ENPM690 REINFORCEMENT LEARNING

FINAL PROJECT

IS RL AS GOOD AS WE THINK IT IS?

ADVERSARIAL ATTACK ON REINFORCEMENT LEARNING ALGORITHM

Submitted by:

Name: Mithun Bharadwaj

UID: 114931115

Table of Contents

ABSTRACT:

Reinforcement learning (RL) is one of three basic machine learning paradigms, alongside supervised learning and unsupervised learning. RL is concerned with how software agents ought to take actions in an environment in order to **maximize the expected cumulative reward**. RL is different from supervised learning in the sense that it **relies on the agent's interaction with the environment** to learn the optimal behavior for the task. There are no labelled input/output pairs necessary for training RL algorithms unlike in supervised learning.

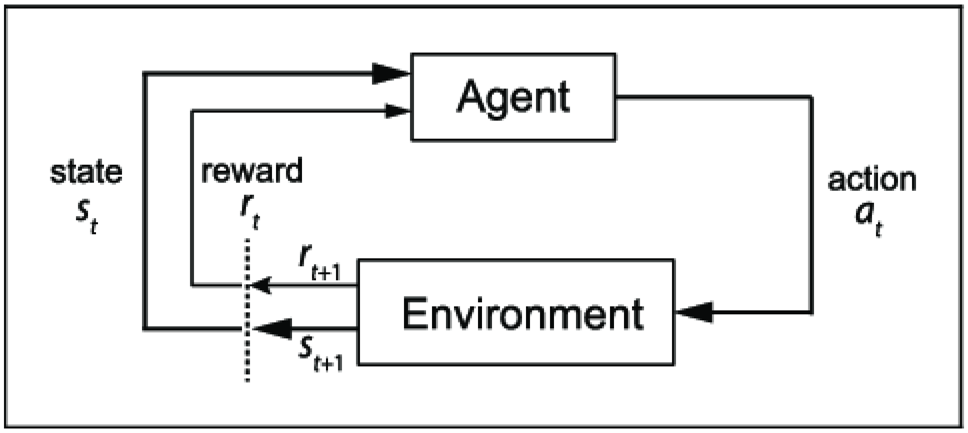


Fig 1. Reinforcement Learning

According to MIT Technology Review, Reinforcement Learning has been gaining popularity in the recent years [1]. It is widely being used in applications such as autonomous driving, gaming, NLP and other prominent fields that require sequential actions as response to the environment.

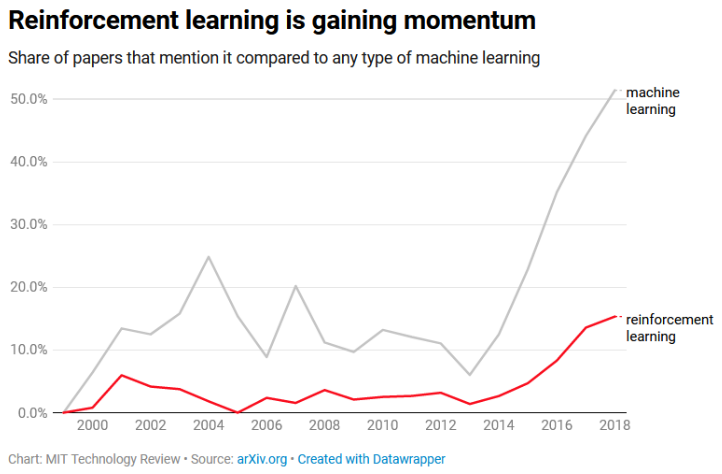


Fig 2. Reinforcement Learning gaining popularity [1]

With the rise in utilization of RL algorithms for high risk tasks, it is important to make sure the models we're using are robust when implemented in the real world. Since what the neural network/ model is learning is still unclear, it is hard to predict the failure points of the decision boundary for the model. But recently, there has been increasing focus given to identifying these vulnerabilities through adversarial attacks and improving robustness through various defenses for these attacks.

