Composite robot learning, and teaching Terry to use a computer

Owen Gallagher
7 May 2020
CS Research

INDEX

I. ABSTRACT

I set out to create a personal assistant program called Terry, both to investigate the integration of multiple robot learning concepts into one system, and to familiarize myself with the technologies involved in implementing such a system. The goal for robot learning approaches was to incorporate key concepts from instruction learning, demonstration learning, and reinforcement learning into the program. The main novelty of Terry is the addition of a different communication channel: the operating system's graphical user interface.

II. BACKGROUND

A. The Current Paradigm

Today's state-of-the-art virtual assistants are well known, unlike most of the codes and acronyms of the software development world, by simple names. Siri (Apple), Alexa (Amazon), and Cortana (Microsoft) are a few such examples. Having no problem with this tradition, I decided to name this one Terry[1]. However, apart from the naming convention I hope to depart from the constrictive, essentially command-line interface (only superficially enhanced by speech-text and text-speech conversion) that popular virtual assistants use. Of course, there are doubtless advantages to this simplication of the robot-user interface; it's typically used on mobile devices as a hands-free alternative to the smartphone's touch GUI, or otherwise takes up little screen real estate to be less intrusive in the user's workflow, and resembles text messaging, a mode of interaction that most users are already comfortable using.

[1] I was between multiple names: Terry (the virtual secreTERRY), Morgan (ready to MORGANize), Neo (lord of *The Matrix*), and Otto (the OTTOmater), to name a few personal favorites.

But there are also some clear and compelling advantages to using a GUI for the assistant. A virtual assistant generally accomplishes tasks that are already doable through the operating system's GUI. Instead of clicking the messaging app shortcut, typing the recipient's name as "Ava Adams", clicking the composer bar, typing the message body, and clicking the send button, the user can instead open the virtual assistant and say, "Tell Ava Adams that I'm going to pick up the card on my way to the station". However, in certain scenarios the user may think of a task that the assistant does not know how to do, like "draw a hand waving with six fingers and send it to Brandon Bishop", or "find a picture of a green egg and set it as my background". In this case, it would be unreasonable for the original programming team to predict that a user might give such an instruction, but not unreasonable for the user to want to be able to do it. Here is where robot learning enters the arena, namely learning via instruction, demonstration, and reinforcement.

B. Composite Robot Learning

There are a number of papers that educate on these methods in robot learning. Researchers Thomaz and Breazeal found that in reinforcement learning, the concept of gaze is very important in facilitating teaching (Thomaz and Breazeal 2008). Terry achieves a viable way for the user/teacher to direct gaze via the cursor. They also found that people adapt their teaching strategy as they develop a mental model for how the robot learns. The OS essentially provides a way for the user to share a familiar perspective with Terry, facilitating the user's understanding of how it learns. Lastly, they point out that reinforcement from the user can be classified into two groups: feedback/reward/penalty and guidance. The OS in tandem with a speech interface works in favor of this also, in that the user can offer guidance with the

mouse, feedback with speech, and properly time both modes of reinforcement by watching Terry progress through the action.

The paper by Argall et al. pointed out challenges with embodiment mapping (between the demonstration and the robot action) and record mapping (between the demonstrator and the demonstration) prevalent in demonstration learning (Argall et al. 2008). Terry avoids to a large extent having to create these mappings because the user/teacher can demonstrate using the same peripherals (keyboard and cursor) that Terry will use to carry out the instructions, acting on the same set of widgets as seen through the same screen.

Lauria et al. implemented a system for instruction learning, applied to a scenario where a user teaches a robot how to navigate a map resembling a suburban neighborhood (Lauria et al. 2002). Here the challenge was to take complex natural language instructions and break them down until arriving at their component instruction primitives, which are a set of initially known actions. I hope to use a very similar system for Terry, where instructions are given through speech. A key improvement that Terry will hopefully contribute to this system is the ability to learn to recognize new widgets, analogous to landmarks in the case of the paper, in addition to recognizing new primitive combinations.

IV. OBJECTIVES

The personal objective of this research project was to explore methods of learning in robots. Initially, I planned to explore robot learning either by comparing various learning methods (demonstration, instruction, reinforcement), or by combining features of multiple and focusing more on the use case. I decided to do the latter,

which has added additional value to this research by providing insight into the technologies required to implement the different components involved.

In terms of general objectives, completion of even a minimally functioning Terry prototype would mean that the core innovation behind it (using the OS GUI interface and incorporating learning techniques) has a useable proof of concept. Such a prototype is particularly feasible with this project because it involves many aspects that can be broken into multiple levels of completion, each level representing a subsequent prototype. For example, the user interface portion will at the fully complete stage incorporate multiple communication categories for: supervision, feedback, guidance, lessons/teaching, commands, queries, and logging. Each of these can be incrementally layered into the interface over time, and each in turn can also be broken into stages.

So, to extend on the UI example, such a “minimally functioning” prototype could rely solely on text-based communication of commands and logging messages, use the demonstration-based learning approach, execute exactly-worded commands from the user, and demonstrate basic control of the mouse and keyboard. The minimal prototype would connect all the modules together, as described further in **IV.B.Requirements**, and a fuller release would include full speech-to-text and text-to-speech support, handle all types of input and output, and all components related to effective learning: instruction derivation, widget definition, demonstration learning, and reinforcement learning.

A. Schedule

My projected schedule was, of course, fairly optimistic. For more details of the categorized task breakdown (data structures, driver, language processor, demonstration processor, output processor, image processor), see **V.D.Documentation, Files, and Directories** as to where to find the full proposed schedule. Below is a list of important milestones, and my projected dates of completion for each.

1. **8 Jan 2020** - A rudimentary driver can demonstrate a series of simple actions (mouse moves, clicks, keyboard typing), but those actions are not associated with actions that can be invoked by instruction. There is also a logger and console window to output state and debugging information.
2. **11 Jan 2020** - The speaker module can say aloud some of the information from the console, and the scribe module can create text transcriptions from spoken instructions.
3. **18 Jan 2020** - The instruction parser can reference existing language mappings[2] (actions, widgets, and lessons in memory), and the prompter can create dialogue windows for the user to interact with if needed. Terry can now also do optical character recognition to find text in an image.
4. **25 Jan 2020** - Terry now has a set of primitive actions that can be invoked by natural language instructions, the prompter can accept screenshots from the user via dialogue window, and the project includes an image recognition library.

