Deep Learning Neural Network Model for Natural Language Processing and Application of Sentiment Analysis to Define the Text Content

A Case Study: BBC Christmas Cooking Recipe

By:
Hazel Kim

Department of Computer Science

Kalamazoo College

SIP Advisor: Alyce Brady

A paper submitted in partial fulfillment of the requirements for the degree of Bachelor of Arts at Kalamazoo College.

2019

i. Acknowledgements

I would like to express my sincere appreciation to Dr. Alyce Brady for supporting me to keep my interest in Computer Science and to shape up my SIP project as my SIP advisor, to Dr. Hannah Apps for always encouraging me to explore any topics I get interested in, and especially to my parents for believing in me with their incessant love. This all could be possible thanks to God's blessings.

ii. Abstract

Recent advances in machine learning have allowed traditional software application domain to integrate AI capabilities. Machine learning, especially deep learning, differs from traditional software engineering in which its implementation is heavily dependent on data from the external world. Yet, there's still a lack of practices for implementing the deep learning model in software development due to the need for curating and processing voluminous, heterogeneous, and often complex data. Since deep learning is able to compose a new model from sub-models and reuse pre-trained models, this "transfer learning" technique can enhance the model if it is used in a strategic way. This research result shows that eliminating unnecessary multiplication by modifying input layers and reducing noise in the vocabulary while training the multilayer perceptions are those strategies with efficient applications of technical debts. In order to implement the deep learning model in software engineering domain, curating voluminous data with proper "transfer learning" technique is a key to increase the accuracy of the model. This paper experiments how to efficiently build and train the deep neural network model with complicated natural language data to enhance the accuracy of the model. It introduces the basic mathematical background of how the model works, and shows two different ways of improving the accuracy for the deep learning model. As a case study, the method descriptions of Christmas cooking recipes is reviewed.

Table of Contents

iii. List of Figures

iv. List of Codes

1. Introduction

Recent advances in machine learning have allowed the traditional software application domain to integrate AI capabilities. Machine learning, especially deep learning, differs from traditional software engineering in which its implementation is heavily dependent on data from the external world. Deep learning automatically learns how to represent data using multiple layers of abstraction. Unlike a traditional application, which needs to build this representation manually, deep learning reduces requirements for manual feature engineering work as well as improves the performance of the model.

Yet, there's still a lack of practices for implementing the deep learning model in software engineering domain due to the need for curating and processing voluminous, heterogeneous, and often complex data. This paper describes experiments on how to efficiently build and train the deep neural network model with complicated natural language data to enhance the accuracy of the model. It will introduce the basic mathematical background of how the model works, and show two different ways of improving the accuracy: eliminating unnecessary multiplication of the redundant words, and reducing the noise in the vocabulary.

As a case study, the method descriptions of Christmas cooking recipes will be reviewed. Then they will be used to build and train the neural network model to identify the recipe content if it is a baked good or an other kind.

2. Data : BBC Cooking Recipe

The data contains details of 1600 Christmas Recipes collected from the BBC Good Food website. It was categorized into title, description, author name, ingredients list and step-by-step methods. The data format is JavaScript Object Notation (JSON). JSON uses human-readable text to transmit data objects consisting of attribute-value pairs in array data types, as shown in Figure 1 below. It is widely used since it is also easy for machines to parse and generate. In this paper, the neural network model was built and trained in Python, and Python has a function to manipulate JSON files. This makes JSON a good format choice to store the data.

{"**Name**": "Christmas pie", "**url**": "https://www.bbcgoodfood.com/recipes/2793/christmas-pie", "**Description**": "Combine a few key Christmas flavours here to make a pie that both children and adults will adore", "**Author**": "Mary Cadogan", "**Ingredients**": ["2 tbsp olive oil", "knob butter", "1 onion, finely chopped", "500g sausagemeat or skinned sausages", "grated zest of 1 lemon", "100g fresh white breadcrumbs", "85g ready-to-eat dried apricots, chopped", "50g chestnut, canned or vacuum-packed, chopped", "2 tsp chopped fresh or 1tsp dried thyme", "100g cranberries, fresh or frozen", "500g boneless, skinless chicken breasts", "500g pack ready-made shortcrust pastry", "beaten egg, to glaze"], "**Method**": ["Heat oven to 190C/fan 170C/gas 5. Heat 1 tbsp oil and the butter in a frying pan, then add the onion and fry for 5 mins until softened. Cool slightly. Tip the sausagemeat, lemon zest, breadcrumbs, apricots, chestnuts and thyme into a bowl. Add the onion and cranberries, and mix everything together with your hands, adding plenty of pepper and a little salt.", "Cut each chicken breast into three fillets lengthwise and season all over with salt and pepper. Heat the remaining oil in the frying pan, and fry the chicken fillets quickly until browned, about 6-8 mins.", "Roll out two-thirds of the pastry to line a 20-23cm springform or deep loose-based tart tin. Press in half the sausage mix and spread to level. Then add the chicken pieces in one layer and cover with the rest of the sausage. Press down lightly.", "Roll out the remaining pastry. Brush the edges of the pastry with beaten egg and cover with the pastry lid. Pinch the edges to seal, then trim. Brush the top of the pie with egg, then roll out the trimmings to make holly leaf shapes and berries. Decorate the pie and brush again with egg.", "Set the tin on a baking sheet and bake for 50-60 mins, then cool in the tin for 15 mins. Remove and leave to cool completely. Serve with a winter salad and pickles."], …}

Figure 1. The first element of the data list of 1600 recipes. Each element is a nested list of name, url, description, author, ingredients, and method.

