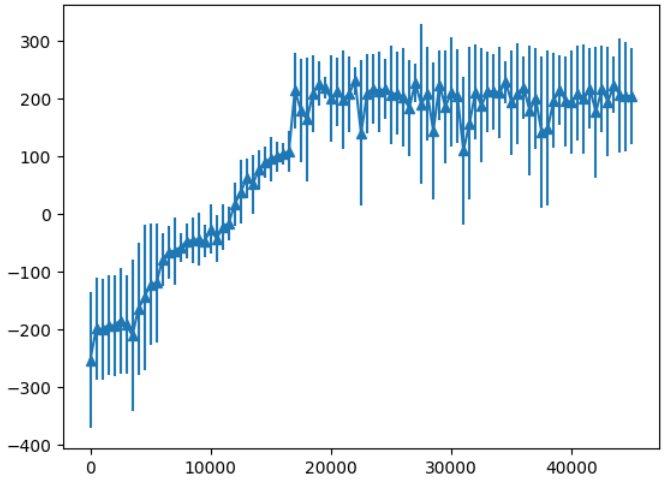


Figure 9: Mean and std. deviation for 100 test episodes every 500 train episodes for N=100

3. Discussion
    - REINFORCE vs A2C:
        - **Convergence**: REINFOCRE had better convergence properties than A2C in the case of LunarLander-v2. REINFORCE generally slows down mainly when the episodes are very long because it needs to wait for the entire episode to end before it can make an update. In the case of LunarLander, the episodes were a maximum of 1000 steps. In fact, towards the end, episode length was only around 500. As such, REINFORCE converged faster than A2C. Following 2 graphs illustrate this. Train rewards in REINFORCE converged in less than 20k episodes but A2C took more than 30k episodes (Note: these graphs were produced from a different setup and platform. As such, they might not match completely with the above plots. Please ignore the straight dark blue line in both the charts. It was because of a loading error in Tensorboard and we didn't want to waste time in cleaning the data.)

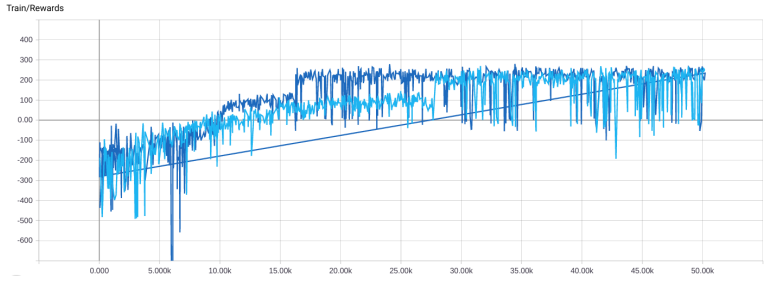


Figure 10: Plot for train rewards for REINFORCE (Dark Blue) and A2C N=100(Light Blue)

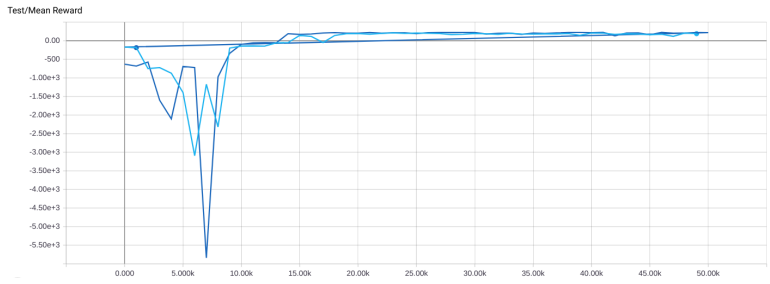


Figure 11: Plot for mean test rewards for REINFORCE (Dark Blue) and A2C N=100(Light Blue). The reason for extreme peaks is explained later.

  - **Variance**: Theoretically, REINFORCE has high variance when compared to A2C. This is mainly because it learns from the entire episode but A2C generalizes using the values provided by the Critic Network. There is more sampling involved in A2C and hence it has low variance. However, we were not able to see clear distinction in variance. It might be because of different parameter setup. Further fine-tuning can help in matching the theoretical stand point.

