∇ giza: A Recipe for AI Languages

Arash Abadpour - arash@abadpour.com

November 4, 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-1.31.1*

Contents

1 Problem Definition **2**

2 Examples **3**
- 2.1 OpenAI code generation . . . 3
- 2.2 Vancouver-Watching . . . 3

3 Concepts **4**
- 3.1 Machines, Shells, and Terraforming . . . 4
- 3.2 Commands . . . 4
  - 3.2.1 Callables . . . 5

4 Mathematical Model **6**

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.

[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*.

[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

wip

In one case, we use Vancouver Watching as an example AI problem and discuss integrating it with We discuss API access to run YOLO [?] object detection models on the stream of images captured by traffic cameras in Downtown Vancouver.

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

3 Concepts

3.1 Machines, Shells, and Terraforming

A *machine* is a state machine that is connected to many other machines and shares its state with them. A *shells* is a stateful access mechanism to a machine to run *commands* 3.2. 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. *Terraforming* is the process of running commands that modify the state of the shell and the machine.

A command may be *invalid*, in which case running it results in a special state change. In bash, for example, the following message is printed,

```
-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 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 Commands

A *command* is a Bash shell command [10] that can be carried in a Python string of characters [11]. Here is an example command,

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

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 practice, 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" state.

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

```
vancouver_watching discover toronto upload --validate 0
```

A *command template* is a representation that yields similar commands. Here is a command template for the above,

```
vancouver_watching discover \
    <area> \
    [~upload] \
    [--validate 1]
```

wip

In a *command* 3.2, the notation ⟨descriptive-name⟩ represents a *placeholder* for a string of characters that is described as *descriptive-name*. See 3.2 for examples.

We can discuss the *validity* of a command in the context of a *terraform* **??**. wip

wip

to alter the validity and outcome of future commands.

Commands generally starts with a *callable* 3.2.1 and is followed by a hierarchical sequence

pieces of this command wip

command may be valid or invalid in the context of a terraform wip

[8] https://aws.amazon.com/
[9] https://www.gnu.org/software/bash/
[10] https://www.gnu.org/software/bash/manual/bash.html#Shell-Syntax
[11] https://docs.python.org/3/library/string.html

3.2.1 Callables

A *callable* is a valid *command* in a terraform **??** that accepts *arguments*, i.e. the following is a valid command, for any ⟨string⟩.

```
callable <string>
```

The *core* **??** is a callable. *plugins* **??** generally define one or more callables.

4 Mathematical Model

References