[2] The term "language mapping" in the context of Terry refers to any object that has an associated language pattern to associate it with, meaning it can be referenced by the user's natural language. The instruction "click the start button" contains mappings to action "click" and widget "start button", for example.

5. **1 Mar 2020** - Widgets can be learned, added to memory, saved and loaded from files in persistent storage, referenced in instructions, and found in the screen using their text labels.
6. **17 Mar 2020** - Sequential composite actions can be learned with instruction learning, and widgets can be learned by type, label, and appearance definition. Also, states can be mapped to peripheral (mouse and keyboard) manipulations, which is a precursor to demonstration learning.
7. **27 March 2020** - Actions with arguments (functional actions) and actions with multiple language patterns can be learned. Actions can also be learned via demonstration. Conditions[3] are now supported at an elementary level, but cannot be learned or invoked.
8. **10 April 2020** - With full conditional action support, Terry is complete.

B. Requirements

Earlier I mentioned a minimally functioning prototype, which outlines my requirements for what Terry should have been able to do by the end of the Spring 2020 semester. In summary, I hoped that it would at least: respond to text-based communication of instructions, print logging messages to a console window, use the demonstration-based learning approach to define new sequential actions, execute exactly-worded actions from the user, and demonstrate basic control of the mouse and keyboard.

A typical virtual assistant would support verbal communication to make the interaction seem more user-friendly, but a prototype would still be able to respond to

[3] A conditional action is a more complex action that allows for control structures (selection, looping, and potentially event handling), discussed further in **V.A.Implementation**.

instructions as long as their typed in a terminal, by carrying out actions and replyaing via text in a console window.

Sequential actions are the simplest form of action I had planned for Terry to support, where composite action **ABCD** consists of a linear sequence of subactions: do **A**, then do **B**, then do **C**, and finally do **D**. Demonstration of this should be the simplest form of learning to implement, as it would just be a matter of recording mouse and keyboard manipulations, matching them up to primitive actions, and associating them with a new composite action name.

When referencing these actions, a more complex assistant would be able to recognize the desired action despite variations in the natural language instruction that contains it. However, a minimal prototype would require that “move mouse to <widget>” be said exactly that way, and not be able to recognize “go to <widget>” or “<move the cursor to <widget>”.

Lastly, among possible primitive actions that Terry could execute, I aimed for it to at least be able to type a character sequence, move the mouse to a given location, and click a mouse button.

V. RESULTS

I’m fairly satisfied with the Terry prototype that has been implemented as of 14 April. Terry runs on both Windows and Mac computers and can, with adecuate permission from the operating system, access the microphone to record an instruction from the user, generate the transcription of that instruction with Google Speech, and pass that transcription to the instruction parser to look for the best matching action or lesson. If it does not recognize any action or lesson in the instruction, Terry logs this in the

log file and in the console window, and can also say so aloud. If the action contains unknown widget names, it asks for a widget lesson to learn them. If the action exists, Terry can then execute it, controlling the mouse and keyboard, taking a screen capture, logging current state variables, running a preset demo, finding a widget on the screen by label or by appearance[4], and shutting down. If the instruction is instead a lesson, if its a widget lesson Terry can then create a new widget with an assigned interaction type, text label, and appearance. Terry also has a watcher component that handles system keyboard and mouse events, which can be used to control Terry even when it doesn't have focus (for example, open Terry's intercom for a new instruction with the global key combination: **CTRL+SHIFT+T**). If the instruction is an action demonstration lesson, the watcher can also be used to create a recording of mouse and keyboard manipulations to be assigned to a new action. This is as far as Terry's development has gotten, because the watcher recordings are not complete enough to be compiled into new actions, as will be explained in more detail in the next section.

However, comparing this list of features to those outlined for the minimally functioning prototype, the outcome of development has been positive. In all respects Terry's capabilities go beyond minimum, except for the watcher input recordings.

A. Implementation

One of the major differences that emerged between the plan and the follow-through was the program's architecture. The original architecture for Terry had a highly modular approach inspired by the classical Input-Process-Output programming paradigm. There were five major modules: the language processor, which encompassed transcription, instruction parsing, and mapping to actions, lessons, and

[4] The Widget.Appearance class is what I use to store widget graphics. This can be the widget's icon or a group of icons in whose regional context the widget is usually found.

widgets; the driver, which compiled actions and executed state transitions to control the mouse, keyboard, and Terry itself; the demonstration processor, which handled peripherals recordings and demonstration learning; the image processor, which could find widgets in a screen capture using character and image recognition; and the output processor, which created windows, logged to the console, and spoke using text-to-speech libraries.

The final architecture had to adapt to the Object-Oriented paradigm that worked much more naturally with Java, chosen for its cross-platform support. The following list describes how different data and processes are assigned to each major class in the project. In general, the major classes that were somewhat analagous to the modules from the original architecture have many static properties and methods since Terry only ever uses one instance of each.

`Terry`: This is the application class with the **`main(String[])`** method runnable by the JVM. It initializes all the other components of the program, including **`Memory`**. If there are no language mappings found in the filesystem, this class generates the necessary action primitives and lessons that Terry starts with before customization via learning takes place.

`Prompter`: This class extends the JavaFX **`Application`** class that runs the main GUI thread, and handles all of Terry's windows: the intercom, the console, the overlay, and dialogues. The intercom is analagous to Terry's face that the user interacts with. Clicking on the intercom window triggers the **`Scribe`** to start and stop recording with the microphone. The console is where the **`Logger`** outputs all text information viewable by the user during execution. The overlay is a semitransparent window that displays over the whole computer screen and can highlight zones in the GUI, or act as

a pane so that the user can select a rectangular region to be screen-captured for defining a **`Widget`** appearance.

`Logger`: All error messages, status changes, debugging messages, and spoken replies are passed through this class, which then outputs messages two three destinations: the current log file to be saved in the filesystem, Terry's console window, and the **`Speaker`**. The routing of messages to these destinations is determined by the message's logging level. The lowest level only gets written to the log file, and the highest gets spoken aloud, printed in the console window, and also added to the log file.

`Speaker`: This class accomplishes speech by using external programs, and invoking them with system commands. On Windows, the command runs `res/speech/voice.exe`, while on Mac the command runs `/usr/bin/say`. In both operating system environments this class has the ability to change the voice, speed, and volume of the audio. However, I did not get to add any primitive actions to expose these options to the user.

