Can my computer guess the language of a word?

Matthew Beardwell

CONTENTS

Learner Name	**Matthew Charles Beardwell**	**Candidate number**	**3010**
Centre Name	**King’s College London Mathematics School**	**Centre Number**	**10953**
Unit Name	**Can my computer guess the language of a word?**	**Unit Number**	**P304**

Introduction

I would like to test whether there is an association between the structure of a word and what language it belongs to. I will be investigating the accuracy of using a feedforward neural network to guess the language of a word – in this case it will be out of a selection of six languages. I believe that if there is a much higher than chance accuracy then this suggests that there is an association.

I will change factors such as the number of hidden layers and the number of nodes in those hidden layers to better the network's guess. I will also test my network with one to three hidden layers and for each I will use 40 to 200 neurons for each hidden layer in increments of 40. Differently shaped networks will give me different accuracies and I hope that testing many types will give me a higher chance of finding the best one. I will run each of the 15 networks with 20 iterations of the training set and, in each iteration, I will run the test set of data through it to get the overall accuracy of the network on the test set at each step. Training for these many iterations should give me a higher accuracy as it has had longer to learn. I am calculating the accuracy of the network on data it has not used so that we can see when it begins to overfit.

Acquiring skills

In this project, I will need to learn how these four main backpropagation formulae work and how to implement them.

Summary: the equations of backpropagation

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \qquad \text{(BP1)}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \qquad \text{(BP2)}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \qquad \text{(BP3)}$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \qquad \text{(BP4)}$$

Figure 1 [1]

I will also need to learn how a feedforward neural network works and how to represent it in Python 3.6.4. I will also need to learn how to train my network and some other smaller details such as writing results to different file types and accessing and making folders.

Terminology

Please note the abbreviations I will use in this document. Where X stands for any number, "Xn" means that there is X neurons per hidden layer in the network and "Xh" means that the network has X hidden layers. When used in conjunction, e.g. "XnYh", it refers to a network that has Y hidden layers. Each of those hidden layers has X neurons.

When I use the word ***overfitting*** [2], I am referring to when a network becomes so good at guessing the language of words in its own set of words it has been trained on, it is basically just memorising them and so the network won't be general enough to guess words it has never seen before.

Backpropagation refers to an algorithm in which the network is permitted to see the true language of a word then changes all its values it's using for its weights and biases slightly so that if it were to guess again, it would be slightly more correct. This uses some calculus. [3]

[N.B. Anything uncited was known prior to research from experience in programming.]

Literature Review

How to implement the Backpropagation Algorithm from Scratch in Python [4] was written as a tutorial on implementing backpropagation in a feed-forward neural network in Python. It gives a written explanation of his code which features forward-propagation, network training, and 'how to apply the backpropagation algorithm to a real-world predictive modelling problem'. The blog has a relatively good reputation - fifty thousand or more applications to receive his free 'crash-course' and good reviews from skilled professionals in the field. He runs the blog alone apparently to support the claim of the quality of his crash-course and to publish his ideas as a sort of professional portfolio and 'how-to' guide. He runs the blog without any apparent sponsors and there is also no reason to suspect a lack of neutrality. He is a professional developer and has higher degrees in Artificial Intelligence.

I have used this source as an example of what my neural network may look like and to practice simplifying and editing so that I might be comfortable with the mathematics of the algorithms. I have also read it to gain a deeper understanding of how to lay my code out and what functions and forms of data storage I should use. For example, the main structure is as follows:

1. Initialize Network
2. Forward Propagate
3. Back Propagate Error
4. Train Network
5. Predict

A Neural Network In 11 Lines of Python (Part 1) [5] gives Trask, himself, the task of trying to rewrite his code from memory. I found this a very useful task as I began to understand why he implemented things such as the reshaping of the weights and biases at the beginning by doubling all their values and subtracting one (This was because they had random values in the range 0 to 1 and we wanted random weights and biases in the range -1 to 1). I ended up using his code as inspiration for the structure of my own such as initialising the network then iteratively updating the weights and biases after a 'forward propagation'.

The writer of the blog claims to be a student from the University of Oxford, a research scientist at Google Deepmind, author of a 'Deep Learning' book, and a 'Deep Learning' instructor on Udacity. Given these qualifications, I would expect the source to be accurate, precise and written from experience. His blog is a page on Github, so it doesn't have obvious sponsors or an excuse to be biased.

Neural Networks with Backpropagation for XOR Using One Hidden Layer [3] is a simple neural network tutorial that goes into depth about the mathematics of the process which has helped me to understand but also formulate solutions for my program. For example, I could update the weights so that my network could make better predictions by using the formula he writes as:

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)}$$

I mainly used this site for the matrix calculus he writes as I trust its accuracy and from my own understanding, it seems correct. After implementing it in my code, I can verify that my network became better at predicting over time in my second program.

Hong has studied at Seoul National University, Carnegie Mellon University, and UC Berkeley and so is at least qualified to write this article. His PayPal donate button on the page suggests that he runs the page for fun but for him to devote as much time as possible to it and to make it free for everyone, he would like to be paid in some amount. This might suggest a lack of quality as he may have a high value set on his own time and would have written these articles more hastily than other writers would have. I would argue, given his qualifications, that any amount of his knowledge that is shared would be helpful and accurate - otherwise he wouldn't have bothered wasting his time writing it. He conveys his complex ideas simply with large diagrams and bolding keywords to make his writing easier to read. He has also ensured paragraphs are short and minimal mathematics is written.

Neural Network Back-Propagation Using Python [6] uses another example of a feedforward neural network. Having a different variant of it being used for a different purpose helps me not only understand and pickup various small things that I might want to use in my own program but also, in general, what I should be thinking when beginning to build a neural network and where I should be starting. Here I interpreted that a good place to start would be to lay down the foundation by writing out the main algorithm before even having defined the functions I'm using. This is like me saying what I want to happen, then working backwards through all the tiny details to make sure it does. The magazine isn't too popular (its Twitter account has about eight thousand followers at the time of writing) but doesn't necessarily have a poor reputation. The writer has spent four years at Microsoft as a Software Design engineer, another seven years as a Software Research Engineer, and has a Master's in Computer Science and a Bachelor's in Mathematics which makes him at least overqualified for writing the article, not to mention his Ph.D. in cognitive psychology and computational statistics. I see no reason for vested interest or a lack of neutrality and the code is available for free download.