The Figure 1 shows that the description of “Method” is made of another list of sentences in a cooking order. Because the neural network model for anticipating the text content needs to count the number of occurrences of each word and calculate the frequency of particular word usage, and because it doesn’t matter which string a word is contained in, it is more convenient, and results in better code performance, to create a long single string containing all of the Method strings merged together, and then just traverse that single string to count word occurrences. Each string is also converged to lowercase. Lowercase is a must for an accurate analysis because otherwise the program is case-sensitive and would not automatically recognize the same words if one of them is capitalized.

```
for i in range(len(data)):
    for j in range(len(data[i]['Method'])):
        data[i]['Method'][j] += " "
for i in range(len(data)):
    data[i]['Method'][0:len(data[i]['Method'])] = [' '.join(data[i]['Method']
[0:len(data[i]['Method'])])]
```

Code 1. Modify the data type to one singly merged string for recipe method.

```
import json

data = [json.loads(line) for line in open('recipes.json', 'r')]
for i in range(len(data)):
    data[i]['Name'] = data[i]['Name'].lower()
    for j in range(len(data[i]['Method'])):
        data[i]['Method'][j] = data[i]['Method'][j].lower()
```

Code 2. Make all the words in name and method categories lowercase to count the capitalized and lowercase letters as the same words.

For the case study, the model needs to have the anticipated output as well as actual output. Because the expected output needs to be consistently compared with the actual output to build the model more accurate, simply a list of recipe name would be convenient to create as expected_output data. Cake, cookie, pie, pudding, brownie, biscuit, scone, muffin, tart, bread, and roll are considered as baked goods while the recipe names which do not contain one of those are labeled as other kinds. This way would lead the model to accommodate the misclassified output.

```
expected_data =[]
for i in range(len(data)):
    if "cake" in data[i]['Name']:
        expected_data.append("Baked Good")
    elif "cookie" in data[i]['Name']:
        expected_data.append("Baked Good")
    elif "pie" in data[i]['Name']:
        expected_data.append("Baked Good")
    elif "pudding" in data[i]['Name']:
        expected_data.append("Baked Good")
    elif "brownie" in data[i]['Name']:
        expected_data.append("Baked Good")
    elif "biscuit" in data[i]['Name']:
        expected_data.append("Baked Good")
    elif "scone" in data[i]['Name']:
        expected_data.append("Baked Good")
    elif "muffin" in data[i]['Name']:
        expected_data.append("Baked Good")
    elif "tart" in data[i]['Name']:
        expected_data.append("Baked Good")
    elif "bread" in data[i]['Name']:
        expected_data.append("Baked Good")
    elif "roll" in data[i]['Name']:
        expected_data.append("Baked Good")
    else:
        expected_data.append("Other")
```

Code 3. Based on the recipe name, the expected data is created as a form of array in order of the original data. The order will be exactly matched with the original array data even though it is an independent data structure.

3. Mathematical Background of Deep Learning Neural Network

The understanding of mathematical background is necessary to know how the code works in a neural network.

3.1 Classification

Neural networks are often used to classify input data into categories. In this case, the neural network has developed to classify recipes into "baked good" or not. When training neural network processes data, such as recipes, the output of the neural network is called the "prediction." This is compared in training to the known classification, called "label." A neural network mimics the process of how the brain operates with neurons that fire bits of information. To accomplish its goal of seeking anticipated output of the data, the neural network model looks a line — in fact, the best possible line — that separates the data. Each word in a recipe represents a data point in this example. If the data is a bit more complicated, it will need more complicated algorithms. In such a case, deep neural network will find a more complex boundary that separates the points. This paper looks for the best line that most accurately anticipates if the recipe method is for a baked good. In other words, a linear regression model would be sufficient to tell whether each recipe is either a baked good or other kind because in this example we only have those classifications. The linear boundary follows the mathematical concept described in Figure 2.

$$\text{Linear function: } w_1x_1 + w_2x_2 + \ldots + w_nx_n + b = 0$$
$$W = (w_1, w_2, \ldots, w_n), \qquad x = (x_1, x_2, \ldots, x_n)$$
$$y\text{ (label) } = 0 \text{ or } 1$$
$$\hat{y}\text{(prediction)} = 1, \text{ if } Wx + b > = 0,$$
$$\hat{y}\text{(prediction)} = 0, \text{ if } Wx + b < 0$$

Figure 2. The mathematical equations illustrates the prediction model. W is indicating a weight value, and x input data, b bias value. Label value is an expected value. In this linear boundary, it simply indicates one of two kinds. Prediction "1" means that the neural network has identified the recipe as a baked good while prediction "0" means that it did not.

3.2 Perceptrons and Feedforward Algorithm

A perceptron is a building block of neural networks and an encoding of our equation into a small graph. A perceptron looks a neuron in the brain. Neurons in the brain take inputs coming from the dendrites. Each neuron functions when it receives the nervous impulses and outputs a nervous impulse or not through the axon. Mimicking the way the brain connects neurons with the output from one becoming the input of another, a perceptron connects mathematical functions, with the output from one becoming the input of the next. The perception output is either 1 (yes) or 0 (no), depending on whether it meets the class criteria. For example, the perception in Figure 3 combines a linear function and a step function to categorize the input. This is known as the feedforward process.

