Contents

1 Introduction **2**

2 Lory: System Design and Workflow **4**
2.1 Wake Word Detection and Listening State 5
2.1.1 Wake Word Detection using Picovoice Porcupine 5
2.1.2 Listening State and Audio Processing 6
2.1.3 Audio Feedback and User Experience 7
2.2 Command Processing 7
2.2.1 Processing the User's Command and Response Handling . 8
2.2.2 Stripping the Response for Further Classification 8
2.2.3 Command Extraction and Error Handling 9
2.3 Classification and Intent Recognition 9
2.3.1 Implementation 9
2.3.2 Data Visualization and Preprocessing 10
2.3.3 Model Preparation and Training 10
2.3.4 Intent Prediction 12
2.3.5 Command Execution Process 14

3 Results and Execution **16**
3.1 Intent Recognition and Slot Extraction 16
3.2 Command Execution: The Role of the controller() Function . . . 17
3.2.1 Executing the Command 17
3.3 Video Demonstration Overview 19

4 Future Enhancements and Conclusion **20**
4.1 Wrapping Up the Watch App 20
4.2 Future Improvements 20
4.3 Conclusion 21

Theses Report: LORY

Nadeer Salem

September 2024

1 Introduction

The rapid development of artificial intelligence (AI) in recent years has opened up new possibilities in how humans interact with technology. Virtual assistants, in particular, have become an integral part of many people's daily lives, with products like Siri, Alexa, and Google Assistant leading the charge. However, these systems often have limitations, particularly in their ability to interact with physical objects or perform more complex, integrated tasks across devices.

The goal of this project is to introduce Lory—an AI assistant built using the GPT API—aims to go beyond those limitations. The initial idea for Lory came from the creation of a simple, hard-coded chatbot, one that could provide fixed responses to specific questions. This type of chatbot would be perfect for businesses or restaurants looking to automate frequently asked questions (FAQs), providing quick, accurate responses to customers. However, as the project evolved, I saw an opportunity to take it further and create a more dynamic assistant that could do more than just answer questions.

Lory was designed to be a hands-free voice assistant, responding to the wake word "Hey Lory." Once activated, Lory listens for a voice prompt, processes the input, and delivers vocal responses, allowing for seamless, hands-free interactions. This functionality makes Lory a highly practical tool for everyday tasks, particularly when multitasking or working in environments where physical interaction with devices may be difficult.

When ChatGPT initially launched, it was primarily a tool for chat completion, excelling at generating text responses but confined to a purely virtual environment. At that point, I envisioned creating an AI that could do more than just respond—it should interact with my computer and control various applications. As an indie developer, it took me some time to build these features, and by the time I did, larger companies had begun releasing similar AI capabilities.

Rather than feeling discouraged, I decided to push the boundaries further and develop Lory into an assistant capable of interacting not just with software but also with the physical world. By adding features like robot arm integration, I am ensuring that Lory can bridge the gap between virtual intelligence and real-world functionality—something that, while being prototyped by other companies, still holds vast untapped potential for innovation.

At this stage, the project is primarily for personal use, and I do not yet intend to release it to the public. The focus remains on refining Lory's capabilities to fit my own needs before considering a broader implementation. However, the potential for expansion is vast, and future iterations may eventually lead to a public version.

Developing Lory has not been without its challenges. As an indie developer, I faced several technical hurdles, particularly when it came to integrating AI with physical devices and computer systems. Debugging the various components, ensuring reliable interactions, and creating a seamless user experience were all learning experiences that shaped the project. These challenges, while difficult, provided a deep understanding of both AI development and practical robotics, which are critical to Lory's functionality.

The development of Lory progressed from a basic FAQ bot to a more comprehensive personal assistant. Now, Lory not only handles communication but also interacts with my computer, opening applications and managing my workspace. One of Lory's key features is the ability to organize tasks through a customized To-Do list that is divided into three sections: 'To-Do,' 'To-Fix,' and 'Done' (because let's be honest, I tend to break things!). In addition to this, Lory can send emails, write notes, search the web, and perform various other administrative tasks that help keep my day running smoothly.