`Scribe`: When the user clicks the intercom window, this class is triggered to start recording using the computer's microphone, and to stop recording when the user is finished saying the next instruction. It then sends the transcript to the Google Speech API endpoint for speech-to-text services, and passes the resulting transcription to the **`InstructionParser`**.

`Driver`: Most actions will invoke this class, which contains a JavaFX **`Robot`** object within it. The robot is able to control the mouse and keyboard, and take screen captures. Currently, Terry needs to be given permission by the OS to enable the

robot, but in the future I had hoped to have Terry properly ask for these permissions on starting.

`InstructionParser`: This class takes care of associating a natural language instruction received from the **`Scribe`** with a **`LanguageMapping`** from **`Memory`**. This is done by using an intermediate class called **`InstructionPossibilities`**. Once the proper language mapping is determined (either an action or a lesson), it can be passed to the **`Compiler`** for execution.

`InstructionPossibilities`: Here is where most natural language processing (NLP) takes place. Most contemporary software that does NLP will use neural networks for instruction classification, but as will be explained, such an approach would not have worked so well for Terry. This class contains a list of objects of class **`InstructionPossibility`**, each of which is associated with one language mapping. As tokens are passed in from the **`InstructionParser`**, each instruction possibility attempts to narrow down which version of the candidate language mapping could be referred to by the instruction. If the mapping's pattern expression is...

```
move ?|mouse,cursor,)) to @#x comma @#y
```

... and the tokens processed so far are...

```
move
```

... then the instruction possibility contains three versions of the language mapping that could be resolved depending on the next token:

```
move
  |-----------to---@#x---comma---@#y
  |---mouse---to---@#x---comma---@#y
```

```
|---cursor--to---@#x---comma---@#y
```

The tokens labeled **@#x** and **@#y** are important, as they are named placeholders for arguments[5]. Both **Action**s and **Lesson**s can contain arguments. Depending on the resolution of the instruction possibility, as can be seen in this example, the expected position of these arguments will change, which explains why simply classifying which language mapping the instruction refers to is not enough. The instruction possibility also handles execution of the resolved language mapping, and takes care of passing on the values of these arguments. The setting and validation of arguments uses the **Arg** class. If the language mapping is an action, the action's **execute** method is invoked. If it's a lesson, then the **learn** method is invoked. If any of the arguments should be widgets based on context but do not exist in **Memory**, then the **Prompter** asks the user to teach that unknown widget. If at the end of the instruction **InstructionPossibilities** no longer contains any possible matching language mappings, Terry assumes it refers to an action it doesn't know how to do yet, and the prompter requests for the user to teach it.

Compiler: This class doesn't currently do much. It has a queue of instruction possibilities that need to be executed, and does a minimal check to make sure there are no null objects in the queue. It can then be asked to compile, which executes each item in the execution queue in sequence. If Terry were to ever support conditional actions with control structures, I had planned on probably implementing them by expanding on this class, and changing the execution queue to a more complex data structure.

[5] @#x stands for a numeric (#) argument (@) called "x". See the **Arg** class for the different argument type characters. Some others are $ for strings and w for widgets, for example.

`Watcher`: This class creates native mouse and keyboard event hooks with the JNativeHook library. This library had some major issues for me to be able to use it effectively, explained in more detail in **V.B.Challenges and Solutions**. It exposes methods to create **`WatcherRecording`**s, which are then used as demonstrations for demonstration learning. Terry never reached the level of learning a functional action, but it does have most of the features implemented for learning sequential actions. Each element in the demonstration is first classified as either **`Keyboard`** or **`Mouse`**. Keyboards are converted to states that execute **`Driver.type()`** and mice are converted to states that execute **`Driver.point()`**, **`Driver.drag()`**, or **`Driver.click()`**. More specifically, the mouse movement states are also categorized into having either location destinations, or widget destinations. These states are then assigned to a new action and added to **`Memory`**.

`Memory`: The data structures that have to do with learning are stored in this class. It has a dictionary hashmap, a trivials list, a hashmap of language mappings, and a shortcut list to widgets. The dictionary aids the **`InstructionParser`** in determining which language mappings will be possible candidates given the first tokens encountered in the instruction transcription, by associating each mapping with all possible leader tokens[6]. The trivials list contains words that are unimportant, which can be skipped to make NLP faster. The language mappings structure hashes by a unique ID number, though this lookup method is not actually used anywhere in the program. The widgets list contains references to a subset of the language mappings hashmap, and helps **`Watcher.Mouse`** when it needs to determine if its destination contains a widget or not before being used in a demonstrated action. The memory

[6] A leader token in a language mapping is the first one that appears in the pattern expression. For example, for the expression `?drink) @$fruit juice` the leader tokens are “drink” (appears optionally) and a string argument.

class also takes care of saving all these data structures to files and reading them from files, and manipulating those structures.

`LanguageMapping`: This is the superclass that **`Action`**, **`Lesson`**, and **`Widget`** all extend. This is the class that makes it possible to reference all of these subclasses with natural language, using **`LanguagePattern`**. It also has a static variable **`count`** which is used for assigning IDs to new language mappings, and members referring to the subclass (action, lesson widget) and ID of that mapping instance. In most contexts it can be treated as if it were an abstract class.

`Action`: This class encapsulates anything that Terry do and makes it available for the user to invoke. Each instance of an action contains a list of states, which are each changed by calling **`State.transition()`** when the action is executed.

`State`: This class has information about the post-conditions of executions. For example, when the **`Driver`** moves the mouse, the associated state is updated to reflect the new resulting mouse location. When something is typed, the associated state stores the string that was typed, and when a screenshot is taken a state stores the image data. **`Terry`** has a hashmap of all the states, hashed by the state's name. Each instance of this class has a name string, a value, a value type, set of argument names and an **`Execution`**, which maps the argument values to the correct variables, executes, and updates the state's value.

`Lesson`: Each instance has a **`Definition`**, which maps arguments to variables used to define new widgets and actions. Most of the details of the creation of new language mappings are implemented in **`Terry`**, where lesson instances are defined.

`Widget`: This class represents anything that can be found on screen. It contains members to store the interaction type (label, graphic, button, etc), the text label for finding the widget with optical character recognition (OCR), the **`Appearance`** for finding the widget with template matching, its size, and then some variables to store search results. The class also has methods for finding itself in a screen capture.

`Arg`: This class is used to handle language mappings with arguments. An argument has a name, a value, and text, the last being the natural language that was used to set the argument. It also has some static constants for defining valid argument types (string, number, widget, widget type, color, speed, direction) and valid argument values for each type. Finally, it has methods to update and read from its instance variables, making sure that its value remains a valid match for its type.