- N's compared:
  - **Convergence**:- In our experiments we observed that N=50 converged faster at around 12k episodes than N=100 (around 18k episodes) but the models for N=1 and 20 did not converge. However, it can be seen that the model for N=20 did achieve a mean reward of 80 before diverging. We believe that further fine tuning the critic and using shared weights across the actor and critic could guide the model to convergence. Theoretically, N=100 should converge faster than N=1 because the higher episode length, allows the actor to learn from a lot more episodic experience rather than relying on the critic network (which hasn't been trained yet) for guidance.
  - **Variance**:- We posit that before convergence is acheived, N=1 will have a lower variance compared to N=100. The reason being that the critic provides an unreliable estimate of the value function of the state. As such the baseline is not yet helpful in reducing the variance for longer observed episodes. However, after convergence, the critic network is trained and serves as a good baseline to reduce the variance of N=100. It's difficult to comment on the overall impact of N on the variance.
- Documentation of other tried and tested concepts:
  - **Effect of Overtraining Critic Network**: We tried over-training Critic Network for the first 1000 episodes, i.e., Actor Network parameters were updated once in every 10 steps for the first 1000 episodes. This overtraining helped in faster convergence. This is intuitive as well because first we are enhancing Critic Network to give a good estimate of state value and then using that to train the Actor Network.

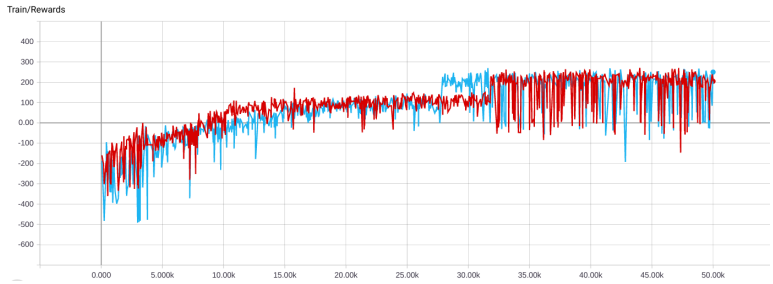


Figure 12: Plot for train rewards for A2C N=100 Overtrained (Light Blue) and normal (Red)

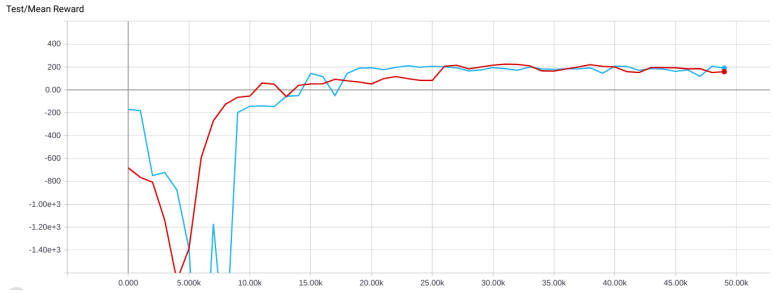


Figure 13: Plot for mean test rewards for A2C N=100 Overtrained (Light Blue) and normal (Red). The reason for extreme peaks is explained later.

- **Subtracting Entropy from Actor Loss**: We tried changing the actor loss by subtracting the entropy of action probabilities, in order to encourage exploration. However, that didn't help in improving performance.
- **Argmax vs random in Test Episodes: Reason for large peaks in the above Test Reward Plots**: For the test episodes, there were extreme peaks observed in some plots. We attribute to the use of Argmax instead of random sampling of action. When we switched to random sampling, these peaks were no more observed

Guidelines on implementation

This homework requires a significant implementation effort. It is hard to read through the papers once and know immediately what you will need to be implement. We suggest you to think about the different components (e.g., Tensorflow or Keras model definition, model updater, model runner, ...) that you will need to implement for each of the different methods that we ask you about, and then read through the papers having these components in mind. By this we mean that you should try to divide and implement small components with well-defined functionalities rather than try to implement everything at once. Much of the code and experimental setup is shared between the different methods so identifying well-defined

reusable components will save you trouble.

Please note, that while this assignment has a lot of pieces to implement, most of the algorithms you will use in your project will be using the same pieces. Feel free to reuse any code you write for your homeworks in your class projects.

This is a challenging assignment. **Please start early!**

References

[1] J Andrew Bagnell. An invitation to imitation. Technical report, DTIC Document, 2015.

[2] Stephane Ross. Interactive learning for sequential decisions and predictions. 2013.

[3] Stephane Ross, Geoffrey J Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In AISTATS, volume 1, page 6, 2011.