Currently, I am working on expanding Lory's capabilities to further enhance its functionality. One of the key advancements includes a smartwatch app that allows me to prepare my office with a single click. By the time I walk into the room, my computer will be up and running, my task lists will be displayed, and I'll have a brief overview of the day's news waiting for me. Additionally, I am developing a robot arm integration, which will enable Lory to interact with the physical world more tangibly. This expansion will open the door for further automation, allowing Lory to assist with tasks that require physical action, bringing my vision of a truly interactive AI assistant even closer to reality.

2 Lory: System Design and Workflow

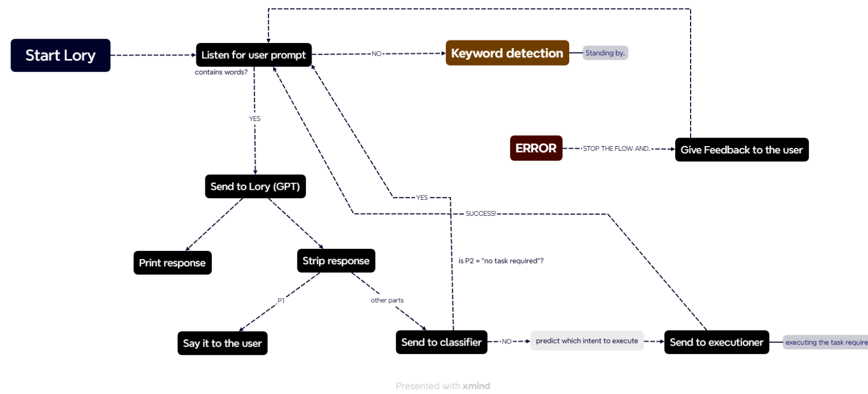

Figure 1: The execution of a task flowchart

The Lory AI assistant operates through a structured process designed to efficiently handle user commands while providing clear feedback and error handling. Below is a brief explanation of the workflow:

1. **Wake Word Detection and Listening State**: Upon startup, Lory enters a direct listening state, ready to capture any command given by the user. If no command is detected within a certain period, Lory transitions into standby mode. In standby, it listens for the wake word "Hey Lory" to reactivate and return to the listening state for user input.

2. **Command Processing**: Once a valid command is identified, it is sent to Lory for further analysis. Lory processes the command by splitting the response into distinct parts. The first part of the response is immediately communicated back to the user vocally, offering instant feedback. The remaining parts are then forwarded to the classification module for deeper analysis.

3. **Classification and Intent Recognition**: The classifier analyzes each part of the stripped response sequentially. If the second part indicates that "no task is required," Lory seamlessly returns to its listening state, waiting for the next user prompt. In the case where a task is necessary, the classifier accurately predicts the user's intent and prepares for task execution.

4. **Task Execution**: Upon intent prediction, the system relays the predicted intent to the executioner module, which carries out the corresponding task. Lory ensures that the task is executed promptly and provides feedback to the user regarding the success or failure of the operation.

5. **Error Handling and Vocal Feedback**: If an error occurs at any stage during the process, Lory responds with vocal feedback, specifying the nature of the error. In cases where the error is not explicitly detailed and Lory simply states that "something went wrong," the user can inquire for more details. Upon request, Lory will provide further information regarding the error, ensuring transparency in error diagnosis and correction.

2.1 Wake Word Detection and Listening State

In the brief overview, I introduced how the wake word detection works with Picovoice Porcupine. Now, let us explore the process more thoroughly, and I will explain how I have adjusted the system for my personal use to make it more efficient.

2.1.1 Wake Word Detection using Picovoice Porcupine

```
def keywordDetection():
    try:
        porcupine = pvporcupine.create(
            access_key= pv_key,
            keyword_paths=["models/wakeWordRaspberry.ppn"]
            )
        recoder = PvRecorder(device_index=-1,
             frame_length=porcupine.frame_length)
        recoder.start()

        while True:
            keyword_index = porcupine.process(recoder.read())
            if keyword_index >= 0:
                recoder.stop()
                porcupine.delete()
                return chat()
    except Exception as e:
        print("error: ", e)
        ms.add_message("console", e)
        recoder.stop()
        porcupine.delete()
```

Picovoice Porcupine is a highly efficient on-device wake word detection engine. It operates entirely offline, ensuring fast response times and privacy. Normally, it continuously listens for the predefined wake word "Hey Lory," filtering out background noise and waiting for a match. When the wake word is detected, it triggers the system to stop listening for the keyword and start processing the user's input.

