Spam detection in social networks

# Abstract

The purpose of this project is to perform a tweets-level spam detection through neural network. Different from other user-level detection, which focusing on detecting the malicious account. In this paper, we design deep learning models for detecting whether a post is a spam or not. The scope of this project is to present the deep learning models we designed for the spam detection task, and the results we obtain during training and test on the dataset.

In order to detect the spams through the tweets, we first design a pre-processing strategy to perform on the dataset. The unique strategy we applied is to simplify the URL by its own domain name only. Secondly, we move on constructing the neural networks. Three neural networks we constructed are from three different papers. The reason of creating three different deep learning model is to check where the problem come from. When a model fails to fit the dataset, we have to prepare another dataset or models, to check the failure is caused by the dataset or the model. The SSCL model is the one that we are the latest published and the one we want to explore. And, the other two models can help us to do the comparison.

The key outcomes we achieve in this project is that we prove the spam can be detected by only the tweets text. When we only input the tweets text to three deep learning models, it achieves the accuracy between 94 to 95 % on test set. After the success on these three text-analysing models, we move to consider other tweets information, such as the number of hash tags, retweets and favourites. Also, the information about the user can also be contained, such as the number of followers, friends, favourites. We expect the extra features can boost the accuracies of these three models since the model are receiving more information. The result of the combined model reaches 95.67% on the GatedCNN model, and the number of trainable parameters in this model is only 995,761, which can be considered as a small model for nature language processing. For future works, these models can be applied on the sequential data. People can follow the manual we provide in this project to add in their dataset, and train tune the models through the GUI software.

# Introduction

As this information age that we are living with, the internet has been necessary for people’s daily life. One convenient tool that the Internet bring to us is the social network. It significantly affects human behaviour of receiving information and interacting with others. Therefore, the social network is part of our daily life. And seldom of people can live without it. Nowadays, people are connected together through the social networks, such as Facebook, Instagram and twitter. These platforms have a common issue, spams. This issue doesn’t like other technical or businesses issues that are emergent, but it’s still fatal doubtlessly. The spam can hide malicious links, which contains virus or malware, in their posts. Moreover, some spams lead you a pay for subscriptions or counterfeit products, then scam your money. These spams can dramatically affect the user experience on the social platform. When a social platform consists of tremendous advertisements and malicious links, no user will trust the platform and stay on it.

Since stopping spams has been an important area worth for investment for the companies that running the social networks, massive effort has been paid on this field. However, most of the current spam detection system focusing on the user-level, which means the model use the information of users to train the model to classify whether a user is a spammer or not. This method may cause some problems. For example, when a malicious user hijack accounts from other normal users. The detection system may detect the user as a spammer and end up deleting the normal account. Therefore, instead of detecting the spammer in the user-level, we will attempt to detect the post in the tweets level. The benefit of the tweets-level detection is we can only remove the spam tweets rather than deleting the whole account.

In this project, our preferred model will be constructed by the neural. The reason of using the deep learning model rather the traditional machine learning model is not only it’s a trendy tool but the ability of processing the text data. Text data is one of the sequential data, which means the order of the data is important and should be considered. The traditional machine learning model have restricted capacity of handling the ordered information. In the other hand, the deep learning has been proved effectiveness on the text-related tasks, such as machine translation, language modelling and text generation. Moreover, the deep learning is well known for coping with the unstructured data, such as the text and image. And the traditional machine learning model is good at dealing with the structured data, including the normal numerical and categorial data.

## Objective

The deep learning is good at two “P”, perception and prediction. Spam detection is an area that neural network doesn’t occur frequent. The traditional machine learning, such as random forest and XGboost, can achieve over 95% easily after some feature engineering. However, to break the bottleneck, we may need the help of deep learning since it usually has more parameters and larger capacity to fit a massive and complicated dataset. To begin with, this project aims to apply the deep learning on the spam detection task. Secondly, we want to prove that deep learning can use only the tweets text as the input to achieve a good performance. Thirdly, we propose to create a GUI (Graph User Interface) software; therefore, the user can manipulate the hyper-parameters and training the models through the software rather than dig into the python code. If some people want to modify the python code, we will also provide a brief manual for instructions. Lastly, we will tune the model and try other strategies to improve the model and let it has a competitive performance.

## Methodology

To conduct this project, we run the Python code on the Ubuntu 18.04 Operating System. The IDE we use if the jupyter notebook for developing stage and run the Python in bash for production stage. The First important package be included is the nltk library, which provide a lot of functions and classes to facilitate the text pre-processing strategy. Moreover, in order to implement the neural network, we have to pick a deep learning framework from three of the popular frameworks, Tensorflow, Keras as Pytorch. The deep learning frameworks can help the user to calculate the derivative and perform backpropagation automatically, which is the most math intensive part of the deep learning.

The final decision we made is the Pytorch since it’s the only pure dynamic framework. Tensorflow has to construct the static graph at the first and feed in the data to run at the second stage. The static mechanism makes the debugging process more difficult because it’s hard to extract a part of neural network to check the parameters inside. In the other hand, the Pytorch allow users to transform the tensor to numpy array; hence, the data status inside the deep learning model can be clearly shown, which makes a large difference in the debugging. In order to attempt new models and ideas, this project can benefit from Pyotorch more than other two frameworks. The last significant library we import to our project is the Tkinter. This library can help us to create GUI software and visualise the training and test outcomes. More details of the applied methodology will be explained in the methodology section.

