AgentSteve Documentation

RockmanZheng

Jun 09, 2018

Contents

CHAPTER 1

Directory Structure

I will not fully describe all funcitions of all directories and files. This should be left to module explanation docs or README file under each directory. Here I will just briefly introduce you to some of the folders and files.

- *docs*. Directory storing files asscioated with (this) documentations.
- *Research*. Directory for storing researching codes.

Under the folder *Research*, there are 2 folders:

- *imitate_experiments*. My currently working directory. All experiments involving imitation module are performed in here. All experiments will be recorded in 3 digits code, such as 001, and stored in a folder in the same name.
- *explore_experiments*. My second currently working directory. Experiments relating exploring module happens here.

CHAPTER 2

Research Logs

Here is a list of research logs, which record all the research ideas and experiments in the project AgentSteve.

2.1 Tue May 22 2018

Current progress: working on imitation module.

I was inspired by this paper *Third-Person Imitation Learning*, which is available on arXiv. Link

There are 3 important parts in the imitation module, namely Df, Dr, Dd, which are feature extractor, performer discriminator, and domain discriminator, respectively, as mentioned in the paper.

Images are first sent into Df which will then extract features from them. These features should be domain invariant (environment agnostic). In other words, one cannot tell from where the initial images come solely based on these features.

The extracted features are then sent into Dd. Dd will classify their source domains. A good Df will output confusing features as discussed above so that a good Dd cannot tell the difference and will give any answer with equal probability.

The features are also sent into Dr. In fact, features from several time-steps will be sent to introduce temporal information. Dr will then give its judgement about whether it is the expert doing the task, or it is an amateur. The environment agnostic natrue of the features should improve the accuracy of discrimination result.

We discovered that 3 components need each other and no one can be singled out. Feature extractor Df is neccessary since dealing higher level concept is better than handling raw pixels. Domain discriminator Dd is also needed because we need to force our Df to be domain invariant. And finally, if without Dr, the Df will easily adjust to output constant value to fool Dd, which is clearly not desired. Thus Dr is crucial for training Df to extract meaningful features.

So we can only implement and test 3 components all at once. If we see them as a new whole integrity then we will feel less pain (:-)

2.1.1 Coding

I had stated a new experiment under the directory *Research/imitate_experiments*, coded 001. And add a few new files, including:

- *domain_discriminator.py*
- *feature_extractor.py*
- *imitateOptimizer.py*
- *performer_discriminator.py*
- *script01.py*

Currently I have made a draft version of algorithm structure in *script01.py*. It merely outlines all elements that I thought of, and cannot be run. Other files are useless for the moment.

In *script01.py* I basically just put some TensorFlow layers API in there, and exposing their function signatures for later development.

Here is the architecture that I am thinking of.

First of all, we will need an input layer. We will use video stream to train our imitator instead of simple images. So the input layer would be a 5 dimension tensor, with dimension [batches,depth,height,width,channels]. Here *batches* refers to the number of training batches, *depth* is the number of frames. And *height* and *width* is the size the our input images. And *channels* typically equals to 3 for colored images, for example those in RGB or HSI format, and equals to 1 for gray scale images.

Df, the feature extractor, can be implemented as the first few layers in a neural network based discriminator. So after replicating a simple MINST discriminator (see this tutorial), we choose the first 4 layers of it to play the role of a feature extractor. Namely:

- Convolutional layer #1, with ReLU activation
- Max pooling layer #1
- Convolutional layer #2, with ReLU activation
- Max pooling layer #2, also named feature layer, since we are constructing a feature extractor

Then we will send output of the last layer into both Dd and Dr.

Dd and Dr, the domain and performer discriminators, will have same structure (at least for now), since they are both classifier and both using output of Df as input. The structure is:

- Dense layer #1, with ReLU activation
- Dropout layer
- logits layer #2 (dense), with 2 units, outputing logits for 2 classes. For Dd, they represent expert domain and amateur domain. For Dr, they represent expert and amateur.

Finally we will use sigmoid function to convert logits to values within [0,1), and view them as probability. Probabilities of 2 units should sum up to 1.

2.1.2 Future Works

Training procedure needs to be constructed. And input feed needs to be prepared.

2.2 Wed May 23 2018

Current Progress: Made a draft of training procedure. Still need to prepare data feeds.