However, for my personal use, I found it more convenient to skip this step. Instead of waiting for the wake word, I have set Lory to directly enter the

listening state as soon as it starts up. This allows me to give my command right away without needing to first say "Hey Lory." This modification helps me streamline my workflow, saving time when I am ready to interact with the system right from the start.

Once the wake word or the initial command is detected, the system clears any resources used for listening to the wake word. This helps free up memory and processing power to focus on understanding and executing the user's command. After this, the system transitions smoothly into the prompt detection phase, where it processes the command that follows.

2.1.2 Listening State and Audio Processing

Now the system enters a continuous listening state, where it actively monitors the microphone for any audio input after detecting the wake word.

```
with sr.Microphone() as source:
          r.adjust_for_ambient_noise(source, 1)
          print("listening...")
          ms.add_message("console", "listening...")
          starting_sound()
          audio = r.listen(source)

       print ("Processing...")
       ms.add_message("console", "processing...")
       confermation_sound()
```

To manage the system's operation in real-time, a while True loop is employed, which allows the assistant to run continuously, handling both listening and processing of user input. Within this loop, the microphone is initialized using the SpeechRecognition library's Microphone object whenever the system is ready to listen for commands. The library's adjust for ambient noise function is called with the source parameter, which dynamically adjusts the system's sensitivity to background noise over a 1-second period. This ensures that Lory can accurately focus on the user's speech while filtering out any environmental sounds that may interfere with recognition. The same loop also oversees the system's transition into the processing phase, where the captured audio is transcribed and responded to. At this point, the assistant enters the listening phase, indicated by a console message and an audio cue (represented by *starting_sound()*). This vocal or auditory feedback informs the user that the system is actively listening for a command.

Once the system is ready, it captures the audio input using the *r.listen()* method. This method pauses the loop until the user speaks, at which point it collects the audio data for further processing. After successfully capturing the audio, the system transitions to the processing phase, providing another auditory signal (confirmation sound()) and updating the console with a "processing" message to inform the user of the next step.

Once the audio input is captured, the system proceeds to analyze and transcribe it. This is where the code discussed earlier comes into play. The audio data is passed to the containsSpeech(audio) function to determine if the input contains valid speech. If valid speech is detected, the system creates a temporary .wav file, transcribes the audio using the Whisper API, and deletes the file afterward to manage resources.

```
    # Check if the audio contains speech
if contains_speech(audio):
    with open("TemporaryFiles/mySpeech.wav", "wb") as f:
        f.write(audio.get_wav_data())

    with open("TemporaryFiles/mySpeech.wav", "rb") as f:
        try:
            prompt = client.audio.transcriptions.create(
            model="whisper-1",
            file=f,
            language= "en",
            response_format="text",
            )
        except Exception as e:
            print(e)


    os.remove('TemporaryFiles/mySpeech.wav')
else:
    prompt = ""
```

The smooth transition from the listening state to the processing state ensures that user input is captured efficiently and converted into a format that can be understood by the assistant.

2.1.3 Audio Feedback and User Experience

The use of auditory cues during the listening and processing phases enhances the user experience by providing real-time feedback. The user is informed when the system is actively listening and when it is working on transcribing their command. These feedback mechanisms are important for hands-free operation, as they give the user a sense of control and awareness without needing to look at a screen.

2.2 Command Processing

Once the audio input is transcribed into a valid text prompt, the system verifies whether the prompt meets the required conditions for further processing. The prompt must have a minimum length and contain at least one recognizable word.

