To what extent do neural networks facilitate speech-based mobile intelligent assistants better than regular rule-based programming?

Session: May 2017

Subject Area: Computer Science

Word Count: 3915

Advisor: Mr. Richard Kick

Abstract

In light of the human population's increasing dependence on smart phone technology, this essay analyzes the basis of both rule-based and neural network programming and the differences between the two to answer the question: **To what extent do neural networks facilitate speech-based mobile intelligent assistants better than rule based programming?** Siri and other smartphone assistants have always been notoriously bad at recognizing colloquial speech, engaging in human-like conversations, or properly "hearing" a command. Software engineers have begun transitioning from traditional rule based programming to neural network programming to combat these issues. Reasoning behind natural language processing; research into the mathematical and logical concepts of each; comparative diagrams and tables; and development of two sets of example code solving the XOR function with collection and graphing of test results were used to explore the functionality of both programming methods. These functionalities were then extrapolated to the purposes of an intelligent assistant in order to analyze benefits as well as drawbacks based on measurements of efficiency and accuracy of the building and execution processes. Neural networks were found to work best for speech-based intelligent assistants because of their use of heuristics applied to self-learning abilities, allowing programs to recognize varying language and speech situations; however, they had drawbacks in supervised training time and necessary data quality. On the other hand, rule based programs were deemed inefficient because of the complexity of the pre-programmed algorithms and the unreasonable time required to execute them when used for intelligent assistants. These results show that software engineers are moving in the right direction by implementing neural networks, a much more powerful alternative to programming with rules.

Word Count: 270

Table of Contents

1. Introduction

In a world where people are becoming increasingly dependent on smartphone technology in their everyday lives, speech-based mobile intelligent assistants like Siri have the potential to become a pivotal force in shaping humans' responsibilities, abilities and schedules. Currently, personal speech-based intelligent assistants are useful for completing simple tasks like scheduling an event or completing an internet search, and have come a long way to existence in the past decade. However, in terms of making decisions and leading conversations, they are often ridiculed for their robotic, programmed responses and inability to make logical conclusions. Futuristic goals of these assistants as human-like, intimate programs can be achieved by shifting the focus away from rule-based programming techniques to relatively new methods of artificial intelligence learning that use neural networks.

1.1 Speech-based mobile intelligent assistants

Software personal assistants, known as intelligent assistants, are programs that assist humans with the functions and capabilities of their technological devices. To use them, people will often trigger a starting mechanism before engaging in dialogue with the mobile intelligent assistant through spoken questions or commands. British artificial intelligence pioneer Oliver Selfridge's idea of "demons," or programs that could interact with their environment, in the 1950s first led to this idea of intelligent assistants (Markoff 2008, Oliver).

Research on intelligent assistants began again in the early 2000s with SRI International's CALO (Cognitive Assistant that Learns and Organizes) funded by the U.S. Pentagon for military purposes, and resulted in spinoff company, Siri Inc. (Markoff 2008, Siri). The widespread use of intelligent assistants in the consumer market really began when Apple Inc., the leading tech company behind Mac computers and iPhones, acquired and released Siri as an integrated feature of the iPhone 4S in 2011. Since

then, other leading tech companies have released their own intelligent assistants, including Amazon's Echo, Microsoft's Cortana, and Google's Google Now, as integrations in their current consumer products.

2. Modes of Artificial Intelligence

2.1 Artificial Intelligence and Natural Language Processing

Artificial intelligence is a field of computer science that seeks to demonstrate natural intelligence through behaviors created by software and hardware (Haaxma-Jurek). This could include varying tasks such as solving purely algorithmical problems by way of easily broken-down procedures and rules, or solving cognition-based problems that require flexible responses according to multiple factors.

"Natural language" refers to the language that humans use. It is difficult for computers to understand natural language, especially colloquial language, because of the ambiguity, incompleteness, and inaccuracies that are usually easily filtered by humans due to experience. Besides analyzing syntax, semantics, and context of a sentence, artificially intelligent programs would also have to know what individual words mean based on a development of pattern recognitions from the past. This is known as natural language processing (Haaxma-Jurek).

Speech-based mobile intelligent assistants are examples of artificial intelligence that must use natural language processing because the goal of these programs is to solve problems spoken to them in order to relieve the duties of their human users. While early assistants utilized heavily algorithmical behavior, this approach proved too inflexible since the grammatical structure of commands had to be spoken in very specific ways in order to be properly processed. Later research has improved this problem by inserting more sets of rules that account for different structural possibilities, but the ultimate, efficient goal is to create assistants that can learn to flexibly respond to a variety of

commands that don’t need to follow a set model. Intelligent assistants have an extra burden besides the processing: responding, with the ability to give answers based on its environment and the spoken commands. Neural networks are one such method of this improvement.

2.2 Rule based programming

Rule-based programming is programming based on logic flow, where a starting set of inputs that form the base of data are processed by applicable rules determined by monitors (Rule). These rules are conditional statements in if-then form: if a condition is met, a certain set of actions will be performed, otherwise another set of actions will be performed; thus, it excludes switch or iterative statements. Often, multiple rules are used in conjunction with each other to produce a final solution. This type of programming is a subcategory of declarative programming, which defines itself with what the program should result in rather than how to get there, as in imperative programming; in other words, the control flow is left for a compiler of the program language to decide.