A Step by Step Backpropagation Example [7] is an article from mattmazur.com that is probably my favourite from those that I have researched because it includes a visual interactive animation of the network working on a link from 'emergentmind' on the page and the calculations he makes includes actual data - it is less theoretical. This makes it easier to understand the mathematics he's using. He is, at the time of writing, a Data Scientist at Help Scout and the mathematics he has written gives me confidence in his expertise. This article seems to be a publication of his ideas and I can see no way in which he might be using his blog to profit. A blog of this type is usually made as a portfolio of one's knowledge and is very valuable in landing a higher paying job at large companies and would give him a wider selection of career paths. I would assume he would put as much effort into each article as possible to ensure there are no mistakes and that they are mathematically and logically accurate.

Predicting Languages with Racist Neural Networks [8] is a video on YouTube that I found from an animator and student programmer who had attempted to build a single hidden layer neural network from scratch for the first time. You input words up to 15 letters long and it outputs its confidences of the word being 12 different languages. I especially like the way he made dynamic graphics to represent the mathematics behind the code being run in real-time. He has no reputation and is only a year or two older than me but he does study at Stanford University. It is also apparently his first attempt at implementing calculus in machine learning rather than just using a library's built-in high-level functions. Nevertheless, his program appears to work on exhibition and the accuracy increases through each iteration to the 90-100% range. I trust that he is a good programmer given that I have watched many more machine learning videos of his. One includes a well-built evolution simulator that builds a character to complete a certain task such as jumping the highest. Not only does he display his knowledge into machine learning after posting this video, but he also describes in a follow-up video the things he could do better if he were to do it again.

After considerable research into neural networks and the calculus involved, it also seems that I can take some ideas of his as they are an improvement to some that I had in mind. For example, the format the word is input as. I may also improve upon some of his ideas. The other sources have far more experience behind them, but I will purpose my neural network that inputs words to guess their language like Huang's as he could display it working with a high accuracy.

Neural Networks and Deep Learning - Chapter 2 [1] is written by Michael Nielson who claims to be a "scientist, writer, and programmer" and is a research fellow at the 'Y combinator' research lab. His most important accomplishment, from this angle, is that he has written a lot of material on mathematics and programming with AI. He has written the books "Neural Networks and Deep learning", "Reinventing Discovery: The New Era of Networked Science", and "Quantum Computation and Quantum Information". This allows me to ignore any other credentials he has as it is obvious here that he is well educated on the topics he teaches.

This book is only six chapters but the last chapter, for example, has a little under 20,000 words. This is enough material to suggest to me that he has put a lot of time and effort into researching and fact checking it. This is also backed up by the Creative Commons License symbol he's placed at the bottom – indicating he'd rather not have this material used without his permission which is uncommon for journalists writing articles. He says that he has written it as an educational resource which includes interactive resources. I can think of few incentives to write a non-fiction educational book and release it online for free other than for the fun of it and to act as a sort of professional portfolio.

I used this resource to get a grasp on the mathematics of the backpropagation algorithm and for the proofs of them. This helped me to implement them correctly as I knew what they did and if there was any error in my code, it would be easier to fix it.

Methodology and Design Principles

Linking back to my Literature Review, this EPQ was mostly made using the articles listed previously. Brownlee's layout [4], Hong's calculus [3] and McCaffrey's [6] use of classes supplied the theory for most of the body of my code.

To build the network, I started by practicing programming a feedforward neural network that would also take words as input so that when I came to program the language-guesser I wouldn't have to change the functions by much. I chose a feed-forward neural network because the first network I happened to learn about was a feedforward neural network on a YouTube video by Siraj Raval [9]. It also seemed the easiest to program. The network would predict the difficulty of a word in Hangman – i.e. the number of guesses the computer will take to get it right. To get this data, I ran each word through a prebuilt program I had made which I believed to be optimal at guessing the word in Hangman. It would return the difficulty of the word, so I could make a large data set after downloading a large English dictionary. The program in fact appeared to not be able to predict the output which suggested to me that the number of guesses it would take the computer was random and had a mean that the 'number of guesses' centred around. This was not a failure from the perspective of my Project Proposal as I still had one other option – language guessing.

I soon later found a video by an animator and student programmer on a YouTube channel called 'carykh' [8] who had written a program, which he represented graphically, that would take a word as an input and guess whether it was one of two languages (although he designed it so that he could theoretically have the program guessing between up to twelve). He only had one hidden layer and his review on it could only be limited in a short video. In this case, I decided that I would use his idea as an inspiration for my own investigation into guessing the language of a word. I came up with some design principles which I had gathered and investigated the benefits for from various sources, including Cary Huang's own video, and altered my original program to gather results and make conclusions.

Luckily, this became a natural 'Segway' from programming the Hangman game to guessing the language of a given word which I had planned for in my brief.

Language

I chose to write the program in Python 3.6.4 because this is the one I know the most about. It was also helpful that most network tutorials were programmed in Python. Python has libraries such as 'TensorFlow' which allow you to do this whole process much better and more efficiently, but I felt like I wanted to work through the equations myself, so I could get a better understanding of why they worked.

A feedforward neural network

It isn't worth me writing an extensive article on how a feedforward neural network works but looking at the diagram below will help summarise it.