2.2.1 Processing the User's Command and Response Handling

If the prompt meets these conditions, the system prepares it to be sent to Lory (GPT API) for processing. The prompt is added to the messages list with the appropriate role ("user") and content (prompt). This list allows for a conversational context to be maintained throughout the interaction.

```
messages.append({"role": "user", "content": prompt})
ms.add_message("chat", "- " + prompt)
```

The prompt is then sent to Lory for processing via the *send_to_lory(messages)*function, which handles the API interaction with the GPT model. The response received from Lory is appended to the chat messages and sent to the connected app for display. Since LoryBox does not have a built-in console or screen, these chat messages and system logs are transmitted to the external application, which shows the conversation and system status on the user's device.

Note: The function 'ms.add_message' is used to send these messages to the external app since LoryBox does not have a built-in console or screen.

```
loryResponse = send_to_lory(messages)
ms.add_message("chat", " " + loryResponse)
```

Now what if the prompt does not meet the validity requirements? If the prompt does not meet the validity requirements, the system plays a sound cue indicating that it is returning to the standby state and waits for the next wake word trigger.

```
else:
    ending_sound()
    print("standing by...")
    ms.add_message("console", "standing by...")
    return keywordDetection()
```

2.2.2 Stripping the Response for Further Classification

Once a valid response is received from Lory, it must be parsed to extract actionable content. The system employs regular expressions to strip the response into specific segments. The first part of the response is extracted by searching for the text before the pipe (—) symbol or until the end of the string. This portion is considered the vocal response that Lory will speak back to the user.

```
stripedResponse = re.search(r'^(.*?)(?=\||$)', loryResponse,
    re.DOTALL).group(1).strip()
openAiSay(stripedResponse)
```

2.2.3 Command Extraction and Error Handling

The next step involves searching for a command in the response. Commands are enclosed within pipe (—) symbols, so the system uses another regular expression to capture the content between the pipes. This command, if detected, is then passed to the classifier for execution. If no command is detected, the system logs an appropriate message, and the original prompt is used as a fallback.

```
try:
    command = re.search(r'\|||(.*?)\|||', loryResponse,
        re.DOTALL).group(1).strip()
except:
    print("no command detected")
    ms.add_message("console", "no command detected")
    command = prompt
```

By splitting the response into distinct parts, the system is able to differentiate between what should be spoken back to the user and what commands should be executed in the background. This structured approach ensures that the system can both interact conversationally with the user and perform tasks as directed.

2.3 Classification and Intent Recognition

One of the essential components of an AI assistant like Lory is its ability to understand what the user is asking. This is done through **classification** and **intent recognition**. The system needs to determine the type of action or task the user wants to perform based on the spoken command. To achieve this, we use a command model that takes the user's input, classifies it into predefined intents, extracts relevant information, and assigns a confidence score to its prediction.

2.3.1 Implementation

The user's spoken command is passed to the model with the function call

```
    intent, slot, confidence = cm.command(command.lower())
```

This step ensures that the command is lowercased for consistent processing. The command model, represented as *cm*, analyzes the input and classifies it into a predefined intent, returning a tuple consisting of the intent, the slot (relevant information for executing the intent), and the confidence score (a float value between 0 and 1 that indicates how certain the model is about the classification). The result is printed to the console using *print(intent, slot, confidence)*. To keep track of the system's decisions, the console logs the returned values with the command *ms.add_message("console", f"[intent, slot, float(confidence)]")*. This ensures that the message is added to the ms object, which handles console messages, formatted as "[intent, slot, confidence]".

```
intent, slot, confidence = cm.command(command.lower())
print(intent, slot, confidence)
ms.add_message("console", f"[{intent}, {slot}, {float(confidence)}]")
```

2.3.2 Data Visualization and Preprocessing

Before diving into the model architecture, it is crucial to understand the dataset that the model is trained on. The dataset consists of several intents, each associated with example patterns (user inputs) and slots (specific information for executing tasks).

In the following sample, we see how the intent name, patterns, and slots work together:

```
- name: turn on
  patterns:
    - run the server
    - fire up the 3D printer again
    - fire up the server
    - please turn on the lights
    - please turn on the Raspberry Pi
    - Lights on
  slots:
    - device: 3D printer, server, lights, light, Raspberry Pi
    - location: my room, desk
```

- The **name** represents the intent that the model will predict, in this case, "turn on."
- **Patterns** are example inputs that the user may say. The model is trained on these patterns to predict the intended action or intent.
- **Slots** specify additional information required to perform the task. For instance, if the user says "turn on the server" the model predicts the "turn on" intent. However, to determine what should be turned on, it looks at the slots to identify the device (in this case, "server").