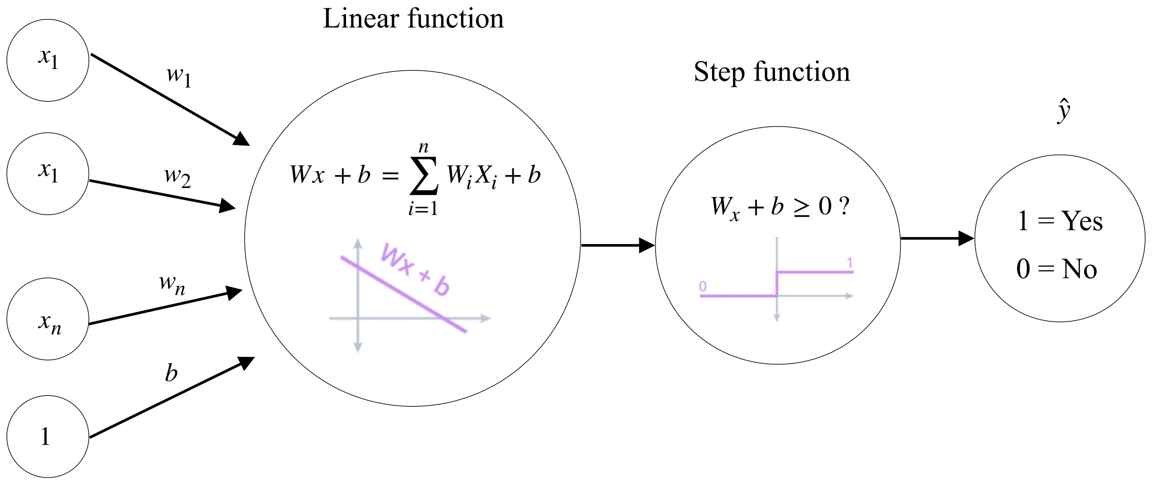


Figure 3. A chart of a perceptron's feedforward process works. Input data x will be multiplied by each corresponding weight values. Bias value b will be multiplied by 1 so that it still gets its own value. If the summation of those values is greater than 0, it will return "yes," otherwise "no."

3.3 Backpropagation

Backpropagation is a process to spread the error to each of the weights. As the feedforward operation compares the output of the model with the desired output, and calculates the error, backpropagation updates the weights. And this feedforward and back propagation loop continues until the model gets better.

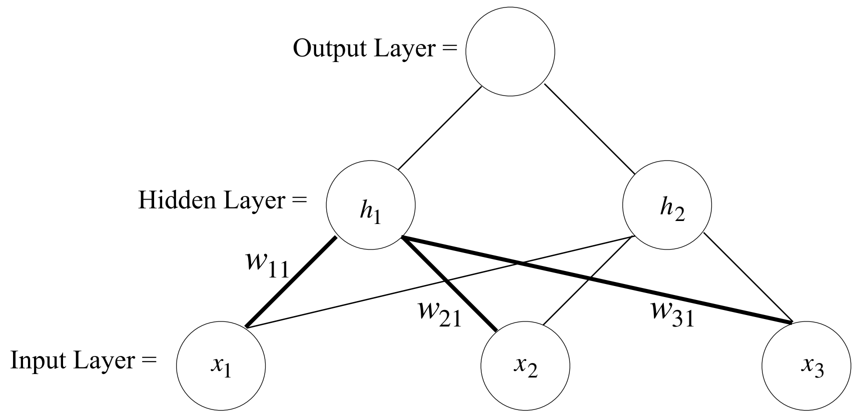
The purpose of backpropagation is to move the line based on misclassified data. If a neural network misclassifies an input, such as a recipe, as "0" when it should have been "1" as indicated by the expected output y, the back propagation process adds a small amount (the "gradient") to each weight and to b (the bias value), where the amount added is a calculation described in section ???. If the neural network misclassifies the input values as "1" the backpropagation process subtracts the same small amount. If the input value is classified correctly, the weights and bias are unchanged. Figure 5 describes this backpropagation algorithm.

```
1. Start with random weights: $w_1, w_2, \ldots, w_n, b$

2. For every misclassified point: $x_1, x_2, \ldots, x_n$

    If prediction == 0 :
        for $i$ in range(0, m) :
            change $w_i$ to $w_i + a x_i$
        change $b$ to $(b + a)$

    If prediction == 1 :
        for $i$ in range(0, m) :
            change $w_i$ to $w_i - a x_i$
        change $b$ to $(b - a)$
```

Figure 5. Pseudocode for perceptron algorithm. For each misclassified point, the model calculates its prediction value through the perceptron algorithm, which is a combination of linear function and step function. After the feedforward process of perceptron, the weight and bias values are updated depending on its prediction value, which is a process of backpropagation. In this figure, *a* is the small multiplier.

3.4 Multilayer Neural Networks

Since the neural network with multilayer perceptron has multiple input units and hidden units, feedforward leads each layer to be calculated from the below layers. The hidden unit is calculated based on this formula: $h_j = \sum_i w_{ij} x_i$. And in the end of the process, the output layer shows us one value, which reflects overall input data.



$$\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \times \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} \qquad h_1 = x_1 w_{11} + x_2 w_{21} + x_3 w_{31}$$

Figure 6. Multilayer perceptron and how hidden layer is calculated based on the input data from the input layer and weight values. A matrix is useful to explain this mathematical application.