`Utilities`: It seems every object oriented software project will inevitably contain a utilities class for all the properties and methods that don't have clear owners. This class is a collection of miscellaneous methods that other classes use. It can convert between JNativeHook system events and JavaFX events, convert between JavaFX key events and character codes and handle control keys for the **`Driver.type()`** method, calculate the edit distance between two strings, and save a screen capture to the file system.

B. Challenges and Solutions

As with any project of this scale, there were plenty of unforseen challenges that had to be dealt with along the way. Below are some of the more important roadblocks, setbacks, and changes of plans.

Coronavirus

No amount of planning beforehand could have prepared me for a possible epidemic crisis to occur, extend Spring Break, and cancel campus classes for the rest of the semester. Luckily, the nature of Terry as a research project made it fairly easy to continue development fairly quickly after returning home; I didn't need any equipment from campus and I didn't have to coordinate with any partners or research participants. In my apartment I also had access to both my Mac laptop and a Windows desktop, so I could continue making Terry compatible for both platforms. Finally, I began development before the semester began, around 20 December, by implementing the intercom window and the first version of the scribe for natural language transcription. This early start gave me approximately one extra week, which compensated nicely for the week lost due to the coronavirus outbreak.

DeepSpeech Transcription Inaccuracy

The early plan for spoken natural language support with Terry was to find a speech-to-text library that didn't need an internet connection so that the user wouldn't need to have an internet connection and wait for transcription requests to send to the server and return with the response. DeepSpeech by Mozilla seemed a good solution that fit my criteria. However, I soon discovered after adding it to Terry that the generated transcriptions were very error-prone. Every few words would be incorrectly transcribed, making the the spoken instructions virtually impossible to match to language mappings.

My next solution to account for mistakes in transcription was to incorporate edit distance using the Wagner-Fischer algorithm[7] to allow approximate matches. Eventually I got this to work too, by marking dictionary entries as matches to the current instruction token during parsing if the edit distance was under a certain threshold proportional to the length of the token. However, even this proved not accurate enough to be useful, and also became problematic when trying to resolve arguments.

I eventually moved to try using Google's Speech API, which had a speech-to-text endpoint and had a reputation for high accuracy. It was a difficult change to make since I'd put so much effort into trying a purely local transcriber, but once I enabled transcription through Google Speech, Terry's transcription accuracy improved significantly.

Permissions for Microphone and Robot

When Terry first runs in either a Windows or Mac environment the operating system will alert the user when the scribe attempts to record a spoken instruction and when the driver attempts to use the robot instance to control the keyboard or mouse or take a screen capture. The user then has to nagivate to their computer's system settings to give Terry these permissions, but if that is not done then the OS will not prompt again, and instead just ignore the attempts by Terry to access these features. For a few days I attempted to incorporate a way for Terry to request these

[7] This version of edit distance is one of the more basic, considering deletion, substitution, and insertion as the possible operations to move between words. I pulled the algorithm from the Wikipedia specification: https://en.wikipedia.org/wiki/Wagner–Fischer_algorithm, and added the method to the Utilities class.

permissions from within the program, but I gave that up to focus my efforts elsewhere.

Java Version Migration

From the beginning of development until about mid-January I wrote source code for Terry exclusively in a Mac environment, as that was the OS of my personal computer and was the most convenient option. After that I began working on a Windows machine in parallel to make Terry cross-compatible, and immediately ran into issues between Java versions. The best solution ended up being to use a newer version of Java for both working environments, which did have some ramifications that required edits to the source code. Terry was first written for JavaSE 1.8, but was moved to JavaSE 10. The major hurdle then was the revelation that JavaFX 13[8], the newer version of JavaFX compatible with JavaSE 10, was no longer included in the JDK, and would need to be included as an external dependency. In the end, this introduced me to the Maven dependency manager, which greatly improved Terry's file organization and maintainability by making external libraries XML references to be downloaded automatically when needed.

JNativeHook Issues

The JNativeHook[9] library is what I used to implement the Watcher class' peripherals recording capabilities, as well as key combination triggers to give Terry new instructions while the app is not visible in the foreground. Upon addition to the project while on a Mac I noticed, however, that most keyboard events were not being registered. This was a major issue because it rendered demonstration learning

[8] JavaFX is the Java GUI framework that Terry uses. Listing chronologically in terms of release date, the Java GUI frameworks are AWT, then Swing, and then JavaFX.
[9] See the entry for Alex Barker in the bibliography for more info about the JNativeHook library.

essentially impossible. Testing on Windows it then appeared to function properly, so I assumed that the underlying issue was probably with the platform-specific C code that the library was using (Libuiohook) when compiled on a Mac. I then set up a demo to test Libuiohook on my mac, and the output showed that all keyboard and mouse events were being properly registered.

At this point it seemed the issue was narrowed down to the interaction between the C event objects and Java event objects that were being captured by Watcher's event listeners, so I set up another test program with a custom version of JNativeHook that I could modify. I then proceeded to insert debugging outputs in the areas of the library that had to do with event handling and tried viewing the results, with no success. Frustrated from lack of progress and little help from the Github community, I decided to move on and attempt other features of demonstration learning, assuming this problem would work itself out eventually. Unfortunately, little progress has been made with this particular issue since then.

Widget Appearance Acquisition and Overlay

Another issue that arose from differences between Mac and Windows platforms was the Prompter's overlay window. The overlay is crucial for locating widgets in the system GUI because it allows the user to define widget appearance from within Terry. One alternative option for doing so would have been for the user to take a screenshot outside of the app, and then upload the image file to Terry, but this process would have been much less streamlined. Additionally, the overlay allows for Terry to give visual cues to the user, like showing where it thinks a widget is on-screen after performing a search. The method using the overlay window for acquiring widget appearance worked pretty well on a Mac, but on Windows the overlay window did not

appear to capture any mouse drags to be able to define the region for the screenshot. I later figured out that Windows will not allow a completely invisible window to have focus, so the problem was fixed by giving the overlay a semi-transparent background instead of being fully transparent. This solution has worked well enough, and overall was probably the better approach, since this way the user can clearly see when the overlay is enabled and when it's not.