  This is just an example of the data structure used in my personal project. In the future, when I publish it, the system will be more dynamic and flexible, allowing it to handle a wider variety of tasks and inputs more effectively.

2.3.3 Model Preparation and Training

The following is responsible for preparing the data, building the neural network, and training the model for intent classification. It processes user inputs, trans-

forms them into machine-readable formats, and defines the model's architecture. Now, we are going to explore what is inside the Commands model (cm).

1. **Data Preprocessing:** The initial step involves tokenizing the patterns (user inputs) and extracting relevant features for training:

```
    wrds = nltk.word_tokenize(pattern) # breaks the pattern (text)
        into tokens (words)
words.extend(wrds) # adds each tokenized word into the 'words' list
docs_x.append(wrds) # stores tokenized patterns
docs_y.append(intent["name"]) # stores the intent name
    corresponding to the pattern
```

   Here, the model gathers all words from the input patterns and associates them with their respective intents. Tokenization helps in breaking down sentences into individual words.

2. **Stemming:** The words are converted to their base form (stems) and lowercased:

```
words = [stemmer.stem(w.lower()) for w in words if w != "?"]
```

   Stemming reduces words to their root form, making the model more efficient by reducing word variations (e.g., "running" becomes "run").

3. **Bag of Words and One-Hot Encoding:** The data is transformed into a format suitable for training the model:

```
    out_empty = [0 for _ in range(len(labels))] # creates an empty
        one-hot vector for labels
for x, doc in enumerate(docs_x):
    bag = []
    wrds = [stemmer.stem(w) for w in doc] # stems each word in the
        pattern
    for w in words:
        if w in wrds:
            bag.append(1) # assigns 1 if the word is in the pattern
        else:
            bag.append(0) # assigns 0 otherwise
    output_row = out_empty[:]
    output_row[labels.index(docs_y[x])] = 1 # one-hot encodes the
        intent label
```

   This creates a numerical representation of the input, where the presence of words is marked by 1, and their absence by 0. Similarly, intents are represented as one-hot encoded vectors, where each intent corresponds to a unique vector.

4. **Model Architecture:** The model is defined using TFLearn's DNN (Deep Neural Network) structure:

```
net = tflearn.input_data(shape=[None, len(training[0])])
net = tflearn.fully_connected(net, 8)
net = tflearn.fully_connected(net, 8)
net = tflearn.fully_connected(net, len(output[0]),
    activation="softmax")
net = tflearn.regression(net)
model = tflearn.DNN(net)
```

   The network consists of an input layer (with the size of the training data), two hidden layers (each with 8 neurons), and an output layer that uses the softmax activation function to classify the input into one of the predefined intents.

5. **Training and Saving the Model:** The model is trained with the preprocessed data:

```
model.fit(training, output, n_epoch=1000, batch_size=8,
    show_metric=True)
model.save("models/brainModel")
```

   The model is trained for 1000 epochs with a batch size of 8. After training, it is saved as brainModel for later use.

6. **Bag of Words Function:** This function converts any user input into the same bag of words format for prediction:

```
def bag_of_words(s, words):
    bag = [0 for _ in range(len(words))]
    s_words = nltk.word_tokenize(s)
    s_words = [stemmer.stem(word.lower()) for word in s_words]
    for se in s_words:
        for i, w in enumerate(words):
            if w == se:
                bag[i] = 1
    return numpy.array(bag)
```

   The *bag_of_words(s, words)* function ensures that new user inputs can be converted into a format the trained model understands.

2.3.4 Intent Prediction

The core function for predicting user intent is responsible for interpreting the input and identifying the most likely task the user wants to perform. The function *command(inp: str)* takes the user's input, processes it, and returns the predicted intent, the detected slots, and the confidence score of the prediction.