## Scope

|  |  |
| --- | --- |
| In Scope | Out of scope |
| * Extract the dataset we need from the SQL. * Design a pre-processing strategy for the tweets text * Construct the neural networks * Testing the neural network algorithm on the small dataset * Create the GUI software by Tkinter library * Train the model through the GUI we made * Tune the model through the GUI * Record the Training and Tuning Result | * Applying other machine learning model to compare with the deep learning model * Comparing multiple pre-processing strategies * Do feature engineering on the dataset (except the tweets text pre-processing) |

## Deliverables

* The methods of pre-processing the tweets
* The pre-processed and training-ready dataset.
* The implementation of the deep learning algorithms.
* The GUI software used for training, testing and tunning the deep learning models.
* The architecture of a neural network that able to receive both tweets text and other structure data.
* The records obtained from the experiments.
* The final report containing all the details of the implementations and experiments.
* The open source GitHub repository (<https://github.com/ChihchengHsieh/SpamDetectionProject>)

# Literature Review

As the spam has been harmful for social platforms, several solutions have been presented. One potential countermeasure to alleviate the problem is to detect and delete the spams on the platform (Heymann, Koutrika, & Garcia-Molina, 2007). People have paid a lot of effort in this field. According to the suggested user and content-based features, the random forest model can reach the performance of 95.7 accuracy (Mccord & Mccord, 2011). In the traditional approach, we use clustering on the text data. After the clustering, we apply machine learning model on the clustering labels and other information. This method has been optimized very well on both efficiency and accuracy. However, only use the clustering label to represent the tweets may cause dramatic information loss. Therefore, deep learning is another alternative of clustering, which can keep more information and be trained as a part of model. Researchers also apply deep learning on the tweets level analysis, some of them are using convolutional neural network (Madisetty & Desarkar, 2018) and some of them using the LSTM (Long-Short Term Memory) (Jain, Sharma, & Agarwal, 2019). One recently published model also combines both LSTM and CNN. Previously, the same reasearch fellows published a CNN-based approch, called Semantic Colvolution Neural Network (SCNN) ) (Jain, Sharma, & Agarwal, 2018). Now, they pulished a Sequential Stacked CNN-LSTM Model, which stack a layer of LSTM on the CNN layer(Jain, Sharma, & Agarwal, 2019). However, not too much detail about the environment and hyper-parameter setting are listed in their papers.

Both analysing the tweets text and perform the classification in the tweets-level are not the main trend in the spam detection field. Therefore, finding a paper that providing details about implementation and the environment they used is hard. Some research result we found only introduce their models briefly and talk about more on the performance. Since no implementation is available, we are hard to benchmark our model with others. The deep learning is a popular in the NLP (Natural Language Processing) but not a famous tool in the spam detection. Therefore, we only found few papers that using deep learning on the spam detection without the details of implementation, which means we are not able to duplicate the work. However, according to the basic concept of architecture provided in the papers, we can try to create a similar neural network based on the same concept. Moreover, we borrow some famous models from the NLP tasks, which can help us to make the comparison of different models.

## HSpam14 Dataset

HSpam dataset (Sedhai & Sun, Hspam14: A collection of 14 million tweets for hashtag-oriented spam research, 2015) is the dataset we use in the project to perform the training and test. The dataset contains 14 million data in total. And, 1,145,136 of them are labelled. This dataset is unbalanced, which contains 82.3% legitimate users and 17.7% malicious users. The reason of using a large dataset is because the deep learning needs a larger amount of data to prevent the overfitting. During training the neural network, even the loss has converged, we still feed in more data till the first epoch has done. We expect the model can be regularised and prevent it from the overfitting through seeing more data.

|  |  |
| --- | --- |
| Tweets | |
| Feature Name | Description |
| text | The text post of tweets. |
| NumberOfHashtags\_c | How many hash tags are in this tweet. |
| Favorite\_count | How many times this post marked as favourite. |
| Retweet\_count | How many times this tweet retweeted by others. |

|  |  |
| --- | --- |
| Users | |
| Feature Name | Description |
| Followers\_count | How many followers this account has. |
| Friends\_count | How many friends this account has. |
| Default\_profile | Does this user use the default profile provided by tweeter? |
| Default\_profile\_image | Does this user use the default profile image provided by tweeter? |
| Favourites\_count | How many posts this user has marked as favourites. |
| Listed\_count | How many lists this user has created. |
| statuses\_count | How many tweets the user has issued (including retweet). |
| Verified | Is this account verified? |

## Sequential Stacked CNN LSTM

SSCL is a recent published model, which is used on the spam detection in the paper. The feature of the SSCL model is it stack two popular neural networks together. SSCL is used for dealing with the sequential data, such as text, which means this model is not restricted in the spam detection field as well. It can be applied on other NLP tasks as well. According to the paper, the CNN at the first layer can work as n-grams to analyse the neighbour words. And the n is decided by the kernel size of the CNN. And reason of applying the LSTM on top of the CNN is to encode the information of position. The text order is important in representing the meaning of the sentence. However, this paper only tell us that they apply these two neural networks to construct SSCL, but it doesn’t explain how the neural network map the input to the output. Since they applied the LSTM in the last layer, there are two methods to map the output of LSTM to the scalar of possibility. One of them is taking the last timestamp output of LSTM, which encode the whole information about the input text. However, it may cause the problem of gradient vanish or gradient explosion. Therefore, we finally apply another method, which sums up all the output in each timestamp. The Second approach can alleviate the gradient problem and still get a good result.

One more interesting point from this paper is the pre-processing strategy they use. Instead of using an embedding layer to transform the word to vector, they use the pre-trained word2vec from the google. If the token doesn’t exist in the google word2vec dictionary, the WordNet or ConceptNet will be applied to find the synonym. However, the datasets they use are not the tweeter dataset. This paper uses the SSCL model to analyse the email and SMS, which will be proper to apply the pre-trained word2vec. However, for the tweets’ dataset, which may contain a lot of slang, URLs and hashtags, the word2vec may not contain the words we need.