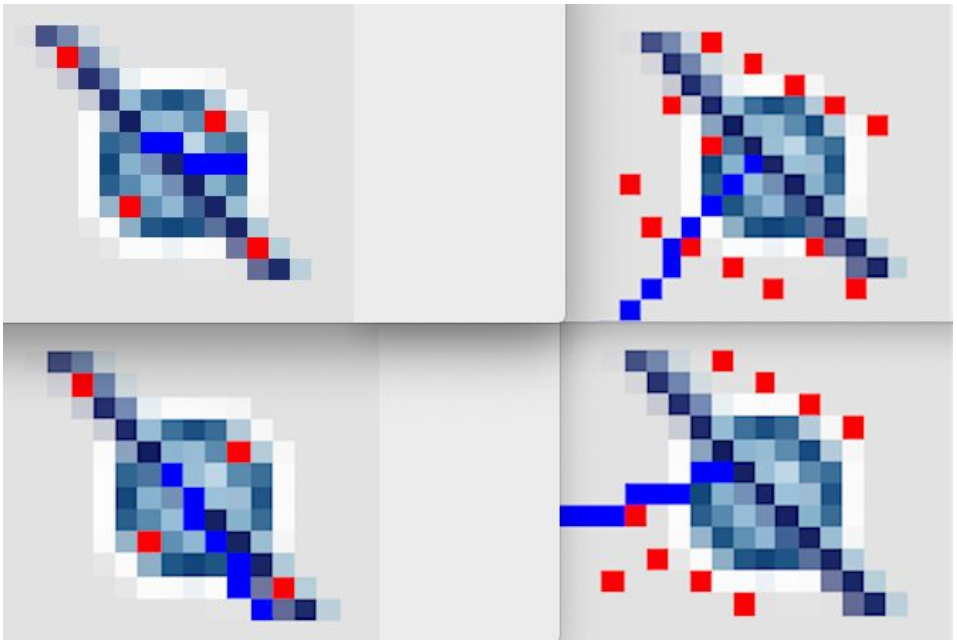
Widget Finding using Feature Recognition vs Template Matching

Once a widget's image on-screen has been acquired, then next challenge is to store that image as an Appearance instance that can be used for the widget to find itself in a screen capture later. My first approach was to try compressing the image into a collection of features, a feature being a keypoint and a descriptor. My keypoints were acquired by using the "fast feature detector" from OpenCV[10] on a grayscale version of the screenshot, and the descriptors expressed change in R,G,B values horizontally (right_neighbor - left_neighbor) and vertically (bottom_neighbor - top_neighbor) around each keypoint in the original screenshot. This representation of a widget appearance was supposed to have two advantages:

1. It would reduce the size of the appearance from hundreds of pixels to a handful of points and small arrays.
2. The features representation of an image, when normalized, should be robust against translation, rotation, and scaling transforms when doing a widget search later.

[10] I used a mix of classes from two different OpenCV libraries for Java. One is JavaCV at https://github.com/bytedeco/javacv, which uses the JNI to call compiled OpenCV C++ code, and the other is the OpenCV Java bindings, which are automatically generated from the C++ header files. The difference is confusing, but currently, JavaCV is used to convert between BufferedImage and Mat objects, and OpenCV is used for template matching.

When doing a widget search by appearance, sections of the full screen shot would be converted to features and compared to the features of the widget to find a match. However, some testing quickly revealed that the features generated for a single image were inconsistent, and therefore nearly impossible to use for image recognition. Eventually I discovered that the transformation between an OpenCV Mat object and a JavaAWT BufferedImage had issues with odd numbers of columns in the image matrix, which helped explain why keypoints would erratically jump between two possibilities (one for even-column images and another for odd-column images).



Feature detection inconsistency. Red pixels are keypoints before normalization, and blue are the principal component vector for normalizing rotation. Not that the left column and right column show a clear distinction between even- and odd- width images.

Odd-width conversion between BufferedImage and OpenCV matrices. The left image is the BufferedImage from the screenshot of the icon, and the right is the grayscale version after conversion to a Mat and then conversion back to a BufferedImage.

However, before realizing this I decided to switch approaches for appearance storage and recognition, and in fact the odd-width problem did not account for 100 percent of the feature variation between different screenshots of the same widget[11]. Now, Terry stores a grayscale image for a widget's appearance, and performs template matching[12] to find the appearance within a screenshot. This method works farily well according to preliminary testing, is more comprehensible, and should continue to work as long as widgets are not transformed or obscured when searching for them.

Instruction Parsing

If I had to pick any single aspect of development that has presented me with the most difficulties and on which I spent the most time writing, debugging, and changing, I would pick instruction parsing. An issue that I encountered early on was how to represent a pattern's expression (for example, **`?a b) +b |@$c,[d_@$e],))`**) as a datastructure that could be traversed an matched to an instruction transcript (for example, **`a b b b d <e>`**). The resulting graph-representation had to include

[11] I also realized later that the fast feature detector does not guarantee the same features to appear for different images with the same objects within it, so I tried switching feature detectors but may not have tested after resolving the odd-width issue. Either way, template matching works well enough.

[12] See OpenCV, "Template Matching" in the bibliography.

branching and cycles, the latter adding complexity to the concept of traversal distance.

After dealing with pattern graph representation, Terry needed to be able to skip trivial words not specified in the expression, but not parse them for argument nodes. String arguments, for example, are allowed to have any tokens within them. Then came the issue of language mappings whose patterns began with argument nodes. These cases complicated the dictionary lookup of mappings, which hashed mappings by leading nodes. The next hurdle was supporting multi-word/multi-token arguments. Now, whenever a possible argument token was encountered Terry could afterward expect either the next node in the pattern graph, or the next token in the current argument node. Each time such a possibility arose, the instruction possibility needed to create a clone of all possible following nodes in the pattern to account for either another argument token or a token for the next node.

The most recent issue in instruction parsing, which I've not solved, is that of multi-instruction transcripts. If a user wants Terry to do something, they should be able to specify a sequence of actions in the same spoken instruction, and not have to decompose their request into singular subrequests and separately record the instructions for each. However, the latter is all that I was able to implement in Terry, because I did not figure out a clear way to pick between continuing to resolve the current instruction and moving on to the next instruction. I do know it would be possible to add this feature in future research without changing Terry's core NLP methodology, but it would require more time than I had within a semester and would increase complexity and resource usage significantly if Terry had to consider starting new instructions at every subsequent token in the transcript.