3.5 Error Function and Gradient Descent

As discussed in the perceptrons section, each perceptron updates weight and bias values to make misclassified points to be closer to the model, which is a boundary line in this case. This process is known as implementing gradient descent. The *gradien*t is a term for rate of change or slope. Hence, the *gradient descent* means a derivative of the slop and aims to minimize the error of the function. However, if an output of a perceptron is a step function and linear, it does not provide a useful derivative but a value of 0 due to its constant slope. This means that the backpropagation would not work due to the zero value of the gradient. For this reason, we use the sigmoid function, $\sigma(z) = 1/1 + e^{-z}$, rather than a step function, at least for every step but the last, it needs to define whether the output means “yes,” or “no,” the final output uses the step function.

$$\sigma(z) = 1/(1 + e^{-z})$$

$$z = \sum_{i=1}^{m} w_i x_i + bias$$

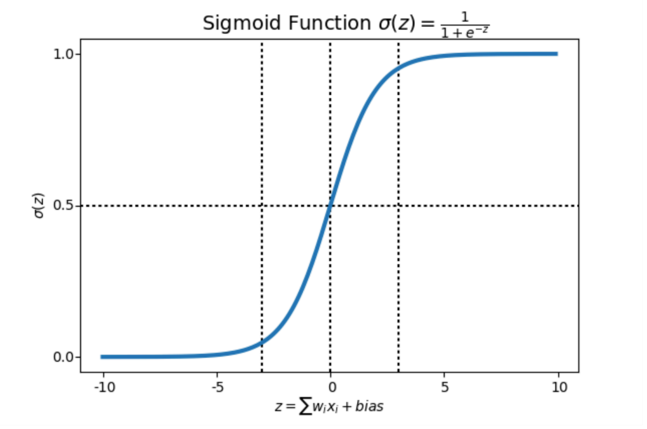


Figure 4. The activation function for the multilayer perceptron output, and its sigmoid function.

To improve the model, it is important to improve the probability of neural network output (the prediction) matching the expected classification (the label). Because the prediction is a product of numbers, natural logarithm is useful for convenient calculation. This unique way of calculating an error value is named *cross entropy.* Cross entropy causes the points that are misclassified to have a larger error values than those that are correctly classified, because a correctly classified point that will have a probability as close to 1 with the negative of the logarithm will get a small value. Accordingly, the negatives of these logarithms works as errors at each point. Points that are correctly classified will have small errors and points that are misclassified will have large errors. This would lead cross entropy to show if a model is good or bad. Consequently, minimizing a cross entropy is a key goal of maximizing a probability and an accuracy of the model.

Prediction function: $\hat{y} = \sigma(Wx + b)$

If $y = 1$,
$P(baked\,good) = \hat{y}$
$Error = -ln(\hat{y})$

If $y = 0$,
$P(other) = 1 - P(baked\,good) = 1 - \hat{y}$
$Error = -(1-y)ln(1-\hat{y}) - yln(\hat{y})$

Error function = $-1/m \sum_{i=1}^{m} (1-y)ln(1-\hat{y}) + yln(\hat{y})$

$$E(W,b) = -1/m \sum_{i=1}^{m} (1-y)ln(1-\sigma(Wx^i + b)) + y_i ln(\sigma(Wx^i + b))$$

Figure 7. Mathematical formula of how to calculate the error function based on the prediction of baked good and of other kinds. The prediction of the multilayer perceptron is a composition of matrix multiplications and sigmoid functions. The error function uses a log-loss cross entropy function to measure how far those points are from the boundary line and how far the errors are from the average error The value *m* is the number of misclassified points.

3.5.1 Minimizing Errors

To minimize the error function, a derivative is necessary. The gradient of the error function is simply the vector formed by all the partial derivatives of the error function with respect to the weights w_i up to w_n and the bias b. First, the sigmoid function has a useful derivative, which is applicable to the gradient descent function; this was the reason why the prediction formula has to have the sigmoid function, with its exponential instead of the step function.

$$\begin{aligned}\sigma'(z) &= \delta/\delta z(1/1 + e^{-z}) \\ &= e^{-z}/(1 + e^{-z})^2 \\ &= 1/(1 + e^{-z}) * (e^{-z}/(1 + e^{-z})) \\ &= \sigma(z)(1 - \sigma(z))\end{aligned}$$

Figure 8. Derivative of Sigmoid Function. In this case z = Wx + b.

$$\hat{y} = \sigma(Wx + b)$$

$$\begin{aligned}\delta/\delta w_j(\hat{y}) &= \delta/\delta w_j(\sigma(Wx + b)) \\ &= \sigma(Wx + b)(1 - \sigma(Wx + b)) * (\delta/\delta w_j)(Wx + b) \\ &= \hat{y}(1 - \hat{y}) * \delta/\delta w_j(Wx + b) \\ &= \hat{y}(1 - \hat{y}) * \delta/\delta w_j(w_1x_1 + \ldots + w_jx_j + \ldots + w_nx_n + b) \\ &= \hat{y}(1 - \hat{y}) * x_j\end{aligned}$$

Figure 9. Derivative of the prediction function, with sigmoid rather than step. This is needed in order to calculate the derivative of the error function with respect to weights, $\delta/\delta w_j(\hat{y})$ needs to be calculated in advance.

$$
\begin{aligned}
\delta/\delta w_j(E) &= \delta/\delta w_j[-ylog(\hat{y}) - (1-y)log(1-\hat{y})] \\
&= -y(\delta/\delta w_j)log(\hat{y}) - (1-y)(\delta/\delta w_j)log(1-\hat{y}) \\
&= -y(1/y)(\delta/\delta w_j)\hat{y} - (1-y)(1/1-\hat{y})(\delta/\delta w_j)(log(1-\hat{y})) \\
&= -y * (1/\hat{y}) * \hat{y}(1-\hat{y})x_j - (1-y)(1/(1-\hat{y}))(-1)\hat{y}(1-\hat{y})x_j \\
&= -y(1-\hat{y})x_j + (1-y)\hat{y} * x_j \\
&= -(y-\hat{y})x_j
\end{aligned}
$$

Figure 10. Derivative of the error E at a point x, with respect to the weight w.