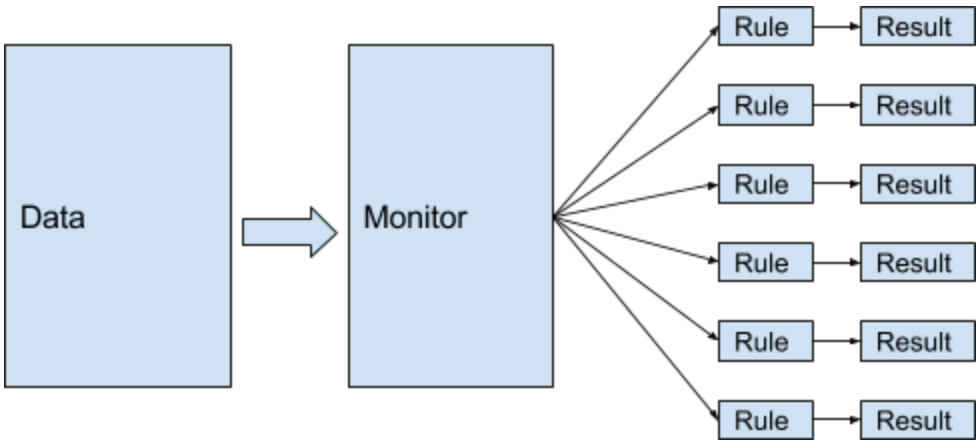


Figure 1: Basic diagram of program flow using rule-based programming, in which data interpreted by a monitor/set of monitors is sent to rule(s) to produce results.

The very primary approach to artificial intelligence uses rule-based programming by assuming that human brains interprets set symbols using logical operators; as a result, all rules must be preset and cannot be changed through growth or learning (Munro). This was first configured in expert systems, which are programs that emulate decision making processes for complex real-world problems in professional fields such as medicine and structural engineering (Markoff 1988).

2.3 Neural Networks for Learning

Artificial neural networks are software computing networks based on philosophies of biological neural networks, which are defined by neurons in the nervous system and their connections or interactions[1]. At a basic level, they are built using interconnected nodes in layers: the input layer of nodes receives external signals; the internal layer of nodes processes input signals from the input nodes and produces output signals as functions; and the output layer of nodes produces a result (Munro).

Neural networks use fuzzy logic, which holds that there exist possible values between 0 (false) and 1 (true), because connections between nodes are weighted so that the likelihood of events influences outputs, much like the concept of probability (Desiano). This contrasts with boolean logic, in which there are only two possibilities of either never getting selected (0) or always getting selected (1). Among other applications, fuzzy logic is an important factor in natural language processing, since words can hold multiple values, for example far, farther, farthest, as far as possible, etc.

As a result of depending on fuzzy logic, neural networks can be self-learning as they adjust the weights on node connections in an attempt for accuracy. One method of this self-learning is backpropagation, which stands for the “backwards propagation of errors” (Rojas). This algorithm updates the weights in the connections by using the gradient descent method of finding the error function’s optimum minimum point, which

[1] *Note: Use of the term “neural network” from this point will imply an “artificial neural network.”*

corresponds to the most accurate solution, and then changing the weight retrospectively. In order to determine whether mistakes or errors exist, neural networks must have some known outputs for some set of inputs to train with so that faulty outputs can be compared and corrected. The backpropagation method can be continuously iterated until the outputs are correct.

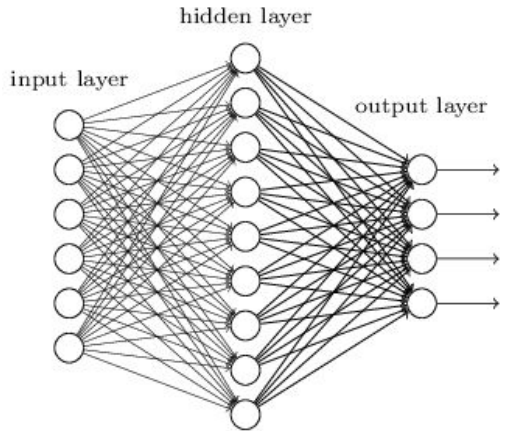


Figure 2: Diagram of a basic neural network with one input layer, one hidden layer of nodes, and one output layer, with weighted connections.[2]

When the concept of neural networks were first created, they only had three basic layers that hindered the complexity of inputs that programs could solve, especially when it came to complicated natural language structures. Deep neural networks were developed to combat this: multiple layers of internal nodes could exist to target a specific aspect of a problem; the descriptor “deep” came from “deep learning”, a branch of machine learning implying multiple layers of analysis. The first layers would fully analyze and recognize the simpler properties before passing down the information to a second layer that would recognize higher-level features, and so on, until conclusive recognitions as a whole are reached (Hof). This provides extra breadth and depth to the analysis, as well as greater potential for fine-tuning the learning process. At this point in time in software development, the term “neural networks” automatically refers to “deep neural networks” because they are now the most advanced and useful form.[3]