Working Directory: *Research/imitate_experiments/001*

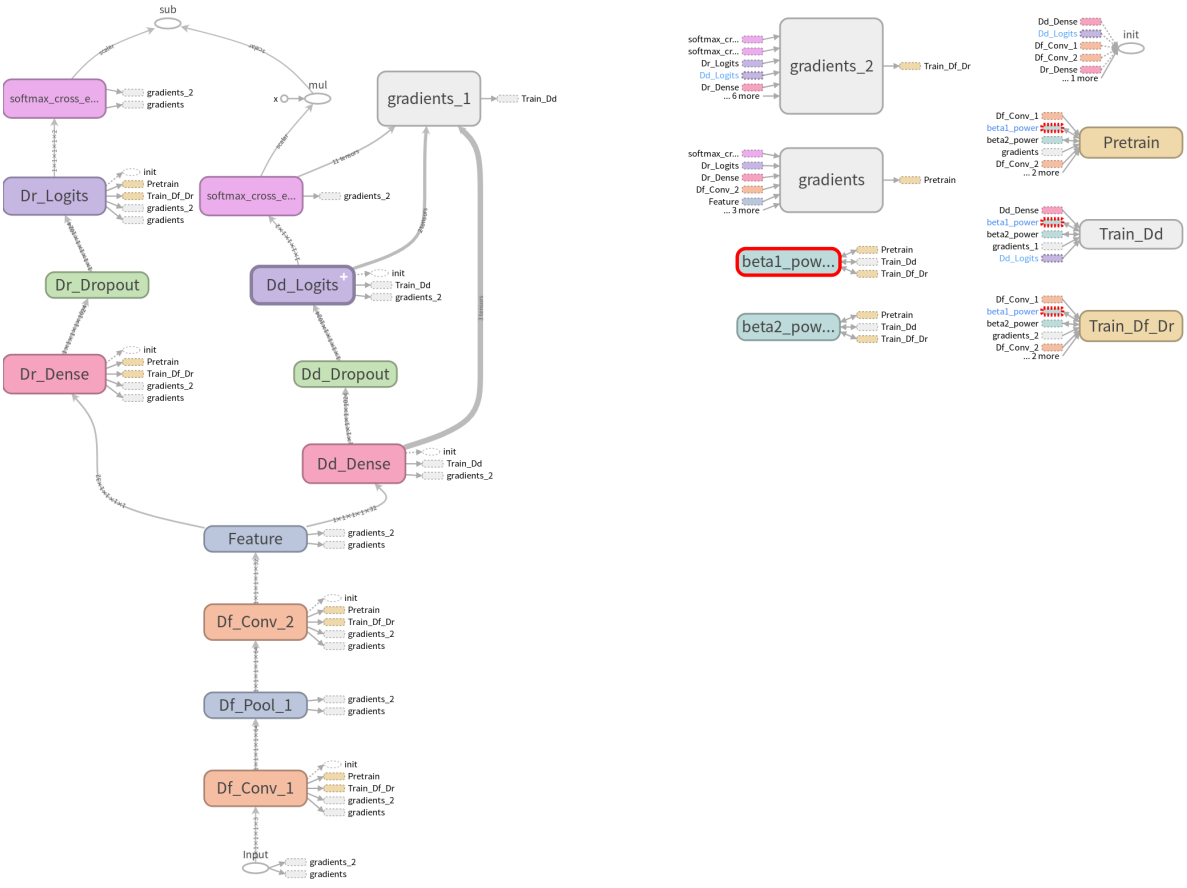
We made a copy of *script01.py* to create *script02.py* and worked on it. This link directs you to the script.

The proposed training procedure is as follow:

1. *Pretraining*. Train Df and Dr, minimizing Lr, where Lr refers to loss of Dr, the performer discriminator
2. Fixed Df, and train Dd to minimize Ld, where Ld refers to loss of Dd, the domain discriminator. The goal of this step is to gain a good Dd.
3. Fixed Dd, train Df and Dr, minimizing $L = L_r - \lambda L_d$, where $\lambda > 0$ is a hyperparameter. Thus we are minimizing Lr and maximizing Ld all at the same time. Here the goal is to obtain an envioronment agnostic Df and a good Dr based on it.
4. Go back to step 2 and repeat until convergence.