3.5.2 Calculating New Weights

Gradient descent is the process of calculating new weights to achieve predictions with smaller errors. The weight gradient, or the change in each weight, is a scalar times the derivative of the error for that point. This is then added or subtracted from the previous weight value (w_i) to generate the next weight value (w_i'). The scalar α is provided as a parameter.

$$
\begin{aligned}
w_i' &= w_i - \alpha[-(y-\hat{y})x_i] \\
&= w_i + \alpha(y-\hat{y})x_i
\end{aligned}
$$

$$b' = b + \alpha(y-\hat{y})$$

Figure 11. How the weight and bias values get updated. The a value should be (1/m)*a for an average value. For a simplified calculation, it could be substituted as a learning rate.

4. Implementation of Sentiment Analysis

Sentiment Analysis is defined as a process of computationally identifying and categorizing opinions expressed in a piece of text in order to determine whether the writer's attitude towards a particular topic is positive, negative, or neutral.

4.1 Building the model

All a neural network really does is search for direct or indirect correlation between two datasets. In order for neural network to train anything, we have to present it with two meaningful datasets. The first dataset must represent what we know. The second dataset must represent what we want to know, what we want the neural network to be able to tell us. As the network trains, it's going to search for correlation between these two data sets, so that eventually it can take one and learn to predict the other.

```
1. print(data[0]['Method'])
```

```
'heat oven to 190c/fan 170c/gas 5. heat 1 tbsp oil and the butter in a frying pan, then add
the onion and fry for 5 mins until softened. cool slightly. tip the sausagemeat, lemon zest,
breadcrumbs, apricots, chestnuts and thyme into a bowl. add the onion and cranberries, and
mix everything together with your hands, adding plenty of pepper and a little salt.  cut each
chicken breast into three fillets lengthwise and season all over with salt and pepper. heat
the remaining oil in the frying pan, and fry the chicken fillets quickly until browned, about
6-8 mins.  roll out two-thirds of the pastry to line a 20-23cm springform or deep loose-
based tart tin. press in half the sausage mix and spread to level. then add the chicken pieces
in one layer and cover with the rest of the sausage. press down lightly.  roll out the
remaining pastry. brush the edges of the pastry with beaten egg and cover with the pastry
lid. pinch the edges to seal, then trim. brush the top of the pie with egg, then roll out the
trimmings to make holly leaf shapes and berries. decorate the pie and brush again with egg.
set the tin on a baking sheet and bake for 50-60 mins, then cool in the tin for 15 mins.
remove and leave to cool completely. serve with a winter salad and pickles.   '
```

Code 4. The dataset of what we want to know. The method description would lead the neural network model to guess if the content is either about baked good or about other kind.

```
expected_data[0]
'Baked Good'
get_target_for_label(expected_data[0])
1
```

Code 5. The dataset of what we know. The expected_data is created as a process of cleaning the data in the beginning of this research. The recipe is categorized into baked good or other kind based on the recipe name.

Since the program transforms the text data into numbers, the neural network would ultimately provide a possibility with decimal point rather than the exact number with integer, as it gets trained with multiple integers with some calculation. First of all, the number of each recipes and total recipes is necessary to calculate the frequency of particular word usages.

```
total_method = Counter()
baked_method = Counter()
other_method = Counter()

for i in range(len(data)):
    if expected_data[i] == "Baked Good":
        for j in range(len(data[i]['Method'])):
            if data[i]['Method'] != None:
                for word in data[i]['Method'][j].split(" "):
                    baked_method[word] += 1
                    total_method[word] += 1


for i in range(len(data)):
    if expected_data[i] == "Other":
        for j in range(len(data[i]['Method'])):
            if data[i]['Method'] != None:
                for word in data[i]['Method'][j].split(" "):
                    other_method[word] += 1
                    total_method[word] += 1
```

Code 6. Calculate how many baked_good or other recipes are there. Plus, the number of total recipes.

Using the code 6, the model also needs to create a set of entire vocabularies that they have used for the method description. This would let the program keep counting the frequency of each word usage. Also the size of the set would let it create the initial layer of the neural network with an appropriate size.

```
vocab = set(total_method.keys())

word2index = {}
for i,word in enumerate(vocab):
    word2index[word] = i
word2index

vocab_size = len(vocab)

layer_0 = np.zeros((1,vocab_size))
```

Code 7. Pre-allocate the vector as an initial layer, and count how many each word is used.

```
def update_input_layer(review):

    global layer_0

    # clear out previous state, reset the layer to be all 0s
    layer_0 *= 0

    # count how many times each word is used in the given review and store the results in layer_0
    for word in review.split(" "):
        layer_0[0][word2index[word]] += 1
```

Code 8. Modify the global layer_0 to represent the vector form of review. The element at a given index of layer_0 should represent how many times the given word occurs in the review.

Building the network needs several functions: pre-processing the data, initiating the network, updating the input layer, training the model, and more. And all those are encapsulated in the neural network class named “SentimentNetwork” in this research.

```
class SentimentNetwork:
    def __init__(self, reviews, labels, hidden_nodes = 10, learning_rate = 0.1):

    def pre_process_data(self, reviews, labels):

    def init_network(self, input_nodes, hidden_nodes, output_nodes, learning_rate):

    def update_input_layer(self,review):

    def get_target_for_label(self,label):

    def sigmoid(self,x):

    def sigmoid_output_2_derivative(self,output):

    def train(self, training_reviews, training_labels):

            ### Forward pass ###
            ### Backward pass ###

            # Update the weights
            # Keep track of correct predictions.


    def test(self, testing_reviews, testing_labels):

    def run(self, review):
```

Code 9. Encapsulate the neural network code in a class “SentimentNetwork.”