[2] *http://i.stack.imgur.com/OH3gI.png*
[3] *Note: Use of the term “neural network” from this point will imply “deep neural network.”*

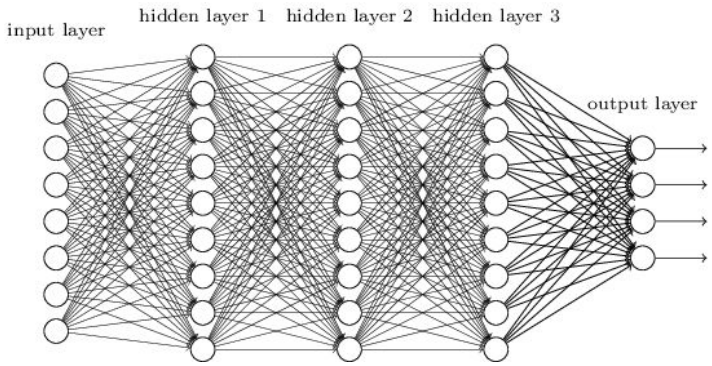


Figure 3: "Deep" neural network with multiple internal hidden layers; this is currently the most advanced form of neural networks.[4]

3. Code Examples and Tests

In order to demonstrate and compare the functionality of rule-based programs and neural networks, two programs were written to solve the XOR logic operator for the first two values in a group of three values, in which one true (1) value, but not two, gives the final true (1) result. The third value is disregarded. Both programs are written in procedural methods using Java.

Table 1: Possible groupings and results of XOR function.

Value 1	Value 2	Value 3	Result
0	0	0	0
0	0	1	0
0	1	0	1
1	0	0	1
1	0	1	1
1	1	0	0
0	1	1	1
0	0	0	0

3.1 Rule-Based Program[5]

No training is required for rule-based programs because the rules must be pre-written. First, establish the input to test with. Each line holds a new case to test.

```
double[][] inputArray = {{0, 0, 0},
```

[4] *http://i.stack.imgur.com/OH3gI.png*
[5] *Full program code can be found in Appendix A.*

```
                                        {0, 1, 0},
                                        {1, 0, 0},
                                        {1, 1, 0}};
```

Iterate through each case and send it to a monitor.

```
for (int i = 0; i < inputArray.length; i++)
{
    monitor(inputArray[i]);
    outputArray[i][0] = finalState;
}
```

This monitor analyzes the input and decides which rule to use based on the first value.

```
public static void monitor(double[] arr)
{
    if (arr[0] == 0)
        xorRules0(arr);
    else
        xorRules1(arr);
}
```

The rules analyze the input's second value and determines the state of the program through global variable finalState, which is saved and stored in an array of ouputs.

```
public static void xorRules0(double[] arr)
{
    if (arr[1] == 0)
        finalState = 0;
    else
        finalState = 1;
}

public static void xorRules1(double[] arr)
{
    if (arr[1] == 0)
        finalState = 1;
    else
        finalState = 0;
}
```

3. 2 Test Results

The program consistently outputs the correct results for the given input: 0.0, 1.0, 1.0, 0.0. Total time to completion is always 1 millisecond. Two rules are used because the XOR function consists of two parts: OR, in which at least one true (1) results in true, and NOT, in which two inputs of the same value (either true/1 or false/0) results in false (0). A function can be established for the maximum number of times the quality of a value is checked y: y = 2x, where x represents the number of numeric possibilities and 2

represents the two parts of the XOR function. More generally, y = kx, where k represents the number of parts of a value being checked for.

3.3 Neural Network Program[6]

Due to the need for connected products and sums of the connection weights and node values, matrices are required for neural network calculations. The JAMA matrix library was imported for this program.

Training is required for the neural network. Therefore, predetermined inputs and outputs are created and stored in matrices. Four cases will be used for testing; each line is a new case.

```
double[][] inputArray = {{0, 0, 0},
                         {0, 1, 0},
                         {1, 0, 0},
                         {1, 1, 0}};
double[][] outputArray = {{0},
                          {1},
                          {1},
                          {0}};
Matrix input = new Matrix(inputArray);
Matrix output = new Matrix(outputArray);
```

The weights between the input and hidden layer, and the hidden and output layer, are initialized in new matrices. The values are randomly chosen between 0 and 1 and are seeded in groups of threes so that values are consistent per training case, which helps increase weight adjustment speed.

```
Matrix weights0 = randomMatrix(3, 4);
Matrix weights1 = randomMatrix(4, 1);

public static Matrix randomMatrix(int row, int col)
{
    double[][] arr = new double[row][col];
    Random rando = new Random(3);
    for (int j = 0; j < row; j++)
    {
        for (int k = 0; k < col; k++)
        {
            arr[j][k] = rando.nextDouble();
        }
    }
```

[6] *Full program code can be found in Appendix B.*

```
        Matrix finalMat = new Matrix(arr);
        return finalMat;
    }
```

The learning inputs are passed through an iteration 100,000 times for training. In the first part of this loop, forward propagation occurs as the values of nodes in the hidden and output layers are calculated by multiplying and summing node and weight values through matrix dot products.

```
for (int i = 0; i < 100000; i++)
{
    Matrix nodes0 = input;
    Matrix nodes1 = sigmoidFunc(nodes0.times(weights0), true);
    Matrix nodes2 = sigmoidFunc(nodes1.times(weights1), true);
```