## Gated-CNN

Gated-CNN (Dauphin, Fan, Auli, & Grangier, 2017) is another model we apply in this project. The model consists of two types of block, including convolution block and the gating block. The gating block is where the model differs from other traditional models. In the gating block, the output of the convolution block will be split in half. And, one of them will pass through the sigmoid layer then multiple another half part. This operation is slightly similar to the self-attention mechanism. The self-attention multiply two matrices together as well. However, the self-attention use the matrix multiplication and the Gated-CNN use the element-wise multiplication. The reason of applying an CNN-based rather than RNN-based neural network is to prevent the gradient explosion and vanishment (Pascanu, Mikolov, & Bengio, 2013). The CNN and other feedforward-based model has three advantages when comparing to the RNN-based model, which are parallelisation, trainability and inference speed (Miller & Hardt, 2018). In order to boost the training speed and solve the training problem caused by gradient, we decide to add in this model as an option that can be selected during training and test.

## Self-Attn

The Self-Attn model is using the self-attention mechanism. And, a classic model using self-attention mechanism is called Transformer (Vaswani, et al., 2017). Transformer model has been proven extremely effective on various NLP tasks. Recently, two famous models, MT-DNN (Liu, He, Chen, & Gao, 2019) and BERT (Devlin, Chang, Lee, & Toutanova, 2018), are dominating the NLP field. Both of them are based on the transformer model but introduced some pretrained strategies to boost the performance. The transformer consists of two-part, one part for encoding the input and another for decoding the output. The transformer was built for text generation; however, we only need one output from the sequential data. Therefore, the Self-Attn model in this project only use the encoding part of the transformer. Same as the Gated-CNN, the transformer is a type of feedforward-based neural network; hence, it can be run on the GPU too reduce the training and inference time. Also, the gradient problem that happens in the RNN model will not occur in this model. Furthermore, in order to better backprop the gradient, a mechanism called “Residual” is added to the transformer architecture. The residual was first published as ResNet (He, Zhang, Ren, & Sun, 2016) . This mechanism enables the neural network to decide whether to do an operation by itself and it also facilitate the propagation of gradient, which makes traning deep neural network trainable.

## Batch Normalization

Batch normalization is a special operation that widely used in the deep learning. The effect of batch normalization is faster and more stable during training. Adopting the batch normalization in the model can significantly smoothen the optimization landscape. A smoother landscape of loss enables a more predictable and stable behaviour of gradient, which can help the model to find the local minima faster (Santurkar, Tsipras, Ilyas, & ˛adry, 2018). Since the benefits above, we adopt the batch normalization to every deep learning model we create. The position of the batch normalization layer is between the matrix manipulation layer and the activation layer. Therefore, a block of neural network will first pass through a feedforward layer (can be convolution or fully-connected layer), then the output will be applied batch normalization and the activation function at the last.

## Dropout

Another popular operation we apply in our project is “Dropout” (Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov, 2014). A serious disadvantage of deep learning is the problem of overfitting. One popular solution is to add a regularization term (L2) in the loss function. A better regularization strategy is to add the dropout layer. Dropout layer randomly drop units from the input, then replace the dropped units by 0. The key concept is to prevent units from co-adapting too much.

# Project methodology

Tools:

1. Operating System, Language and IDE

Three operating system can be selected to conduct the deep learning project, including Ubuntu, MacOS and Windows. We determine the Ubuntu 18.04 to be main operating system we are working with. The reason of not picking Windows is the programming language we use.

Python is our selection on running the machine learning algorithm. Other languages also have deep learning framework, such as JavaScript, Julia and R. However, Python provide the most completed solution for both data processing and deep learning. To run python on the machine, the Windows operating system need Anaconda to create an environment to work with Python. And the installation of Python under windows environment is a little bit tricky. In terms of the MacOS, this operating system doesn’t support Nvidia GPU well, which we will use to perform the parallel processing. Therefore, the Ubuntu 18.04, which is Python friendly and compatible with Nvidia GPU, is our final pick.

In terms of the IDE, two of them we’re using in this project. First one is the Jupyter Notebook, which is extremely debugging friendly. Working with Jupyter Notebook has three main advantages. Firstly, it allows the user to run only one block of code rather than the whole implementation. Running only certain part of code can help us to track the processes and the variables easily. Another benefit of applying Jupyter Notebook is the markdown mode it gave. Rather than opening another word file to record the findings, the Jupyter Notebook allow users to write markdown block in the code. Lastly, the visualization is more convenient than other IDEs, such as PyCharm or Spyder. Another IDE we selection is Visual Studio Code. Unlike Visual Studio, the Visual Studio Code is a lightweight and cross-platform IDE, which can be run on Windows, Linux and MacOS. Moreover, Visual Studio Code has a great extension system, which can help me to refactoring, modularising and formatting our code. Two IDEs are used for this project since we want to use Jupyter Notebook to facilitate the development process. After the development stage, we can easily transfer the .iPython file, which Jupyter Notebook use, to .py file. Finally, we can run the Python code in the bash, which always be the most stable method.

1. Extracting data from SQL dataset

The original data source is a SQL dataset. Hence, the software we use to interact with the SQL dataset is MySQL Workbench, which can be used for querying the dataset and exporting data.

