∇ giza: A Recipe for AI Languages

Arash Abadpour - arash@abadpour.com

November 5, 2023

Abstract

In this paper, we discuss the Mathematics of building Machine Vision AI systems in Linux. We review the general challenge of translating the description of an AI operation in human language into a human-readable, machine-executable script. We select multiple Machine Vision AI challenges that we first describe in human language. Then, in each case, we build the language to convert the description in human language into one or more scripts we execute on machines. We use AWS SageMaker [1] for development and training and AWS Batch [2] for inference and discuss API calls. The main contribution of this paper is a mathematical framework for building an AI language for a practical use-case in Machine Vision. We hope that researchers in other fields of AI use and extend this framework in their disciplines. We present a reference implementation [3] of this framework and multiple use-cases [4] - *revision-1.43.1*

Contents

[1] `https://aws.amazon.com/sagemaker/`
[2] `https://aws.amazon.com/batch/`
[3] `https://github.com/kamangir/awesome-bash-cli`, *awesome-bash-cli*, *abcli*.
[4] bird watching in downtown Vancouver with AI, `https://github.com/kamangir/Vancouver-Watching`, *Vancouver-Watching*, *vanwatch*.

background

Almost five years ago, on Thursday, November 8, 2018, I acquired a Raspberry Pi [5] on Amazon. Since then, my personal and professional lives have focused on Linux. Professionally, I do AI and, more recently, geospatial AI. In my personal life, I mix AI, cloud, and mathematics into minimal forms that seek survival [6]. Over the years, I have built a set of mechanisms for building AI systems that I will document in this paper. Therefore, this is an attempt to produce formal mathematical definitions for the AI mechanisms that I will collectively refer to as *giza*. I seek to understand these mechanisms through this effort better so that I can use them more optimally and also along new dimensions.

[5] `https://www.raspberrypi.org/`
[6] `https://github.com/kamangir`

1 Problem Definition

wip

2 Examples

2.1 OpenAI code generation

Experiments w/ the OpenAI API [7]. wip

2.2 Vancouver-Watching (vanwatch)

vanwatch is a callable that performs the following actions,

1. terraforms the machine 5.2.1.
2. discovers the cameras in an area 2.2.1.
3. ingests images from the cameras in an area and detects the objects in them 2.2.2.
4. analyzes the objects detected in an area 2.2.3.

2.2.1 vanwatch discover

```
vancouver_watching discover \
    [area=vancouver|iran,~upload] \
    [-|<object-name>] \
    [<args>]
```

Creates a list of the cameras available in *area* in ⟨object-name⟩ 3.8 and tags 5.2.7 this object for use by ingest 2.2.2.

2.2.2 vanwatch ingest

```
vancouver_watching ingest \
    [area=vancouver|iran,count=<-1>,detect,dryrun,model=<model-id>,~upload] \
    [-|<object-name>] \
    <args>
```

Finds the latest object discovered 2.2.1 from *area* and ingests *count* images into ⟨object-name⟩ and detects the objects in them.

2.2.3 vanwatch analyze

```
vancouver_watching analyze \
    [~download,~upload] \
    [.|<object-name>] \
    [<args>]
```

Additional analysis on ⟨object-name⟩, including generating a heat map of the objects in the area.

[7] https://github.com/kamangir/openai

3 Concepts

3.1 Machines and Shells

A *machine* is a state machine that is connected to many other machines and shares some of its state with them for read and write. A *shell* is a stateful access mechanism to a machine that an *operator* 3.2 uses to run *commands* 3.3. Running a command in a shell can modify the state of the shell, the machine on which the shell is running, and potentially the states of all other machines. Two examples of machines are a Raspberry Pi that runs Linux and is connected to AWS [8] and a docker container running in AWS Batch. GNU Bash [9] is an example of a shell.

3.2 Operators

The *operator* generates commands and runs them on different shells on different machines. The operator attempts to maximize an objective function that depends on the state of multiple machines.

3.3 Commands

A *command* is any Bash command [10] and can be represented in a Python string of characters [11]. Here is an example command,

```
vancouver_watching ingest \
    vancouver \
    dryrun \
    . \
    --count 12
```

The above command and the one below are *identical*.

```
vancouver_watching discover vancouver ~upload --validate 1
```

Two commands are identical if running them on two machines in identical states yields the same states. In theory, the state of any machine depends on the state of any other machine, and it is almost impossible to run two commands in identical states, including the time of execution. Therefore, when we refer to two identical commands, we either use a derivation-based proof of identity or consider a validation in a limited "relevant" subset of the state representation.

For any shell on any machine, at known states, there is a mapping between the set of all commands and $\{True, False\}$ that we address as "whether the command is found". In Bash, for example, the following message is printed when a command is not found.

```
-bash: void: command not found
```

Note that writing to the standard streams *stdin* and *stdout* are examples of state changes in the shell and the machine. *Terraforming* is the process of running commands that modify the state of the shell and the machine in ways that make additional commands found. Terraforming is also intended to modify the state change caused by a set of commands favourable to the interest of an operator. For convenience, we address a command that is found as a *valid* command and *invalid* otherwise. Terraforming may also ensure the machine's state after a reboot or when a shell starts. Terraforming generally includes a modification of *bashrc*, *bash_profile*, *screenrc*, and *desktop* files.