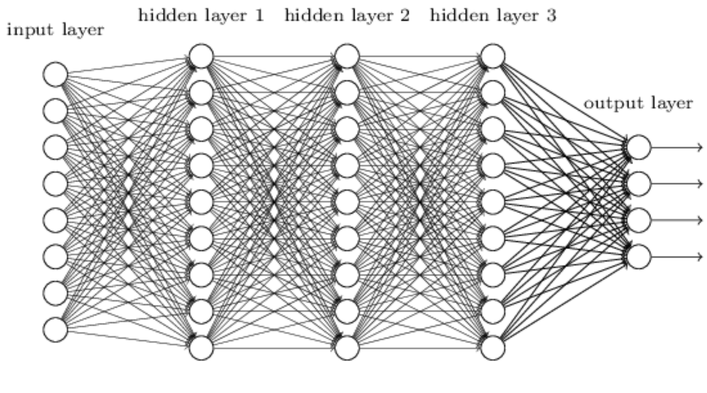
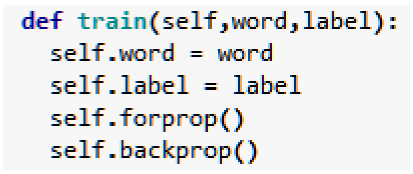


Figure 2 [10]

As you can see, the input is a list of numbers and the output is another list of numbers. The arrows represent the propagation of this “signal” through to the output. This network essentially works as a function – it takes a vector of numbers and outputs another vector. Each node of a hidden layer multiplies all the different weights along the arrows by the nodes at the end of the arrows in the previous layer. It adds these up and adds another number to it called a bias then passes it into a non-linear activation function. The non-linear function I used was the sigmoid function. This then becomes the value of that single node. This happens to all nodes in a layer. It is essentially a mix of multiplying and adding with the previous layer to get the next one. To make the network better at guessing, we compare our output with the real one that we know it is supposed to be and use calculus to pass the difference backwards through the network to all the weights and biases to change them slightly so that the output is close to the known one. This is called ‘backpropagation’. [1] [3] [4] [6] [7]

I chose a feedforward neural network as the machine learning technique to implement because it might be the one that has the simplest mathematics and the simplest structure. I didn’t want to use a machine learning library to do all the maths for me, so I will choose to manually program the network. I will only use some help from the ‘NumPy’ module to multiply my matrices and add them.

After finishing the program, this was the resulting training function:



```
def train(self,word,label):
  self.word = word
  self.label = label
  self.forprop()
  self.backprop()
```

Ignoring the ‘self’ keywords, every time I want to the train the network on a single word, I can input the word and its label (language) and update the weight’s and biases of the network by running the forward the backwards propagation algorithm.

I have not included these functions as they are just Python implementations of someone else’s algorithm and they prove to be even harder to explain than their algebraic form.

Inputs

Because of the mathematical nature of machine learning, I will have to turn my word into some list of numbers so that I can put it into the network. I am going to use one-hot encoding for this after trialling some different methods as I learnt it was one of the better methods from a Cary Huang video [8]. Instead of a number between 1 and 26 for each input letter, which would indicate letters close together in the alphabet would be close together in value, it would instead take in 26 digit array for each letter where all digits would be 0, apart from a 1 at the position where the letter is in the alphabet (i.e. a = [1,0,0,0,0,0,…,0], b = [0,1,0,0,0,0,…,0], etc.). If the network looked at the letter ‘l’ without one-hot encoding [11], it would look at it as it would similarly to the letter ‘j’ as they are only two letters apart in the alphabet whereas the letter ‘j’ is much rarer than ‘l’ in some languages. This would be a problem as the distance of letters in our alphabet is almost completely useless information here as the alphabet is a random ordering.

Instead, the network would be learning the patterns of zeroes and ones which would be far more accurate than giving it a vector of the letters’ scalar values (i.e. ‘14’ for ‘n’). In this model, the network not only cares about the position of each letter, but it treats each letter as though it could be 40% of ‘a’, 20% of ‘c’, 30% of ‘e’, and 10% of ‘f’ but instead we say that a letter is e.g. 100% an ‘a’ and 0% all of the other letters. Because of this, each letter is treated as having its own place and distinct “meaning” such that the network can learn the patterns of rarity of each letter type in each position in a word. One downside to this is that it might require more computing power since I will be manipulating 390 inputs (if the words are limited at 15 letters long) and a lot more weights and biases.

Below I have represented the encoding of the German word ‘anderungsstand’. The black cells are ones and the white, zeroes. The array of ones and zeroes are placed end-to-end for all the letters and then some more zeroes are added on so that the length of the input stays the same for all words that could be input. Essentially, the network would be looking at a flattened version of this image where some pixels are black, and some are white, and deciding what language the word is out of six possible options.

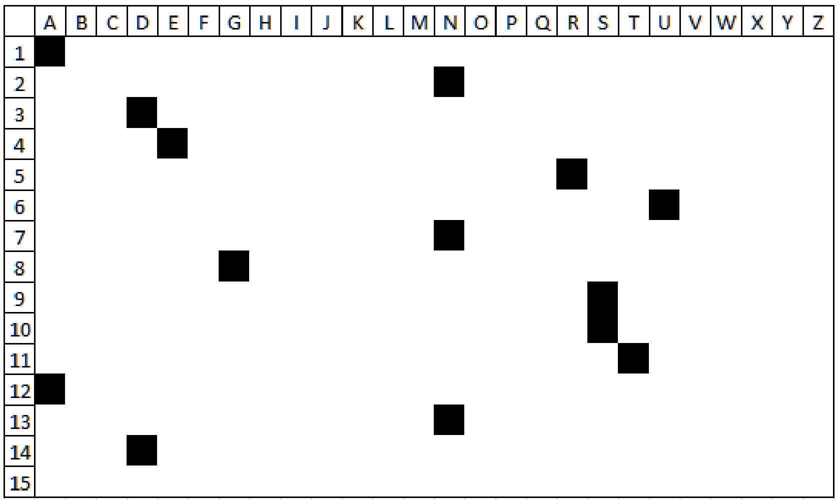


Figure 3

In this encoding, the same shapes of black pixels, just moved vertically up and down in the ‘picture’ could represent word endings such as ‘-ness’ which to the English reader would suggest that the word is English. Unfortunately, an ‘-ness’ in the letter positions 4 to 7 is completely different in the programs eyes to ‘-ness’ in the letter positions 5 to 8 which is one drawback to inputting the word like this.

This is the surprisingly short code for this section. I have included explanations.

```
def word2vec(a,N):           # a is the word, N is the max # of the letters long
                             # returns vector with 26*N dimensions (eg. if N = 15 then size = 390)
  vec = []
  if len(a) < N:             # if the word is short enough
    a = a + "|"*(N-len(a))   # add spaces to the end so it fills up N letters
  for i in a:
    vec += letter2vec(i)     # convert letters to eg. [0,0,1,0..] and put them all together
  return np.array(vec)

def letter2vec(l):
  letterPattern = [[0]]*26                # ie. [0,0,0,0,0,0,..]
  if l != "|":                            # if it's a letter and not an empty space
    letterPattern[ord(l.lower())-97] = [1] # change letter position in [0,0,0,..] to 1 from 0
  return letterPattern                    # return eg. [0,0,0,1,0,..]
```

The code works but the odd-looking ‘97’ included and the obscure variable names could be changed.