Platform-Specific Commands for Speaker

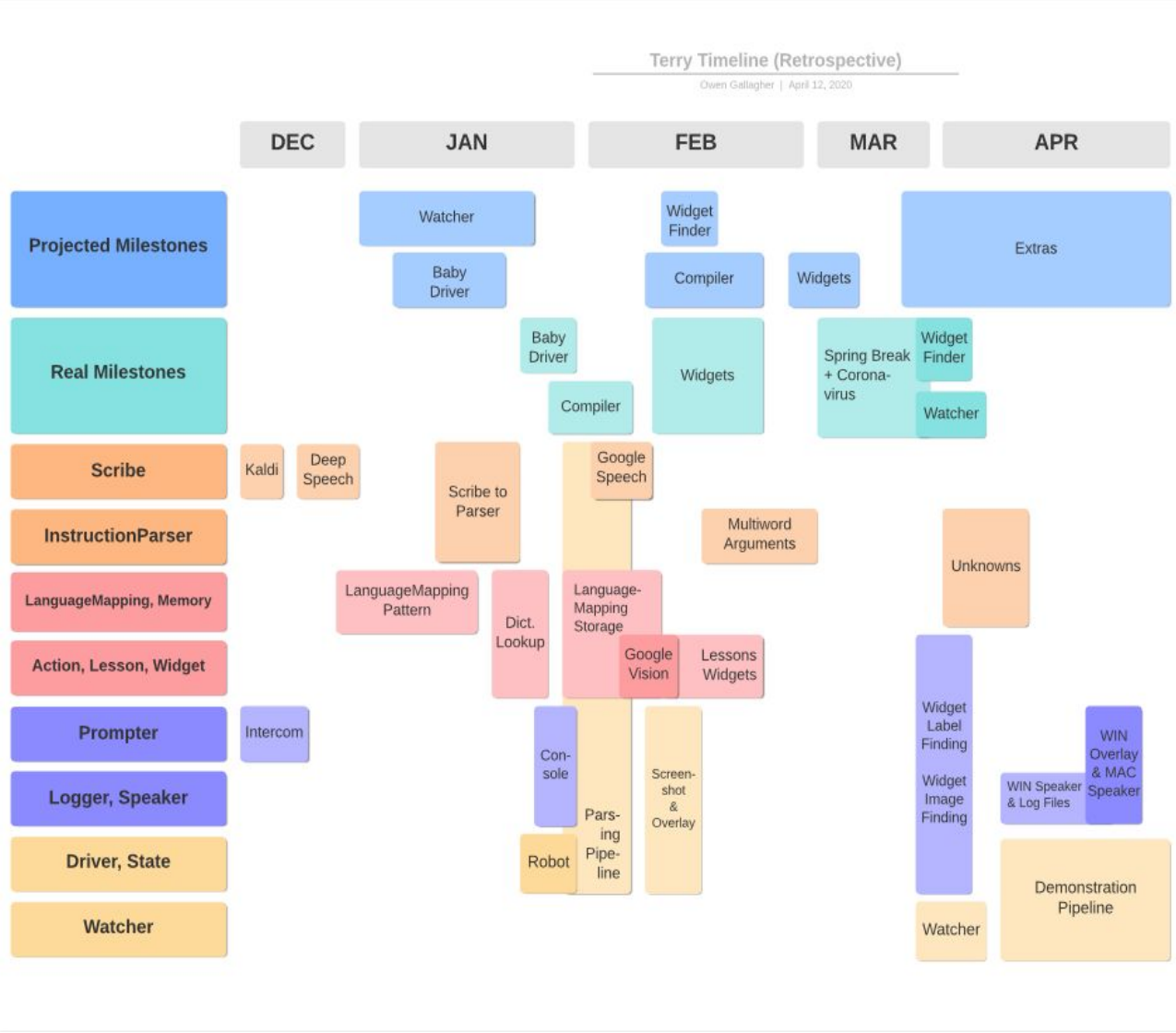
The current version of Terry only uses system commands directly (commands to be executed in a command line shell by the operating system) for the Speaker. Though the Scribe needs to access the microphone through the operating system, Java's sound packages take care of that interaction without having to execute any system commands. The Speaker, on the other hand, calls external programs to handle text-to-speech and playback through the computer's audio output device.

Firstly, this was difficult because Windows did not have any system commands to invoke speech (Terry instead uses a program called voice.exe[13] on Windows), and moreover because the syntax for system commands are quite different between Mac and Windows. Two major details I learned for using system commands in Java on Windows are that (1) though Windows uses the back-slash as a delimiter, Java's filesystem classes handle the conversion most of the time, and (2) when executing programs via system commands, most of the time one will need to use the absolute path to those programs, as opposed to the relative path.

C. Timeline

On the next page is a retrospective timeline for the project from around 20 December 2020 until 12 April 2020. The top row has a selection of milestones representative of projected start and completion dates, and the second row has equivalent milestones with actual dates. The rows beneath are major tasks throughout the project categorized by classes involved, and the right side is a key that describes each task in more detail.

[13] See the bibliography entry for Fulkerson, Eli: "voice.exe".



Kaldi - created first scribe transcript with Kaldi Aspire
Intercom - created the prompter intercom window with the glasses icon, that registers mouse clicks
DeepSpeech - created scribe transcript with DeepSpeech
LanguageMapping.Pattern - the pattern class stores a graph representation of ways an instruction can be said, and can convert from expression representation (ex: |hi,[whats_up],) ?world))
Scribe to Parser - handles the connection between two asynchronous classes to pass the transcript to the instruction parser
Dict. Lookup - created a dictionary for quick reference to known language mappings in memory, accounting for edit distance
Robot - first primitive actions with driver: point, type, click, shut-down; first driver demonstration with primitive action sequence
Console - added console window to prompter, connected to logger
Parsing Pipeline - the pipeline fully connects the intercom, to the scribe, to the instruction parser, to memory, to the driver for execution
Google Speech - improved transcription accuracy by switching to Google Speech API
LanguageMapping Storage - actions and dictionary can now be read and written to files in src/res/memory/
Google Vision - Google Vision API added for optical character recognition
Screenshot & Overlay - driver can now take a screen capture hiding prompter windows; prompter can create an invisible overlay over GUI
Lessons - first lesson for widget definition created
Widgets - widgets with labels can be created via lesson and referenced in instructions
Multiword Arguments - struggles with multi-word arguments in instruction expressions, with eventual success
Spring Break + Coronavirus - about a 1-month pause for Spring break, the break extension, and later cancellation of campus classes at JC
Widget Label Finding - uses Google Vision to find the widget label in a screenshot, then can direct the driver to move the mouse to that location
Widget Image Finding - defines widget appearance and can search for a widget by appearance to move the mouse to that location
Watcher - created watcher to read system keyboard and mouse events with JNativeHook. I have been struggling to get it to work consistently even now
Unknowns - Terry partially recognizes unknown actions and widgets to be learned
WIN Speaker & Log Files - added speaker to windows using voice.exe and enabled log file creation
WIN Overlay & MAC Speaker - fixed windows overlay and added mac speaker
Demonstration Pipeline - drafted the workflow for demonstration learning, but without a functioning watcher