2.2.1 Coding

Training procedure draft is in *script02.py*. It can now be executed (with TensorFlow successfully installed) and produce the following graph in *model* folder (visualized using TensorBoard):



We can use this graph to verify the whole training procedure is constructed as we expected. We will now briefly discussed how we have coded the training graph. To view this graph on TensorBoard, please check this page for a quick guide.

As mentioned above, one of Df, Dr and Dd will be blocked from updating in the training. In TensorFlow's language, in some phase of training, *trainable variables* of some component will not be updated through backpropagation. To realize this effect, we will be utilizing the `var_list` signature of the method `tf.train.Optimizer.minimize`. By providing a specific collection of `tf.Variable`'s to `var_list`, we are able to force `tf.train.Optimizer.minimize` to minimize the given loss only by updating those `tf.Variable`'s and leaving others, i.e. other parts of the model unchanged.

So you will see something like:

```
# Create an Adam optimizer
adam_op = tf.train.AdamOptimizer()
# Only updating tf.Variable's in update_vars when minimizing loss function Loss
train_op = adam_op.minimize(loss = Loss,var_list = update_vars)
```

So back to the training procedure proposed in the begining, in pretraining phase and step 3, we only need to update `tf.Variable`'s in Df and Dr; in step 2, we only need to update `tf.Variable`'s in Dd. In order to retrieve those `tf.Variable`'s of each part, we will be using the property `tf.layers.Layer.trainable_variables` that can give us access to all trainable variables in a particular `tf.layers.Layer`. Thus in *script02.py*, we construct layers using API of the class `tf.layers.Layer`, instead of their corresponding functional forms. For example, we use `tf.layers.Conv3D` rather than `tf.layers.conv3d`. You can theck this guide for basic usage of `tf.layers.Layer`.

So you will see something like:

```
# List of tf.Variable's that will be passed to var_list
train_vars = []
# A dense layer
dense_layer = tf.layers.Dense(units=10)
# Must first initialize all variables
# Otherwise, dense_layer.trainable_variables will be an empty list
sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)
# Add trainable variables to list
train_vars+=dense_layer.trainable_variables
```

One important thing needs to be notified, that before adding trainable variables to list, we must first initialize all variables by running `tf.global_variables_initializer()` in a `tf.Seesion`, as shown above. Otherwise, we would get an empty list, and later no `tf.Varialbe` will be updated!

With knowledge of above things, we created 3 optimizing operations corresponding 3 training steps, namely `pretrain_op` for pretraining, `dd_train_op` for training domain discriminator Dd, and `train_op` for training environment agnostic feature extractor Df and performer discriminator Dr based on it. After this, we need to run variables initialization `tf.global_variables_initializer()` again to make things work.

Now we are prepared to run these optimizing operations to train our imitation model by simply calling `tf.Session.run` method.

2.2.2 Future Works

We still need to prepare input data feeds.

2.3 Tue May 29 2018

Last time we ran Malmo on my Ubuntu virtual machine, the frame rate was so slow that we could not even see what is going on. So today we enlarged display memory capacity to 128M and discovered that the game runs more smoothly. Later we performed a render speed test and found that smaller rendering size could result in higher frame rate and lower data throughput.