In order to initialize the environment to build and train the network model, the class creates a corresponding settings with input parameters such as method description data, expected_data, hidden_nodes(for multilayer perceptions), and learning_rate. Then the model assigns a seed to the random number generator to ensure that it could be reproducible results while training the model. After that, it proceeds the method description data and their associated expected_data list with function defined pre_process_data() for all the required data to be ready. Finally, function defined init_network() could be used to build the network to have the number of hidden nodes and the learning rate that were passed into this initializer as shown in code 10.

```
def __init__(self, reviews,labels,hidden_nodes = 10, learning_rate = 0.1):

    np.random.seed(1)
    self.pre_process_data(reviews, labels)
    self.init_network(len(self.review_vocab),hidden_nodes, 1, learning_rate)
```

Code 10. Create a SentimentNetwork with the given settings.

The pre_process_data() method populates the method description and expected_data with all of the words in the given method data. And it converts the vocabulary set to a list so that it can access words via indices. Then it creates a dictionary of words in the method description as well as in the expected_data mapped to index positions. Finally init_network() method sets number of nodes in input, hidden and output layers. Then it initialize weights between the input layer and the hidden layer and those between the hidden layer and the output layer. It ultimately creates an input layer, which is a two-dimensional matrix with shape 1 * input_nodes.

```
def init_network(self, input_nodes, hidden_nodes, output_nodes, learning_rate):
    self.input_nodes = input_nodes
    self.hidden_nodes = hidden_nodes
    self.output_nodes = output_nodes

    self.learning_rate = learning_rate


    self.weights_0_1 = np.zeros((self.input_nodes,self.hidden_nodes))
    self.weights_1_2 = np.random.normal(0.0, self.output_nodes**-0.5,
                                        (self.hidden_nodes, self.output_nodes))

    self.layer_0 = np.zeros((1,input_nodes))
```

Code 11. Init_network method with parameters input_nodes, hidden_nodes, output_nodes, learning_rate. It ultimately produces an initial input layer.

4.2 Training the model

```
def train(self, training_reviews, training_labels):


    for i in range(len(training_reviews)):

        # Get the next review and its correct label
        review = training_reviews[i]
        label = training_labels[i]

        ### Forward pass ###
       # Input Layer
        self.update_input_layer(review)

        # Hidden layer
        layer_1 = self.layer_0.dot(self.weights_0_1)

        # Output layer
        layer_2 = self.sigmoid(layer_1.dot(self.weights_1_2))

        ### Backward pass ###
        # Output error
   # Output layer error is the difference between desired target and actual output.
        layer_2_error = layer_2 - self.get_target_for_label(label)
        layer_2_delta = layer_2_error * self.sigmoid_output_2_derivative(layer_2)

        # Backpropagated error
  # errors propagated to the hidden layer
        layer_1_error = layer_2_delta.dot(self.weights_1_2.T)
  # hidden layer gradients - no nonlinearity so it's the same as the error
  layer_1_delta = layer_1_error

        # Update the weights
  # update hidden-to-output weights with gradient descent step
        self.weights_1_2 -= layer_1.T.dot(layer_2_delta) * self.learning_rate
  # update input-to-hidden weights with gradient descent step
        self.weights_0_1 -= self.layer_0.T.dot(layer_1_delta) * self.learning_rate

        # Keep track of correct predictions.
        if(layer_2 >= 0.5 and label == 'Baked Good'):
            correct_so_far += 1
        elif(layer_2 < 0.5 and label == 'Other'):
            correct_so_far += 1
```

Code 12. Training the model. While looping through all the given reviews and run a forward and backward pass, the algorithm updates weights for every item.

4.2.1 Eliminating Unnecessary Multiplications

To make the original neural network model more efficient, the model can eliminate unnecessary multiplications and additions that occur during forward and backward propagation. First, it can copy the SentimentNetwork class from the previous project and remove the update_input_layer function and modify the init_network not to use a separate input layer anymore. Instead, it can use the old hidden layer more directly. For implementing this modification, it can get rid of self.layer_0 and create self.layer_1, a two-dimensional matrix with shape of 1*hidden_nodes. Accordingly in the train method, the update_input_layer must be removed as well. Instead, it will use self's layer_1, not a local layer_1 object. This would lead the forward pass to replace the code that updates layer_1 with new logic that adds the weights for the indices used in the method description. It would update weights_0_1, which means that it would only update the individual weights which had been used in the previous forward pass. This would be useful to train the model with more valid value of weights.

```
def init_network(self, input_nodes, hidden_nodes, output_nodes, learning_rate):
    # Set number of nodes in input, hidden and output layers.
    self.input_nodes = input_nodes
    self.hidden_nodes = hidden_nodes
    self.output_nodes = output_nodes

    # Store the learning rate
    self.learning_rate = learning_rate

    # Initialize weights

    # These are the weights between the input layer and the hidden layer.
    self.weights_0_1 = np.zeros((self.input_nodes,self.hidden_nodes))

    # These are the weights between the hidden layer and the output layer.
    self.weights_1_2 = np.random.normal(0.0, self.output_nodes**-0.5,
                                        (self.hidden_nodes, self.output_nodes))

    ## Removed self.layer_0; added self.layer_1
    # The input layer, a two-dimensional matrix with shape 1 x hidden_nodes
    self.layer_1 = np.zeros((1,hidden_nodes))
```

Code 13. Edited code for eliminating unnecessary input layer in forward and backward propagation. It removes self.layer_0 and adds self.layer_1 in init_network method.