1. Pre-processing

In order to pre-processing the text input, the NLTK package is heavily in use. This library provides several useful functions. For example, the TweetTokenizer can recognise the URL and hashtags and better tokenise the text than other tokenizers. Moreover, it also offers Potter Stemmer and Stopwords Removal to easily pre-process the input text. One more important class NLTK offers is the nltk.Text class. The Text class can help the user to quickly create a dictionary and use the created dictionary to map the token to index. Another library we frequently use in this stage is Pandas, which can form the input data to data frame for better manipulation. Also, the read\_csv() and read\_html() functions are useful when loading data.

1. Constructing Neural Networks

In order to facilitate the process of development, deep learning framework has to be adopted. Using Deep learning framework has three main benefits. Firstly, the deep learning framework can calculate the derivative and perform back-propagation automatically. Without worrying the back-propagation, we can concentrate on designing the architecture of neural network. Another advantage is the functions it provided. Some popular and common function has been optimized in the deep leaning framework, which can provide a better efficiency. For example, the cross entropy, batch normalization and dropout have been built in the frameworks. Hence, we can call the function instead of writing the code from scratch. The last reason for using frameworks is the GPU support. The frameworks have been setup to work with both CPU and GPU. It can detect the GPU on the machine and run with CUDA driver. The GPU boost for deep learning is important. At most of the cases, a GPU can boost the training speed to more than double.

Keras, Pytorch and Tensorflow are three popular frameworks can be run in Python. We first remove the selection of Keras since it’s higher level framework that using Tensorflow as the backbone. In this project, we may need to modify the algorithm in a low level. Rather than relying on a high-level framework, we want to start constructing the neural network from a low level. Therefore, the last two choices are Tensorflow and Pytorch.

Tensorflow is a mature framework with a large community, which also contribute to a “contrib” package in tensorlfow framework. Tensorflow 2.0 is extremely popular around the world for both research and production. In the other hand, Pytorch is a younger framework, which just release their 1.0 version. However, it also works with Microsoft to improve their production process. Moreover, the Pytorch is completely written in Python. And, part of the Tensorflow is written in C language. For the readability, Pytorch will be a better option to understand the base part of framework. The major difference between these two frameworks is the Tensorflow is static and the Pytorch is dynamic. The static computational graph require user to construct their neural network first, and feed in the data to run the session latter. This separation may negatively affect the debugging experience since the architecture can’ t be checked till you run the session. Moreover, the tensor in the Pytorch framework can be easily transform to the numpy array, which allow users to check the input and output of each operation. Tensorflow also provide “eager-mode”, which makes Tensorflow dynamic. However, this eager mode doesn’t work well with other static functions and the contrib library in tensorflow. Therefore, I choose Pytorch as the deep learning framework on this project, which is dynamic and better for debugging and research.

1. Creating GUI

To create the graph user interface, Tkinter is the library we use. This library allows users to create a window and design the widget inside the window. This toolkit has several functions enable users to locate and interacting with widgets, including “CheckButton”, “Entry” and “Text”. Out project use this library to create multiple widgets. Hence, user can interact with the “args” class, which contains all the hyper-parameters. For example, the “CheckButton” can set the Boolean value and the “Entry” can handling scalar or string value in the hyper-parameters. Moreover, “Text” is the class we use for showing the training and test result.

1. Hardware:

The machine we use is a laptop, Alienware-15-R4. This machine currently dual boot Windows 10 and Ubuntu 18.04; nonetheless, this project only use the Ubuntu system. The memory in this machine is 16GB, which is able to load whole HSpam dataset. The Processer we have is Intel® Core™ i7-8750H CPU @ 2.20GHz × 12. This processor mainly used in pre-processing phase, such as generating dictionary, tokenizing sentences and mapping word to index. The GPU is GeForce GTX 1070/PCIe/SSE2, which is compatible with CUDA and Pytorch. We run most of the training and test parallelly on the GPU rather than CPU.

## Procedures

### Extracting data from SQL

After scanned through the dataset, we found some informative features for tweets and user.

|  |  |  |
| --- | --- | --- |
| Name | Tweets | Users |
| Features | text  numberOfHashtags\_c  favorite\_count  retweet\_count  possibly\_sensitive | followers\_count  friends\_count  default\_profile  default\_profile\_image  favourites\_count  listed\_count  statuses\_count  verified |

Table (1)

Two types of files we use for extracting the data. We firstly try to extract the tweets’ text to csv and json file. However, both of these wo file types will have problem during reading since the special characters in the tweets may affect the reading process. Therefore, we end up storing the text data in html format, which will not be affected by the special characters. For other user and tweets features, we stored them at .csv file since those features do not contain special characters. Moreover, the size of .csv data is considerably smaller than .csv file, which also affects the speed of reading into memory. The whole dataset consists of 1,308,468 observations. 1,145,136 of them are labelled and can be used for training. Therefore, we extract all labelled data as our dataset for training, validation and test.

### Pre-processing

The pre-processing strategy applied in our project consists of 7 steps:

Initial example: We are SANDWICHES @lunch [www.sandwiches.com/shopping/beef](http://www.sandwiches.com/shopping/beef)

|  |  |  |
| --- | --- | --- |
| Step | Operation | Example |
|  | Replace the URLs by its own domain name only. | We are SANDWICHES 4.99 dollars @lunch [www.sandwiches.com](http://www.sandwiches.com/shopping/beef) |
|  | Remove all digits. | We are SANDWICHES dollars . @lunch  [www.sandwiches.com](http://www.sandwiches.com/shopping/beef) |
|  | Tokenise the sentence. | [ We, are, SANDWICHES, dollars, ., @lunch, [www.sandwiches.com](http://www.sandwiches.com/shopping/beef)] |
|  | Lower every character | [ we, are, sandwiches, dollars, ., @lunch, [www.sandwiches.com](http://www.sandwiches.com/shopping/beef)] |
|  | Remove stopwords. | [ sandwiches, dollars, ., @lunch, [www.sandwiches.com](http://www.sandwiches.com/shopping/beef)] |
|  | Remove the token length < 3 | [ sandwiches, dollars, @lunch, [www.sandwiches.com](http://www.sandwiches.com/shopping/beef)] |
|  | Apply Potter Stemming | [ sandwich, dollar, @lunch, [www.sandwiches.com](http://www.sandwiches.com/shopping/beef)] |
|  | After the above pre-processing steps, the tokens will be mapped to index through the dictionary created by the training set. | |