Baby Driver - functioning driver demonstration with primitive actions
Widget Finder - theoretically find widgets on screen by label and appearance
Compiler - instruction parser to driver pipeline complete
Widgets (milestone) - widgets can be learned, referenced, found, saved, and loaded
Watcher (milestone) - create watcher connected ot peripheral input events
Extras - everything I hoped to do but did not reach: functioning demonstration learning, instruction learning, conditional and argument actions, multipattern actions

D. Documentation, Files, and Directories

Below is a list of important resources included within this research project. It includes locations for the source code, documentation, and assets.

https://youtu.be/suchWkpWD7o, https://bit.ly/terrydemo

This is a video demonstration of some of Terry's capabilities as of now, on Mac and Windows computers (as well as a customized forwarding link from bitl.y).

terry/

The git project, accessible at github.com/ogallagher/terry. The associated pages in GitHub are informative, both to see the source code and to view edit history and current and past issues that I published there throughout the semester.

terry/docs/

A directory that includes this research paper, as well as well as my research proposal, and other documentation.

...docs/color_scheme.html

I didn't end up using this much, but if I were to continue working on Terry I'd make its UI elements ascribe to these colors

...docs/pattern_results.txt

A text file with some specifics on the syntax and grammar that I used for my custom instruction parser.

...docs/primitive_actions.md

A list of all the primitive actions I included with Terry, that would serve as a basis for all learned actions.

...docs/terry_architecture.md

A diagram of my proposed architecture for the different modules Terry would include. The final implementation ended up looking quite different though, to account mostly for object-oriented programming.

...docs/terry_timeline.pdf

My projected timeline, separated into columns by module, created before I began implementation.

...docs/terry_timeline_real.pdf

My retrospective timeline, separated into rows by the main classes involved, that reflects how implementation really progressed.

...docs/use_deepspeech_cli.md and ...docs/use_kaldi_aspire.md

My notes on two different transcription library candidates I had in mind, but which I did not end up using in favor of the Google Speech API.

terry/src/com/terry/

The source code folder that contains everything needed to compile into a runnable .jar file, apart from terry/lib-javacv/. The file with the main executable class is Terry.java.

src.../res/

A directory that contains the icon assets, Google API credentials (not visible on GitHub), the Windows-compatible text-to-speech program, JavaFX style sheets, and some configuration files.

VI. FUTURE WORK

There are plenty of improvements that could be made to Terry given time for further development. Demonstration learning is nearly implemented into a functioning prototype, but full keyboard and mouse event handling are still missing. That would require figuring out why certain events are not being passed to JNativeHook to be handled by the Watcher. The Speaker could also be improved by adding actions for the user to customize the voice, volume, and playback speed. The methods exist in the Speaker, only they are not available to the user and have not been tested. This would be more of an aesthetic and usability improvement rather than a contribution to the research aspect of the project. Other improvements of this nature would be to enable customizable system-wide keyboard shortcuts (dependent on Watcher input event handling) and proper handling of OS permissions for the Driver and Scribe.

In addition to improvement of existing features, there were some other key features that I didn't reach, like instruction and reinforcement learning. I had originally hoped for Terry to use a combination of multiple learning approaches, but I didn't accomplish adding either of these two in addition to demonstration learning. Instruction learning support would mean the user can describe new actions, and the description is mapped to states and state transitions, while reinforcement learning would allow the user to provide guidance and feedback. My plan for implementing reinforcement learning was never finalized, but one quick option would be to have

confidence variables, dependent on user feedback, that govern how fast Terry executes each action.

Currently all actions that Terry can learn (and pretty much all of the primitive actions that Terry has pre-learned) are sequential series of subactions. The next step would be for Terry to be able to learn functional actions, which pose the considerable challenge of identifying arguments in action lessons, and mapping those arguments to the correct states. After functional actions in terms of complexity are conditional actions. In this category I'm considering both (1) actions that contain control structures (selection and looping), and (2) actions that respond to events. Event-triggered actions would be associated with other states, and would check for conditions to be met by the values of those states. Apart from learning, one composite action that is event-triggered, which Terry already has implemented and that would serve as an example for future work in this area, is **`mouseToWidget`**. This action calls a subaction **`locateWidget`** to take a screen capture and search in the capture for the given widget destination, and is triggered to call **`Driver.point()`** when the **`widgetlocationupdated`** state sets to **`True`**, meaning that the result of that widget search is stored in the **`widgetlocation`** state.

There will be scenarios when using Terry where a widget might appear at multiple instances on the screen (for example, multiple exit buttons for multiple open windows). There is currently nothing in place to handle these situations. Further development would be well applied if it added support for multiple candidate results of a widget search, and other similar situations of ambiguity. Future research could also improve the robustness of Terry's instruction parsing and extend it to support

multiple instructions[14] in the same transcript. Being able to invoke a series of actions from a single user recording would greatly improve the usability of Terry and its effectiveness in illustrating the capabilities of this project's approach.

[14] To clarify, an instruction is a reference in natural language to a lesson or an action. A transcript is the result of speech-to-text conversion of a recording from the microphone, created by the Scribe.

VIII. BIBLIOGRAPHY

Abdulkader, Lakshmiratan, Zhang (2016) “Introducing DeepText: Facebook’s text understanding engine”. *Facebook: Engineering*. engineering.fb.com/ml-applications /introducing-deeptext-facebook-s-text-understanding-engine

> This article explains in accessible language how a group within Facebook’s AI team worked on an engine for understanding user text content (written in natural language) for extracting things like facts, context, intent, and preference. The engine uses a convolutional neural network, and transforms the input text with a word embedding that preserves position in meaning space, both for semantics within a language and across languages.