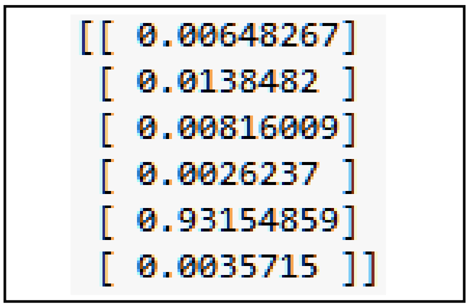
Testing

I am going to test the network with hidden layers of 40, 80, 120, 160, and 200 nodes. For each I will test the network with one, two and three hidden layers. This will help me decide how best to design the network to maximise my guess accuracy. Using a range of various networks will give me a better idea of what changes I could make to make the network perform better (e.g. If I see a positive association between the number of nodes and the network's accuracy, I may choose to keep increasing the number of nodes to better the accuracy). I will graph the accuracy of the network over each iteration and see which network performs the best. I will run various types of the same network through 20 iterations of training and for each iteration, I will test the network on 90,000 words (15,000 for each language) as a test set of data that the network has not used to improve. Each iteration uses the same test set and training set. This is 300 iterations with 120,000 words training and 90,000 words being tested for every single iteration and with most of the networks being sufficiently large to be significantly slower than a 1h. Running these large training and test sets 20 times through 15 different networks will be very time consuming so I will run the program overnight for as many nights as necessary.

To collect the test set, I ended up looking up large word dictionaries only for French, German, Latin, Italian, a version of Japanese and Greek using only our 26-letter alphabet. I wrote a program that formatted these, mixed them and put the majority in one file and a smaller number in another. The larger became my training set and the smaller became my test set. The program opened all dictionaries at once and, at each step, picked a random word from all six, turned the language it belonged to into a number and wrote it into the file as a line looking like, e.g. "3,anderungsstand". This made it easier to read the word and language pair from the file again. The dictionary for Latin had far fewer words in it which limited the number of data pairs I could run through the network for all the languages as I didn't want an imbalance on the number of words for each language being used. This meant that I could run the 20 iterations of training and testing faster but it may have also made my network less accurate as it has seen less data for each language. This only took one afternoon and so I would meet the deadline I set for myself in the brief.

Outputs

The network will take the encoded word or 'image' in and return six numbers out. These will be its confidence that it is each language. The one with the highest confidence will be the prediction. If guessing between six languages becomes very difficult – i.e. the accuracy is close to random luck – then I will incrementally decrease the number of languages it is guessing from.



```
[[ 0.00648267]
 [ 0.0138482 ]
 [ 0.00816009]
 [ 0.0026237 ]
 [ 0.93154859]
 [ 0.0035715 ]]
```

Figure 4: Example output suggesting Japanese (on the fifth row) with a 96.4% confidence

Points to note after finishing the program

An important parameter to decide, which I had not covered, was the learning rate – how big the changes to the network are allowed to be. I did not know this would be an issue and after research, I can find few sources on the explanation of the following problem hence the lack of citations. Making the following changes saw the issue disappear so in my opinion, the size of the learning rate was what was causing the problem.

I assumed that if this number was small, i.e. I used 0.01, then there would be no issues and it would converge quickly. Unfortunately, as you can see here, this learning rate was too high and although it gave me very high accuracies at some points, its accuracy would oscillate as the training went on. To prevent this, I changed the learning rate to 0.001 which gave smooth curves, unlike the example of an 80n1h network training below.

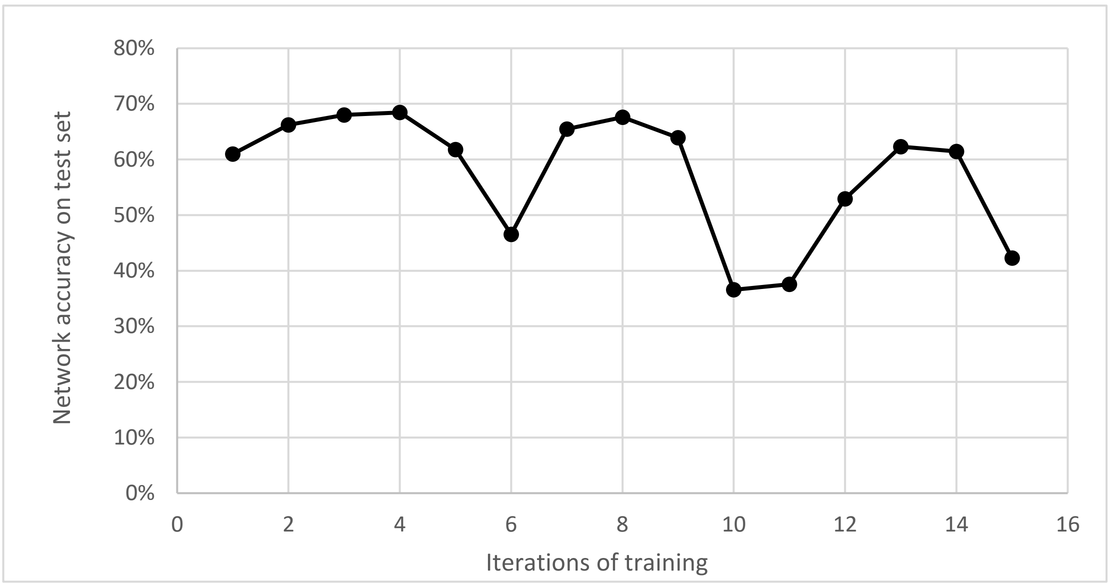
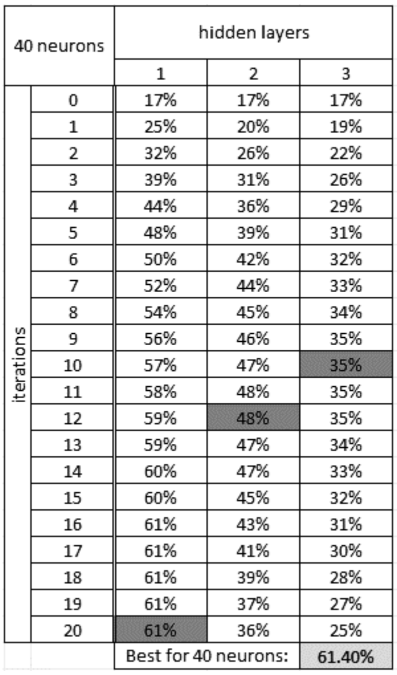


