∇ 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-1.33.1*

Contents

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 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 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.

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
```

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* 3.4, *arguments* 3.5, *objects* 3.6 for the next rules. Here is a command template for the two above,

[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

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

3.4 Options

3.5 Arguments

3.6 objects

4 Mathematical Model

References