Adversarial attacks have been studied considerably for classifiers and supervised tasks. In this project I would like to explore the weak points of RL models through a simple Fast Gradient Sign Method (FGSM) adversarial attack and study the effect of small perturbations to the state on the final policy of the model. The attack is studied for a model trained using Deep Q-Learning in the Pong environment in the OpenAI Gym. (Ping-Pong Atari game). The goal is to expose the weak robustness of these policies that generally perform well but produce totally unexpected and sub-optimal results even with a negligible perturbation to the states.

INTRODUCTION:

The problem setting in this case is FGSM attack on a trained Deep Q-Learning model for the Pong environment. Understanding the problem will make the necessity of studying this area and its impact much clearer. The main aim of this project is to expose the weak points in the decision boundary of RL models, and although this has been studied extensively in the classifier and supervised learning setting, it has not been paid too much attention in the RL setting. Observing the rise in significance of RL in recent times, it's import to study the robustness of the policies learned.

The ease with which a model can be fooled is counter intuitive considering how well it has learned the data, based on the accuracy of training and validation. Considering Machine Learning models are deployed everywhere constantly, and a lot of organizations and funding are focused around AI technology, it becomes increasingly important to study their robustness as the impact of AI spreads especially in high risk applications such as autonomous driving where lives are at stake.

This project implements a simple FGSM attack which perturbs the state by a small amount considering the gradient of the loss with the state in a way that increases the loss. The perturbation is so slight that visually it doesn’t make any difference in the state and can be considered as noise (0.1% of the max value of the state in this case). This small amount of perturbation is enough to fool the system to choose a different action than it normally would have, meaning the policy is not robust enough even though it does well under normal circumstances.

FGSM is one of the most efficient but simplest attacks to understand, and yet as the results show later, it brings a huge change to how the policy acts. There have been much more complicated and effective attacks studied for the classifier case which can cause more havoc with lesser perturbations. Moreover, the model will be much more complicated for a self-driving car compared to an agent in a pong game, which means there are many weaker spots in the former model. This goes to show although ML models have worked reliably for most applications and are used widely, we need to be weary of the consequences of using them everywhere without checking their robustness.

BACKGROOUND/ RELATED WORK:

Atari Environment

We consider tasks in which an agent interacts with an environment $\boldsymbol{\varepsilon}$, in this case the Atari emulator, in a sequence of actions, observations and rewards. At each time-step the agent selects an action $\boldsymbol{a_t}$ from the set of legal game actions, $\boldsymbol{A} = \{\boldsymbol{1}, \ldots, \boldsymbol{K}\}$. The action is passed to the emulator and modifies its internal state and the game score. In general $\boldsymbol{\varepsilon}$ may be stochastic. The emulator’s internal state is not observed by the agent; instead it observes an image $\boldsymbol{x_t} \in \mathbb{R}^{\boldsymbol{d}}$ from the emulator, which is a vector of raw pixel values representing the current screen. In addition, it receives a reward $\boldsymbol{r_t}$ representing the change in game score. Note that in general the game score may depend on the whole prior sequence of actions and observations; feedback about an action may only be received after many thousands of time-steps have elapsed. Since the agent only observes images of the current screen, the task is partially observed and many emulator states are perceptually aliased, i.e. it is impossible to fully understand the current situation from only the current screen $\boldsymbol{x_t}$. We therefore consider sequences of actions and observations, $\boldsymbol{s_t} = \boldsymbol{x_1, a_1, x_2, \dots \dots, a_{t-1}, x_t}$ and learn game strategies that depend upon these sequences. In this implementation, 4 pervious states have been considered.

The goal of the agent is to interact with the emulator by selecting actions in a way that maximizes future rewards.

Deep Q-Learning

***Q*-learning** is a model-free (doesn't need to know the exact MDP) reinforcement learning algorithm to learn a policy telling an agent what action to take under what circumstances. It does not require a model (hence the connotation "model-free") of the environment, and it can handle problems with stochastic transitions and rewards, without requiring adaptations. The basis of this is the value function. There are 2 types of value function:

1. **State value function:** For each state, it yields the expected return if the agent started in that state and then followed the policy for the remaining time steps.
2. **Action value function (Q-value):** For each state **s**, and action **a**, the action value function yields the discounted return if the agent starts in state **s**, takes action **a** and follows the policy $\boldsymbol{\pi}$

The action value function basically gives the expected reward if you take the action **a** from state **s**. This means if we can calculate the action values accurately for all the valid actions that can be taken from that particular state, then the action with the highest Q-value is the optimal action to take. For any finite Markov decision process (FMDP), *Q*-learning finds an optimal policy in the sense of maximizing the expected value of the total reward over any and all successive steps, starting from the current state [10].