Figure 5

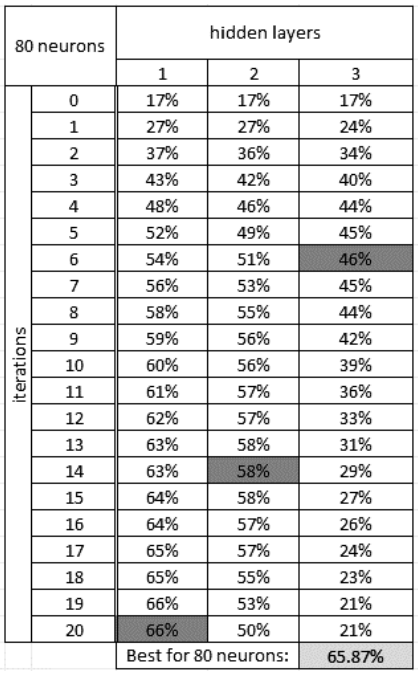
I had allocated two weeks for debugging in my brief, so this error had not caused too many issues. At this stage, I had hoped the backpropagation algorithm would be fully functional and since I had hit this dead-end, I thought I wouldn't meet my objective. Issues with debugging is the key factor in becoming a month late for my second milestone deadline. Nevertheless, making the change fixed the network and I was able to continue to the data collection.

Presentation of Findings

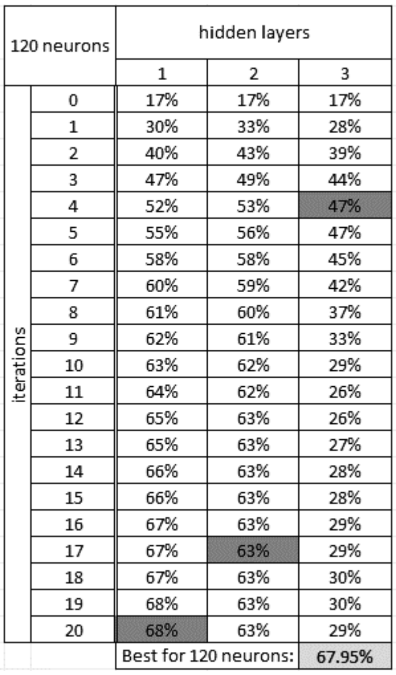
The Data



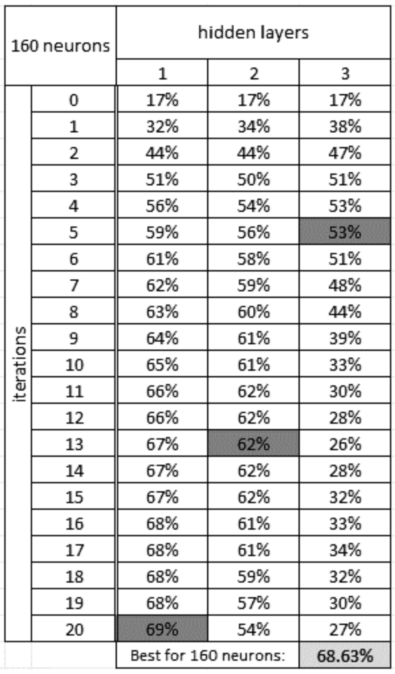
40 neurons		hidden layers		
		1	2	3
iterations	0	17%	17%	17%
	1	25%	20%	19%
	2	32%	26%	22%
	3	39%	31%	26%
	4	44%	36%	29%
	5	48%	39%	31%
	6	50%	42%	32%
	7	52%	44%	33%
	8	54%	45%	34%
	9	56%	46%	35%
	10	57%	47%	35%
	11	58%	48%	35%
	12	59%	48%	35%
	13	59%	47%	34%
	14	60%	47%	33%
	15	60%	45%	32%
	16	61%	43%	31%
	17	61%	41%	30%
	18	61%	39%	28%
	19	61%	37%	27%
	20	61%	36%	25%
		Best for 40 neurons:		61.40%



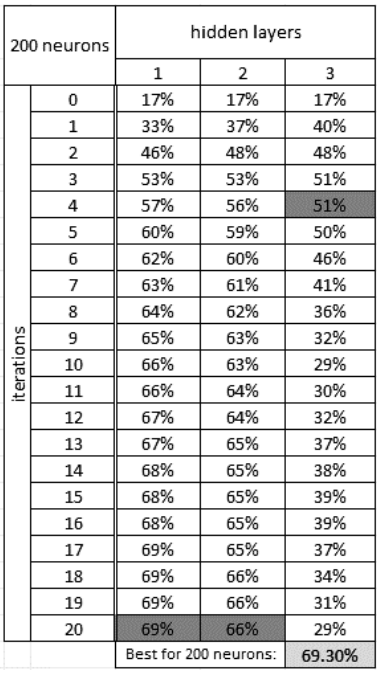
80 neurons		hidden layers		
		1	2	3
iterations	0	17%	17%	17%
	1	27%	27%	24%
	2	37%	36%	34%
	3	43%	42%	40%
	4	48%	46%	44%
	5	52%	49%	45%
	6	54%	51%	46%
	7	56%	53%	45%
	8	58%	55%	44%
	9	59%	56%	42%
	10	60%	56%	39%
	11	61%	57%	36%
	12	62%	57%	33%
	13	63%	58%	31%
	14	63%	58%	29%
	15	64%	58%	27%
	16	64%	57%	26%
	17	65%	57%	24%
	18	65%	55%	23%
	19	66%	53%	21%
	20	66%	50%	21%
		Best for 80 neurons:		65.87%