The sigmoid function is used as an activation function, which refers to the mathematical processing used. The sigmoid function is generally used because it always produces values between 0 and 1, which is useful for its nonlinearity and probability conversions; however, any other function can also be used (Iamtrask). If passed true, the function uses the regular sigmoid function; if passed false, it uses the derivative of the sigmoid function, *s(x)(1-s(x)),* in the form of *x(1-x)* given that values passed have already been acted on by the sigmoid function.

```
public static Matrix sigmoidFunc(Matrix mat, boolean bool)
{
    double[][] arr = mat.getArrayCopy();
    for (int j = 0; j < arr.length; j++)
    {
        for (int k = 0; k < arr[j].length; k++)
        {
            if (bool)
                arr[j][k] = 1.0 / (1.0 + Math.pow(Math.E, -1.0*arr[j][k]));
            else
                arr[j][k] = arr[j][k] * (1.0 - arr[j][k]);
        }
    }
    Matrix finalMat = new Matrix(arr);
    return finalMat;
}
```

The derivative function comes in use in the second half of the iteration, in which back propagation occurs. At the final and hidden layer of nodes, difference in errors are calculated by comparing the given results and the actual results, and the required change of the corresponding connection weights is calculated from the error using the

sigmoid derivations. This change of weights is then added to the weights themselves, and the process is repeated.

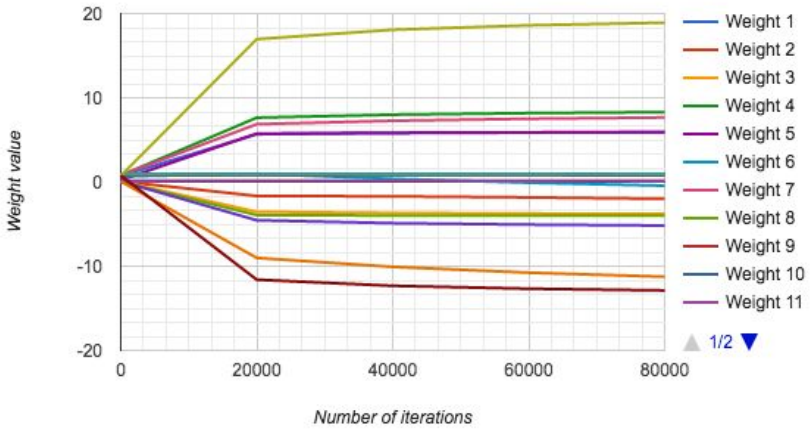
```
Matrix endingError = output.minus(nodes2);
Matrix endDeltaChange = endingError.arrayTimes(sigmoidFunc(nodes2, false));

Matrix midError = endDeltaChange.times(weights1.transpose());
Matrix midDeltaChange = midError.arrayTimes(sigmoidFunc(nodes1, false));

weights1.plusEquals(nodes1.transpose().times(endDeltaChange));
weights0.plusEquals(nodes0.transpose().times(midDeltaChange));
```

After the iterations teaching the neural network and adjusting its connective weights, the program should be ready to take new inputs and can correctly calculate results using the forward propagation.

```
double[][] testInputArray = {{0, 0, 1},
                             {0, 1, 1},
                             {1, 0, 1},
                             {1, 1, 1}};
Matrix testResults = sigmoidFunc(sigmoidFunc(inputTest.times(weights0),
true).times(weights1), true);
```

3.4 Test Results[7]

Graph 1: Change of connective weight values over 80,000 iterations

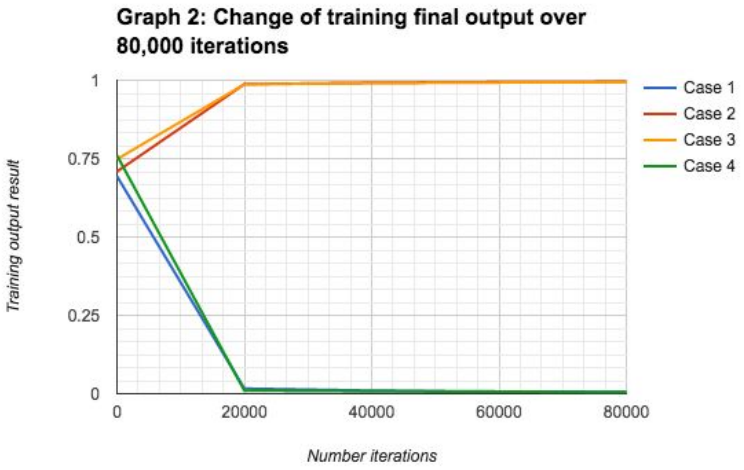


Test consistently run at 700-1000 milliseconds. They also reveal that the weights, at first randomly distributed between 0 and 1, will very quickly increase or decrease in the first few iterations and then plateau at an optimal value (Graph 1). Even after 80,000 iterations, the values are still changing, albeit very slightly. For example, weight 4 starts at 0.76816, jumps to 7.64241 by 20,000 iterations,

[7] *Full test results from neural network program found in Appendix D.*

and then continues to increase to 7.99536, 8.18452 and 8.30143. Values beyond the [0, 1] range are reached.