Here is an overview of how the function works:

1. **Input Processing:**

```
if inp != "no task required":
    results = model.predict([bag_of_words(inp, words)])[0]
    results_index = numpy.argmax(results)
    name = labels[results_index]
```

   The user's input is first processed into a bag of words format. The trained model then predicts the intent by returning a list of probabilities. The highest probability corresponds to the predicted intent, which is found using *numpy.argmax(results)*.

2. **Threshold Check:**

```
if results[results_index] >= 0.85:
```

   A confidence threshold of 85% is set. If the model's confidence in the predicted intent is greater than or equal to 85%, the system proceeds to extract slots (additional information such as room or object).

3. **Slot Extraction:**

```
for d in intent["slots"]:
    for slot, values in d.items():
        slot_values = [v.strip() for v in values.split(',')]
        slots[slot] = slot_values
```

   Once the intent is predicted, the system searches for any slots associated with that intent. Slots provide the context needed to execute the task, such as specifying a room or object. The system tokenizes the input and checks for the presence of the slot values.

4. **Slot Detection:**

```
for i in range(len(tokens) - value_length + 1):
    if tokens[i:i+value_length] == value_tokens:
        slot_values_detected[slot].append(value)
```

   The system compares the tokenized user input with the possible slot values. If any of the slot values match, they are stored as *slots_mentioned*.

5. **Return the Result:**

```
return intent_name, slots_mentioned, results[results_index]
```

If the confidence level is high enough, the function returns the predicted intent name, the detected slots, and the confidence score. If no match is found or the confidence is low, it returns None for both the intent and slots, along with the confidence score.

6. **Handling 'No Task Required':**

```
else:
    return None, None, 1
```

In case the system detects no task is required (e.g., the input is "no task required"), the function returns None for both intent and slots but sets the confidence to 1 to avoid any errors in the system's return handling.

2.3.5 Command Execution Process

Once the system successfully recognizes an intent and identifies the relevant slots from a user's input, the next step is to execute the command. This process is initiated by sending the predicted intent, along with the associated slots and prompt, to the executioner component of the system using the following function call in the main code:

```
commandExecutionar.controller(intent, slots, prompt, loryResponse)
```

The controller() function acts as the decision-maker, using a series of if-statements to determine which task should be executed based on the provided parameters:

```
def controller(intent:str, slots:dict, prompt="", loryResponse=""):
```

- **intent:** This is the core instruction, such as "turn on" or "search web," that determines what kind of task the system needs to perform.
- **slots:** These provide context or specifics for the task. For instance, in the command "turn on the server," the intent is "turn on," and the slot is "server."
- **prompt:** The user's full input is passed through this parameter, allowing the system to extract additional details when necessary. For example, in a web search command, the prompt helps the system know what specifically needs to be searched.
- **loryResponse:** Lory generates structured responses, some of which are not shown to the user but guide the system. In tasks like managing a to-do list or writing notes, this parameter helps the system organize content—for example, identifying the task's content and determining where to add it in the list.

Through these parameters, the controller() function evaluates the input and triggers the appropriate task. For instance, if the intent is "turn on," and the slot is "server," the function will proceed to check the server status and perform the required actions to turn it on.

3 Results and Execution

In this chapter, I will demonstrate the full execution flow of the system from intent recognition to task completion. Using the example command, "turn on the server," we will show how the system processes user input, predicts the intent, identifies the slots, and executes the task.

3.1 Intent Recognition and Slot Extraction

The first step in the execution process is recognizing the user's intent and extracting the relevant slots from the input. The *command()* function is responsible for processing the user's input and predicting the intent.

For the input *"turn on the server,"* the function works as follows:

```
def command(inp:str):
    if inp != "no task required":
        results = model.predict([bag_of_words(inp, words)])[0]
        results_index = numpy.argmax(results)
        name = labels[results_index]
```

Here, the input is first converted into a bag-of-words representation, allowing the model to process it. The model then predicts a list of confidence scores for each possible intent, and the intent with the highest confidence is selected using *numpy.argmax()*. In this case, the predicted intent is "turn on."