4.2.2 Strategically Reducing Noise in the Vocabulary

The second way of enhancing the neural network model is adding minimum count of the words and cut off the polarity of the least used words to reduce the noise in the data. T-distributed Stochastic Neighbor Embedding and Vector T-SNE for More Polarized Words and Frequency Histogram are used to see the polarity of the words usage. In this research, the most effective polarity cut-off rate 0.8 was used based on those visual data and the actual experiments.

```
baked_counts = Counter()
other_counts = Counter()
total_counts = Counter()

for i in range(len(method_description)):
    if(expected_data[i] == 'Baked Good'):
        for word in reviews[i].split(" "):
            baked_counts[word] += 1
            total_counts[word] += 1
    else:
        for word in reviews[i].split(" "):
            other_counts[word] += 1
            total_counts[word] += 1
```

Code 14. Count the number of baked goods' recipes and others' recipes.

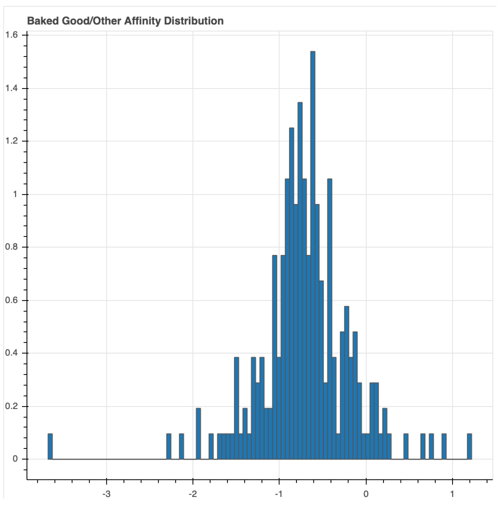


Figure 12. Frequency Histogram

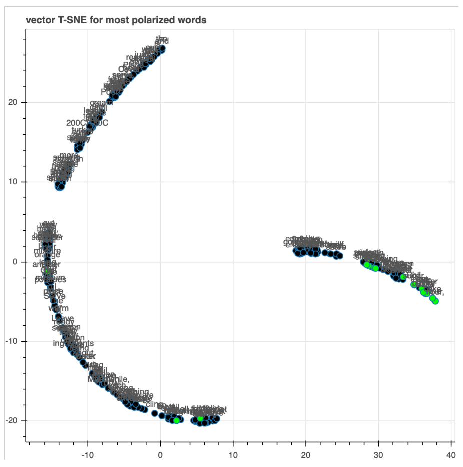


Figure 13. T-distributed Stochastic Neighbor Embedding and Vector T-SNE for More Polarized Words.

```
baked_other_ratios = Counter()

for term,cnt in list(total_counts.most_common()):
    if(cnt >= 50):
        baked_other_ratio = baked_counts[term] / float(other_counts[term]+1)
        baked_other_ratios[term] = baked_other_ratio

 for word,ratio in baked_other_ratios.most_common():
    if(ratio > 1):
        baked_other_ratios[word] = np.log(ratio)
    else:
        baked_other_ratios[word] = -np.log((1 / (ratio + 0.01)))
```

Code 15. Calculate baked_good-to-other_kind ratios for words before building vocabulary.

```
review_vocab = set()
for review in reviews:
    for word in review.split(" "):
        if(total_counts[word] > min_count):
            if(word in baked_other_ratios.keys()):
                if((baked_other_ratios[word] >= polarity_cutoff) or (baked_other_ratios[word] <= -polarity_cutoff)):
                    review_vocab.add(word)
            else:
                review_vocab.add(word)
```

Code 16. The model add only the words that occur at least min_count times and for words with baked_good/ other_kinds ratios, only add words that meet the polarity_cutoff.

5. Result

The result of training neural network model shows the learning rate of 0.01 is the best condition for all the three types of models. And the third model, which is strategically reducing noise in the vocabulary has the highest training accuracy regardless of different learning rates. The strategy of cutting off relatively least used vocabulary data for training the model significantly enhances the accuracy of the model. This method results in more significant improvement even though the second experiment, which is eliminating unnecessary multiplication by getting rid of a local input layer and directly applying the updated self input layer, still produced a better accuracy than the original method.