Tests also consistently reveal that the final node values increase or decrease very quickly in the first few iterations before plateauing to, but not entirely reaching, 0 or 1 (Graph 2). In Case 1 (0,0,0), the final output reaches 0.00553; in Case 2 (0,1,0), 0.99534; in Case 3 (1,0,0), 0.99409; and in Case 4 (1,1,0), 0.00533. Such inconsistencies also appear when running brand new, non-testing cases in the neural network: (0,0,1) produces 0.00232 rather than 0, (0,1,1) produces 0.98242 rather than 1, (1,0,1) produces 0.99005 rather than 1, and (1,1,1) produces 0.00274 rather than 0.



4. Analyzing rule-based intelligent assistant systems

4.1 Benefits

Because rule-based intelligent assistant systems are not heuristic, or self-learning, and can only draw conclusions based on whatever they already know, they can either make a correct response or a totally incorrect response. Occasionally this leads to the discovery of a unique solution: “they cannot advance the state of the art or reach novel conclusions (though they can often reach surprising conclusions that an expert might overlook)” (Desiano). However, it is likely that this phenomenon of an unintended yet accurate solution is relatively rare, considering the likelihood of concluding something

already not known, especially of the straightforward tasks that intelligent assistants work with.

4.2 Drawbacks

As previously mentioned, systems built on rules can very easily fail due to errors in its creation and lack of self-learning: “If rules are programmed incorrectly, the engine will reach incorrect conclusions or fail without warning and sometimes without indication” (Desiano). This makes rule-based programming an inefficient, faulty method to build assistants with; additionally, the process of building the assistant in the first place with the many rules that need to be accurate will be tedious. Furthermore, rule-based programming makes conversing with assistants and assistants’ responses stale and orderly, unlike a natural conversation with a real person, because they cannot learn to structure sentences in a variety of ways. The grammar and pronunciation would need to be impeccable in order for the assistant to recognize speech, and in turn, it would respond in pre-programmed ways to only select prompts, such as being able to say “What is the weather today?” but not “How is the weather looking?”

4.3 Evidence from code and tests

The results from running the XOR rule based code on the four given cases were exactly 0.0 or 1.0, proving that if programmed exactly, a rule-based program will produce very consistent and accurate results. However, if this case was extrapolated to solve not a two-step XOR case but rather a many-stepped case like recognizing speech and language structure, many more than two rules will need to be written. Additionally, since the monitor will have to decide which rule to use, it may have to check hundred or thousands of rules before arriving at the correct rule to send input data to, not just two as in the XOR code. This in turn increases the execution time of the program: the XOR code took 1 millisecond to monitor and check the rules for four cases, so for a language processing program that needs to check multiple cases of words spoken or different grammatical sections of a sentence, the many rules would increase the time greatly.

This is also evidenced by the y = kx function defining the maximum number of iterations for the program to individually check every value for every rule.

Rule based programming limits the capabilities of intelligent assistants by holding them to what already exists. Their lack of realism makes both their development and usage inefficient.

5. Analyzing neural network based intelligent assistant systems

5.1 Benefits

One of the major beneficial characteristics of neural networks is their use of heuristics, or approximations when exact answers cannot be found, for self-learning applications (Heuristics). Based on the growth and learning curve of human brains, these programs "can 'train' themselves to discover similarities and patterns in data, even when their human creators do not know the patterns exist" (Markoff 2014). Speech-based intelligent assistants can thus improve their speech and language structure recognition upon "practice" or exposure to procedures, leading to accuracy.

Creation of image recognition software is analogous to training speech-based assistants: one merely deals with pixels, while the other deals with sound wave data. In 2014, researchers from Google and Stanford University created a program that could recognize not just the objects within pictures, but also the actions being done within. The program was built on "weaving together" neural networks, one to recognize images and one to recognize typed language; the researchers then trained the program with annotated images, and tested their learning on new images. The result was that "the programs were able to identify objects and actions with roughly double the accuracy of earlier efforts, although still nowhere near human perception capabilities" (Markoff

2014). This type of training is called “supervised learning”, in which the creators who are initially “smarter” than the program trains it with set inputs and outputs until the internal weights are properly adjusted. Unsupervised and reinforcement learning will be the next steps, in which the program recognizes patterns itself or gauges its success based on effects of its outcomes so that program creators do not have to account for all possibilities in their code, but rather a general method of traversing and editing nodes and connections (Shiffman).

5.2 Drawbacks

Drawing directly from the image recognition software analogy just mentioned, the process of training neural networks is not easy, as the supervised training process requires many iterations, which produce mediocre results compared to superior human understanding. Also, these iterative tests are based on data that already exists, which may not be accurate enough. For example, one issue with the image recognition software training was that “billions of images and hours of video available online...are often poorly described and archived,” a problem that the program was trying to solve in the first place but ended up getting hindered by as the creators needed to find or come up with good test material (Markoff 2014). Presumably, speech samples of a variety of tone, clarity, and other numerous characteristics will also need to be found or freshly produced for speech based mobile assistants; data itself can thus become an obstacle.