Argall, Chernova, Veloso, Browning (2008) “A survey of robot learning from demonstration”. *Robotics and Autonomous systems*, vol. 57, pp. 469-483

> The overview of processes and methods involved in demonstration learning from this paper is very thorough. It touches on many important concepts that have helped me find the right vocabulary for explanation and inquiry, evaluate the effectiveness of demonstration learning, and identify and prioritize common challenges that I’ll need to address when implementing it. It explains, for example, that demonstration learning is broken into two phases: example gathering, and policy deriving. It’s from this paper that I first became aware of the challenge of correspondence, and the solutions of record mapping and embodiment mapping.

Barker, Alex (2006-2020) “JNativeHook: Global keyboard and mouse listeners for Java”. *Github*. github.com/kwhat/jnativehook

> This library uses the Java Native Interface (JNI) to call methods written in C that can be compiled to the specific platform the JVM is running in. The C code it uses comes from another library by the same author, at the following URL: github.com/kwhat/libuiohook.

Barrett, Bronikowski, Yu, Siskind (2018) “Driving under the Influence (of Language)”. *IEEE Transactions on Neural Networks and Learning Systems*, vol. 29, no. 7

> This approach uses natural language route commands for navigation and spatial understanding. A sentence's meaning for this particular application of natural language control gives information about both the robot's path and the objects in a floorplan, which can be thought of as a mix of commands and assertions. A robot constructs a path through a floorplan by satisfying constraints over time (t0=right of cone, t1=behind stool, t3=between stool and chair) with a very probabilistic model. The outcome of this research is a method of “grounding natural language semantics in robotic driving”, which can be applied, at least in

theory, to my application since Terry will be driving the cursor, so to speak, in a spatial environment (the GUI).

Calinon (2018) “Learning from Demonstration (Programming by Demonstration)”. Springer, Berlin, Heidelberg. doi.org/10.1007/978-3-642-41610-1

This paper gives an overview of demonstration learning. It lists various techniques involved, different applications, and comparisons with other methods used in robot learning. A key classification within demonstration learning that I’ve seen in other papers as well is divided into observational and kinesthetic learning, and a key problem prevalent in demonstration learning is correspondence between the demonstrator and robot.

Chen, Mooney (2011) “Learning to Interpret Natural Language Navigation Instructions from Observations”. *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, pp. 859-865

The researchers tackle the problem of following natural language instructions for navigation (without robot learning) using a virtual indoor environment with clear visual cues. The complexity remains fairly constrained though, interestingly, they require that the human navigators give the instruction set from memory, which introduces a seemingly unnecessary variable to muddy the results.

Fulkerson, Eli (2017) “voice.exe”. *Eli Fulkerson .com*. elifulkerson.com/projects/commandline-text-to-speech.php

This is a compiled C# program that uses the System.Speech namespace classes to access Windows’ text-to-speech and audio playback support. Terry uses voice.exe for the Speaker class in Windows environments.

King (2015) “Top 8 virtual personal assistants”. *Raconteur*, Technology/Artificial Intelligence for Business. raconteur.net/technology/top-8-virtual-personal-assistants

Leo King gives a list of 8 exemplary virtual personal assistants from differing environments. When I wanted to compare Terry to existing mainstream assistant programs I primarily used this post, in addition to personal experience. This list supports my assertion that the unique idea that Terry brings to the table is its use of the OS GUI, the same that the user uses, for demonstration learning.

Lauria, Bugmann, Kyriacou, Klein (2002). “Mobile robot programming using natural language”. *Robotics and Autonomous Systems*, vol. 38, pp. 171-181

This paper was essentially my entry into the robot learning field, and was the one that I presented in class. The authors’ research is in response to the question of how to design robots in a world where the applications breach

environments with increasingly naive users, as is the case of domestic robots. They decide to use an instruction-based learning approach, where the robot learns (is programmed) to do new tasks by breaking complex natural language commands into their primitive pieces. The paper also has a useful amount of detail regarding implementation specifics.

Misra, Sung, Lee, Saxena (2016) “Tell me Dave: Context-sensitive grounding of natural language to manipulation instructions”. *The International Journal of Robotics Research*, vol. 35, no. 1-3, pp. 281–300

This is the project within which the Tell Me Dave corpus was created. It will require more reading on my part to see how the corpus was used exactly, but so far it looks like the corpus is a collection of natural language domestic task instructions.

OpenCV (2020) “Template Matching”. *OpenCV: Open Source Computer Vision*, docs.opencv.org/master/d4/dc6/tutorial_py_template_matching.html

This article describes how to perform template matching with OpenCV, including the algorithms used, the syntax of the important classes, and an example program. This is what Terry uses for finding widgets on the screen by image recognition.

Ringaard, Gupta (2017) “SLING: A Natural Language Frame Semantic Parser”. *Google AI Blog*. ai.googleblog.com/2017/11/sling-natural-language-frame-semantic.html

This post proposes an alternative to the modular pipeline solution for language parsing. This typical Natural Language Processing (NLP) system does grammar tagging, followed by dependency parsing. SLING does both steps at once by creating context-dependent frames for words.

Saponaro, Jamone, Bernardino, Salvi (2017) “Interactive Robot Learning of Gestures, Language and Affordances”. arxiv.org/pdf/1711.09055.pdf

Here the researchers propose a learning model to unify input from word descriptions, affordance exploration, and human gestures. This scenario is relevant to my research because, apart from dealing with robot learning in general, it also seeks to combine multiple modes of input for learning.

Thomaz, Breazeal (2008) “Teachable robots: Understanding human teaching behavior to build more effective robot learners”. *Artificial Intelligence*, vol. 172, pp. 716-737

This paper closely investigated the relationship between the teacher and the learner and the different types of communication that pass between them. It makes a number of assertions that I have accepted as truths moving forward, which dictate which methods of human-to-robot teaching should work better. They state that people, in general, want to direct the robot's attention to guide

exploration, have a positive reward bias, and adapt their teaching strategy as they develop a mental model for how the robot learns. To address these findings, the authors suggest ways for the teacher to control the robot's gaze, or area of focus for visual input, and to view what the robot is sensing.

Zamani, Magg, Weber, Wermter (2018) "Deep reinforcement learning using symbolic representation for performing spoken language instructions". *Journal of Behavioral Robotics*, vol. 9, pp. 358-373. doi.org/10.1515/pjbr-2018-0026

These authors, like Lauria et al., considered the domestic task domain as justification for their work. It also references the advantages of combining instruction learning with the reinforcement learning studied by Thomaz and Breazal. Here, they emphasize the need to compartmentalize the task of hybrid learning into distinct modules: intention detection (parse user's natural language instruction), deriving environment state (environment model and context for instruction), and the reward function (handling positive and negative reinforcement).