Table (2)

### Creating Deep learning model

Two types of model are created in this project. One of them only receive the tweets’ text as input. Another accepts both tweets’ text and other user and tweet features listed in the Table (1). And, the architecture of these two models are shown below.

|  |
| --- |
| Only receive tweets text |
|  |
| This model is only used for handling the text data. The text model can be one of SSCL, GatedCNN or SelfAttn. The embedding layer we used here is trainable, which can be trained with the text model. The output of this model will be a scalar with value between 0 and 1. |

|  |
| --- |
| Receive both text and other user and tweets information |
|  |
| This model can handling both text data and the structured data. The left side of the model is almost the same as the last model. The only different between the left hand side model and the last model is the output dimension. The output dimension of last model has to be 1 to calculate the loss value. However, in this model, we doesn’t use the textModel output to calculate the loss; therefore; it can be any value. The right hand side, Info Model, is used for handling other tweet and user information. The Info Model consist of four fully connected layers. The Combine Model is consist of two fully connected layer. And, the the input and output dimension of Combine Model are (infoModel\_outDim + textModel\_outDIm) and 1 respectively. |

### Create GUI

To create the graph user interface, the most important part is to decide which hyper-parameters the GUI can tune. After deterring the controllable hyper-parameters, we can put the widgets in the window. The GUI is only one of the methods to interacting with the models, but it’s more convenient and visualisable.

|  |
| --- |
| GUI Demo |
| https://github.com/ChihchengHsieh/SpamDetection-TweetsAndUserInfo/raw/master/TrainingResult/runningOnSamllDataset.png |

1. Train the Neural Network

The training can be started by the GUI software. Also, the training and validation result will be shown at the right side. The test accuracy and loss will be shown after the end of training. Moreover, the “Show Model” button can show the number of parameters and the structure of current model. When the training start, the information about the dataset in use will be shown in the result box.

|  |
| --- |
| Training Demo |
| https://github.com/ChihchengHsieh/SpamDetection-TweetsAndUserInfo/raw/master/TrainingResult/SSCL_MultiTask.png?raw=true |
| The left bottom text box will show the training and test records. And, the right top canvas represents the training loss and accuracy. The right bottom plot both training and validation records for accuracy and loss. |

# OUTCOMES

The graph user interface of the neural network is one of our deliverables. We use this GUI to train, test and tune the deep learning models. Therefore, we want to provide a brief instruction of the GUI. This instruction will show how can the GUI interact with the created neural networks.

## Brief Manual:

|  |
| --- |
| DEMO of the GUI |
|  |
| 1. The section 1 is for deciding the dataset and overall architecture. Currently, only on dataset, HSpam14, can be selected. After selecting the dataset, the user has to decide whether they want to use infoModel or textModel. If the user decides to include the textModel, they have also to determine which textModel they want to use. The options for textModel include SSCL, GatedCNN and SelfAttn. 2. The section 2 is used for designing the textModel architecture. After selecting a different textModel, this section will be changed according to the model you select. 3. The section three is mainly used for managing the training, including the dataset split, vocabulary size and other parameters affecting the training but not architecture. Although this section will not affect the architecture and the number of trainable parameters, this section is still important. For example, the learning is an important hyper-parameter that can affect optimization process. 4. This section only contains two Boolean hyper-parameters, which allow user to use different type of dataset. 5. The training and the text result will be shown in the section 5. Moreover, when the “Show Model” button in the section 1 is clicked, the model architecture will be shown in this section as well. 6. Section 6 show the training loss and accuracy. The frequency of plotting the training records can be adjusted by the hyper-parameter, “log\_freq”, in section 3. 7. The training and validation records can be seen in this section. The first graph shows the loss and the second graph show the accuracy. This section is the main section for supervising the training. From this graph, we can determine whether the training converge, or the overfitting occur. |

## All Hyper-parameters

Section 1

|  |  |
| --- | --- |
| Name | Description |
| HSpam14 | The dataset that the user wants to load. Currently, only one dataset is supported. This option must to be selected or the training will not start. |
| Model (SSCL, GatedCNN, SelfAttn) | If the user wants to apply the textModel for processing the input tweets’ text, one of these models has to be selected. |
| Using \_textModel,  Using\_infoModel | Whether to use the textModel or infoModel. The user can select both of them or one of them. Moreover, if using\_textModel is ticked, user has to select one textModel above. |
| MultiTask\_FCHidden | This is the hyper-parameter for controlling the number of hidden units in the infoModel. The infoModel include four fully connected layers. The 1st and 2nd layers will have MultiTask\_FCHidden\*2 hidden units. The number of hidden units in 3rd layer will be MultiTask\_FCHidden. And the last layer has infoModel\_outDim hidden units. And, the last layer will also be the output of the infoModel. |
| textModel\_outDim | If only the textModel is selected, this hyper-parameter will be fixed as 1 automatically. If both textModel and infoModel are selected, this hyper-parameter will be adjustable. This hyper-parameter controls the output dimension of the textModel. |
| infoModel\_outDim | Similar to the above hyper-parameter, this hyper-parameter will only be adjustable when both infoModel and textModel are in use. It controls the output dimension of infoModel. |
| combine\_dim | This parameter is used for deciding the number of hidden units of the combine model. The combine model only consists of two fully connected layer. The input dimension of the combine model is (infoModel\_outDim + textModel\_outDim) and the output dimension is 1. The combine\_dim adjust the number of hidden units in 1st layer in combine model. |
| Show Model Button | Sometimes, we don’t want to run through the whole training process, but we still want to check the models we design. Then this button is used for this case. This button will show the number of parameters in the whole model. Also, it will show the details of the architecture; therefore, the user can easily check how the parameters on the GUI affect the neural network architecture. |
| Start Training Button | Click this button to load the data and start the training. The training records will be shown in the section 5, 6 and 7. |