Additionally, neural networks’ self-learning quality is not foolproof. Sometimes, a correct solution will be eliminated on accident, requiring a backtrack: “Once the system realizes that the correct answer was eliminated, it will retrace its steps, reversing the narrowing effect of each heuristic it used, until it finds the answer” (Desiano).

Finally, even if an intelligent assistant based on neural networks is successfully developed, the marketing and distribution aspect is another impediment. The revised

assistants will need to be incorporated in many mobile devices and other electronics either by software update or entirely new installations; additionally, these programs will need to heed to private companies and databases to acquire the information necessary to make the best decisions. For example, Viv Labs is creating Viv, a “more complex Siri” that writes its own programs and connects to databases to make multi-faceted decisions for everyday users. While Viv has succeeded in handling such requests, investigative journalist Steven Levy commented: “I think the big challenge really is going to be - able to bring all these other businesses in and get them interested in it. And the second big challenge is to get a position of prominence on people's devices there” (Inskeep).

5.3 Evidence from code and tests

The results of the neural network code were not exactly 1.0 or 0.0 as expected, but were within 0.02 of the desired output. This could have been a result of using the “long” Java type to store unrounded decimals for accuracy; however, it still reveals neural networks’ inexactness as the machine has to adjust by itself. This learning process depends heavily on the training input, which has to correspond to the correct “pattern”, in this case given by the XOR function, in order to lead to the correct weights. For example, since the third value in the cases was a disregarded element of the problem, all the cases in the training set had to have 0 as the last element, not 1’s or variations of 1’s, in case the network recognized a separate pattern from the XOR pattern.

Table 2: Third values that would have caused the neural network program to recognize a different pattern other than the desired XOR of first two values compared to correct third values

Final Desired Output	**Incorrect third values 1: Opposite correlation**	**Incorrect third values 2: Similar correlation**	**Correct third values 1: No correlation**	**Somewhat correct third values 2: No correlation**
0	1	0	0	1
1	0	1	0	1
1	0	1	0	1

0	1	0	0	1

In general, by 10,000-20,000 iterations the weights were accurate enough for producing output within 0.5 of the correct final output, but another 60,000 iterations was helpful in increasing the accuracy in noticeable amounts.These iterations in general took 7-10 seconds, an amount that can multiply greatly for neural networks of much larger magnitude and depth. However, the program code is relatively simple, relying on processes using matrix math and functions or derivatives applicable to all cases. The new input cases after the network was trained and properly weighted usually took the program a speedy 0-1 milliseconds to compute outputs for.

While neural networks provide an efficient and heuristic approach to speech-based intelligent assistants, their training and incorporation into society poses obstacles to its continued development in this field.

6. Conclusion

6.1 Code test results and extrapolation

In the creation of both programs for the XOR case, rule based programming was definitely more straightforward to use and more accurate compared to the neural network programming method, especially due to the lack of a need to “train”. However, if these methods were extrapolated to handle the more demanding speech and language recognition tasks of mobile intelligent assistants, the neural networks would fare much better because of its ability to handle many more non-programmed cases in its deep, “hidden” layers and its ability to create nodes and inputs itself after initial training, compared to rule-based programming need to have all grammatical rules and speech recognition pre-programmed.

This need for pre-programming such a large and complex amount of rules causes rule-based programming to become the more time-consuming method to create, as neural networks are created to build and adjust their layers themselves. Rule-based programs would also take more time not just to create, but also to run, since the monitor would need to check for all valid rules to apply on the input while neural networks can immediately feed input into their forward propagating activation functions. The one loss neural network based programs have versus rule based programs in time efficiency is that neural networks require initial periods of supervised training with very precise, correct data to establish a basis of connective weights, while the rules program is set for use right after coding is completed.

Finally, while rule based programs can be very exact, this property can also be its downfall in comparison to neural networks. In language processing, neural networks can become more flexible to and unaffected by trivial nuances in colloquial speech. In the case of mistakes, rule programs have no way of understanding what is wrong: it just is wrong. Meanwhile, neural networks can recognize their faults and backtrack through layers of nodes and connections using derivative math and inverted matrices while making adjustments to fix errors before reattempting to process a result.

6.2 Neural Networks prevail

Neural networks have several advantages over rule based programming in the building speech-based intelligent assistants. More efficient to develop, more accurate in its usage, and overall more powerful, neural networks provide the framework necessary to build programs capable of naturally interacting with humans, while rule based programming is too inefficient for creating intelligent assistants as it is more tailored to solving straightforward problems. However, it is important to note that neural networks still require much training and may pose ethical risks that will need to be addressed, including the coding of moral values and storage of identifying information within nodes and their connections.

References

Desiano, S. D. (n.d.). Expert systems. In *Computer sciences* (2nd ed.). Detroit, MI: Macmillan Reference USA.

Haaxma-Jurek, J. (2014). Artificial intelligence. In *The Gale Encyclopedia of Science* (5th ed.). Farmington Hills, MI: Gale. Retrieved April 28, 2016.