We make the standard assumption that future rewards are discounted by a factor of $\boldsymbol{\gamma}$ per time-step, and define the future discounted return at time $\boldsymbol{t}$ as $\boldsymbol{R_t} = \sum_{t'=t}^{T} \boldsymbol{\gamma^{t'-t} r_{t'}}'$, where $\boldsymbol{T}$ is the time-step at which the game terminates. We define the optimal action-value function $\boldsymbol{Q}_*(\boldsymbol{s},\boldsymbol{a})$ as the maximum expected return achievable by following any strategy, after seeing some sequences and then taking some action $\boldsymbol{a}$,

$$\boldsymbol{Q}_*(\boldsymbol{s},\boldsymbol{a}) = \boldsymbol{max_\pi}\ \mathbb{E}[\boldsymbol{R_t}|\boldsymbol{s_t} = \boldsymbol{s}, \boldsymbol{a_t} = \boldsymbol{s}, \boldsymbol{\pi}] \qquad \textbf{(1)}$$

Where $\boldsymbol{\pi}$ is a policy mapping sequences to actions (or distributions over actions).The optimal action-value function obeys an important identity known as **the Bellman equation.**

$$\boldsymbol{Q}(\boldsymbol{s},\boldsymbol{a}) = \boldsymbol{r}(\boldsymbol{s},\boldsymbol{a}) + \boldsymbol{\gamma} \max_{\boldsymbol{a}} \boldsymbol{Q}(\boldsymbol{s'},\boldsymbol{a}) \qquad \textbf{(2)}$$

Equation (2) shows the Bellman equation using Q-values.

As the number of valid states increases, and it usually does pretty rapidly (For example, the number of valid states in the game Go are of the order 10^{170}), the calculation of Q values become extremely difficult and intensive. Using Neural Networks to estimate the Q-value in such cases is the Seep Q-Learning algorithm. The network is usually referred to as the *approximator* or the *approximating function*, and denoted as $\boldsymbol{Q}(\boldsymbol{s},\boldsymbol{a};\ \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ represents the trainable weights of the network.

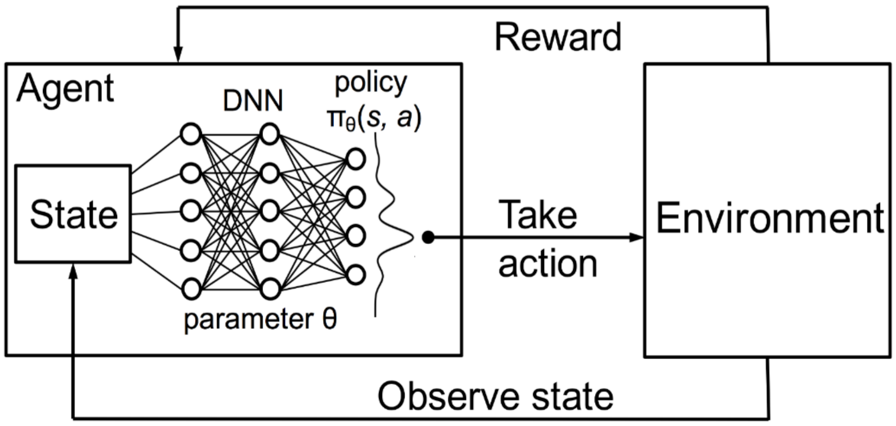


Fig 3. Deep Q Learning

Adversarial attack and FGSM:

Adversarial examples are inputs to machine learning models that have been intentionally optimized to cause the model to make a mistake. We call an input example a “clean example” if it is a naturally occurring example, such as a photograph from the ImageNet dataset. If an adversary has modified an example with the intention of causing it to be misclassified, we call it an “adversarial example.” Of course, the adversary may not necessarily succeed; a model may still classify the adversarial example correctly.