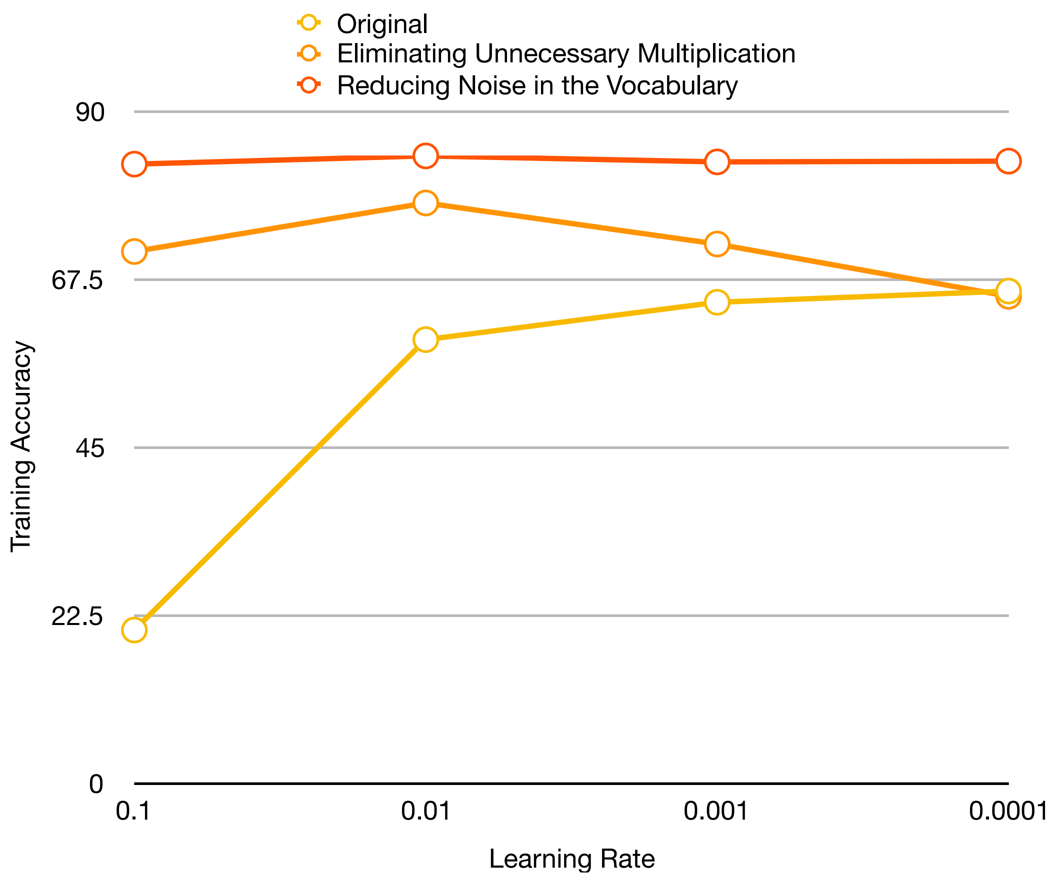


Figure 14. The graph for tracking the training accuracy with various learning rates from three different types of neural network models: original version, enhanced version with eliminating unnecessary input layer, and another enhanced version with reducing noise in the vocabulary.

	Original	Eliminating Unnecessary Input Layer	Reducing Noise In the Vocabulary
0.1	20.4%	71.1%	82.8%
0.01	59.3%	77.6%	83.9%
0.001	64.3%	72.1%	83.1%
0.0001	65.8%	65.1%	83.2%

Table 1. The table for tracking the training accuracy with various learning rates from three different types of neural network models: original version, enhanced version with eliminating unnecessary input layer, and another enhanced version with reducing noise in the vocabulary.

6. Discussion

Machine learning, especially deep learning, is heavily dependent on output data from the external world. Since deep learning automatically learns how to represent data using multiple layers of abstraction, it has a potential disadvantage of technical debt, a long-term costs incurred by data dependencies. Deep learning also makes data possible to compose complex models from a set of sub-models and potentially reuse pre-trained parameters with so called “transfer learning” techniques. This not only adds additional dependencies on the data, but also on external models that may be trained separately and may also change in configuration over time. It frequently happens that the supporting code and infrastructure incur significant technical debt. Consequently, dependency debt is noted as one of the key contributors to technical debt in software engineering domain. Since the data dependency debt results in a significant impact to improve or to weaken the accuracy of the model, the application of deep learning requires a careful consideration of its influence on the output.

7. Conclusion

Since deep learning is able to compose a new model from sub-models and reuse pre-trained models, this “transfer learning” technique can also enhance the model if it is used in a strategic way. As this research result shows, eliminating unnecessary multiplication by modifying input layers and reducing noise in the vocabulary while training the multilayer perceptions are those strategies with efficient applications of technical debts. In order to implement the deep learning model in software engineering domain, curating voluminous, heterogeneous data with proper “transfer learning” technique is a key to increase the accuracy of the model.

8. References

Amershi, Saleema, Begel, Andrew, Bird, Christian, DeLine, Robert, Gall, Harald, Kamar, Ece, Nagappan, Nachiappan, Nushi, Besmira, and Zimmermann, Thomas. “Software Engineering for Machine Learning: A Case Study.” Microsoft, 2019.

Arpteg, Anders, Brinne, Bjö̈rn Brinne, Crnkovic-Friis, Luka, and Bosch, Jan. “Software Engineering Challenges of Deep Learning” in 2018 44th Euromicro Conference on Software Engineering and Advanced Applications , 2015, pp. 50-59.

Chen, Danqi and Manning, Christopher. A fast and accurate dependency parser using neural net- works. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), pp. 740–750, Doha, Qatar, October 2014. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/D14-1082.

Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoffrey E. “Imagenet classification with deep convolutional neural networks,” in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

Maas, Andrew L and Ng, Andrew Y. A probabilistic model for semantic word vectors. In NIPS Workshop on Deep Learning and Unsupervised Feature Learning, 2010.

Ré, Christopher, Divy, Agrawal, Magdalena, Balazinska, Michael, Cafarella, Michael, Jordan, Kraska, Tim, and Ramakrishnan, Raghu. “Machine learning and databases: The sound of things to come or a cacophony of hype?” in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 283–284

Trask, Andrew, Gilmore, David, and Russell, Matthew. Modeling order in neural word embeddings at scale. CoRR, abs/1506.02338, 2015. URL http://arxiv.org/abs/1506.02338.

Zhang, Xiang, Zhao, Junbo, and LeCun, Yann. Character-level convolutional networks for text classification. CoRR, abs/1509.01626, 2015. URL http://arxiv.org/abs/1509.01626.