Heuristics. (2007). Retrieved March 15, 2016, from http://ic.galegroup.com/ic/scic

Hof, R. D. (2013). Deep Learning. *MIT Technology Review*. Retrieved April 28, 2016, from https://www.technologyreview.com/s/513696/deep-learning/

Iamtrask. (2015, July 12). *A Neural Network in 11 Lines of Python (Part 1).* Retrieved Sept. 6 2016 from http://iamtrask.github.io/2015/07/12/basic-python-network/

Inskeep, S. (Writer). (2014, August 12). Meet Viv, a more complex Siri [Transcript, Radio series episode]. In *Morning edition*. National Public Radio.

Markoff, J. (1988, May 15). IDEAS AND TRENDS: Can Machines Learn to Think? *The New York Times*.

Markoff, J. (2008, December 14). A software secretary that takes charge. *New York Times*.

Markoff, J. (2008, May 3). Pursuing the next level of artificial intelligence. *New York Times*. Retrieved March 15, 2016.

Markoff, J. (2014, November 18). Advance reported in content-recognition software. *New York Times*. Retrieved March 15, 2016.

Munro, P. (2013). Neural networks. In *Computer sciences* (2nd ed.). Detroit, Michigan: Macmillan Reference USA.

Oliver Selfridge. (2008, December 22). *The Telegraph*. Retrieved April 28, 2016, from http://www.telegraph.co.uk/news/obituaries/3903053/Oliver-Selfridge.html

Rojas, Raul. (1996). *Neural Networks - A Systematic Introduction.* New York: Springer-Verlag.

Rule Based Programming. (n.d.). Reading. Retrieved April 28, 2016, from https://inst.eecs.berkeley.edu/~selfpace/studyguide/3S.rdngs/rule-based.prog.pdf

Shiffman, D. (2012). Neural Networks. *The Nature of Code.* Retrieved Sept. 6, 2016 from http://natureofcode.com/book/chapter-10-neural-networks/

Siri. (2016). Retrieved April 28, 2016, from https://www.sri.com/work/timeline-innovation/timeline.php?timeline=computing-digital#!&innovation=siri

Appendix A: Rule-based program code

```
import java.util.*;
import java.lang.*;
/**
 * @Victoria Juan
 * @September 2016
 * @Written for IB Extended Essay: To what extent do neural networks facilitate speech-based
mobile
 * intelligent assistants better than rule based programming?
 */
public class RuleBasedExample
{
    public static int finalState; //global value to track current state of program

    //rule monitor: if first element is 0, send to xorRules0, else first element is 1, so send
to xorRules1
    public static void monitor(double[] arr)
    {
        if (arr[0] == 0)
            xorRules0(arr);
        else
            xorRules1(arr);
    }

    //xorRules0: decides state based on array's second value with given first value as 0
    public static void xorRules0(double[] arr)
    {
        if (arr[1] == 0)
            finalState = 0;
        else
            finalState = 1;
    }

    //xorRules1: decides state based on array's second value with given first value as 1
    public static void xorRules1(double[] arr)
    {
        if (arr[1] == 0)
            finalState = 1;
        else
            finalState = 0;
    }

    //method to print array
    public static void printArr(double[][] arr)
    {
        for (int i = 0; i < arr.length; i++)
            System.out.println(arr[i][0]);
    }
```

```
    public static void main(String[] args)
    {
        //test rules (no training required)
        double[][] inputArray = {{0, 0, 0},
                                 {0, 1, 0},
                                 {1, 0, 0},
                                 {1, 1, 0}};
        double[][] outputArray = new double[4][1];

        final long startTime = System.currentTimeMillis(); //begin timer
        //for each set of input values in array, send to monitor to decide rules
        for (int i = 0; i < inputArray.length; i++)
        {
            monitor(inputArray[i]);
            outputArray[i][0] = finalState;
        }

        printArr(outputArray);
        final long endTime = System.currentTimeMillis();

        System.out.println("Total testing execution time: " + (endTime - startTime) );
    }
}
```