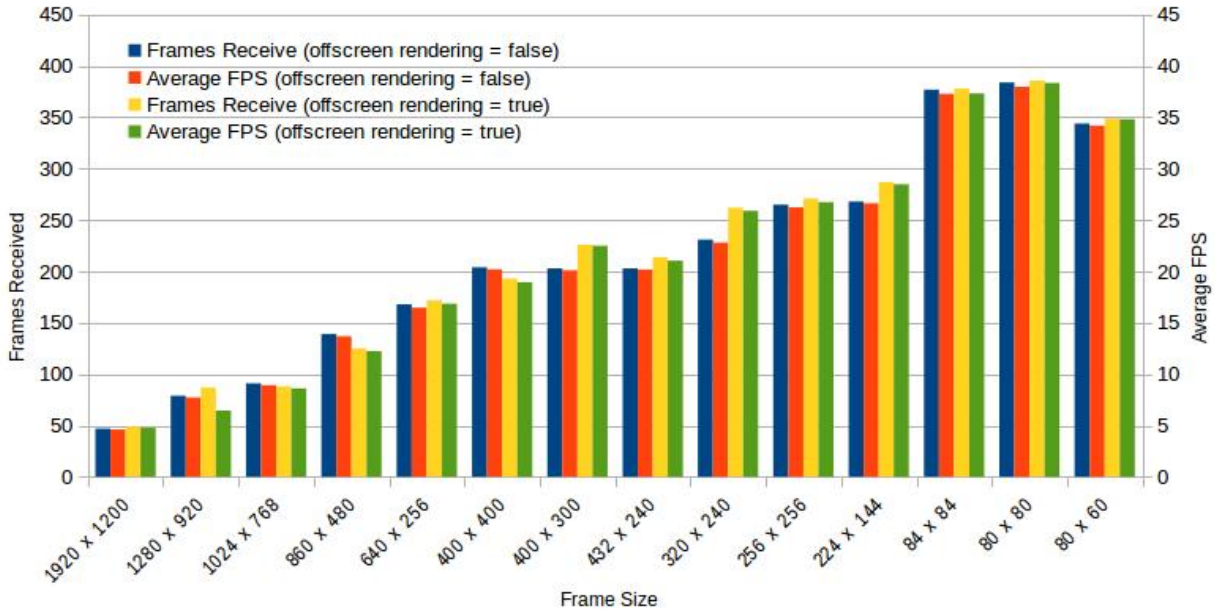
My virtual machine's configuration:

- OS: Ubuntu 16.04 64bit
- Memory: 5120MB
- Display Memory: 128MB
- 4 CPUs: Intel Core i5-6300HQ at 2.30GHz

Working Directory: *AgentSteve/Malmo-0.34.0-Linux-Ubuntu-16.04-64bit_withBoost_Python2.7*

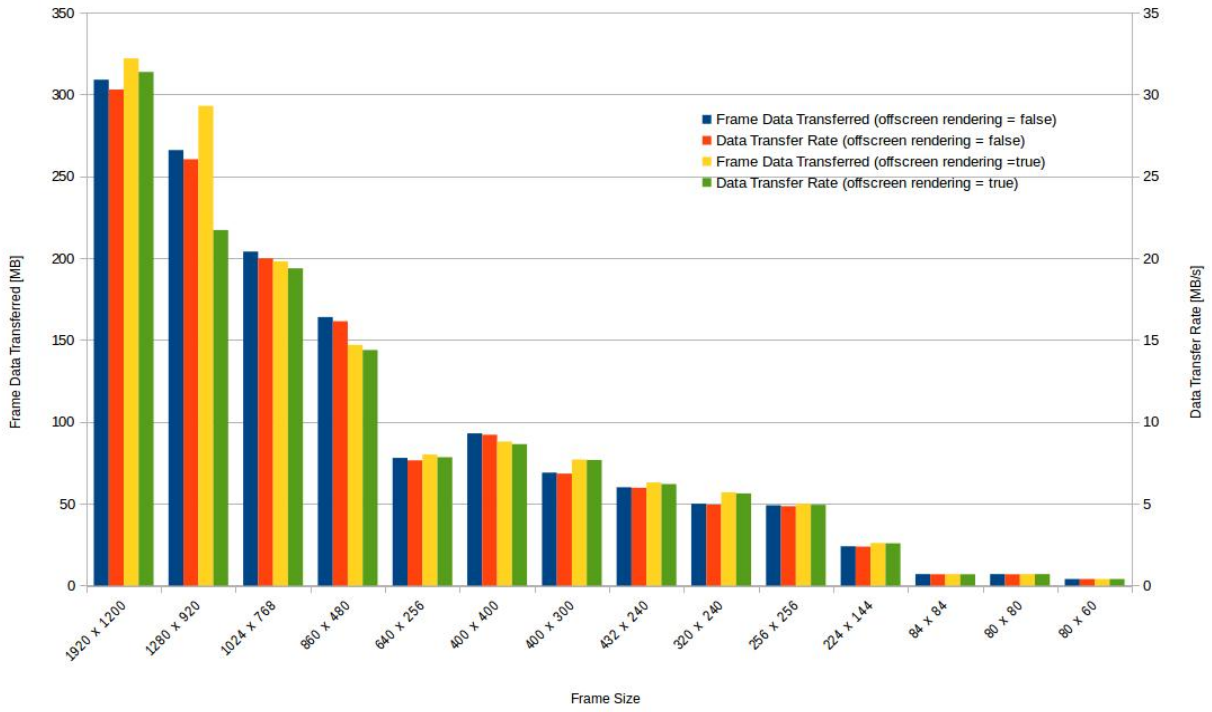
Now, launch the Malmo platform. Under *Python_Examples* folder, we run the script *render_speed_test.py* in terminal. Below are the results, we have visualized them with charts, where x axis represents different frame size (e.g. 1024 X 768).