Attack scenarios can be classified by the amount of knowledge the adversary has about the model:

- **White box:** In the white box scenario, the adversary has full knowledge of the model including model type, model architecture and values of all parameters and trainable weights.
- **Black box with probing**: In this scenario, the adversary does not know very much about the model, but can probe or query the model, i.e. feed some inputs and observe outputs. There are many variants of this scenario—the adversary may know the architecture but not the parameters or the adversary may not even know the architecture, the adversary may be able to observe output probabilities for each class or the adversary may only be to observe the choice of the most likely class.
- **Black box without probing**: In the black box without probing scenario, the adversary has limited or no knowledge about the model under attack and is not allowed to probe or query the model while constructing adversarial examples. In this case, the attacker must construct adversarial examples that fool most ma-chine learning models.

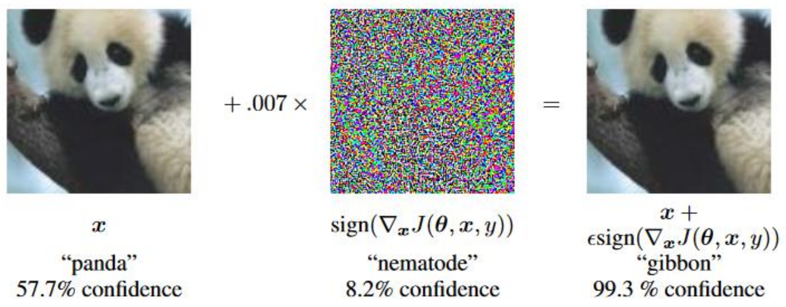

+ .007 ×
=
x
"panda"
57.7% confidence
$\text{sign}(\nabla_x J(\theta, x, y))$
"nematode"
8.2% confidence
$x + \epsilon\text{sign}(\nabla_x J(\theta, x, y))$
"gibbon"
99.3 % confidence

Fig 3. Adversarial example [4]

Fast Gradient Sign Method:

To test the idea that adversarial examples can be found using only a linear approximation of the target model, the authors of [4] introduced the fast gradient sign method (FGSM), which is a type of white-box attack.

FGSM works by linearizing loss function in L_∞ neighborhood of a clean image and finds exact maximum of linearized function using following closed-form equation:

$$\boldsymbol{x^{adv}} = \boldsymbol{x} + \boldsymbol{\varepsilon}\, \boldsymbol{sign}(\boldsymbol{\nabla_x J}(\boldsymbol{x}, \boldsymbol{y_{true}})) \qquad \textbf{(3)}$$

APPROACH:

The high level approach to:

1. Understand the environment, i.e the state, action space, episode and reward.
2. Keep the input constant and train a Deep Q-Learning network to develop a policy that can consistently beat the computer.
3. Keep the weights/parameters constant and implement FGSM attack. To implement FGSM:
    a. Forward propagate the state through the network to get the loss
    b. Backward propagate through the network to obtain gradients.
    c. Perturb the state according to equation (3)
    d. Get the corresponding action for this state.
4. Simulate the environment using the action for the perturbed state.
5. Observe the output, i.e if the policy is corrupted.
6. As a control implement a simulation which just adds random noise to the state and observe if the policy degrades.

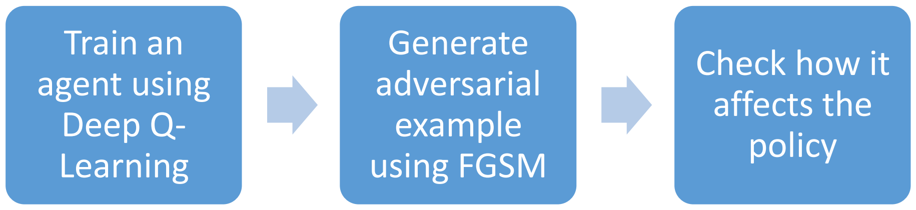

Train an agent using Deep Q-Learning
Generate adversarial example using FGSM
Check how it affects the policy