120 neurons		hidden layers		
		1	2	3
iterations	0	17%	17%	17%
	1	30%	33%	28%
	2	40%	43%	39%
	3	47%	49%	44%
	4	52%	53%	47%
	5	55%	56%	47%
	6	58%	58%	45%
	7	60%	59%	42%
	8	61%	60%	37%
	9	62%	61%	33%
	10	63%	62%	29%
	11	64%	62%	26%
	12	65%	63%	26%
	13	65%	63%	27%
	14	66%	63%	28%
	15	66%	63%	28%
	16	67%	63%	29%
	17	67%	63%	29%
	18	67%	63%	30%
	19	68%	63%	30%
	20	68%	63%	29%
		Best for 120 neurons:		67.95%



160 neurons		hidden layers		
		1	2	3
iterations	0	17%	17%	17%
	1	32%	34%	38%
	2	44%	44%	47%
	3	51%	50%	51%
	4	56%	54%	53%
	5	59%	56%	53%
	6	61%	58%	51%
	7	62%	59%	48%
	8	63%	60%	44%
	9	64%	61%	39%
	10	65%	61%	33%
	11	66%	62%	30%
	12	66%	62%	28%
	13	67%	62%	26%
	14	67%	62%	28%
	15	67%	62%	32%
	16	68%	61%	33%
	17	68%	61%	34%
	18	68%	59%	32%
	19	68%	57%	30%
	20	69%	54%	27%
		Best for 160 neurons:		68.63%

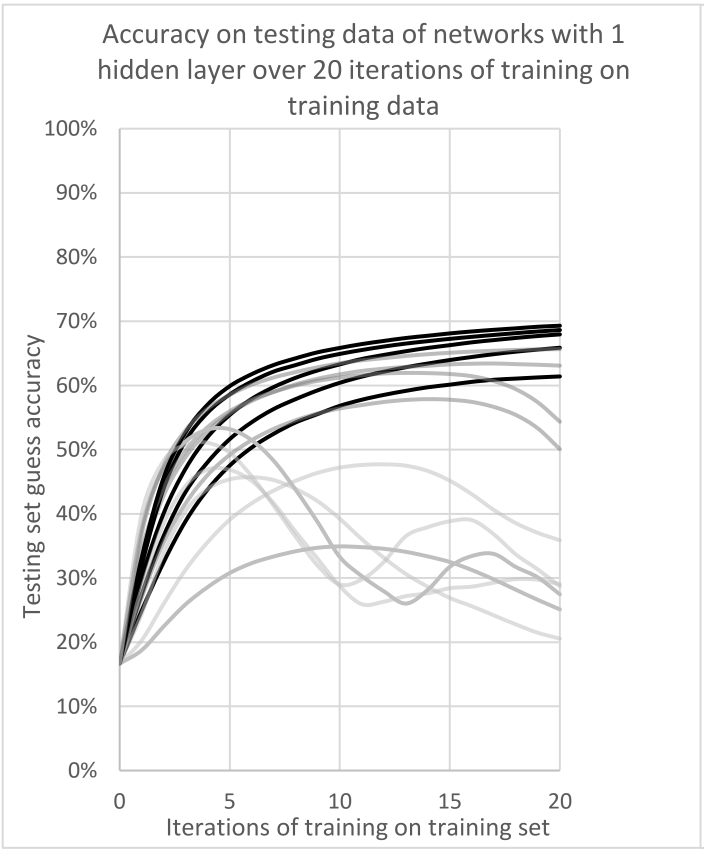


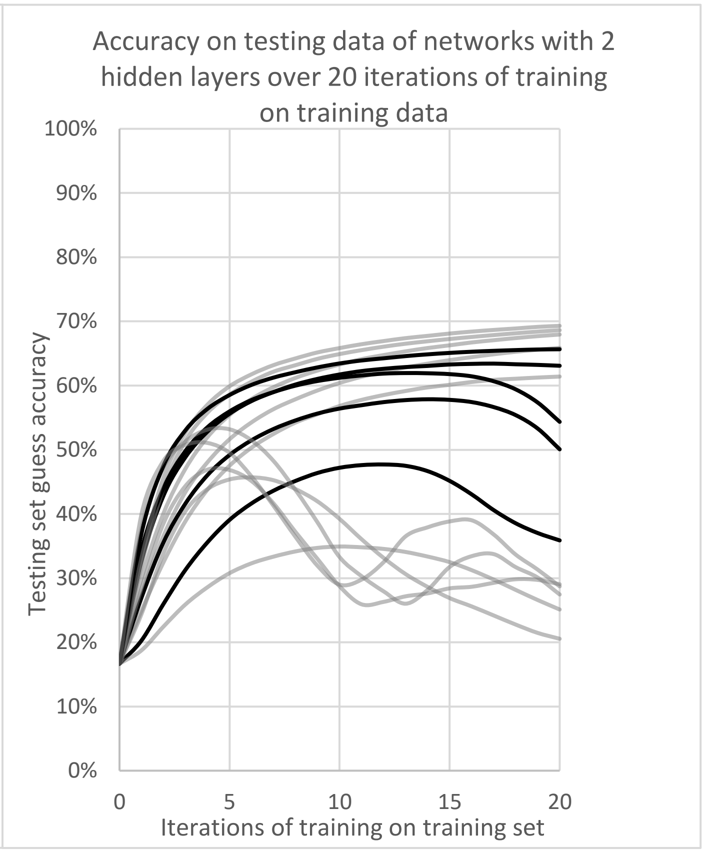
200 neurons		hidden layers		
		1	2	3
iterations	0	17%	17%	17%
	1	33%	37%	40%
	2	46%	48%	48%
	3	53%	53%	51%
	4	57%	56%	51%
	5	60%	59%	50%
	6	62%	60%	46%
	7	63%	61%	41%
	8	64%	62%	36%
	9	65%	63%	32%
	10	66%	63%	29%
	11	66%	64%	30%
	12	67%	64%	32%
	13	67%	65%	37%
	14	68%	65%	38%
	15	68%	65%	39%
	16	68%	65%	39%
	17	69%	65%	37%
	18	69%	66%	34%
	19	69%	66%	31%
	20	69%	66%	29%
		Best for 200 neurons:		69.30%