First look at results related with frames.



In general, as can be seen, as frame size decreases, we will have higher FPS and receive more frames. So in our future experiments, we should use a relatively small frame size to obtain high FPS, so that the game runs smoothly. And in most cases, turning on the offscreen rendering option will result in higher FPS.

Then let's look at results related with data throughput.



From the picture above, we can discover that as frame size decreases, data throughput also decreases, rapidly. So in our later experiments, we should use a relatively small frame size to obtain low data throughput, so that our system doesn't need to handle high pressure of data.

2.3.1 Conclusion

The performance of Malmo platform can be speed up by reducing the rendering frame size, as large frame is not necessary for learning system.

2.4 Wed 6 Jun 2018

2.4.1 Getting Started With Python

In order to import module `MalmoPython`, you will need to find the shared library *Python_Examples/MalmoPython.so* under Malmo project and copy it to your Python's site package location(s), which can be checked by running

```
import site;site.getsitepackages()
```

If you are going to use `malmoutils` as well, please also copy the file *Python_Examples/malmoutils.py* to your site package directory.

2.4.2 Windows Support

Malmo seems to be not supporting Windows system very well yet. In fact, it seems that Minecraft mod development on Linux is the main stream. For more information, check *Minecraft Coder Pack*, and Minecraft Forge.

Remark: Malmo project is built upon Minecrft Forge.

2.4.3 Video Recording

First you can check *Research/explore_experiments/001/script01.py*.

I used `VideoProducer` to specify output frames to be the size of 256x256. And use the API `MalmoPython.MissionRecordSpec.recordMP4` to specify we want to record video into MP4 format.

Run *script01.py* with arguments as:

```
python script01.py --record_video --recording_dir=mission_records
```

The first argument `record_video` tells Malmo to record video. The second argument `recording_dir` specify in which directory we store our records. While running mission, Malmo will temporarily store all records including video records under a folder named `mission_records`. After the mission is completed, Malmo will compress all the records and store it to destination specified by API `MalmoPython.MissionRecordSpec.setDestination`. In *script01.py* we set destination to be *mission_records/Mission_1.tgz*.

After a mission is completed, we decompress *mission_records/Mission_1.tgz* and play video record to review the mission.

We discovered that in first person view, the hotbar slots and hands of our agent **did not** get recorded into video. So we may need to allow the agent to observe the game in third person view to better let it understand the world.

2.5 Thu 7 Jun 2018

record video streams as individual frames rather than video - eg `MalmoPython.MissionSpec.recordBitmaps(MalmoPython.FrameType.DEPTH_MAP)`. But this API seems to be not working.

From Malmo/src/MissionRecordSpec.h on Malmo's GitHub page, bitmaps and MP4 cannot both be recorded for a given video producer.

After checking what frame types Malmo defined, I tried the type `VIDEO`, and things run successfully. That is, I used `MalmoPython.MissionSpec.recordBitmaps(MalmoPython.FrameType.VIDEO)`

Frames are recorded in ppm format, which can be processed in a very straightforward way (but this is also a highly inefficient format).