Section 2 (SSCL)

|  |  |
| --- | --- |
| Name | Description |
| SSCL\_RNNHidden | The hidden dimensions of LSTM. |
| SSCL\_CNNDim | The hidden dimensions of Convolution Neural Network. |
| SSCL\_CNNKernel | The kernel size of the CNN. This hyper-parameter also represents the n-gram strategy we want to apply. |
| SSCL\_CNNDropout | The dropout possibility of the units after the CNN layer. |
| SSCL\_LSTMDropout | The dropout possibility of the units after the LSTM layer. |
| SSCL\_LSTMLayers | Number of LSTM layers. Too many layers may considerably reduce the training and inference speed. |
| SSCL\_embeddingDim | Dimension for the embedding layer. Another way to describe this hyper-parameter is the size of the word vector. |

Section 2 (GatedCNN)

|  |  |
| --- | --- |
| Name | Description |
| GatedCNN\_embeddingDim | Dimension for the embedding layer. Another way to describe this hyper-parameter is the size of the word vector. |
| GatedCNN\_convDim | The number of filters in the CNN layer. |
| GatedCNN\_kernel | The kernel size of CNN layer. |
| GatedCNN\_stride | The stride of the CNN layer. The stride represents the move distance after every convolutional operation. |
| GatedCNN\_pad | Padding for the CNN layer on the width and height dimension. |
| GAtedCNN\_layers | Number of CNN layers. A CNN block consists of a CNN layer, a batch normalisation and an activation function. |
| GatedCNN\_dropout | The dropout rate after each CNN block. |

Section 2 (SelfAttn)

|  |  |
| --- | --- |
| Name | Description |
| SelfAttn\_LenMaxSeq | The max sequence length in the dataset after the pre-processing steps. Since we’re using the tweets as the input data, which are restricted in 280 words. Therefore, the default value is set as 280. |
| SelfAttn\_ModelDim | The model dimension for operating the self-attention. This model dimension will also represent the value of embedding dimension since they are restricted to be the same in order to perform self-attention. |
| SelfAttn\_FFInnerDim | This hyper-parameter is used for deciding the dimension in PositionwiseFeedForward block. |
| SelfAttn\_NumLayers | Number of the self-attention blocks. One self-attention block consists of one MultiHeadAttention block followed by a PositionwiseFeedForward block. |
| SelfAttn\_NumHead | Number of heads in the MultiHeadAttention block. We can understand this as “having more heads means wider hidden layer”. Therefore, this hyper-parameter affects the width of the neural network rather than depth. |
| SelfAttn\_KDim | The K dimension that needed for operating the self-attention. This value can’t be too large, or the matrix multiplication in the self-attention operation may cause memory leaks. |
| SelfAttn\_VDim | The V dimension that needed for operating the self-attention. This value can’t be too large, or the matrix multiplication in the self-attention operation may cause memory leaks. |

Section 3

|  |  |
| --- | --- |
| Name | Description |
| Vocab\_size | The vocabulary size for creating the nltk.Text object. All the tokens will be ranked by their occurrence. And, the final vocabulary will contain the most frequent “vocab\_size” words. In default, the most frequent 30000 tokens will be stored in the vocabulary corpus. |
| Validation\_portion | Used for controlling the validation dataset size. However, the final dataset size will also be determined with both validation\_portion and test\_protion. |
| Test\_portion | Used for controlling the test set size.  If the whole dataset size is N, then:  Training set size -> N\*(1-validation\_portion)  Validation set size -> N\*(validation\_portion) \* (1-test\_portion)  Test set size -> N\*(validation\_protion) \* (test\_protion) |
| Batch\_size | The batch\_size is how many observations you want to feed in the neural network during training. A larger batch\_size can enable a stable training. However, it will also be restricted by the memory. |
| L2 | The L2 is used for punishing the weight in the neural network. It’s a term that can prevent the overfitting. |
| n\_epoch | Number of epochs that the training will run. |
| earlyStopStep | Number of steps that the training will run for early stop. |
| earlyStopEpoch | Number of epochs that the training will run for early stop. (This will almost have the same effect as n\_epoch). However, the training will stop at min (n\_epoch, earlyStopEpoch, earlyStopStep), which means when the model reach one of the stop condition, then training will stop. |
| Val\_freq | The frequency to run the model on the validation set to have validation loss and accuracy. The section 7 results will be controlled by this hyper-parameter. A higher value of val\_freq will plot more details in section 7. However, running on the validation set will not adjust the weight, which means a frequent plotting may cause a lower training efficiency. |
| Val\_steps | How many steps to run during recording the validation loss and validation accuracy. Running more steps can provide more accurate average loss and accuracy; nonetheless, it will slow down the training as well. |
| Log\_freq | The frequency for plotting the section 6. |
| Model\_save\_freq | The frequency for saving the weight of neural network to the local machine. The saved model can be found in ./HSpam14\_”model\_name”/Model if the HSpam14 dataset is in used. |
| Model\_name | The name of the current model. This model name will affect the folder name for storing the weights and training records. |
| Scheduler\_step  Schedular\_gamma  Scheduler\_minLr | These three hyper-parameters are used for setting the scheduler to adjust the learning rate. From these three arguments, the dynamic learning rate will be set as “After each schdular\_step, the learning rate will multiple scheduler\_gamma.”. And the minimum learning rate is scheduler\_minLr. If we don’t want to apply learning rate schedular, the shcedular\_gamma can be set to 1. |