Appendix B: Neural network program code

```
import Jama.Matrix; //Java Matrix library from http://math.nist.gov/javanumerics/jama/
import java.util.*;
import java.lang.*;

/**
 * @Victoria Juan
 * @September 2016
 * @Created for IB Extended Essay: To what extent do neural networks facilitate speech-based
mobile
 * intelligent assistants better than rule based programming?
 */
public class NeuralNetExample
{
    //Sigmoid function. Nested for loops iterate through each matrix value. If passed true,
use sigmoid
    //function; if false, use derivative in which the input is already the sigmoid output.
    public static Matrix sigmoidFunc(Matrix mat, boolean bool)
    {
        double[][] arr = mat.getArrayCopy();
        for (int j = 0; j < arr.length; j++)
        {
            for (int k = 0; k < arr[j].length; k++)
            {
                if (bool)
                    arr[j][k] = 1.0 / (1.0 + Math.pow(Math.E, -1.0 * arr[j][k]));
                else
                    arr[j][k] = arr[j][k] * (1.0 - arr[j][k]);
            }
        }
        Matrix finalMat = new Matrix(arr);
        return finalMat;
    }

    //Random matrix function. Generates a random matrix of values between 0 and 1 for node
connection
    //weights. Seeded in multiples of 3 for consistency between separate cases.
    public static Matrix randomMatrix(int row, int col)
    {
        double[][] arr = new double[row][col];
        Random rando = new Random(3);
        for (int j = 0; j < row; j++)
        {
            for (int k = 0; k < col; k++)
            {
                arr[j][k] = rando.nextDouble();
            }
        }
```

```
        Matrix finalMat = new Matrix(arr);
        return finalMat;
    }

    public static void main(String[] args)
    {
        //increases learning speed of backpropogation
        double learningRate = 1.2;
        //inputArray used to test and set neural network
        double[][] inputArray = {{0, 0, 0},
                                 {0, 1, 0},
                                 {1, 0, 0},
                                 {1, 1, 0}};
        //outputArray used to store results for neural network learning
        double[][] outputArray = {{0},
                                  {1},
                                  {1},
                                  {0}};
        Matrix input = new Matrix(inputArray);
        Matrix output = new Matrix(outputArray);

        //matrices with weights between initial and hidden layer/hidden and output layer
        Matrix weights0 = randomMatrix(3, 4);
        Matrix weights1 = randomMatrix(4, 1);

        final long startTime = System.currentTimeMillis(); //begins learning timer
        for (int i = 0; i < 100000; i++)
        {
            //begin forward propogation
            Matrix nodes0 = input;
            //hidden layer values: sigmoid function of dot product of input values and
connection weights
            Matrix nodes1 = sigmoidFunc(nodes0.times(weights0), true);
            //final output values: sigmoid function of dot product hidden values and
connection weights
            Matrix nodes2 = sigmoidFunc(nodes1.times(weights1), true);

            //final error: difference between intended output and actual results
            Matrix endingError = output.minus(nodes2);
            //necessary change to end weights: scalar product of error and sigmoid derivatives
of final results
            Matrix endDeltaChange = endingError.arrayTimes(sigmoidFunc(nodes2, false));

            //hidden layer error: dot product of necessary change to end weights and ending
weights
            Matrix midError = endDeltaChange.times(weights1.transpose());
            //necessary change to mid weights: scalar product of hidden layer error and
sigmoid derivatives of hidden results
            Matrix midDeltaChange = midError.arrayTimes(sigmoidFunc(nodes1, false));
```

```
            //make changes to weights by adding dot product of nores and necessary change
            weights1.plusEquals(nodes1.transpose().times(endDeltaChange));
            weights0.plusEquals(nodes0.transpose().times(midDeltaChange));

            //print ending node results every 10,000 iterations
            if (i % 10000 == 0)
            {
                nodes2.print(7, 5);
            }
        }
        final long endTime = System.currentTimeMillis(); //end learning timer
        System.out.println("Total learning execution time: " + (endTime - startTime) );

        //test neural network with new inputs
        double[][] testInputArray = {{0, 0, 1},
                                     {0, 1, 1},
                                     {1, 0, 1},
                                     {1, 1, 1}};
        Matrix inputTest = new Matrix(testInputArray);

        final long startTime1 = System.currentTimeMillis();
        //calculate and print results for test values
        Matrix testResults = sigmoidFunc(sigmoidFunc(inputTest.times(weights0),
true).times(weights1), true);
        testResults.print(7, 5);
        final long endTime1 = System.currentTimeMillis();

        System.out.println("Total testing execution time: " + (endTime1 - startTime1) );
    }
}
```

Appendix C: Complete neural network test data

Iterations		0	20,000	40,000	60,000	80,000
Input-to-hidden weights	Weight 1	0.73106	5.67369	5.80186	5.86981	5.91286
	Weight 2	0.07099	-1.66136	-1.70394	-1.83165	-1.99386
	Weight 3	0.06712	-3.54320	-3.67707	-3.74542	-3.79243
	Weight 4	0.76816	7.64241	7.99536	8.18452	8.30143
	Weight 5	0.22733	5.74862	5.84960	5.89628	5.92353
	Weight 6	0.66032	0.97443	0.37069	-0.08183	-0.44009
	Weight 7	0.80667	6.86520	7.27380	7.50966	7.66277
	Weight 8	0.02982	-3.92802	-3.98525	-3.99443	-3.99714
	Weight 9	0.81117	0.81117	0.81117	0.81117	0.81117
	Weight 10	0.94542	0.94542	0.94542	0.94542	0.94542
	Weight 11	0.15273	0.15273	0.15273	0.15273	0.15273
	Weight 12	0.91028	0.91028	0.91028	0.91028	0.91028
Hidden-to-output weights	Weight 13	0.73106	16.96619	18.07930	18.60800	18.92157
	Weight 14	0.07099	-4.54597	-4.89156	-5.06160	-5.17271
	Weight 15	0.06712	-9.02079	-10.07901	-10.76833	-11.25455
	Weight 16	0.76816	-11.60388	-12.33080	-12.67600	-12.87984
Final output node values	Case 1	0.69395	0.01627	0.00984	0.00704	0.00553
	Case 2	0.70887	0.98759	0.99213	0.99416	0.99534
	Case 3	0.74832	0.98701	0.99095	0.99287	0.99409
	Case 4	0.76013	0.01005	0.00767	0.00634	0.00533
Total learning execution time						999 millisec
Testing output node values					Case 1	0.00232
					Case 2	0.98242
					Case 3	0.99005
					Case 4	0.00274