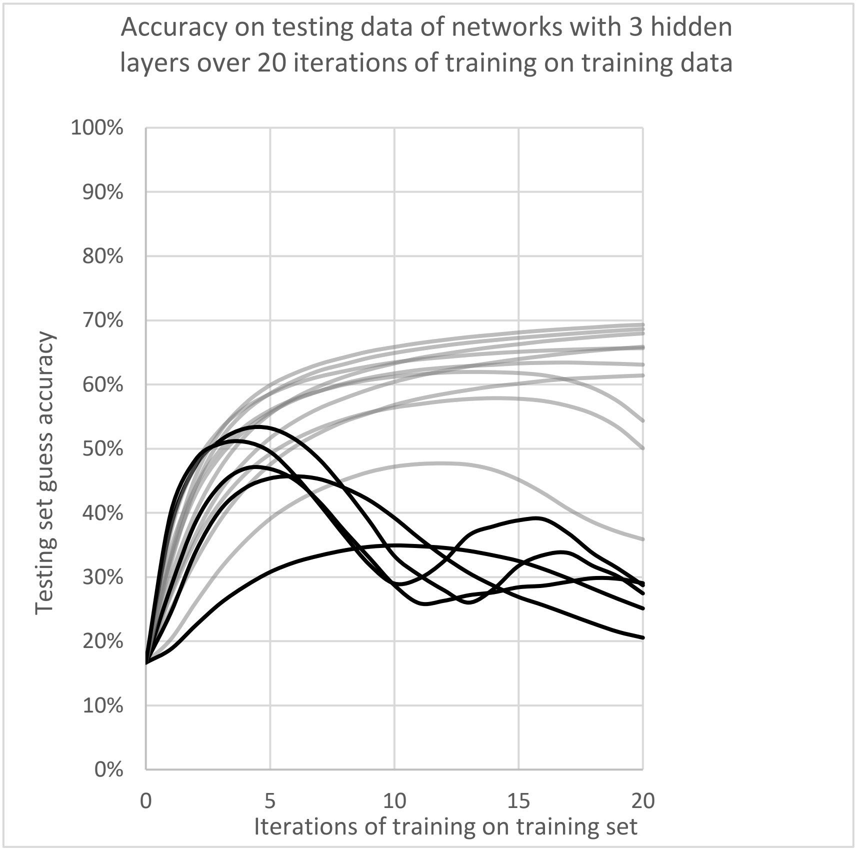
I collected the data by sending a batch of 90,000 new words through the network and recorded the percentage that it guessed correctly. I did this again and again for each iteration. The accuracy appeared to become reasonably decent, i.e. in the 60-70% range and so I could move on to interpreting the data.

I had not met my first set milestone as the Project Outcome had to be written at the same time as I wrote the program. I also went a little over my second milestone, but this didn't become an issue. All documents and the artefact were written just after the start of March 2018.

Representing the Data







I will use the above graphs to make conclusions on my network's performance.

Conclusions

I can quickly conclude from these graphs that the fewer the number of hidden layers, the better the network performed and the longer the network improved without overfitting [2]. I quickly had issues with overfitting from the 3h networks as you can see the accuracy of the network starts going down quickly after only four iterations for the 200n network. The best accuracy these many-hidden-layer networks could reach was also much lower than the 1h networks. It also seemed that the more hidden neurons per layer on a given network, the better the peak accuracy was, with only one or two exceptions. This can be seen below.

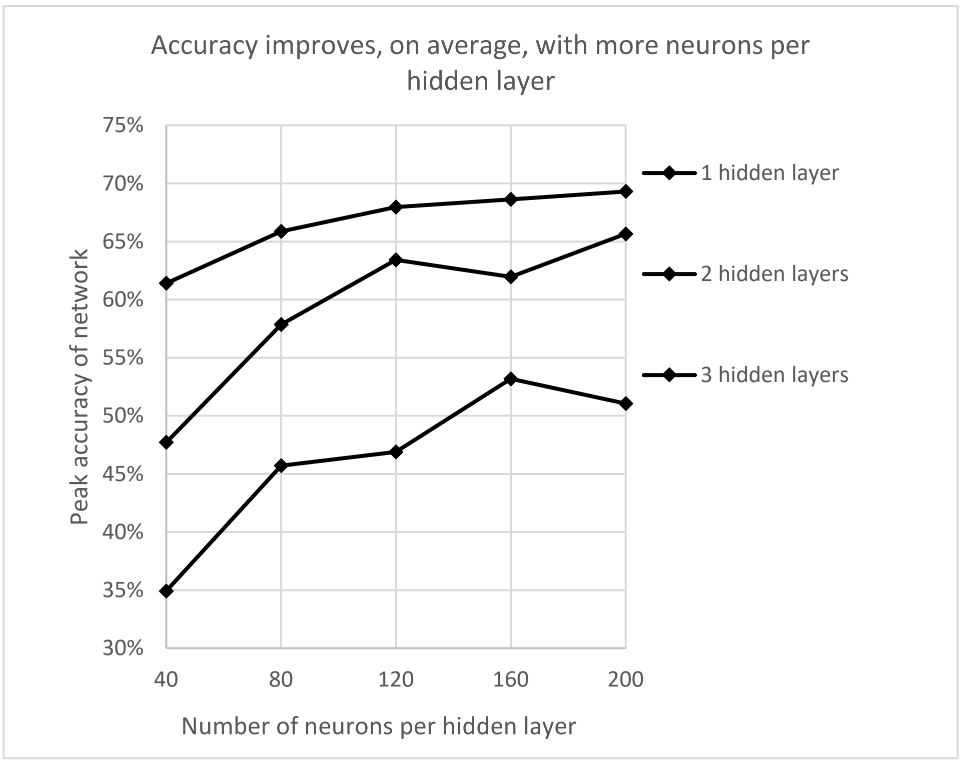
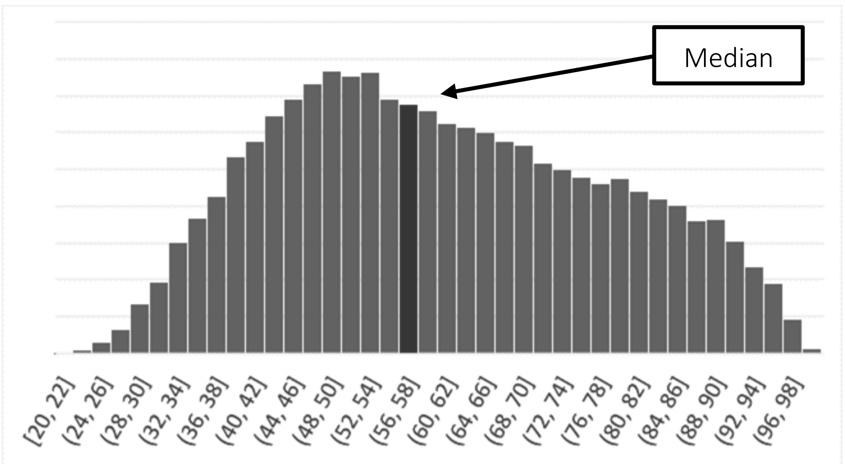


Figure 6

This fit with my conclusion that the most accurate network, with an accuracy of 69.3% (bear in mind that a random, untrained network has an accuracy averaging 16.7%), was the 200n1h network. I believe this could get in the 70-80% accuracy range, but it would be impractical to find the amount of time to train and test the network for that long.