Next, the function checks if the confidence level of the prediction is high enough (in this case, a threshold of 0.85 is used). For the input "turn on the server," the confidence score returned is extremely high: *0.9999249*. Since the score exceeds the threshold, the function proceeds to extract the slots:

```
if results[results_index] >= 0.85:
    slots_mentioned = {}
    for intent in data["intents"]:
        if intent["name"] == name:
            intent_name = name
            slots = {}
            for d in intent["slots"]:
                for slot, values in d.items():
                    slot_values = [v.strip() for v in values.split(',')]
                    slots[slot] = slot_values
```

At this point, the system has determined that the intent is "turn on," and now it looks for any relevant slots, such as "server." The slot information is stored in the *slots_mentioned dictionary*, and the system uses token matching to find slot values within the user input:

```
tokens = word_tokenize(inp)
slot_values_detected = {}
```

```
for slot, values in slots.items():
    slot_values_detected[slot] = []
    for value in values:
        value_tokens = word_tokenize(value)
        value_length = len(value_tokens)

        for i in range(len(tokens) - value_length + 1):
            if tokens[i:i+value_length] == value_tokens:
                slot_values_detected[slot].append(value)
                break

slots_mentioned.update(slot_values_detected)
```

In this case, the function would return:

- **Intent:** "turn on"
- **Slots:** "device": ["server"]
- **Confidence:** 0.9999249

3.2 Command Execution: The Role of the controller() Function

Once the intent and slots are identified, the *controller()* function takes over to execute the command. For the example command *"turn on the server,"* the *controller()* function uses a series of if-statements to determine the appropriate actions based on the intent and slots.

Here is the relevant code snippet for handling the "turn on" intent:

```
if intent == "turn on":
    if slots:
        if "device" in slots.keys():
            if "server" in slots["device"]:
                ip = "192.168.x.x"
                plug = PyP100.P100(ip, email, password)
```

In this case, the *controller()* function first checks if the intent is "turn on" and verifies that slots are present. It then checks if the slot key "device" is available and specifically looks for "server" in the slot values.

3.2.1 Executing the Command

1. **Checking Raspberry Pi Status:** If the "server" slot is detected, the function proceeds by checking the status of the Raspberry Pi. This involves using a smart plug to power on the Raspberry Pi if it is off:

```
if not is_raspberrypi_ready("192.168.x.x", "nadeer", "password"):
```

```
openAiSay(random.choice(["Raspberry pi is not online, sir. I
    will turn it on right away.",
                    "I found that the raspberry is still off,
                        I am truning it on right now.",
                    "I couldn't connect to the raspberry,
                        sir. it seems to be turned off, I
                        will activate it and try again."]))
plug.turnOn()
```

2. **Starting the Server:** After the Raspberry Pi is ready, the function attempts to start the server. It uses SSH to execute a command that activates a virtual environment and starts the server script:

```
try:
command = 'ssh nadeer@192.168.x.x "source
    /home/nadeer/Desktop/ve/bin/activate; cd
    /home/nadeer/Desktop/LoryCommands; nohup python server.py
    &"'
subprocess.Popen(command, shell=True)
```

The function then monitors the server's status by sending HTTP requests and handles retries if the server does not respond immediately:

```
while True:
try:
    response = requests.get("http://192.168.x.x:5000")
    if response.status_code == 200:
        break
except (requests.ConnectionError, requests.Timeout):
    attempts += 1
    openAiSay(f"trying to connect again, attempt number
        {attempts}")
    if attempts >= 5:
        raise Exception("server not responding")
time.sleep(2)
```

3. **Providing Feedback:** Finally, the function provides feedback based on whether the server was successfully activated or if an error occurred:

```
openAiSay(random.choice(activation_success_dict["server"]))
```

If an error occurs during the activation process, it is caught and reported:

```
except Exception as e:
print("error: ", e)
openAiSay(random.choice(activation_error_dict["server"]))
```

3.3 Video Demonstration Overview

In the accompanying video, I demonstrate the multitasking capabilities of the Lory assistant, specifically within the context of my work on the robotic arm. The video begins with a voice command to Lory, explaining that there is an issue with the robot arm. I then instruct Lory to search Google for a NEMA 17 stepper motor on Amazon, highlighting its ability to perform web searches. Following this, I command Lory to open the Fusion 360 software, which I use for 3D modeling, and simultaneously request it to turn on the 3D printer, showcasing the system's ability to handle multiple tasks concurrently.