Fig 4. Main approach followed

IMPLEMENTATION:

1. **Atari Environment (Pong):**
    a. **States:** Pixel values (0, 255) of the frame
    b. **Action:** 3 actions → Go up, go down, stay still
    c. To get the environment to work properly, we need **Atari wrappers** provided in the OpenAI gym baselines repository which

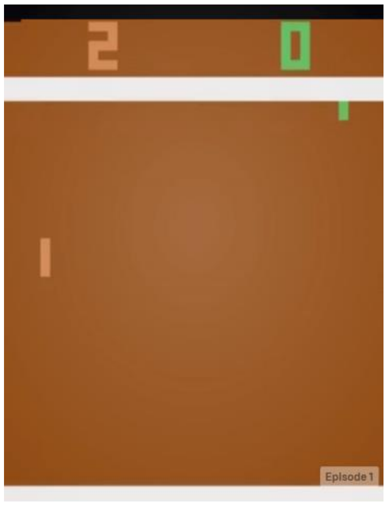
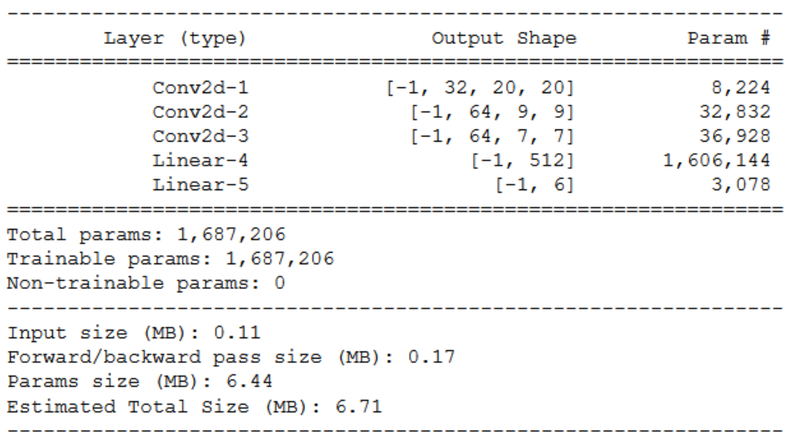


Fig 5. Pong environment

2. **Deep Q-Learning:**
    a. **Software/ Modules:**
        i. Python
        ii. PyTorch
        iii. OpenAI Gym
        iv. Torchsummary

b. Model architecture:

Model summary is as per the below image



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 20, 20]           8,224
            Conv2d-2             [-1, 64, 9, 9]          32,832
            Conv2d-3             [-1, 64, 7, 7]          36,928
            Linear-4                  [-1, 512]       1,606,144
            Linear-5                    [-1, 6]           3,078
================================================================
Total params: 1,687,206
Trainable params: 1,687,206
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.11
Forward/backward pass size (MB): 0.17
Params size (MB): 6.44
Estimated Total Size (MB): 6.71
----------------------------------------------------------------
```

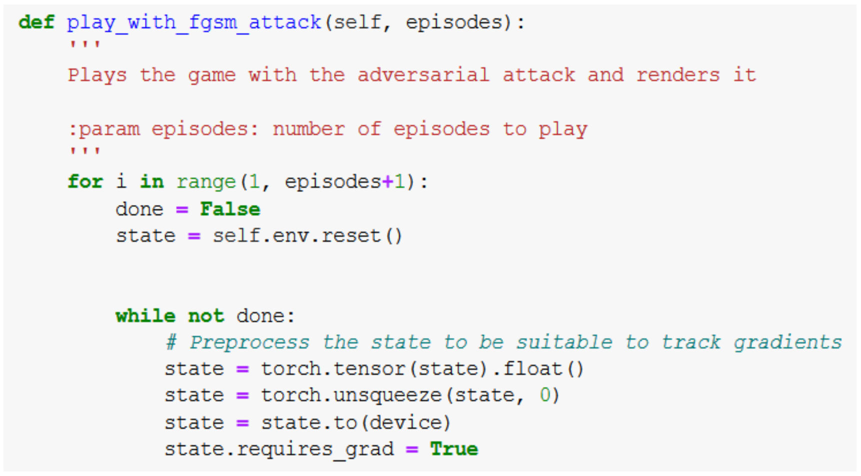
Fig 6. Model summary

c. Reward structure:

-1 if the agent loses the point and +1 if the agent wins. Episode vs rewards table cab be seen in the results section.

3. FGSM Implementation:

a. Preprocess: Since we need the gradient of the loss with respect to the state, we need to convert the state to a Tensor that can be processes by PyTorch and make it keep track of the gradients through the network.



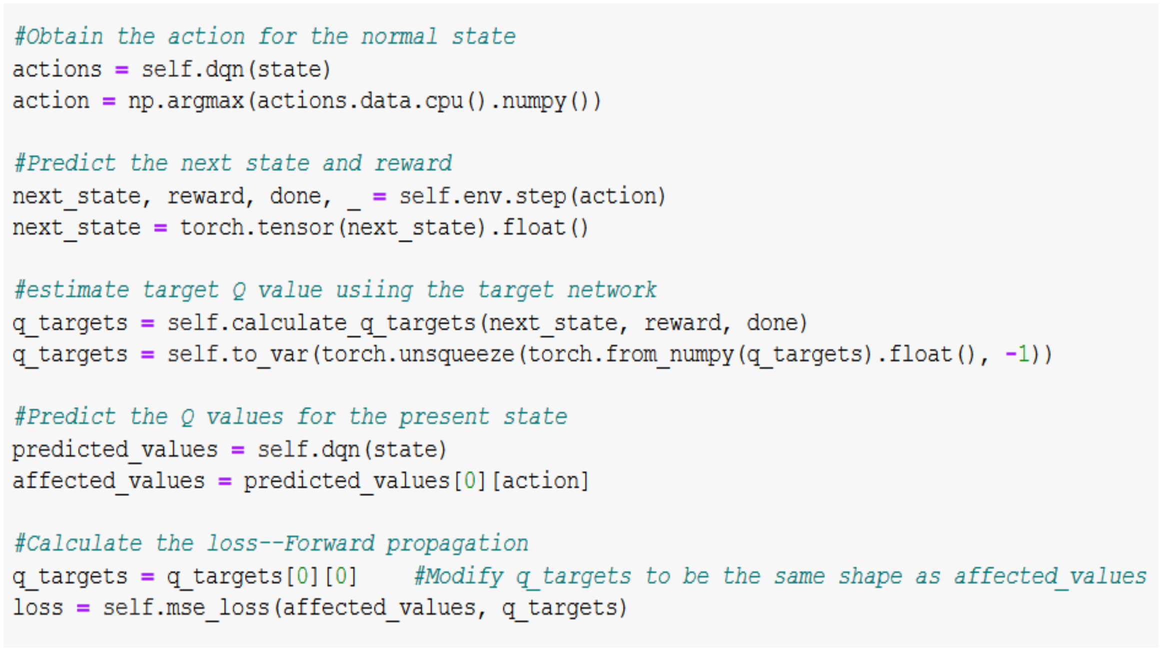
```python
def play_with_fgsm_attack(self, episodes):
    '''
    Plays the game with the adversarial attack and renders it

    :param episodes: number of episodes to play
    '''
    for i in range(1, episodes+1):
        done = False
        state = self.env.reset()


        while not done:
            # Preprocess the state to be suitable to track gradients
            state = torch.tensor(state).float()
            state = torch.unsqueeze(state, 0)
            state = state.to(device)
            state.requires_grad = True
```

Fig 7. Preprocessing step

b. **Forward propagation:** Propagate the processed state through the network to calculate the loss



```
#Obtain the action for the normal state
actions = self.dqn(state)
action = np.argmax(actions.data.cpu().numpy())

#Predict the next state and reward
next_state, reward, done, _ = self.env.step(action)
next_state = torch.tensor(next_state).float()

#estimate target Q value usiing the target network
q_targets = self.calculate_q_targets(next_state, reward, done)
q_targets = self.to_var(torch.unsqueeze(torch.from_numpy(q_targets).float(), -1))