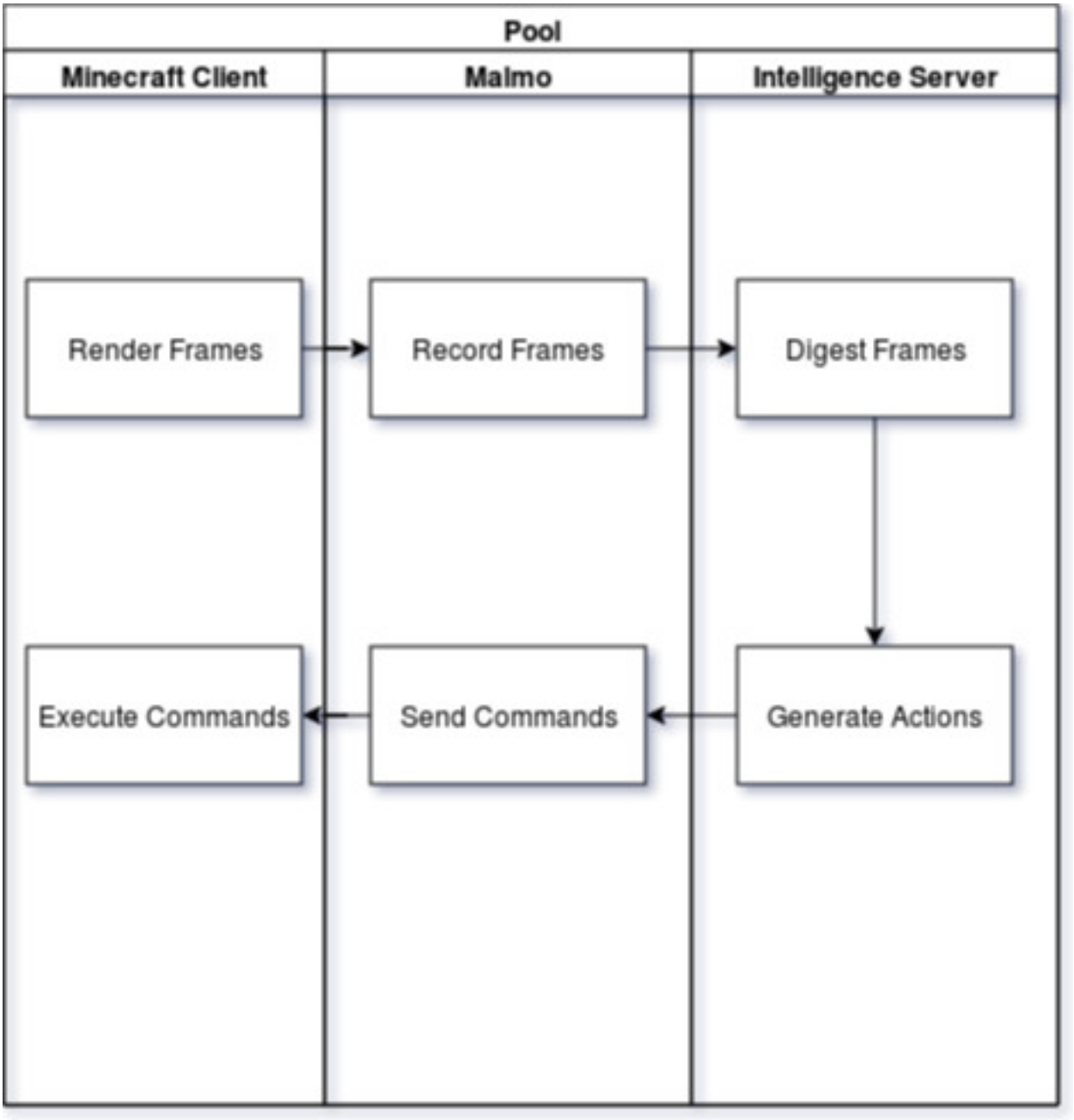
You can find *Research/explore_experiments/001/script02.py* and under that directory, run:

```
python script02.py --record_video --recording_dir=mission_records
```

Then you will find all recordings including frames in *mission_records/* folder.

2.5.1 Intelligence System

I am going to seperate intelligence system, which plays the role of brain for our agent, from Malmo. The working relationship between Minecraft, Malmo and this system can be summarized as in the following flowchart:



2.5.2 Networking Prototype

- Directory: *Research/work_table/*
- Files: *server02.py*, *client02.py*

In order to build the system, we need to facilitate video stream feed from Malmo end to Intelligence system end. We decided to use *socket* to achieve this.

Today I have built a very simple server and client pair, as can be easily found under work table directory. Simply run (with Python 2.7)

```
python server02.py
```

to start up the server and run

```
python client02.py
```

to open a client. Then you should see server consistenly sending "HEY YOU" message to client, as displayed on client side. You can open at most 100 clients simultaneously. The server will simply provide the same service to every client connected to it, since our ultimate goal is to provide one single video stream to these clients.