The network's not particularly high accuracy is reflected by its low confidence in its choices. As you can see there is a large spread of confidences on the 200n1h network:



The median is about 59% and almost 32% of the confidences the network has had on its predictions have been below 50%. Given this information and that there is such a huge spread, the network is clearly not very confident with a lot of its predictions. This roughly aligns with the 31% of predictions the network gets incorrect. I don't think there is a strong association between the average prediction confidence and the number of predictions the network gets correct, but this does suggest that if the network had a higher accuracy, there would be a smaller spread of confidences and the median prediction confidence would be higher than 59% - the network would be more confident in its choices if the network was better at guessing.

The answer to my question is that, yes, I can predict the language of a word – with a 69% accuracy with six different possible languages a word could be. Any more languages and I would think it would become too difficult to predict with a reasonable accuracy. If the number of languages to pick from were lower, e.g. two, I would think that the accuracy would be much higher. Possibly even in the 90-100% range.

Afterthoughts

Initially, I wasn't too confident in my own programming abilities and my computer performance. I also found the task of learning to program the network very overwhelming and complicated and I had many, many hurdles with getting the backpropagation algorithms to work. I also made illustrations of the inner workings of my network such as a representation of the network as an image filter by looking at rows of the networks biases and weights collapsed into one matrix. Unfortunately, I forgot that I was using non-linear activations [12] between each successive weight multiplication and bias addition and, so I didn't believe my models were correct. Nevertheless, I built a working feedforward neural network with an accuracy that was much higher than random chance, and I have made other correct descriptions and illustrations of my network and its performance.

I was expecting a much lower accuracy rate since Huang's code [8], using two languages, achieved such a high accuracy rate but with three times as many languages to choose from, I don't think even a person could get as high as a 69% accuracy when guessing between six languages. To make this accuracy higher, I would suggest running through 50 iterations instead of 20, since it looked like it would take a lot of iterations for a 1h network to start overfitting and the accuracy to decrease. These iterations would be run on the 1h network with as many neurons as possible. The only reason I would consider limiting the number of neurons in the hidden layer and the number of iterations of training is because it would take a very long time to complete the training of the network and at extreme numbers of neurons per layer, overfitting [2] may become a big problem. I think it is also possible to consider different models for the network or another type of network entirely. For this, I would have to do more research and have some more experience in machine learning.

If I were to do this project again, I think I would've devoted some more effort in trying out different activations functions to see how they affect the efficiency and accuracy of the network. I only used the sigmoid activation function because I did not spend much time testing the others and, in the time I tested them for, I could only get the sigmoid activation function to work. Overlooking the importance of the activation function may have affected the accuracy of my network and the speed at the rate at which it converged.

Linking conclusions to the brief

I was able to answer the title of my brief, and with an answer that proved the amount of effort I put into the project was worth it – I achieved a high guess accuracy.

Most of my deadlines that I set were met apart from the one regarding debugging which I have discussed previously.

All my project objectives were met – I learned how to build a feed-forward neural network and the mathematics of the process. I was then able to explain this in front of an audience. I failed in finding a link between the word and how difficult it was to guess in Hangman, but I then went on to guess the language of the word successfully.

BIBLIOGRAPHY

[1] P. Michael A. Nielson, “Neural Networks and Deep Learning - Chapter 2,” 2015. [Online]. Available: http://neuralnetworksanddeeplearning.com/chap2.html.

[2] O. Dictionaries, “Overfitting Oxford Dictionary Definition,” [Online]. Available: https://en.oxforddictionaries.com/definition/overfitting.

[3] P. K. Hong, “Neural Networks with Backpropagation for XOR Using One Hidden Layer,” August 2014. [Online]. Available: http://www.bogotobogo.com/python/python_Neural_Networks_Backpropagation_for_XOR_using_one_hidden_layer.php.

[4] D. J. Brownlee, “How to Implement the Backpropagation Algorithm From Scratch In Python,” November 2016. [Online]. Available: https://machinelearningmastery.com/implement-backpropagation-algorithm-scratch-python/.

[5] A. W. Trask, “A Neural Network In 11 Lines of Python (Part 1),” 12 July 2015. [Online]. Available: https://iamtrask.github.io/2015/07/12/basic-python-network/.

[6] P. James D. McCaffrey, “Neural Network Back-Propagation Using Python,” June 2017. [Online]. Available: https://visualstudiomagazine.com/articles/2017/06/01/back-propagation.aspx.

[7] M. Mazur, “A Step by Step Backpropagation Example,” March 2015. [Online]. Available: https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/.

[8] C. K. Huang, “Predicting Languages with Racist Neural Networks,” 21 November 2016. [Online]. Available: https://www.youtube.com/watch?v=evTx5BoKcc8.

[9] S. Raval, “YouTube,” 2016. [Online]. Available: https://www.youtube.com/watch?v=h3l4qz76JhQ.

[10] P. Michael Nielson, “Neural Networks and Deep Learning - Chapter 5,” 2015. [Online]. Available: http://neuralnetworksanddeeplearning.com/chap5.html.

[11] D. J. Brownlee, “Machine Learning Mastery,” 2017. [Online]. Available: https://machinelearningmastery.com/how-to-one-hot-encode-sequence-data-in-python/.

[12] A. Sharma, “Understanding Activation Functions in Neural Networks,” 30 March 2017. [Online]. Available: https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0.