A timelapse sequence is then shown, capturing the process of my work on the robot arm, illustrating the hands-on aspect of the project. After completing the task, I instruct Lory to put the computer to sleep, demonstrating the assistant's ability to manage device states efficiently.

This video serves as a practical demonstration of Lory's interactive features and multitasking capabilities, particularly in a real-world use case involving both software and hardware components.

You can watch the demo by clicking here!

4 Future Enhancements and Conclusion

4.1 Wrapping Up the Watch App

As I approach the conclusion of my work on the Lory assistant, I have successfully wrapped up the development of the watch app. This app now allows Lory to prepare my entire workspace with just one click, streamlining my daily routine and improving efficiency. The app was developed using Android Studio, which provided a versatile platform for designing and managing the various interfaces and features necessary for seamless interaction between my watch and the Lory system.

In the development process, I utilized Android Studio's built-in support for Wear OS, which helped me connect the watch app to LoryBox via HTTP. The watch app sends commands to the LoryBox, allowing it to execute tasks like turning on devices or loading up specific setups. This functionality, as demonstrated in my results video, showcases how Lory integrates seamlessly with my workspace, allowing for efficient and automated setup with just one click.

4.2 Future Improvements

Looking ahead, Lory's capabilities have room for expansion, especially in interacting with the physical world. After completing the robot arm, I plan to explore the integration of lightweight drones that are fully aware of their surroundings. These drones would leverage modules like the Arduino Nano and ESP32-CAM to process visual data and navigate autonomously. This expansion into real-world spaces would allow Lory to control objects with even greater precision and responsiveness. Although this may not seem like a project directly related to my current work and might appear not worth the time, I chose it as my first step because of the unique challenges I'll face. These challenges include optimizing drone navigation, improving real-time processing of visual data, and ensuring seamless communication between Lory and the drones. Overcoming these obstacles will help me build the skills necessary for achieving larger goals in the future. Plus, the process itself is enjoyable for me, making it both a valuable and fun endeavor.

Improving Lory's file management capabilities is another key area for enhancement. Lory could become more adept at organizing, categorizing, and retrieving files, making it an even more powerful assistant in handling both digital and physical tasks. Additionally, I envision Lory assisting with online businesses by performing business analysis, generating statistics, and providing clients with their needs. This would allow Lory to not only support my workflow but also contribute to business management and client services.

My vision for Lory's future extends beyond drones. While I plan to start with lightweight drones, my broader goal is to expand Lory's control over various systems and environments. One idea is integrating Lory with a CCTV system, enabling it to track my movements and adjust its behavior dynamically—whether

controlling lights, managing devices, or responding to my presence in specific areas. The CCTV system could also serve as a traditional security tool, with Lory monitoring my workspace or home when needed.

These are just a few of the projects I envision, and as Lory evolves, it has the potential to handle increasingly complex tasks, seamlessly blending digital and physical worlds.

Lastly, I am considering adding features that enable Lory to assist in creating 3D models. While still a developing field for AI assistants, 3D modeling has vast potential for applications such as prototyping and design. This would bring Lory closer to offering a comprehensive set of tools for creators, expanding its utility even further.

4.3 Conclusion

Throughout this project, I have explored the development of Lory, an AI assistant designed to streamline tasks and integrate seamlessly into both digital and physical environments. From improving personal productivity with the ability to prepare my workspace via a watch app, to future expansions like robot arms, drones, and CCTV integration, Lory's potential continues to grow.

The journey hasn't been without its challenges, but each obstacle has pushed me toward finding innovative solutions. With future enhancements in sight—such as improved file management, business analytics, and even 3D modeling—the possibilities for Lory's applications are vast. This project represents the foundation of something much larger, and I am excited to continue expanding Lory's capabilities in the future.

Lory is not just an assistant—it's a continually evolving system that will one day be capable of handling complex tasks across different domains, from helping manage my workspace to integrating into online businesses and the physical world.