#Predict the Q values for the present state
predicted_values = self.dqn(state)
affected_values = predicted_values[0][action]

#Calculate the loss--Forward propagation
q_targets = q_targets[0][0]    #Modify q_targets to be the same shape as affected_values
loss = self.mse_loss(affected_values, q_targets)
```

Fig 8. Forward propagation

c. **Backprop:** Back propagate through the network to calculate the gradient of the loss with respect to the state and ass the perturbation to the state.

4. Simulate the resulting action:

After perturbing the state, get the action corresponding to the perturbed state and simulate the environment.

5. Compare with a control simulation:

Add random noise which is not calculated to maximize loss and observe if the policy degrades.

RESULTS:

Results during training

Episode	Reward
10	-20
20	-17
30	-20
40	-20
50	-21
100	-15
200	-7
250	13
300	17
350	19
400	12

Table 1: Episode vs Reward

FGSM Attack results and control simulation results:

To view the code for this project, video results of the trained model, attacked model and control simulation, please check the repository for this project on my GitHub.

Project GitHub repo: https://github.com/mithun-bharadwaj/RL_Final_Project

ANALYSIS:

- Deep Q-Learning performed well in the environment and the agent was trained to an acceptable proficiency for the game as expected based on the theory of Q-Learning.
- FGSM was quite effective in attacking the network and corrupting the policy as the intuitive understanding of FGSM is that it tries to use the gradient of the loss w.r.t the

input to maximize loss and therefore the calculated perturbations cause the trained network to take actions that it previously would not have taken.

- As expected the control simulation with random noise added to the state did not make much of a difference to the policy as it is not optimized to maximize the loss function and hence does not affect the gradients too much.

CONCLUSION

1. Deep Q-Learning is very effective when the number of valid states increase rapidly.

2. Although it is effective, Q-Learning is very intensive on the hardware, and therefore I had to use cloud resources to train the model.

3. Although FGSM is quite simple and intuitive to understand, as the results show it can have huge impact on the policy.

4. Therefore, we should consistently question the reliability and robustness of the networks we deploy.

FUTURE WORK:

1. Check the vulnerabilities of the policy to other type of attacks.
2. Make the policy more robust to such perturbations by defending these attacks.
3. Certifiable robustness where we can assure if the state is perturbed by certain amount, it won’t have any impact on the policy.
4. All of these involve in-depth understanding of the models, what’s happening under the hood and how the model arrives at the decision boundary.

BIBLIOGRAPHY

1. Karen Hao, *MIT Technology Review article on RL*, https://www.technologyreview.com/2019/01/25/1436/we-analyzed-16625-papers-to-figure-out-where-ai-is-headed-next/
2. Alexey Kurakin, Ian Goodfellow, et al; *Adversarial Attack and Defenses Competition*, **NIPS 2018**, (https://arxiv.org/pdf/1804.00097.pdf)
3. Volodymyr Mnih, Koray Kavukcuoglu, et al.; *Playing Atari with Deep Reinforcement* Learning, **NIPS 2013** (https://arxiv.org/pdf/1312.5602v1.pdf)
4. Ian Goodfellow, Jonathan Shelns, Christian Szegedy; *Explaining and Harnessing Adversarial Examples*, NIPS 2015 (https://arxiv.org/pdf/1412.6572.pdf)
5. Xiaoyong Yuan, Pan He, Qile Zhu, Xiaolin Li; *Adversarial Examples: Attacks and Defenses for Deep Learning*
6. Adam Gleave, Michael Dennis, et al; *Adversarial policies: Attacking Deep Reinforcement Learning* (https://arxiv.org/abs/1905.10615)
7. Lerrel Pinto, James Davidson, et al; *Robust Adversarial Reinforcement Learning* (http://proceedings.mlr.press/v70/pinto17a/pinto17a.pdf)
8. OpenAI baseline (https://github.com/openai/baselines)
9. Pong Environment, OpenAI Gym (https://gym.openai.com/envs/Pong-v0/)
10. Q-Learning, Wikipedia (https://en.wikipedia.org/wiki/Q-learning)