Overall Structure of the implementation

|  |  |
| --- | --- |
| All\_Models.py | This file contains all of the implementation of neural networks, which includes MultiTaskModel, SSCL, GatedCNN and SelfAttnModel. If the user wants to implement more models to the |
| Constants.py | This file storing the constant used for padding and mapping the token to index. The <SOS> (start of sentence) will be attached to the start of the sentence. <EOS> (end of sentence) will also be appended in the end. The <UNK> will be used for replacing the tokens not in the vocabulary corpus. Moreover, the <PAD> is used for padding the sequence to the same length, which enables the batch training. |
| Datasets.py | The file store the torch dataset classes that we use to feed the data into neural networks. If the user wants to add a new dataset, the new structure of the dataset should be added into this file. |
| LoadData.py | This file contains the functions for loading and pre-processing the data. If user want to add a new dataset, one loading function can be added into this file. Moreover, the function “preprocessingInputTextData” in this file can also be modified to apply on the new dataset. |
| SubModels.py | Since the implementation of SelfAttn model is complicated, some sublayer classes are stored in this file. |
| TkinterTweetsWithUserSpamDetection.ipynb | This is the main file that we want to run. Two version of the file are provided. One is the Jupyter Notebook .ipython file. Another is the .py file, which can be run in the bash. If the user want to adjust the GUI system, this file is where the user want to modify. Moreover, the ipython file can work better with this implementation since some tools in the Tkinter library are set with Jupyter Notebook. |
| TkinterTweetsWithUserSpamDetection.py | This is almost the same as the ipython file above. If the user wants to run the GUI software in the bash, this file should be executed through Python 3. |
| Trainers.py | This is the .py file containing the Trainer class. The Trainer class has several useful functions, including plotting training history, saving trained models and performing feedforward. If the user wants to adjust the training strategy, the optimization or loss functions can be changed in this file. |
| utils.py | This file storing multiple useful functions that can be used in every part of the implementation. |

Training Outcomes:

|  |
| --- |
| The first training with SSCL model. |
| https://github.com/ChihchengHsieh/SpamDetectionProject/raw/master/ModelLog/WithOutPunishment/All_Hist_SSCL.png?raw=true |
| In this training, although the accuracy is higher than 85%, the model doesn’t work well. When we check the confusion matrix, we can find the model classify all the inputs as the legitimate user, which means the model did not learn useful information. In order to alleviate this problem, two strategies we apply. One of them is to push the malicious user more. For example, when a data of malicious user is fed into the model, if the model didn’t classify this instance correctly, the weights in the model should be adjusted more than the input of legitimate user. Another is to use weighted random sampling, which will sample more malicious user from the initial dataset. This strategy can make the number of malicious users equals to the number of legitimate users, which can alleviate the problem. |

|  |
| --- |
| After applying the punishment on the malicious users. |
| https://github.com/ChihchengHsieh/SpamDetectionProject/raw/master/ModelLog/WithPunishmentOnSpammer/All_Hist_SSCL.png?raw=true |
| The problem is still the same, this model still recognises every input as legitimate user. In order to make the model focus on the malicious user, we want to increase the ratio of punishment. The current punishment ratio is around 1.2, which means the loss of a malicious users will be multiplied by 1.2. |

|  |
| --- |
| Increase the punishment ratio to 1.4 |
| https://github.com/ChihchengHsieh/SpamDetectionProject/raw/master/ModelLog/WithPunishmentAndRatio1.4/All_Hist_SSCL.png?raw=true |
| The model still can’t be trained. However, the model starts to classify all the input as the malicious user now. We can see from the training accuracy. The classifier recognises all the inputs as malicious users or legitimate users. The next strategy is to apply the weighted random sampling. |

|  |
| --- |
| After applying the weighted random sampling |
| https://github.com/ChihchengHsieh/SpamDetectionProject/raw/master/ModelLog/LargeModelWithWeightedRandomSampling/All_Hist_SSCL.png?raw=true  Confusion Matrix:  https://github.com/ChihchengHsieh/SpamDetectionProject/raw/master/ModelLog/LargeModelWithWeightedRandomSampling/ConfusionMatrix.png?raw=true |
| The classifying problem still occurs. Therefore, instead of modifying the dataset and the sampler, we want to check the trainability of the model. The current model is using the hyper-parameter from the paper. However, some details are not provided in the paper, which means some parts of code are implemented by our own strategy. We consider the problem may be caused by the gradient explosion or vanishment, which commonly occurs in all RNN-based models. To solve the problem, we want to sum up the outputs of LSTM in each timestamp. Summing up the output can improve the propagation of gradient theoretically. |

After summing the outputs of each timestamp, the model can be trained now.