Commands can be similar when considered as strings of characters. Here is a command that is similar to the above,

```
vancouver_watching ingest toronto upload . --count 3
```

[8] https://aws.amazon.com/

[9] https://www.gnu.org/software/bash/, *Bash*, *bash*.

[10] https://www.gnu.org/software/bash/manual/bash.html#Shell-Syntax

[11] https://docs.python.org/3/library/string.html

3.4 Command Templates

A *command template* is a representation that yields similar commands, given the following three rules. First, ⟨description⟩ can be replaced with any string of characters that can be described as "description". See *options* 3.6, *arguments* 3.7, *objects* 3.8 for the next rules. Here is a command template for the two above,

```
vancouver_watching ingest \
    <area> \
    [dryrun,~upload] \
    [<object-name>] \
    [--count <-1>]
```

3.5 Callables

A *callable* is a valid command with no space and control operators [12]. The list of callables depends on the machine's state and is generally extended through terraforming. Some of the well-known callables are *git* [13], *docker* [14], *pushd* [15], *nano* [16].

Theorem 1 *For any callable ⟨callable⟩, and any string ⟨suffix⟩, "⟨callable⟩ ⟨suffix⟩" is a valid command.*

Theorem 2 *For any valid command ⟨command⟩, and any string ⟨suffix⟩, "⟨command⟩ ⟨suffix⟩" is a valid command.*

3.6 Options

A string representation of a dictionary, such as `<keyword>=<value>,<keyword>,-<keyword>,<...>`.

3.7 Arguments

Any list of arguments that is consumable by a Linux tool. An example is a series of `--<arg> <value>`.

3.8 Objects

Objects are accessible on every machine by their *name* and an object may be *selected* 5.2.6. Objects may be uploaded 5.2.8 and downloaded 5.2.2, fully or partially, or listed **??**. Objects have metadata **??**.

Commands 3.3 consume and generate objects. If the object name is not provided or is provided as `-`, then an object with a unique name is created and used. Otherwise, the object is either provided by name, as in ⟨object-name⟩, or by reference, as in one of ".", ".." or "...".

[12] https://www.gnu.org/software/bash/manual/bash.html#index-control-operator
[13] https://git-scm.com/docs/git
[14] https://docs.docker.com/engine/reference/commandline/cli/
[15] https://www.gnu.org/software/bash/manual/bash.html#Directory-Stack-Builtins
[16] https://www.nano-editor.org/

4 Mathematical Model

operations: pre and post-fixing, batch multi-dimensional, deep injection.

5 TL;DR

This section provides a set of practical guidelines for building a Machine Vision AI language based on a hybrid of Python [17] and Bash.

5.1 Before you Start

Code is maintained in one or more repository systems. Here we use *git* 5.2.3. The core 5.2 and each individual plugin 5.3 are maintained in separate repositories.

5.2 Build the Core

The *core* is a callable that loads the plugins and terraforms the machine.

5.2.1 @conda

5.2.2 @download

```
@download object <object-name> [filename=<filename>,open]
```

5.2.3 @git

5.2.4 @help

When the command ends with `help` a help message should print.

5.2.5 @relations

Two objects can be related in many ways.

```
@relations get <object-name-1> <object-name-2>
@relations get <object-name-1> <object-name-2> <relation>
@relations get <object-name-1>
@relations get <object-name-1> <relation>

@relations search <relation>

@relations set <object-name-1> <object-name-2> <relation>
```

5.2.6 @select

```
@select object <object-name>
@select <type> <type-name>
```

When an object is selected, '.' represents ⟨¡object-name¿⟩. Similarly, '..', '...', and so on, as deep as needed, refer to the previously selected object and the one before that.

Commands default the objects they consume and modify to '.', '..', and so on. Because the commands in a script use the same objects, selecting the objects enables their names to be omitted in a script.

An object may have a 'type', for example, 'model' or 'dataset'. Commands that consume objects specify a 'type' for the argument. This enables the user to simultaneously select different types of objects and run commands on them. Here is an example from *hubble* [18], wherein the user selects an object and then a hubble dataset and an object in that dataset to download in the object.

```
abcli select; open .
hubble select dataset hst
hubble select object public/u4ge/u4ge0106r
hubble download ~dryrun
```

[17] https://www.python.org/
[18] https://github.com/kamangir/hubble

5.2.7 @tag

An object can have many tags. A tag is a boolean or valued property of the object.

```
@tags get <object-name>
@tags get <object-name> <keyword>

@tags search <keyword=value,~that,this>

@tags set <object-name> <keyword=value,~that,this>
```

5.2.8 @upload

```
@upload object <object-name> [filename=<filename>]
```

5.2.9 Scripts

The script is a bash script [19].

5.3 Establish a Plugin Mechanism

A plugin generally defines one or more callables.

[19] https://en.wikipedia.org/wiki/Shell_script