About Design And Codes

Python simply does not provide a ready-to-use module for our video streaming feed purpose. `SocketServer.TCPServer` seems to fit. But a little research revealed that it is mainly designed for being the base class for `BaseHTTPServer` etc. Real time stream protocol (RTSP) seems not supported. So I need to write my own server and client to enable consistent data feed.

We used 1 thread for listening and accepting incoming connect request. When we accept a client, we create a service thread for it to consistently provide service. Specific service will be implemented in derived class by overriding `MyBaseServer.serve` method.

On the other hand, client is single-threaded. And any derived class of `MyBaseClient` only needs to override method `work` to specify jobs on client side.

CHAPTER 3

Change Logs

Here is a list of change logs, which record all the changes happened in the AgentSteve project.

3.1 Wed May 23 2018

- Initialize project on GitHub.
- Update documentations.
- Add new experiment as *Research/imitate_experiments/001/script02.py*.
- Add new folder *Research/imitate_experiments/model* to store model files for visualization with TensorBoard.

This project is aimed at constructing an agent in the popular video game Minecraft that is capable of learning, exploring and helping, with the power of deep learning!

You can fork this project on GitHub.

CHAPTER 4

Introduction

Minecraft is a 2011 sandbox video game created by Markus Persson. It has no specific goals to accomplish. Players can build houses, develop agriculture and raise livestock. They can also create armors and swords, brew magic portions to fight with mummies, dragons, and other hostile creatures. Villages, desert palaces, underseas ancient cities, and other amazing constructions are waiting for players to discover. Please visit the game's website for more information!

Nowadays, deep learning can achieve pretty astonishing goal, such as beating human players in extremely intelligent chess game Go, see AlphaGo, created by team DeepMind. But making machine to play a game like Minecraft is still an active research area. It requires machine to have intrinsic interests to its surroundings, desires for new knowledge and the ability to require them, just like our human do, instead of simply fulfilling a given goal.

In this project, we intend to (at least partly) achieve the goal to create an agent that is smart enough to be capable of:

- **learning from others**, including human players and other agents. For example, learn from demonstration and try to replicate the result, or in other words, the ability of imitating.
- **fulfiling tasks**, including those assigned by human players, other agents and itself. It knows how to plan, how to divide the big and abstract goal into smaller pieces, and conquer them one by one to accomplish the ultimate goal.
- **exploring**. Agent should be curious about its surrounding world and have intrinsic desire for knowledge. It likes novel things and hates boring life.
- **social networking**. The agent is encouraged by others' recognition and accompany. It is desired for friendship and always willing to help. It should be able to communicate with others about its own motives and intensions, especially to human player. It should also understand others' meaning and reasoning.
- **memorizing**. It has an experience database. All past experiences and knowledge will be efficiently stored and can be retrieved fastly. Unimportant or too ancient memories will be dumped. Important and recent memories will be highlighted.
- **predicting**. Agent should be able to predict the near and far futures. Predicing power will be heavily based on past experiences (experience database). Having a vision even for the near future and knowing the consequences of its own actions would be very beneficial for agent to achieve its ultimate goal.
- **transferring knowledge**. Knowledge obtained by one system should be able to be utilized by other system (may be through experience database). For example, skills learned by imitating, or knowledge gained through exploring should be able to be applied to better fulfilling tasks later.

CHAPTER 5

Getting Started

This project uses Malmo project by Microsoft to provide the Minecraft game environment. Malmo is a platform for Artificial Intelligence experimentation and research built on top of Minecraft. To install and run Malmo platform, see docs on its GitHub page. Personally, I recommend using Ubuntu 16.04 64 bit system, with Python 2.7.

We will be using TensorFlow as the main tool for researching deep learning capability. TensorFlow is an open source machine learning framework developed by Google. The latest version is 1.8, and we will start experimenting on this version. Please visit their site for installation and more information.

I am currently working on the first module, i.e. the imitation system. Major inspiration came from this paper: *Third-Person Imitation Learning*, by Bradly Stadie, Pieter Abbel and Ilya Sutskever. You can access this paper on arXiv on this link.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search