|  |
| --- |
| SSCL |
| https://github.com/ChihchengHsieh/IFN702SpamDetection/raw/master/TrainingResults/SSCL_1.png?raw=true |
| The training start to converge. And, the SSCL model reaches 93.84% accuracy on the test set. The reason of setting the earlyStopStep as 4000 is the problem of overfitting. When this model train to more than 10000 steps, it will have overfitting problem, which may be caused by the large number of trainable parameters. The other two models, GatedCNN and SelfAttn, also have this problem. |

|  |
| --- |
| GatedCNN |
|  |
| The GatedCNN achieve 94.76% on the test set. However, from the training history plot, we can find that sometimes the model is broken on the validation set. The low accuracy on the validation set may be caused by the outlier in the training set, which dramatically adjust the weights and break the model. However, from training records above, we can see the training accuracy is not affected; hence, the cause should simply be the overfitting. |

|  |
| --- |
| SelfAttn |
| https://github.com/ChihchengHsieh/IFN702SpamDetection/raw/master/TrainingResults/SelfAttn_3.png?raw=true |
| The SelfAttn model reach the accuracy of 93.45% on the test set. The reason to set the SelfAttn\_Droptout is to prevent the overfitting, which occurs when SelfAttn\_Dropout = 0.1. |

Even the model has become trainable, the above three models still have the overfitting problem. To solve the overfitting problem, we reduce the number of trainable parameters in each model to be around 1,000,000. Moreover, another general method to alleviate the overfitting is to feed more data to the dataset. Therefore, we will still run the model till one epoch even the model has converged. Lastly, the test portion can be larger to gain more objective accuracy on the test set. Therefore, the test set size become 5762.

|  |
| --- |
| SSCL |
| https://github.com/ChihchengHsieh/SpamDetection-TweetsAndUserInfo/raw/master/TrainingResult/textModelOnly_SSCL.png?raw=true  Number of Parameters: 971,713 |
| From the screenshot above, we can clearly know the parameter we set. Almost all of the parameters are reduced in this case to reduce the capacity of model; hence, the overfitting problem can be solved. Moreover, this model reach accuracy of 94.15% on a larger test set. And, only 67 malicious users cannot be recognised in this model. Also, we can see the model converged around 2000 steps. However, we continue to train the model as a regularization strategy. |

|  |
| --- |
| GatedCNN |
| https://github.com/ChihchengHsieh/SpamDetection-TweetsAndUserInfo/raw/master/TrainingResult/textModelOnly_GatedCNN.png.png?raw=true  Number of Parameters: 991,585 |
| From above plot, an obvious phenomenon we can see. After the training accuracy converge, the validation accuracy is still unstable. This interesting phenomenon may be caused by the overfitting. After feeding more data to the model, the validation accuracy become stable as well. |

|  |
| --- |
| SelfAttn |
| https://github.com/ChihchengHsieh/SpamDetection-TweetsAndUserInfo/raw/master/TrainingResult/textModelOnly_SelfAttn.png?raw=true  Number of Parameters: 1,026,145 |
| In the training and validation plot, we can see both training and validation accuracy decrease for roughly 5 steps. The outliers may contribute to this. Another method to explain the decrease is the loss surface. An input data causes a large loss and weight adjustment, which breaks the model. However, since the adjusted weight cannot fit other training data, the new weight will be changed immediately according to the loss and gradient. |

After testing the textModel, we found that some useful information in the dataset are not used (see at literature review HSpam14). Therefore, we want to include more information about the user and tweet to boost the performance. For feeding more data as input, we made an ensemble-like model. Unlike some ensemble model using voting are averaging, our model allows the propagation of gradient; hence, the whole model can be trained together.

|  |
| --- |
| SSCL + InfoModel |
| https://github.com/ChihchengHsieh/SpamDetection-TweetsAndUserInfo/raw/master/TrainingResult/SSCL_MultiTask.png?raw=true  Number of Parameters: 975, 889 |
| After adding the infoModel and combineModel, the accuracy has been improved from 94.15% to 94.9%. Moreover, the training become more stable. In the case of textModel only, the textModel may not be able to get over 90% accuracy every time. This may be affected by the dataset or the optimization method. However, after applying the infoModel, the accuracy is easy to reach 93 or 94%. |

|  |
| --- |
| GatedCNN + infoModel |
| https://github.com/ChihchengHsieh/SpamDetection-TweetsAndUserInfo/raw/master/TrainingResult/GatedCNN_MultiTask.png?raw=true  Number of Parameters: 995,761 |
| This model reaches 95.67% accuracy, which is the highest accuracy in the whole project. Combining the GatedCNN and infoModel considerably boost the performance. Moreover, the GatedCNN was a model that easy to suffer from overfitting. After applying the ensemble, the overfitting problem rarely happens. |

|  |
| --- |
| SelfAttn + infoModel |
| https://github.com/ChihchengHsieh/SpamDetection-TweetsAndUserInfo/raw/master/TrainingResult/SelfAttn_MultiTask.png?raw=true  Number of Parameters: 1,030,321 |
| I expect this model to work best since it’s a very popular NLP model. Also, this model has the highest accuracy when we only use textModel. However, according to the training and validation plot, the training is not stable, and the accuracy is reduced to 94.92% from 95.3%. The SelfAttn model may not work well with infoModel. |

From the GatedCNN + infoModel and SSCL + infoModel, we can see the number of parameters only increase for a small amount. However, the training become more stable and performance is improved.

After using the ensemble model, we think the infoModel maybe another trainable model. Therefore, we extract the infoModel from ensemble model. And train the infoModel only.

|  |
| --- |
| InfoModel |
| https://github.com/ChihchengHsieh/SpamDetection-TweetsAndUserInfo/raw/master/TrainingResult/OnlyInfoModel.png?raw=true  Number of Parameters: 1,596,417 |
| Even the number of parameters has been set to be more than the ensemble model, the model still can’t be trained. It implies that the infoModel is a supportive role for the textModel. The textModel with infoModel can be better and more stable. In the other hand, the infoModel without textModel will not be trainable. |

Discussion:

Conclusion:

Reflection: