



國立臺北科技大學

自動化科技研究所

碩士學位論文

結合 CNN 與 LSTM 模型
建構分類惡意程式方法

Development of a Malware Classification
Approach Based on CNN and LSTM Models

研究生：黃綱正

指導教授：陳文輝 博士

中華民國一百零八年十一月

摘要

論文名稱：結合 CNN 與 LSTM 模型建構分類惡意程式方法

頁數：五十二頁

校所別：國立臺北科技大學 自動化科技研究所碩士班

畢業時間：一百零八學年度 第一學期

學位：碩士

研究生：黃綱正

指導教授：陳文輝 博士

關鍵詞：惡意程式分類、深度學習、卷積神經網路、長短期記憶模型

本論文主要是利用深度學習模型進行惡意程式分類。目前已有研究利用機器學習方式實作(如：SVM、Decision Tree 等)，近年研究已利用卷積神經網路及循環神經網路等類神經網路模型進行分類，相較於機器學習方式，具備較高準確率。本論文提出之模型，利用 Malimg 資料集作為惡意程式分類實驗，並與原提出論文作比較，準確率由 84.92%提高至 87.79%。所運用模型結合卷積神經網路與循環神經網路之優點，解決惡意程式樣本之特徵提取後資料序列分類問題。

ABSTRACT

Title: Development of a Malware Classification Approach Based on CNN and LSTM Models

Pages: 52

School: National Taipei University of Technology

Department: Graduate Institute of Automation Technology

Time: November 2019

Degree: Master

Researcher: Gang-Cheng Huang

Advisor: Wen-Hui Chen, Ph.D.

Keywords: Malware Classification, Deep Learning, Convolution Neural Network, Long Short-Term Memory

This thesis proposed a malware classification method by using deep learning models. In recent years, there have been implemented not only by machine learning (e.g., SVM, decision tree, etc.) but also by convolution neural networks (CNNs), and recurrent neural networks (RNNs). Many studies had experimented that using deep learning models had much higher accuracy than machine learning models. The proposed model in this thesis had achieved an improvement of the accuracy from 84.92% to 87.79% for evaluating the “Maling” malware dataset. This architecture involved using the advantage of CNNs for feature extraction on the input data with LSTMs to support sequence classification.

誌 謝

在整個研究過程中，由衷感謝為我的指導教授陳文輝教授，無論是課堂、課後及論文研討期間，都以嚴格方式要求學生，藉由論文、實作練習，讓學生增廣見聞，進而精益求精，實事求是；另老師也非常鼓勵與支持學生參加校外活動與研討會，廣開見聞，由於個人並非專業於人工智慧領域，初次踏入偶有陌生，陳教授不厭其煩地解釋其中演算法及原理，有如醍醐灌頂般。

在就讀碩士中，參加第三屆教育部臺灣好屬駭，所幸透過徵選擁有資安實務導師培訓資格，資安實務導師為國立臺灣大學資訊工程學系暨研究所蕭旭君老師，以引導方式提供利用深度學習方式解決惡意程式分類之相關論文文章。另外為國立臺灣科技大學資訊管理學系吳宗成教授，於臺灣好屬駭之期中報告，藉由成果發表所提出之問題給予建議與鼓勵。另外於期末發表時，感謝崑山科技大學資訊工程學系曾龍副教授針對研究上提出不足之處，在教育部第三屆台灣好屬駭中，也很榮幸獲得表現優異獎，感謝各位資安先進支持。在研究之餘，也感謝國防大學理工學院資訊安全訓練中心主任暨電機電子工學學系張克勤副教授於資訊安全相關技術提供建議與方向。

碩士論文口試校外委員分別為國立臺灣大學資訊工程學系暨研究所蕭旭君副教授、國防大學理工學院翁旭谷副教授及財團法人資訊工業策進會資安科技研究所毛敬豪所長，三位老師給予不少指導，增進研究成果。

其次在惡意程式、漏洞研究及資訊安全相關知識，徐千洋(Tim Hsu)先生在國家安全會議中針對資訊安全的 BOB(Best of Best)課程中，利用專題報告時機，以導師引導方式調整我的學習方向。再來是感謝辦公室同仁，同時也是碩士班的同學洪維德，這兩年期間共同選課，無論是課前課後，在不停的研討，一起研究作業及課業上問題。

最後感謝是家人支持與鼓勵，在就讀碩士中，平日工作之餘，晚上赴學校上課，假日回家也準備溫暖的環境，方可專心於研究上，順利完成碩士學位。

黃綱正 於

國立臺北科技大學自動化科技研究所

中華民國一百零八年十一月四日

目 錄

摘要	i
ABSTRACT	ii
誌謝	iii
目 錄	iv
表目錄	v
圖目錄	vi
第一章 緒論	1
1.1 研究動機	1
1.2 研究方法	4
1.3 論文架構	4
第二章 相關研究	5
2.1 惡意程式研究	5
2.2 類神經網路研究	16
2.2.1 多層感知器與 Dropout	17
2.2.2 卷積神經網路	20
2.2.3 循環神經網路	20
2.2.4 交叉熵誤差與梯度下降	21
2.2.5 誤差反向傳播法	22
2.3 基於類神經網路對於惡意程式分類技術	23
2.3.1 樣本資料前置處理	23
2.3.2 微軟惡意程式分類比賽第一名演算法	24
2.3.3 作者 Daniel Gribert 提出演算法	25
2.3.4 作者 Abien Fred M. Agarap 演算法	26
第三章 系統設計	27
3.1 資料前置處理	27
3.2 類神經網路設計	31
第四章 實驗結果分析與討論	32
4.1 實驗環境	32
4.2 實驗結果與比較	32
4.2.1 CNN 模型	33
4.2.2 RNN 模型	35
4.2.3 LSTM 模型	37
4.2.4 GRU 模型	39
4.2.5 CNN+RNN 模型	41
4.2.6 CNN+LSTM 模型	43
4.2.7 CNN+GRU 模型	45
4.3 實驗綜合比較	47
第五章 結論與未來工作	48
參考文獻	49

表目錄

表 1.1	惡意程式種類名詞定義.....	2
表 1.2	網路攻擊狙殺鍊說明.....	3
表 2.1	檔案標頭常見區段與說明.....	7
表 3.1	惡意程式樣本 Malimg 詳細資訊.....	27
表 4.1	硬體及系統環境說明.....	32
表 4.2	使用參數比較.....	47
表 4.3	結果比較.....	47



圖目錄

圖 1.1	TWNIC 惡意程式統計數據圖.....	2
圖 1.2	網路攻擊狙殺鍊手法.....	3
圖 2.1	DOS 標頭資料結構.....	5
圖 2.2	PE 標頭資料結構.....	6
圖 2.3	IMAGE_OPTIONAL_HEADER32 資料結構.....	6
圖 2.4	利用 UPX 加殼處理及一般程式碼編譯後執行檔.....	7
圖 2.5	利用 UPX 加殼器反組譯情況.....	8
圖 2.6	利用除以 0 產生例外處理進而修改程式執行邏輯.....	8
圖 2.7	無意義加花指令.....	9
圖 2.8	利用判斷式進行程式流程混淆.....	9
圖 2.9	以雙引號字串資料作為初始化.....	10
圖 2.10	以字元方式壓入線中建立字串.....	10
圖 2.11	利用 Boost C++ 於原始碼中加解密處理.....	11
圖 2.12	於執行階段中密文與明文對比.....	11
圖 2.13	利用 PE Info 查看程式所使用 DLL 及 API 函數.....	12
圖 2.14	實作抓取 Kernel32.dll 位址.....	13
圖 2.15	實作 GetProcAddress 目的是後續可動態取得 API 函數.....	13
圖 2.16	利用動態載入、呼叫函數方式.....	14
圖 2.17	利用 PE Info 查看利用動態呼叫、載入方式之惡意程式.....	14
圖 2.18	利用 cdecl 與編譯後結果.....	15
圖 2.19	利用 stdcall 與編譯後結果.....	15
圖 2.20	裸函數實作，其中裡面藏有加花指令.....	16
圖 2.21	監督式學習與非監督式學習比較圖.....	17
圖 2.22	多層感知器(MLP, Multilayer Perceptron)運作圖.....	17
圖 2.23	每一層利用 0.5% 機率丟棄之神經元.....	18
圖 2.24	沒有使用 Dropout 之神經網路圖.....	19
圖 2.25	使用 Dropout 之神經網路圖.....	20

圖 2.26	透過矩陣相乘運算得到可能特徵.....	20
圖 2.27	循環神經網路神經元執行流程圖.....	21
圖 2.28	長短期記憶模型執行流程圖.....	21
圖 2.29	反向傳播法以圖式顯示.....	23
圖 2.30	於 DL4MD 論文中提出惡意程式資料前置處理方.....	24
圖 2.31	惡意程式樣本圖像化.....	24
圖 3.1	資料集長寬大小(shape)與規一化數值範圍值(max 與 min).....	27
圖 3.2	類別 1- Adialer.C.....	28
圖 3.3	類別 2- Agent.FYI.....	28
圖 3.4	類別 3- Allapple.A.....	28
圖 3.5	類別 4- Allapple.L.....	28
圖 3.6	類別 5- Alueron.gen!J.....	28
圖 3.7	類別 6- Autorun.K.....	28
圖 3.8	類別 7- C2Lop.P.....	29
圖 3.9	類別 8- C2Lop.gen!G.....	29
圖 3.10	類別 9- Dialplatform.B.....	29
圖 3.11	類別 10- Downloader Dontovo.A.....	29
圖 3.12	類別 11- Fakerean.....	29
圖 3.13	類別 12- Instantaccess.....	29
圖 3.14	類別 13- Lolyda.AA 1.....	29
圖 3.15	類別 14- Lolyda.AA 2.....	29
圖 3.16	類別 15- Lolyda.AA 3.....	29
圖 3.17	類別 16- Lolyda.AT.....	29
圖 3.18	類別 17- Malex.gen!J.....	30
圖 3.19	類別 18- Downloader Obfuscator.AD.....	30
圖 3.20	類別 19- Rbot!gen.....	30
圖 3.21	類別 20- Skintrim.N.....	30
圖 3.22	類別 21- Downloader Swizzor.gen!E.....	30
圖 3.23	類別 22- Downloader Swizzor.gen!I.....	30

圖 3.24	類別 23- VB.AT.....	30
圖 3.25	類別 24- Downloader Wintrim.BX.....	30
圖 3.26	類別 25- Yuner.A.....	31
圖 3.27	運用 CNN 與 RNN 類神經網路模型結合示意圖.....	31
圖 4.1	利用 matplotlib 繪製 CNN 訓練之正確率(y)與訓練週期(x).....	33
圖 4.2	利用 matplotlib 繪製 CNN 訓練之 log loss(y)與訓練週期(x).....	34
圖 4.3	利用 matplotlib 繪製 CNN 模組訓練之混淆矩陣.....	34
圖 4.4	利用 matplotlib 繪製 RNN 訓練之正確率(y)與訓練週期(x).....	35
圖 4.5	利用 matplotlib 繪製 RNN 訓練之 log loss(y)與訓練週期(x).....	36
圖 4.6	利用 matplotlib 繪製 RNN 模組訓練之混淆矩陣.....	36
圖 4.7	利用 matplotlib 繪製 LSTM 訓練之正確率(y)與訓練週期(x).....	37
圖 4.8	利用 matplotlib 繪製 LSTM 訓練之 log loss(y)與訓練週期(x).....	38
圖 4.9	利用 matplotlib 繪製 LSTM 模組訓練之混淆矩陣.....	38
圖 4.10	利用 matplotlib 繪製 GRU 訓練之正確率(y)與訓練週期(x).....	39
圖 4.11	利用 matplotlib 繪製 GRU 訓練之 log loss(y)與訓練週期(x).....	40
圖 4.12	利用 matplotlib 繪製 GRU 模組訓練之混淆矩陣.....	40
圖 4.13	利用 matplotlib 繪製 CNN 訓練之正確率(y)與訓練週期(x).....	41
圖 4.14	利用 matplotlib 繪製 CNN+RNN 訓練之 log loss(y)與訓練週期(x).....	42
圖 4.15	利用 matplotlib 繪製 CNN+RNN 模組訓練之混淆矩陣.....	42
圖 4.16	利用 matplotlib 繪製 CNN+RNN 訓練之正確率(y)與訓練週期(x).....	43
圖 4.17	利用 matplotlib 繪製 CNN+LSTM 訓練之 log loss(y)與訓練週期(x).....	44
圖 4.18	利用 matplotlib 繪製 CNN+LSTM 模組訓練之混淆矩陣.....	44
圖 4.19	利用 matplotlib 繪製 CNN+LSTM 訓練之正確率(y)與訓練週期(x).....	45
圖 4.20	利用 matplotlib 繪製 CNN+GRU 訓練之 log loss(y)與訓練週期(x).....	46
圖 4.21	利用 matplotlib 繪製 CNN+GRU 模組訓練之混淆矩陣.....	46

第一章 緒論

1.1 研究動機

近年來，隨著資訊技術蓬勃發展，到現今社會人手一隻智慧型手機，一台電腦，資訊逐漸普及化，網路發展也隨之興起，但網路犯罪(cybercrime)也隨之發生，尤其是網路地下市場販售之軟體，透過散播惡意程式(malware)達到勒索、竊取情資或對資料、設備等造成一定程度之損害。防毒軟體(antivirus software)為大部分電腦使用者使用之軟體，目的在保護電腦不受駭客攻擊，常見惡意程式如：蠕蟲(computer worm)、木馬程式(Trojan horse)、巨集病毒(macro virus)、廣告軟體(adware)等，名詞定義如下表 1.1。在傳統防毒軟體防護技術中，特過病毒碼(virus pattern)及掃描引擎(scan engine)檢查電腦中，是否含有惡意程式於系統中執行。而病毒碼資料庫是透過防毒軟體公司蒐集樣本後，從中擷取惡意程式中唯一特徵碼，後續進行檔案掃描時，即可透過大量之病毒碼資料庫進行比對，藉由比對過程中判斷是否為惡意程式。在現今隨著惡意程式推陳出新，日新月異，網路犯罪日益猖獗，即使資安相關產業已利用各種技術與方法蒐集一切，但仍具有空窗期，而駭客於期間進行網路駭客行為，其缺點就是無法立即有效於第一時間進行防護。

於 2015 年微軟(Microsoft)於 Kaggle 大賽中，提供超過 400GB 惡意程式樣本(解壓縮後)[1]。最後於賽後取得第一名隊伍採用方式為隨機生成樹(random forest tree)，透用 xgboost 函式庫實作。近年也有學者利用卷機神經網路(CNN, Convolution Neuron Network)或循環神經網路(RNN, Recurrent Neuron Networks)等方式進行惡意程式分類[2][3]。

由財團法人台灣網路資訊中心(TWNIC)於 2019 年 6 月電子報中，紀載自 2015 年至 2019 年掌握之惡意程式總數，如下圖 1.1，惡意程式總數於 2019 達到最高峰，計有 906.97 百萬隻惡意程式樣本。此數據為已被掌握個數，而在網際網路世界中，尚未發現之個數必定遠高於已公開數。當個人、政府部門或公司企業等組織肇生資安事件時，乃須透過嚴謹資安鑑識流程，因此電腦科學鑑識(computer forensics science)等相關技術、研究也隨之發展。

表 1.1 惡意程式種類名詞定義[4]

類	別	定	義
蠕蟲(Worm)		透過漏洞(exploit)、檔案分享、電子郵件或文件附檔(attachment)等方式，進行自我複製感染其他資訊設備。	
木馬(Trojan Horse)		偽裝成正常程式於電腦中啟動後門程式(backdoor)，駭客可藉此進行遠端非法存取。	
間諜軟體(Spyware)		在未經使用者允許下安裝於電腦中，目的蒐集使用者電腦情資，透過分析後在從網路上下載。	
廣告軟體(Adware)		會於系統中彈跳或佔用螢幕顯示廣告資訊。	
勒索軟體(Ransomware)		近期造成世界最嚴重事件為「WannaCry」勒索病毒，將電腦檔案加密後像使用者勒索，透過虛擬貨幣(如：Bitcoin、Litecoin 等)方式交付贖金。	

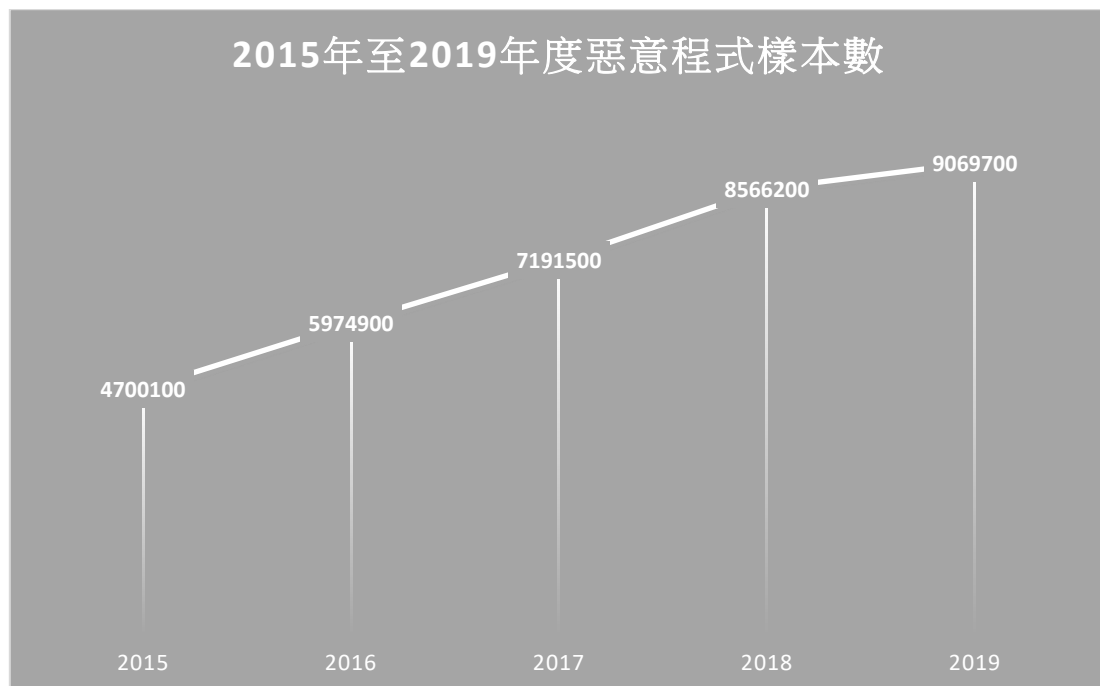


圖 1.1 TWNIC 惡意程式統計數據圖[5]

針對網路攻擊狙殺鍊(cyber kill chain)，如下圖 1.2，駭客會經過偵查(reconnaissance)、武裝(weaponization)、傳遞(delivery)、弱點攻擊(exploitation)、控制(control)、執行(execute)、維持(maintain)等 7 個步驟[6]，各步驟說明如下表 1.2。駭客於網路滲透時會以此 7 種方式交互運用，特別為武裝階段中，主要是將惡意程式或結合弱點(vulnerability)進行包裝，透過使用者瀏覽文件、網頁等正常行為，在背景執行下植入程式，因此駭客可透過後門程式所建立之隧道，達到非法遠端存取(remote access)行為。

表 1.2 網路攻擊狙殺鍊說明

步驟	方法	說明
1	偵查	駭客通常利用網路拓撲掃描，針對電腦、網路設備等進行服務開啟狀態、弱點等掃描，或藉由社群媒體等公開資訊，蒐集資訊進行進一步利用與滲透。
2	武裝	通常駭客藉由自製惡意程式，通常會將載具結合文件、資料檔或壓縮檔內，誘使使用者瀏覽釣魚網站、惡意文件等，達到植入後門進行遠端操控。
3	傳遞	通常利用儲存媒體設備(如：USB 隨身硬碟、電子郵件附件等)方式將惡意程式附載於其中。
4	弱點攻擊	如利用已知或零時差(zero day)弱點攻擊，達到遠端執行代碼(RCE, Remote Code Execution)、阻斷服務(Dos, Denied of Service)等方式影響使用者。
5	控制	於目標主機安裝後門程式後，駭客藉此遠端操控。
6	執行	針對使用者使用之內部網路進行內往滲透，進而資料竊取。
7	維持	駭客為了長久於該目標網路環境內運作，會清除電磁紀錄、滲透軌跡等，並保持原有電腦稽核紀錄，讓資安鑑識人員難以察覺異樣，達到行為匿蹤效果。

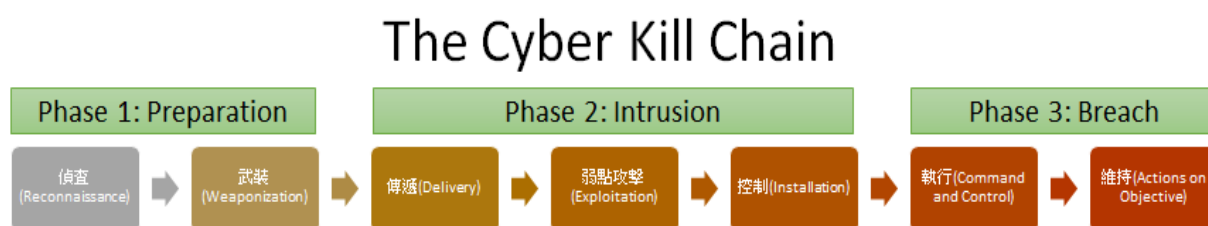


圖 1.2 網路攻擊狙殺鍊手法[7]

近年全球最大網路攻擊為 2017 年「WannaCry」勒索軟體事件，此為電腦蠕蟲病毒之一，係透過微軟作業系統之檔案分享通訊協定(SMB, Server Message Block)漏洞進行遠端注入攻擊，達到檔案加密以實現勒索效果。其中使用漏洞編號為 CVE-2017-0143、CVE-2017-0144、CVE-2017-0145、CVE-2017-0146、CVE-2017-0148[8]。而往年著名電腦蠕蟲如：2008 年 Conficker[9]、2006 年熊貓燒香[10]等。由於蠕蟲病毒影響範圍甚廣，針對分類需使用快速及準確率高之自動化方式，歸納出惡意程式族群，進而輔助威脅情資(threat intelligence)分析。

1.2 研究方法

本論文先期研究微軟惡意程式分類比賽資料集進行實驗，透過研究樣本資料之前置處理及惡意程式分類之演算法後，再利用蠕蟲資料集 Malimg 做方法設計與實驗，並比較相關利用此資料集之研究[11]。囿於惡意程式設計除需考慮特徵擷取外，仍需考量程式邏輯先後順序問題。駭客針對木馬免殺(anti anti-virus)技術中，藉由增加無意義垃圾程式碼(俗稱花指令、加花)、改變程式邏輯等手法達到越過(bypass)防毒軟體檢查機制，面對大量惡意程式以及樣本數大幅成長，要以快速及準確率較高之自動化方式歸納惡意程式族群與類別，增進威脅情資族群分析效率。本論文主要將卷機神經網路與循環神經網路之模型結合，進而解決惡意程式特徵擷取、程式邏輯之順序性問題。

1.3 論文架構

除第一章緒論以外，本論文的架構分別為：

- 第二章 相關研究：從資訊安全到深度學習，逐一介紹以駭客角度編撰惡意程式反防毒軟體查殺方式，到深度學習之應用。
- 第三章 系統設計：從樣本到類神經網路學習，探討目前針對樣本資料前置處理及惡意程式分類方法之演算法研究。
- 第四章 實驗結果分析與討論：以 Malimg 為惡意程式資料集，藉由演算法、參數使用及辨識之正確率與其他研究論文作比較。
- 第五章 結論與未來工作。

第二章 相關研究

2.1 惡意程式研究

在 Windows 作業系統下有多種文件，其中副檔名(file extension)為*.exe(可執行檔)、*.dll(動態連結函示庫)、*.sys(驅動程式)等都是遵守 PE(Portable Executable)文件檔案結構[12]。PE 格式主要是由 Unix 中 COFF 格式修改而成[13]，大致結構於區分為 DOS 文件標頭、PE 文件標頭及區段表[14]。

在 DOS 標頭中，會以起始 MZ 作為標記(該標記代表為 0x5A4D，該數值定義於 Win32 SDK 中使用#define IMAGE_DOS_SIGNATURE)，在作網路封包、惡意程式等分析時，通常會優先尋找這組字串，藉由找到此文字可將之視為程式檔。該 DOS 標頭主要考量相容性問題，而大部分可在 Windows NT 系統執行之程式是無法在 MS-DOS 作業系下執行，一般正常未被竄改時會顯示「This program cannot run in DOS mode」或「This program requires run under Windows」等字眼。該標頭之資料結構於 WinNT.h 定義，如下圖 2.1 依 DOS 標頭中最後一個 e_lfanew 參數可引導找到可執行檔之文件標頭，可藉此判斷是否為 PE 文件。

```
typedef struct _IMAGE_DOS_HEADER { // DOS .EXE header
    WORD e_magic; // Magic number
    WORD e_cblp; // Bytes on last page of file
    WORD e_cp; // Pages in file
    WORD e_crlc; // Relocations
    WORD e_cparhdr; // Size of header in paragraphs
    WORD e_minalloc; // Minimum extra paragraphs needed
    WORD e_maxalloc; // Maximum extra paragraphs needed
    WORD e_ss; // Initial (relative) SS value
    WORD e_sp; // Initial SP value
    WORD e_csum; // Checksum
    WORD e_ip; // Initial IP value
    WORD e_cs; // Initial (relative) CS value
    WORD e_lfarlc; // File address of relocation table
    WORD e_ovno; // Overlay number
    WORD e_res[4]; // Reserved words
    WORD e_oemid; // OEM identifier (for e_oeminfo)
    WORD e_oeminfo; // OEM information; e_oemid specific
    WORD e_res2[10]; // Reserved words
    LONG e_lfanew; // File address of new exe header
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

圖 2.1 DOS 標頭資料結構

在 PE 文件標頭中，會以起始 PE\x00\x00 作為標記(該標記於資料結構中 Signature 代表 0x50450000，該數值定義於 Win32 SDK 中使用#define IMAGE_NT_SIGNATURE，

其中第三、四字節為 ASCII 空值)，而 PE 文件標頭是 Windows NT 下判斷是否為有效之執行檔之唯一有效結構，資料結構如下圖 2.2。

```
typedef struct _IMAGE_NT_HEADERS {  
    DWORD Signature;  
    IMAGE_FILE_HEADER FileHeader;  
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;  
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

圖 2.2 PE 標頭資料結構

另外於 PE 標頭中 ImageBase 值為程式進入點，數值位於資料結構中 IMAGE_OPTIONAL_HEADER32，如下圖 2.3，其數值通常為 0x00400000 最為程式起始進入點。接續為區段(section)，較常出現如下表 2.1。

```
typedef struct _IMAGE_OPTIONAL_HEADER {  
    //  
    // Standard fields.  
    //  
    WORD Magic;  
    BYTE MajorLinkerVersion;  
    BYTE MinorLinkerVersion;  
    DWORD SizeOfCode;  
    DWORD SizeOfInitializedData;  
    DWORD SizeOfUninitializedData;  
    DWORD AddressOfEntryPoint;  
    DWORD BaseOfCode;  
    DWORD BaseOfData;  
  
    //  
    // NT additional fields.  
    //  
    DWORD ImageBase;  
    DWORD SectionAlignment;  
    DWORD FileAlignment;  
    WORD MajorOperatingSystemVersion;  
    WORD MinorOperatingSystemVersion;  
    WORD MajorImageVersion;  
    WORD MinorImageVersion;  
    WORD MajorSubsystemVersion;  
    WORD MinorSubsystemVersion;  
    DWORD Win32VersionValue;  
    DWORD SizeOfImage;  
    DWORD SizeOfHeaders;  
    DWORD CheckSum;  
    WORD Subsystem;  
    WORD DllCharacteristics;  
    DWORD SizeOfStackReserve;  
    DWORD SizeOfStackCommit;  
    DWORD SizeOfHeapReserve;  
    DWORD SizeOfHeapCommit;  
    DWORD LoaderFlags;  
    DWORD NumberOfRvaAndSizes;  
    IMAGE_DATA_DIRECTORY DataDirectory[ IMAGE_NUMBEROF_DIRECTORY_ENTRIES];  
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

圖 2.3 IMAGE_OPTIONAL_HEADER32 資料結構

表 2.1 檔案標頭常見區段與說明

名稱	說明
.text	此為可執行程式碼(Executable code)區段
.data	可讀寫之區段
.idata	導入(Import)數據資訊區段
.edata	導出(Export)數據資訊區段
.rdata	只可讀初始化之數據
.rsrc	資源(Resource)區段

樣本逆向工程，可透過 IDA Pro[15]、OllyDbg[16]等工具進行輔助分析，其中大多數惡意程式，除利用 PE 加殼(packer)技術達到(如：UPX[17]、ASPPack[18]等)，進行程式加密或壓縮方式達到數據偽裝，另外一種免殺技巧為利用組合語言、垃圾碼等方式，隱藏真正執行指令，藉此達到免殺效果。在一般撰寫完程式透過編譯器編譯成可執行檔後(實驗以利用 Microsoft Visual Studio C++編譯器，作業系統為 Windows)與透過加殼技術處理之比較，如下圖 2.4，發現檔案標頭已被處理。另外利用 IDA Pro 工具針對程式已利用 UPX 加殼器處理後之情況，發現無法直接反組譯原始程式設計原生產生之程式檔，如下圖 2.6。其中要原因是該加殼工具會將解密、解壓縮等程式段落寫於標頭區，當程式載入記憶體時，將會把程式碼進行脫殼(unpacker)，最後執行原始程式。

Address	Standard PE Hex	Standard PE ASCII	Packed PE Hex	Packed PE ASCII
00000000	4D 5A 90 00 03 00 00 00	MZ.....	4D 5A 90 00 03 00 00 00	MZ.....
00000010	B8 00 00 00 00 00 00 00@.....	B8 00 00 00 00 00 00 00@.....
00000020	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00000030	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00000040	0E 1F BA 0E 00 B4 09 CD!.L.Th	0E 1F BA 0E 00 B4 09 CD!.L.Th
00000050	69 73 20 70 72 6F 67 72	is program canno	69 73 20 70 72 6F 67 72	is program canno
00000060	74 20 62 65 20 72 75 6E	t be run in DOS	74 20 62 65 20 72 75 6E	t be run in DOS
00000070	6D 6F 64 65 2E 00 00 00	mode...\$.....	6D 6F 64 65 2E 00 00 00	mode...\$.....
00000080	37 D2 57 55 73 B3 39 06	7.WUs.9.s.9.s.9.	37 D2 57 55 73 B3 39 06	7.WUs.9.s.9.s.9.
00000090	60 D5 38 07 70 B3 39 06	.8.p.9...q.9.	60 D5 38 07 70 B3 39 06	.8.p.9...q.9.
000000A0	60 D5 3C 07 6B B3 39 06	<.k.9...=.9.	60 D5 3C 07 6B B3 39 06	<.k.9...=.9.
000000B0	1C D7 38 07 77 B3 39 06	.8.w.9.s.8.5.9.	1C D7 38 07 77 B3 39 06	.8.w.9.s.8.5.9.
000000C0	32 D4 3C 07 71 B3 39 06	2.<q.9.2...r.9.	32 D4 3C 07 71 B3 39 06	2.<q.9.2...r.9.
000000D0	32 D4 3B 07 72 B3 39 06	2...r.9.Richs.9.	32 D4 3B 07 72 B3 39 06	2...r.9.Richs.9.
000000E0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
000000F0	50 45 00 00 4C 01 03 00	PE.L...e.\.....	50 45 00 00 4C 01 08 00	PE.L...G.].A...
00000100	00 00 00 00 E0 00 02 01@.....	00 00 00 00 E0 00 02 01@.....
00000110	00 10 00 00 C0 01 00 00@.....	00 10 00 00 C0 01 00 00@.....
00000120	00 10 02 00 00 00 40 00@.....	00 10 02 00 00 00 40 00@.....
00000130	06 00 00 00 00 00 00 00@.....	06 00 00 00 00 00 00 00@.....
00000140	00 20 02 00 00 10 00 00@.....	00 20 02 00 00 10 00 00@.....
00000150	00 00 10 00 00 10 00 00@.....	00 00 10 00 00 10 00 00@.....
00000160	00 00 00 00 10 00 00 00@.....	00 00 00 00 10 00 00 00@.....
00000170	0C 11 02 00 10 01 00 00@.....	0C 11 02 00 10 01 00 00@.....
00000180	00 00 00 00 00 00 00 00@.....	00 00 00 00 00 00 00 00@.....
00000190	EC 12 02 00 10 00 00 00@.....	EC 12 02 00 10 00 00 00@.....
000001A0	00 00 00 00 00 00 00 00@.....	00 00 00 00 00 00 00 00@.....
000001B0	00 00 00 00 00 00 00 00@.....	00 00 00 00 00 00 00 00@.....
000001C0	00 00 00 00 00 00 00 00@.....	00 00 00 00 00 00 00 00@.....
000001D0	00 00 00 00 00 00 00 00@.....	00 00 00 00 00 00 00 00@.....
000001E0	00 00 00 00 00 00 00 00@.....	00 00 00 00 00 00 00 00@.....
000001F0	00 00 00 00 00 00 00 00@.....	00 00 00 00 00 00 00 00@.....
00000200	00 00 00 00 00 00 00 00@.....	00 00 00 00 00 00 00 00@.....
00000210	55 50 58 31 00 00 00 00	UPX1.....	55 50 58 31 00 00 00 00	UPX1.....
00000220	00 34 00 00 00 04 00 00	4.....rsrc.	00 34 00 00 00 04 00 00	4.....rsrc.
00000230	00 00 00 00 40 00 00 E0@.....	00 00 00 00 40 00 00 E0@.....
00000240	00 10 00 00 00 10 02 00@.....	00 10 00 00 00 10 02 00@.....
00000250	00 00 00 00 00 00 00 00@.....	00 00 00 00 00 00 00 00@.....
00000260	00 00 00 00 00 00 00 00@.....	00 00 00 00 00 00 00 00@.....
00000270	00 00 00 00 00 00 00 00@.....	00 00 00 00 00 00 00 00@.....
00000280	00 00 00 00 00 00 00 00@.....	00 00 00 00 00 00 00 00@.....
00000290	00 00 00 00 00 00 00 00@.....	00 00 00 00 00 00 00 00@.....
000002A0	00 00 00 00 00 00 00 00@.....	00 00 00 00 00 00 00 00@.....
000002B0	00 00 00 00 00 00 00 00@.....	00 00 00 00 00 00 00 00@.....
000002C0	00 00 00 00 00 00 00 00@.....	00 00 00 00 00 00 00 00@.....
000002D0	00 00 00 00 00 00 00 00@.....	00 00 00 00 00 00 00 00@.....
000002E0	00 00 00 00 00 00 00 00@.....	00 00 00 00 00 00 00 00@.....
000002F0	00 00 00 00 00 00 00 00@.....	00 00 00 00 00 00 00 00@.....
00000300	00 00 00 00 00 00 00 00@.....	00 00 00 00 00 00 00 00@.....
00000310	00 00 00 00 00 00 00 00@.....	00 00 00 00 00 00 00 00@.....

圖 2.4 利用 UPX 加殼處理及一般程式碼編譯後執行檔

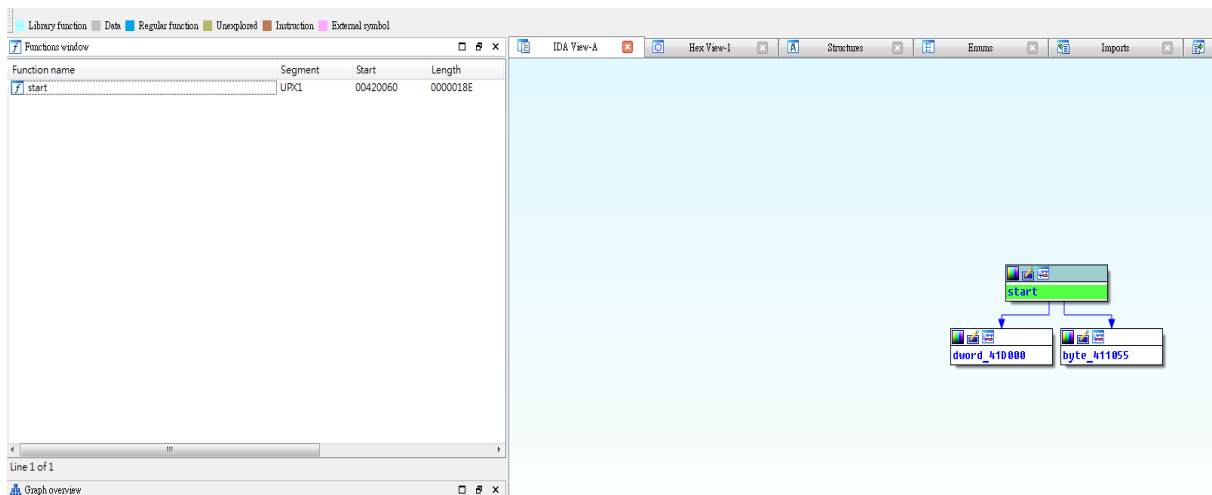


圖 2.5 利用 UPX 加殼器反組譯情況

利用修改程式執行邏輯(以 VC#為實驗)，如下圖 2.6，利用 `try{...}catch{...}` 機制，藉由產生例外處理(exception handling)將預執行惡意指令(malicious codes)寫入於 catch 中，特意將程式發生錯誤。當程式遇到於執行階段(run-time)出現錯誤時，將會透過拋(throw)將控制權移交，產生例外(exception)後進行處理。在大多數程式語言中(如：C#、Java、PHP、Python 等)，都有內建異常機制，可隨著函數堆疊狀態觀察，逐一逆向方式進行除錯(debug)[19]。

```
try
{
    int i = 100;
    int j = 0;

    if (i / j == 100)
    {
        return;
    } // end if
} // end try
catch (Exception ex)
{
    string my_registry_code = ex.Message;
    my_registry_code = "HKEY_CURRENT_USER\\Software\\Microsoft\\Windows\\CurrentVersion\\Explorer\\Shell Folders";
    Microsoft.Win32.Registry.SetValue(registry_code, "Startup", Environment.SpecialFolder.ApplicationData + "notepad");
} // end catch
```

圖 2.6 利用除以 0 產生例外處理進而修改程式執行邏輯

其次是利用無意義程式碼進行混淆，該段程式碼並不會造成整體程式執行邏輯影響，但會使分析人員進行靜態分析時，增加查殺難度，提高惡意程式空窗期時間，

如下圖 2.7，其中該程式片段使用組合語言撰寫，該指令僅作加、減動作，互相抵消方式毫無影響暫存器數值，最後可利用跳轉指令(jump)跳入預執行程式[20]。

```
_asm
{
    push ebp;
    mov ebp, esp;
    push edx;
    pop edx;
    add esp, 0x10;
    add esp, -0x10;
    add esp, 0x20;
    add esp, -0x20;
    xor eax, eax;
    cmp eax, 0;
    jz equal;
    jnz no_equal;
}
```

圖 2.7 無意義加花指令

另外可用判斷式方式改變病毒碼特徵，如利用 if{...}else if{...}else{...} 或 switch{case:... case:... default:...} 等方式，如下圖 2.8，該判斷式為判斷一年是否有 100 個月，如果條件成立將會呼叫 MessageBoxA 函數，由於該函數為 Win32 API 有效函數(會載入 User32.dll 動態函示庫)，編譯器在執行優化過程中並不會因垃圾程式碼予以刪除，且程式執行過程中實際上不可能被執行，造成一定混淆程度。

```
SYSTEMTIME g_stcTime;
if (g_stcTime.wMonth == 100)
{
    MessageBoxA(NULL, "Alert", "Alert", MB_YESNOCANCEL);
} // end if
```

圖 2.8 利用判斷式進行程式流程混淆

以程式語言 C 為例，宣告一個字串變數，以雙引號字串資料為初始化，如下圖 2.9，原程式碼與編譯後組合語言做為比較。另外一種為字元堆疊方式，如下圖 2.10，以陣列初始化中將各個字元壓入線(stack)中，最後須以\x00 作為陣列結尾，是以程式方式建立字串。其中前者為編譯期間產生，後者為執行期間產生。

```

5:      char word_1[] = "NTUST";
00414C28 A1 88 6D 41 00      mov     eax,dword ptr [string "NTUST" (0416D88h)]
00414C2D 89 45 F0          mov     dword ptr [word_1],eax
00414C30 66 8B 0D 8C 6D 41 00 mov     cx,word ptr ds:[416D8Ch]
00414C37 66 89 4D F4          mov     word ptr [ebp-0Ch],cx

```

圖 2.9 以雙引號字串資料作為初始化

```

7:      char word_2[] = { 'N','T','U','S','T',0 };
00414C3B C6 45 E0 4E          mov     byte ptr [word_2],4Eh
00414C3F C6 45 E1 54          mov     byte ptr [ebp-1Fh],54h
00414C43 C6 45 E2 55          mov     byte ptr [ebp-1Eh],55h
00414C47 C6 45 E3 53          mov     byte ptr [ebp-1Dh],53h
00414C4B C6 45 E4 54          mov     byte ptr [ebp-1Ch],54h
00414C4F C6 45 E5 00          mov     byte ptr [ebp-1Bh],0

```

圖 2.10 以字元方式壓入線中建立字串

在撰寫程式中，若字串未經過處理時，可直接利用 WinHex[21]等工具解析出，為了隱藏機敏資訊(如：私鑰、密碼等)，需撰寫特定加解密或雜湊(hash)方式對字串做處理。如字串加解密運用 Boost C++函式庫[22]，主要是將字串中每個字元做 XOR 處理，如下圖 2.11，該程式碼將字串加密並透過 BOOST_PP_SEQ_ENUM 處理，需透過 DEFINE_HIDDEN_STRING 巨集，最後於程式中呼叫 GetEncryptionKeyN 解密。如下圖 2.12，會在程式執行階段中將密文解密。其中密文為 C ，明文為 E ，密鑰為 K ，字元位置數為 n ，字串長度為 L ，條件必須為 $K \geq L$ ，其加解密方法如下表示：

$$\text{加密：} C_n = E_n \oplus (K - n), \quad (1.1)$$

$$\text{解密：} E_n = C_n \oplus (K - n)。 \quad (1.2)$$

```

5  #include <boost/preprocessor/cat.hpp>
6  #include <boost/preprocessor/seq/for_each_i.hpp>
7  #include <boost/preprocessor/seq/enum.hpp>
8
9  #define CRYPT_MACRO(r, d, i, elem) ( elem ^ ( d - i ) )
10
11 #define DEFINE_HIDDEN_STRING(NAME, SEED, SEQ)\
12 static char* BOOST_PP_CAT(Get, NAME)()\
13 {\
14     static char data[] = {\
15         BOOST_PP_SEQ_ENUM(BOOST_PP_SEQ_FOR_EACH_I(CRYPT_MACRO, SEED, SEQ)),\
16         '\0'\
17     };\
18
19     static bool isEncrypted = true;\
20     if ( isEncrypted )\
21     {\
22         for (unsigned i = 0; i < ( sizeof(data) / sizeof(data[0]) ) - 1; ++i)\
23         {\
24             data[i] = CRYPT_MACRO(_, SEED, i, data[i]);\
25         }\
26
27         isEncrypted = false;\
28     }\
29
30     return data;\
31 }
32
33 DEFINE_HIDDEN_STRING(EncryptionKey0, 0x67, ('1')('9')('2')('.')('1')('6')('8')('.')('1')('1')('.')('2')('5')('1'))
34 DEFINE_HIDDEN_STRING(EncryptionKey1, 0x55, ('1')('9')('2')('.')('1')('6')('8')('.')('1')('1')('.')('2')('5')('1')('.')('8')('0'))
35 DEFINE_HIDDEN_STRING(EncryptionKey2, 0x10, ('8')('0'))
36 DEFINE_HIDDEN_STRING(EncryptionKey3, 0x54, ('1')('6')('3')('8')('4'))
37 DEFINE_HIDDEN_STRING(EncryptionKey4, 0x26, ('M')('o')('z')('i')('l')('l')('a')('r')('5')('1')('0')(' ')('')('W')('i')('n')('d')('o')('w')('s')('')

```

圖 2.11 利用 Boost C++ 於原始碼中加解密處理

.data:01225079	a7J	db 'otepad.exe',0	
.data:01225084	byte 1225084	db 0	; DATA XREF: sub_1221454+E9↑r
.data:01225084			; sub_1221454+103↑w
.data:01225085		align 4	
.data:01225088	byte_1225088	db 45h	; DATA XREF: sub_1221454+121↑w
.data:01225088			; sub_1221454:loc_1221586↑o
.data:01225088	aCrp	db '-crp',0	
.data:0122508E	byte_122508E	db 1	; DATA XREF: sub_1221454+113↑r
.data:0122508E			; sub_1221454+12C↑w

圖 2.12 於執行階段中密文與明文對比

另可利用雜湊方式，如 ROR13 處理[23]，此方法可做比對使用，常被用於尋找要呼叫之作業系統函數，對函數名稱做雜湊處理進行比對，令 hash 為雜湊值，其範圍為 $0x0000 \leq \text{hash} \leq 0xFFFF$ ，a 與 b 可自定義任意數值，c 為字串中字元，處理方法如下演算法表示：

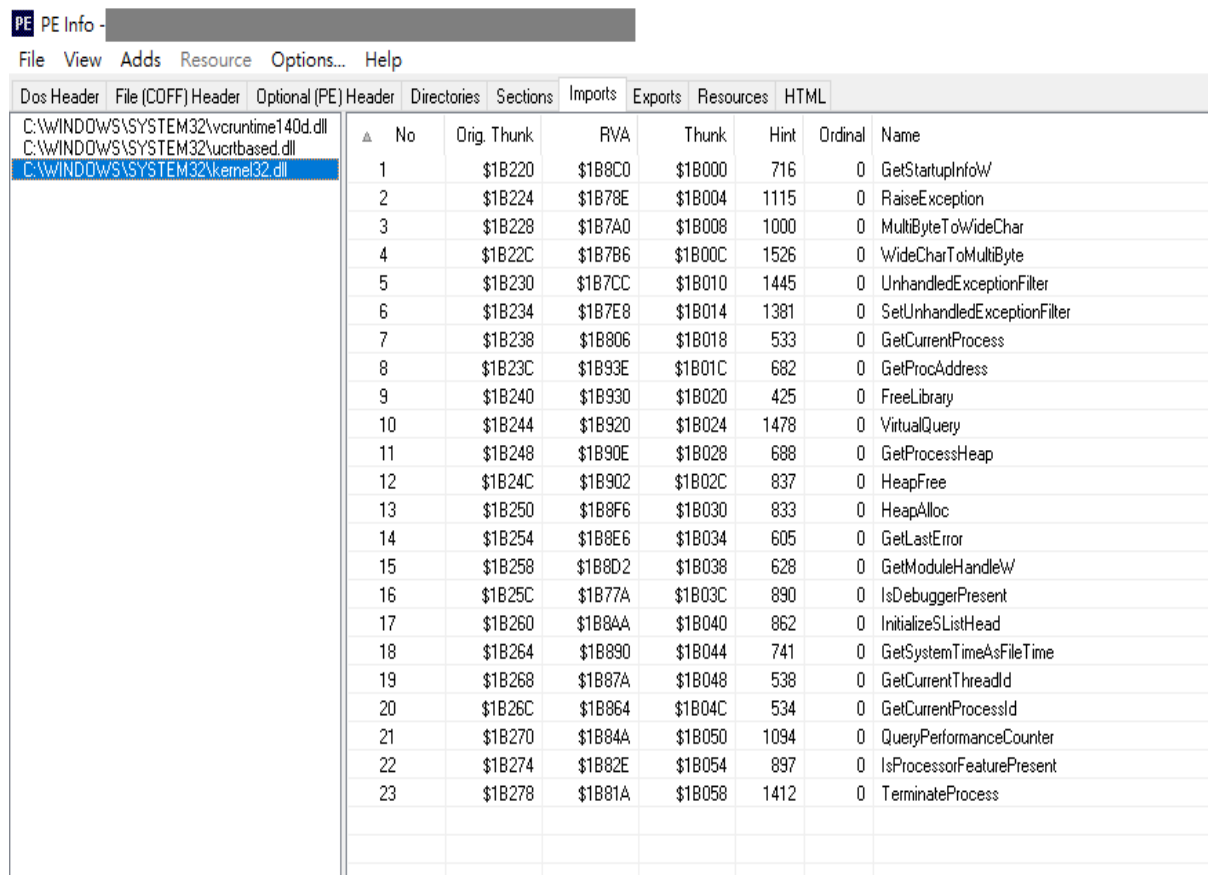
```

hash = 0
for c in string:
    hash = (hash >> a) | (hash << b)
    hash = hash + c

```

在程式被執行時，會將儲存媒體(如：硬碟、光碟機等)之程式載入到記憶體，而在 Windows 作業系統中，最主要載入動態函示庫為 kernel32.dll，在依程式設計者所需呼叫之函數時，於編譯期間在作連結(linker)時載入。當程式被執行時，會透過執行

檔中導入表找到對應之 API 函數之特定動態函式庫，如下圖 2.13，利用 PE Info 工具查看導入狀況。



Δ	No	Orig. Thunk	RVA	Thunk	Hint	Ordinal	Name
	1	\$1B220	\$1B8C0	\$1B000	716	0	GetStartupInfoW
	2	\$1B224	\$1B78E	\$1B004	1115	0	RaiseException
	3	\$1B228	\$1B7A0	\$1B008	1000	0	MultiByteToWideChar
	4	\$1B22C	\$1B7B6	\$1B00C	1526	0	WideCharToMultiByte
	5	\$1B230	\$1B7CC	\$1B010	1445	0	UnhandledExceptionFilter
	6	\$1B234	\$1B7E8	\$1B014	1381	0	SetUnhandledExceptionFilter
	7	\$1B238	\$1B806	\$1B018	533	0	GetCurrentProcess
	8	\$1B23C	\$1B93E	\$1B01C	682	0	GetProcAddress
	9	\$1B240	\$1B930	\$1B020	425	0	FreeLibrary
	10	\$1B244	\$1B920	\$1B024	1478	0	VirtualQuery
	11	\$1B248	\$1B90E	\$1B028	688	0	GetProcessHeap
	12	\$1B24C	\$1B902	\$1B02C	837	0	HeapFree
	13	\$1B250	\$1B8F6	\$1B030	833	0	HeapAlloc
	14	\$1B254	\$1B8E6	\$1B034	605	0	GetLastError
	15	\$1B258	\$1B8D2	\$1B038	628	0	GetModuleHandleW
	16	\$1B25C	\$1B77A	\$1B03C	890	0	IsDebuggerPresent
	17	\$1B260	\$1B8AA	\$1B040	862	0	InitializeListHead
	18	\$1B264	\$1B890	\$1B044	741	0	GetSystemTimeAsFileTime
	19	\$1B268	\$1B87A	\$1B048	538	0	GetCurrentThreadId
	20	\$1B26C	\$1B864	\$1B04C	534	0	GetCurrentProcessId
	21	\$1B270	\$1B84A	\$1B050	1094	0	QueryPerformanceCounter
	22	\$1B274	\$1B82E	\$1B054	897	0	IsProcessorFeaturePresent
	23	\$1B278	\$1B81A	\$1B058	1412	0	TerminateProcess

圖 2.13 利用 PE Info 查看程式所使用 DLL 及 API 函數

但部分惡意程式為規避靜態分析使用 API 函數，會採用動態方式載入，方法為先抓取 kernel32.dll 位址，其實作原始碼如下圖 2.14，從中尋找 LoadLibraryA 函數，尋找函數之原始碼如下圖 2.15，最後呼叫需使用函數及載入動態函式庫，如下圖 2.16。透過動態呼叫及載入方式，最後再透過 PE Info 查看，如下圖 2.17，發現無法直接藉由靜態方式取得該程式所使用之 DLL 與 API 函數。

```

/* This is very important function !! */
HMODULE __stdcall GetKernel32BaseAddress()
{
    PPEB pPeb = 0;
#ifdef _M_IX86
    __asm mov    eax, pPeb;
    __asm mov    eax, dword ptr fs : [0x30];
    __asm mov    pPeb, eax;
#endif
    PLDR_DATA_TABLE_ENTRY pLdrDataTableEntry = (PLDR_DATA_TABLE_ENTRY)pPeb->InMemoryOrderModuleList.Flink;
    PLIST_ENTRY pListEntry = pPeb->Ldr->InMemoryOrderModuleList.Flink;

    while (pListEntry != NULL)
    {
        DWORD hash = UnicodeROR13Hash(pLdrDataTableEntry->FullDllName.Buffer);

        if (hash == C_KERNEL32_DLL)
            break;
        else if (hash == KERNEL32_DLL)
            break;

        pListEntry = pListEntry->Flink;
        pLdrDataTableEntry = (PLDR_DATA_TABLE_ENTRY)(pListEntry->Flink);
    } // end while

    return (HMODULE)pLdrDataTableEntry->Reserved2[0];
} // end func

```

圖 2.14 實作抓取 Kernel32.dll 位址

```

SIZE_T __stdcall GetProcAddress(HMODULE hModuleBase, DWORD dwHash)
{
    SIZE_T pFunctionAddress = NULL;
    SIZE_T size = 0;
    PIMAGE_DOS_HEADER dos = (PIMAGE_DOS_HEADER)hModuleBase;
    PIMAGE_NT_HEADERS nt = (PIMAGE_NT_HEADERS)((SIZE_T)hModuleBase + dos->e_lfanew);
    PIMAGE_DATA_DIRECTORY expdir = (PIMAGE_DATA_DIRECTORY)(nt->OptionalHeader.DataDirectory + IMAGE_DIRECTORY_ENTRY_EXPORT);
    SIZE_T addr = expdir->VirtualAddress;
    PIMAGE_EXPORT_DIRECTORY exports = (PIMAGE_EXPORT_DIRECTORY)((SIZE_T)hModuleBase + addr);
    PULONG functions = (PULONG)((SIZE_T)hModuleBase + exports->AddressOfFunctions);
    PSHORT ordinals = (PSHORT)((SIZE_T)hModuleBase + exports->AddressOfNameOrdinals);
    PULONG names = (PULONG)((SIZE_T)hModuleBase + exports->AddressOfNames);
    SIZE_T max_name = exports->NumberOfNames;
    SIZE_T max_func = exports->NumberOfFunctions;

    for (SIZE_T i = 0; i < max_name; i++)
    {
        SIZE_T ord = ordinals[i];

        if (i >= max_name || ord >= max_func)
        {
            return NULL;
        } //end if
        if (functions[ord] < addr || functions[ord] >= addr + size)
        {
            if (dwHash == ROR13Hash((PCHAR)hModuleBase + names[i]))
            {
                pFunctionAddress = (SIZE_T)((PCHAR)hModuleBase + functions[ord]);
                break;
            } // end if
        } //end if
    } // end for

    return pFunctionAddress;
} // end func

```

圖 2.15 實作 GetProcAddress 目的是後續可動態取得 API 函數

```

void __stdcall InitializeWindowsFunctionAddress()
{
    // kernel32.dll
    HMODULE hKernel32 = GetKernel32BaseAddress();

    LoadLibraryA = (HMODULE(__stdcall*)(LPCSTR))GetProcAddress(hKernel32, FUNC_LOAD_LIBRARY_A);
    GetModuleFileNameA = (DWORD(__stdcall*)(HMODULE, LPSTR, DWORD))GetProcAddress(hKernel32, FUNC_GET_MODULE_FILE_NAME_A);
    GetComputerNameExA = (BOOL(__stdcall*)(COMPUTER_NAME_FORMAT, LPSTR, LPDWORD))GetProcAddress(hKernel32, FUNC_GET_COMPUTER_NAME_EX_A);
    Sleep = (VOID(__stdcall*)(DWORD))GetProcAddress(hKernel32, FUNC_SLEEP);
    DeleteFileA = (BOOL(__stdcall*)(LPCSTR))GetProcAddress(hKernel32, FUNC_DELETE_FILE_A);
    CopyFileA = (BOOL(__stdcall*)(LPCSTR, LPCSTR, BOOL))GetProcAddress(hKernel32, FUNC_COPY_FILE_A);
    ExitProcess = (VOID(__stdcall*)(UINT))GetProcAddress(hKernel32, FUNC_EXIT_PROCESS);
    MultiByteToWideChar = (int(__stdcall*)(UINT, DWORD, LPCSTR, int, LPWSTR, int))GetProcAddress(hKernel32, FUNC_MULTI_BYTE_TO_WIDE_CHAR);
    WideCharToMultiByte = (int(__stdcall*)(UINT, DWORD, LPCWSTR, int, LPSTR, int, LPCSTR, LPBOOL))GetProcAddress(hKernel32, FUNC_WIDE_CHAR_TO_MULTI_BYTE);
    WaitForSingleObject = (DWORD(__stdcall*)(HANDLE, DWORD))GetProcAddress(hKernel32, FUNC_WAIT_FOR_SINGLE_OBJECT);
    GetCurrentDirectoryA = (DWORD(__stdcall*)(DWORD, LPSTR))GetProcAddress(hKernel32, FUNC_GET_CURRENT_DIRECTORY_A);
    GetLocalTime = (DWORD(__stdcall*)(LPSYSTEMTIME))GetProcAddress(hKernel32, FUNC_GET_LOCAL_TIME);
    IsDebuggerPresent = (BOOL(__stdcall*)(VOID))GetProcAddress(hKernel32, FUNC_IS_DEBUGGER_PRESENT);
    GetCurrentProcess = (HANDLE(__stdcall*)(VOID))GetProcAddress(hKernel32, FUNC_GET_CURRENT_PROCESS);
    CreateToolhelp32Snapshot = (HANDLE(__stdcall*)(DWORD, DWORD))GetProcAddress(hKernel32, FUNC_CREATE_TOOLHELP32_SNAPSHOT);
    Process32First = (BOOL(__stdcall*)(HANDLE, LPPROCESSENTRY32))GetProcAddress(hKernel32, FUNC_PROCESS32_FIRST);
    Process32Next = (BOOL(__stdcall*)(HANDLE, LPPROCESSENTRY32))GetProcAddress(hKernel32, FUNC_PROCESS32_NEXT);
    VirtualProtect = (BOOL(__stdcall*)(LPVOID, SIZE_T, DWORD, PDWORD))GetProcAddress(hKernel32, FUNC_VIRTUAL_PROTECT);
    IsWow64Process = (BOOL(__stdcall*)(HANDLE, PBOOL))GetProcAddress(hKernel32, FUNC_IS_WOW64_PROCESS);
    CreateFileA = (HANDLE(__stdcall*)(LPCSTR, DWORD, DWORD, LPSECURITY_ATTRIBUTES, DWORD, DWORD, HANDLE))GetProcAddress(hKernel32, FUNC_CREATE_FILE_A);
    GetFileSizeEx = (BOOL(__stdcall*)(HANDLE, PLARGE_INTEGER))GetProcAddress(hKernel32, FUNC_GET_FILE_SIZE_EX);
    ReadFile = (BOOL(__stdcall*)(HANDLE, LPVOID, DWORD, LPDWORD, LPOVERLAPPED))GetProcAddress(hKernel32, FUNC_READ_FILE);
    SetFileAttributesA = (BOOL(__stdcall*)(LPCSTR, DWORD))GetProcAddress(hKernel32, FUNC_SET_FILE_ATTRIBUTES_A);
    CloseHandle = (BOOL(__stdcall*)(HANDLE))GetProcAddress(hKernel32, FUNC_CLOSE_HANDLE);
}

```

圖 2.16 利用動態載入、呼叫函數方式

PE Info -									
File View Adds Resource Options... Help									
Dos Header	File (COFF) Header	Optional (PE) Header	Directories	Sections	Imports	Exports	Resources	HTML	
C:\WINDOWS\system32\kernel32.dll									
			▲ No	Orig. Thunk	RVA	Thunk	Hint	Ordinal	Name
			1	\$4078	\$4080	\$4000	772	0	IsProcessorFeaturePresent

圖 2.17 利用 PE Info 查看利用動態呼叫、載入方式之惡意程式

在現今 Windows 作業系統中，常見呼叫慣例(calling conventions)為 cdecl[24]、stdcall[25]等。其中 cdecl 為 C 語言呼叫約定如下圖 2.18，與 stdcall 為 Windows 作業系統實作 API 常用呼叫慣例，其方式如下圖 2.19，cdecl 在呼叫函數回傳(return)後調整堆疊，而 stdcall 在呼叫函數回傳前調整堆疊。其中可使用裸函數(naked)實作函數(僅在 32 位元下使用)，且必須利用組合語言實作線堆疊，如下圖 2.20，在 VC++使用 __declspec(naked)做前墜[26]。

```

1 #include <stdio.h>
2
3 int __cdecl func_1(int a, int b, int c, int d, int e)
4 {
5     return a + b + c + d + e;
6 } // end func
7
8 int __stdcall func_2(int a, int b, int c, int d, int e)
9 {
10    return a + b + c + d + e;
11 } // end func
12
13 int main(void)
14 {
15     int val_1 = func_1(1, 2, 3, 4, 5);
16     int val_2 = func_2(1, 2, 3, 4, 5);
17 } // end main
    
```

```

int __cdecl func_1(int a, int b, int c, int d, int e)
{
    push    ebp
    mov     ebp, esp
    sub     esp, 0C0h
    push    ebx
    push    esi
    push    edi
    lea     edi, [ebp-0C0h]
    mov     ecx, 30h
    mov     eax, 0CCCCCCCCh
    rep stos dword ptr es:[edi]

    return a + b + c + d + e;
    mov     eax, dword ptr [a]
    add     eax, dword ptr [b]
    add     eax, dword ptr [c]
    add     eax, dword ptr [d]
    add     eax, dword ptr [e]
} // end func
pop     edi
pop     esi
pop     ebx
mov     esp, ebp
pop     ebp
ret
    
```

Equal to
"pop; pop; pop; pop; pop;"

圖 2.18 利用 cdecl 與編譯後結果

```

1 #include <stdio.h>
2
3 int __cdecl func_1(int a, int b, int c, int d, int e)
4 {
5     return a + b + c + d + e;
6 } // end func
7
8 int __stdcall func_2(int a, int b, int c, int d, int e)
9 {
10    return a + b + c + d + e;
11 } // end func
12
13 int main(void)
14 {
15     int val_1 = func_1(1, 2, 3, 4, 5);
16     int val_2 = func_2(1, 2, 3, 4, 5);
17 } // end main
    
```

```

int __stdcall func_2(int a, int b, int c, int d, int e)
{
    push    ebp
    mov     ebp, esp
    sub     esp, 0C0h
    push    ebx
    push    esi
    push    edi
    lea     edi, [ebp-0C0h]
    mov     ecx, 30h
    mov     eax, 0CCCCCCCCh
    rep stos dword ptr es:[edi]

    return a + b + c + d + e;
    mov     eax, dword ptr [a]
    add     eax, dword ptr [b]
    add     eax, dword ptr [c]
    add     eax, dword ptr [d]
    add     eax, dword ptr [e]
} // end func
pop     edi
pop     esi
pop     ebx
mov     esp, ebp
pop     ebp
ret     14h
    
```

POP the stack for 5 times
(int 4 bytes * 5 params)

圖 2.19 利用 stdcall 與編譯後結果


```

282 void __declspec (naked) FakeMain(void)
283 {
284     __asm
285     {
286         // flower
287         nop; nop; nop; nop; nop;
288         nop; nop; nop; nop; nop;
289         nop; nop; nop; nop; nop;
290         nop; nop; nop; nop; nop;
291         nop; nop; nop; nop; nop;
292         sub esp, 0x50;
293         nop; nop; nop; nop; nop;
294         nop; nop; nop; nop; nop;
295         nop; nop; nop; nop; nop;
296         nop; nop; nop; nop; nop;
297         nop; nop; nop; nop; nop;
298         mov ebp, esp;
299         nop; nop; nop; nop; nop;
300         nop; nop; nop; nop; nop;
301         nop; nop; nop; nop; nop;
302         nop; nop; nop; nop; nop;
303         nop; nop; nop; nop; nop;
304         add esp, 0x0c;
305         nop;
306         push eax;
307         pop eax;
308         nop;
309         push eax;
310
311         nop;
312         push ebp;
313         mov ebp, esp;
314         push ebx;

```

圖 2.20 裸函數實作，其中裡面藏有加花指令

2.2 類神經網路研究

在機器學習中，主要分類有監督式學習(supervised learning)、非監督式學習(unsupervised learning)及強化式學習(reinforcement learning)，如下圖 2.21。在監督式學習中，資料須具備特徵(feature)與標籤(label)，透過設計之演算法建立模型，主要在作分類(classification)與回歸(regression)；另外為非監督式學習，在資料中並沒有設定好的標籤，透過演算法為將資料分別為相異性較大之群組，而各群組內資料之相似度為最高，主要在作分群(clustering)、歸類(association)與降維(dimension reduction)。

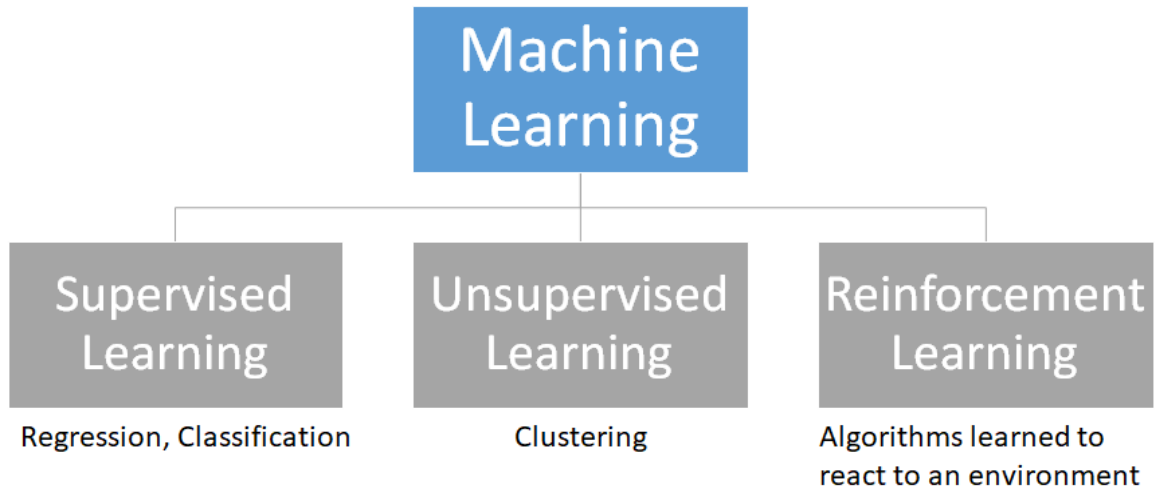


圖 2.21 監督式學習與非監督式學習比較圖[27]

本論文探討類神經網路(NN, Neural Network)，主要是模擬生物大腦神經網路，透過數學方式，針對各非線性函數(nonlinear)進行尋找逼近值，其所用之類神經模型為監督式學習。

2.2.1 多層感知器與 Dropout

神經網路是由多個感知器(perceptron)組成，其中將組合成輸入層(input layer)、隱藏層(hidden layer)與輸出層(output layer)，如下圖 2.22。而將輸入訊號轉換成輸出訊號函數稱為活化函數(activation function)[28]，本論文主要是利用 ReLU(rectified linear unit)及 Softmax 函數，其中 y 為輸出函數， x 為輸入函數，全部代表有 n 個，而 k 代表為第 k 個。

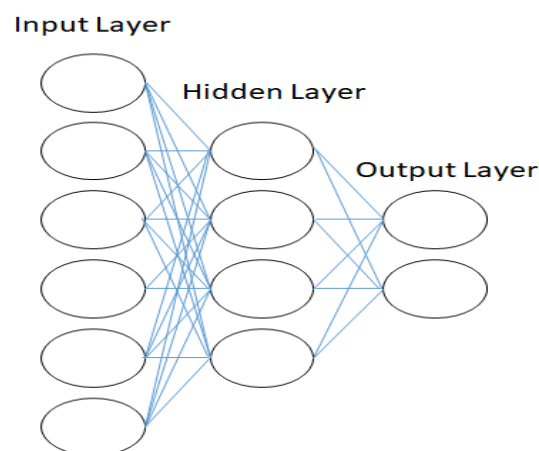


圖 2.22 多層感知器(MLP, Multilayer Perceptron)運作圖

$$\text{ReLU: } y = \begin{cases} 0 & (x < 0) \\ x & (x \geq 0) \end{cases} \circ \quad (2)$$

$$\text{Softmax: } y = \frac{e^{a_k}}{\sum_{i=1}^n e^{a_i}} \circ \quad (3)$$

在訓練神經網路模型中，可使用 Dropout 方式解決過度學習問題(overfitting)，此方法將會隨機挑選神經元(neuron)後刪除[29]，而被刪除神經元之梯度(gradient)為 0，主要是在訓練每一次迭代(epoch)中，如下圖 2.23，以一定機率丟刪除神經元，而被刪除部分不會傳遞訊息。

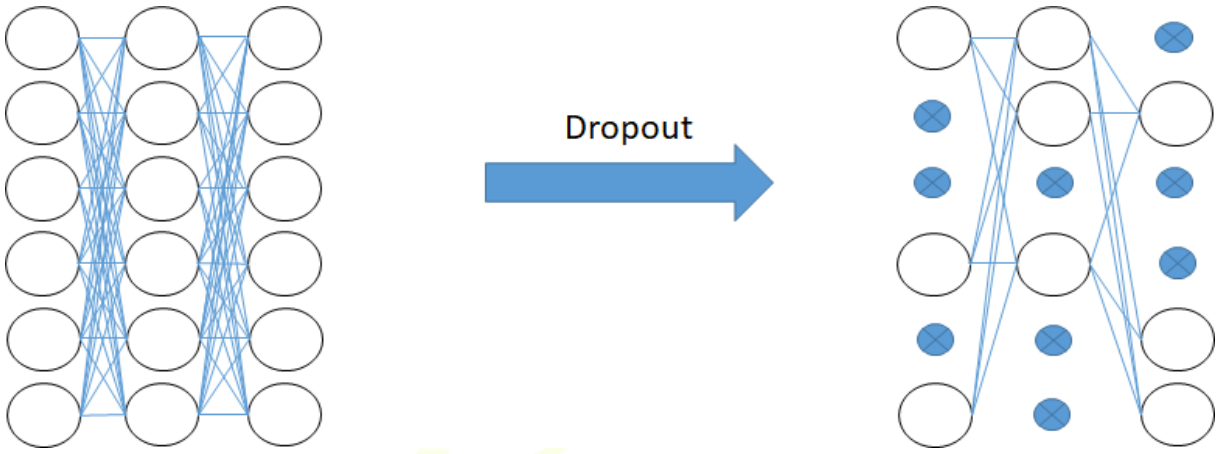


圖 2.23 每一層利用 0.5% 機率丟棄之神經元[30]

在沒有使用 Dropout 方式之類神經網路，如下圖 2.24，令 m 為類神經網路中第幾層， n 為神經元連接數， i 為在第 m 層中的某個神經元， y 為表示神經輸出(如：第 m 層中第 i 個神經元輸出代表 y_i^m)， w 為權重係數(weight)， b 為偏差係數(bias)， z 為各神經元輸出加上偏差值， f 為活化函數，表示法如下：

$$z_i^{m+1} = w_i^{m+1} y_i^n + b_i^{m+1}, \quad (4.1)$$

$$y_i^{m+1} = f(z_i^{m+1}) \circ \quad (4.2)$$

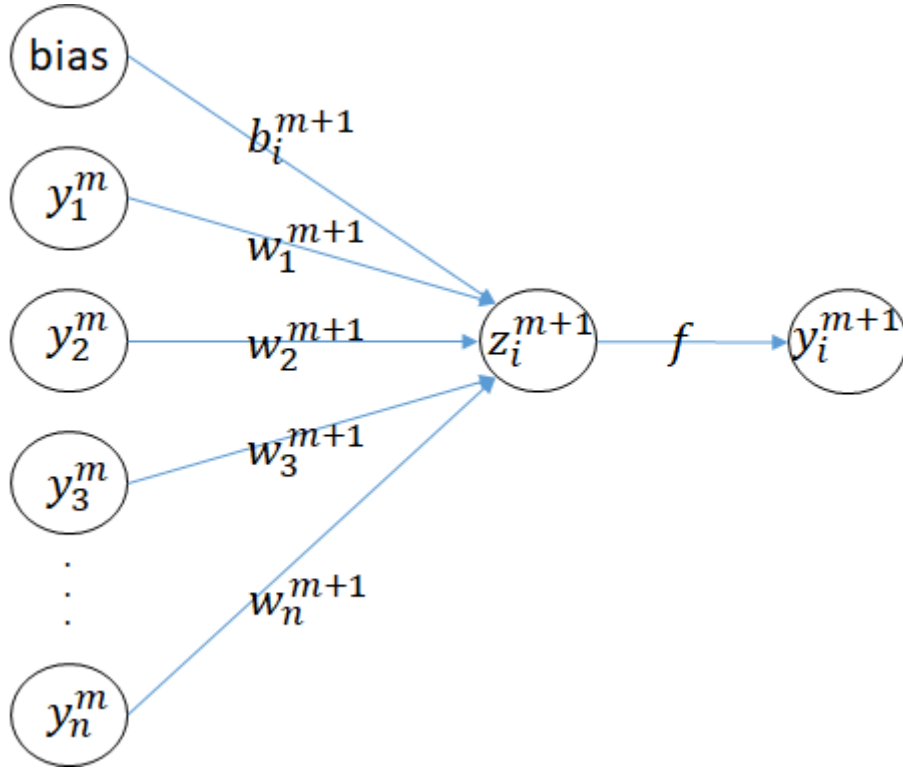


圖 2.24 沒有使用 Dropout 之神經網路圖

在使用 Dropout 方式之類神經網路，如下圖 2.25，令 r 為使用 Bernoulli 函數[31]產生 0 與 1 之機率向量， p 為機率，表示法如下：

$$r_i^m \sim \text{Bernoulli}(p), \quad (5.1)$$

$$\tilde{y}^m = r^m * y^m, \quad (5.2)$$

$$\tilde{z}_i^{m+1} = w_i^{m+1} \tilde{y}_i^n + b_i^{m+1}, \quad (5.3)$$

$$y_i^{m+1} = f(\tilde{z}_i^{m+1}). \quad (5.4)$$

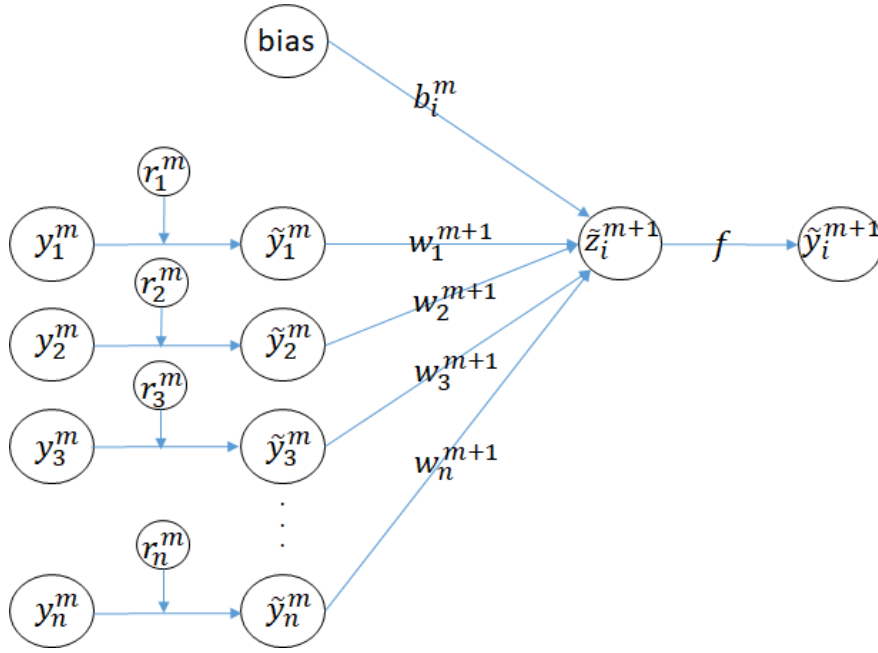


圖 2.25 使用 Dropout 之神經網路圖

2.2.2 卷積神經網路

卷積神經網路主要是由 Yann LeCun 所提出，而卷積層主要是透過輸入資料進行處理，輸入及輸出須相同的資料維度(dimension)。此演算透過特徵圖(feature map)，將影像資料特徵學習起來[32]。如下圖 2.26，透過數值相乘得到圖片特徵(黑色為-1，白色為 1)，經過重複運算後，製作新的矩陣得到圖片的可能特徵。

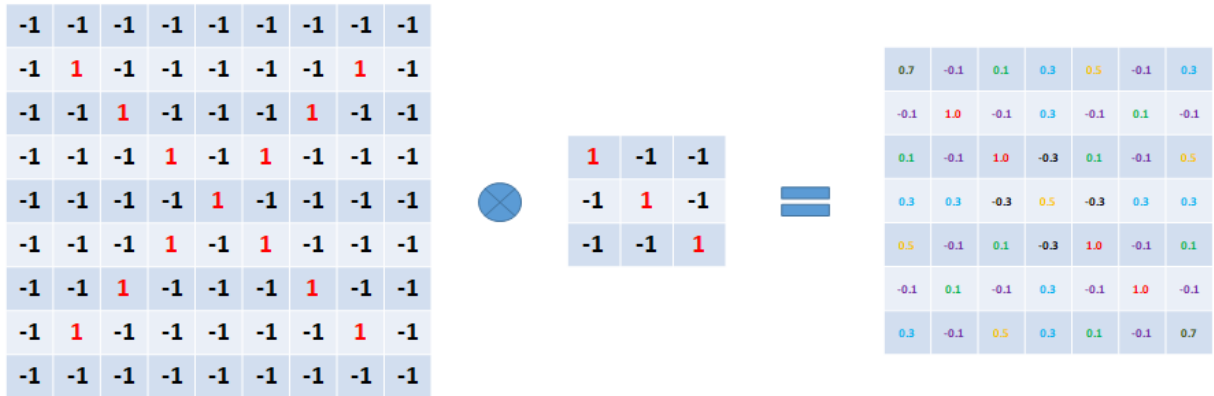


圖 2.26 透過矩陣相乘運算得到可能特徵[33]

2.2.3 循環神經網路

語言、文章為具有順序性特性，須考慮前後文才可分析出詞句、語句意義，如建構具有時間性及順序性問題時，可利用循環神經網路，進而提高準確率。從簡易的

RNN 模型中，如下圖 2.27。在 RNN 公式表示，令 X_t 為在時間 t 之輸入函數， O_t 為在時間 t 之輸出函數， (U, V, W) 中， W 代表在時間 $t-1$ 時輸入狀態而也代表在時間 t 時輸出狀態， S 為 X_t 加上 S_{t-1} (其中 S_{t-1} 代表最後狀態) 加上 U 及 W ，表示法如下公式：

$$S_t = func([U]X_t + [W]S_{t-1}) \quad (6)$$

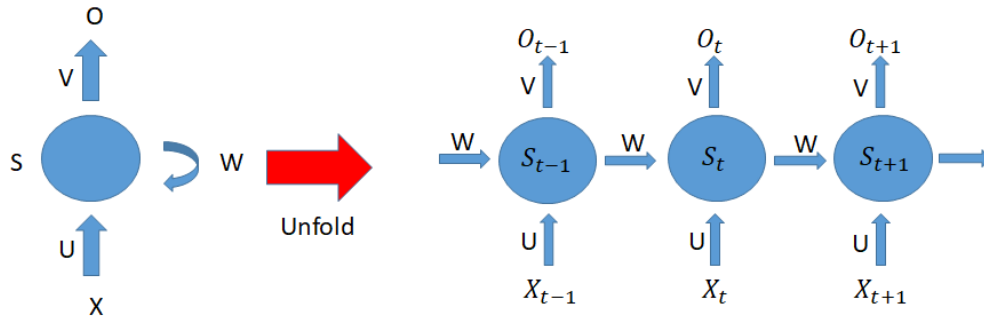


圖 2.27 循環神經網路神經元執行流程圖

本文主要運用 LSTM(long short-term memory)模型，此模型主要解決 RNN 之梯度消失問題(gradient vanishing)，誤差梯度會隨著時間長度增加，以指數成長的速度消失[34][35]。如下圖 2.28， X_t 代表輸入向量(vector)， Y_t 代表輸出向量，而 LSTM 主要控制閘門，分別為 I_t (input gate)、 F_t (forget gate) 及 O_t (output gate)，最後由 C_t 決定狀態，其中向量之間運算採用哈達瑪乘機(Hadamard product)。

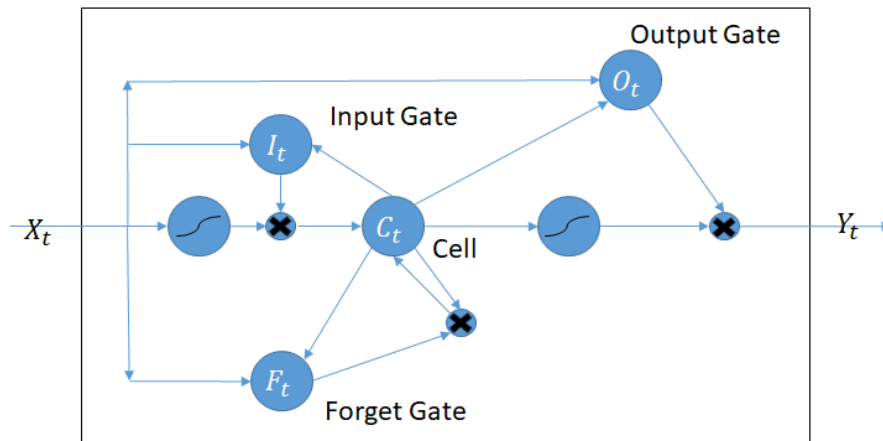


圖 2.28 長短期記憶模型執行流程圖

2.2.4 交叉熵誤差與梯度下降

在神經網路中，會尋找一個指標找到最佳參數，而所用的指標為損失函數(loss function)，較常使用如有均值誤差(mean squared error)與交叉熵誤差(cross-entropy

error)，本論文所運用損失函數為交叉熵誤差。令 \log 以 e 自然常數為底數， y_k 為神經網路輸出， t_k 為分類後標籤，然後進行批次學習，設總資料集個數為 N ，而 t_{nk} 代表在 n 個資料中第 k 個資料(其中 y_{nk} 為神經網路輸出， t_{nk} 為訓練資料)，結果如下表示：

$$E = -\sum_k t_k \log y_k, \quad (7.1)$$

$$E = -\frac{1}{N} \sum_n \sum_k t_{nk} \log y_{nk}. \quad (7.2)$$

其中學習目的是要找到最適合權重(weight)及偏權值，意指在找尋最適合參數為損失函數之最小值，因而會運用梯度下降法(gradient descent method)尋找函數最小值，以微分方式說明，表示方法如下定義：

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h)-f(x)}{h}. \quad (8)$$

於微分定義中，隨著 x 的改變函數 $f(x)$ 也將會有變化，若 h 趨近於 0 時，可以代表 $f(x)$ 之細微變化。藉由數值微分(numerical differentiation)方式求解微分方程。而當全部變數作偏微分當向量表示時，稱為梯度(gradient)。令 η 為學習率(learning rate)，即為每次學習之更新量，結果如下表示：

$$X = X - \eta \frac{\partial f}{\partial X}. \quad (9)$$

接下來以輸出資料為 $m \times n$ 大小形狀為例，在一個神經網路中，令權重為 W ，以 L 代表損失函數，而可以利用 $\frac{\partial L}{\partial W}$ 表示梯度，實際結果如下表示：

$$W = \begin{pmatrix} w_{11} & \cdots & w_{1m} \\ \vdots & \ddots & \vdots \\ w_{n1} & \cdots & w_{nm} \end{pmatrix}, \quad (10.1)$$

$$\frac{\partial L}{\partial W} = \begin{pmatrix} \frac{\partial L}{\partial w_{11}} & \cdots & \frac{\partial L}{\partial w_{1m}} \\ \vdots & \ddots & \vdots \\ \frac{\partial L}{\partial w_{n1}} & \cdots & \frac{\partial L}{\partial w_{nm}} \end{pmatrix}. \quad (10.2)$$

2.2.5 誤差反向傳播法

反向傳播演算法(BP, backpropagation)是由 Geoff Hinton 所提出[36]，其計算圖如下圖 2.29，如要計算 $f(x)$ ，針對節點進行偏微分($\frac{\partial y}{\partial x}$)，並針對上一層資訊 E ，進行相乘。

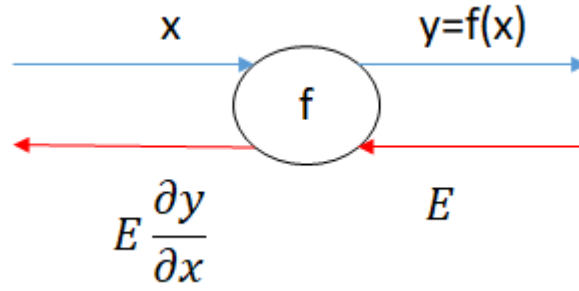


圖 2.29 反向傳播法以圖式顯示

接下來在正向傳播時，權重乘以輸入加上偏權值所代表之神經元，一樣進行反向傳播時接各參數進行偏微分，令 Y 代表輸出結果， X 代表輸入， W 代表權重， B 代表偏權值及 L 代表輸出損失，表示如下公式：

$$\text{正向傳播：} Y = X \cdot W + B, \quad (11.1)$$

$$\text{反向傳播} X : \frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} \cdot W^T, \quad (11.2)$$

$$\text{反向傳播} W : \frac{\partial L}{\partial W} = X^T \cdot \frac{\partial L}{\partial Y}, \quad (11.3)$$

$$\text{反向傳播} B : \frac{\partial L}{\partial B} = \sum \frac{\partial L}{\partial Y}. \quad (11.4)$$

2.3 基於類神經網路對於惡意程式分類技術

2.3.1 樣本資料前置處理

目前利用機器學習、類神經網路等演算法進行惡意程式分類方式，在對各惡意程式樣本執行資料前置處理時，有針對將樣本所呼叫之 Windows API 擷取出，如下圖 2.30。

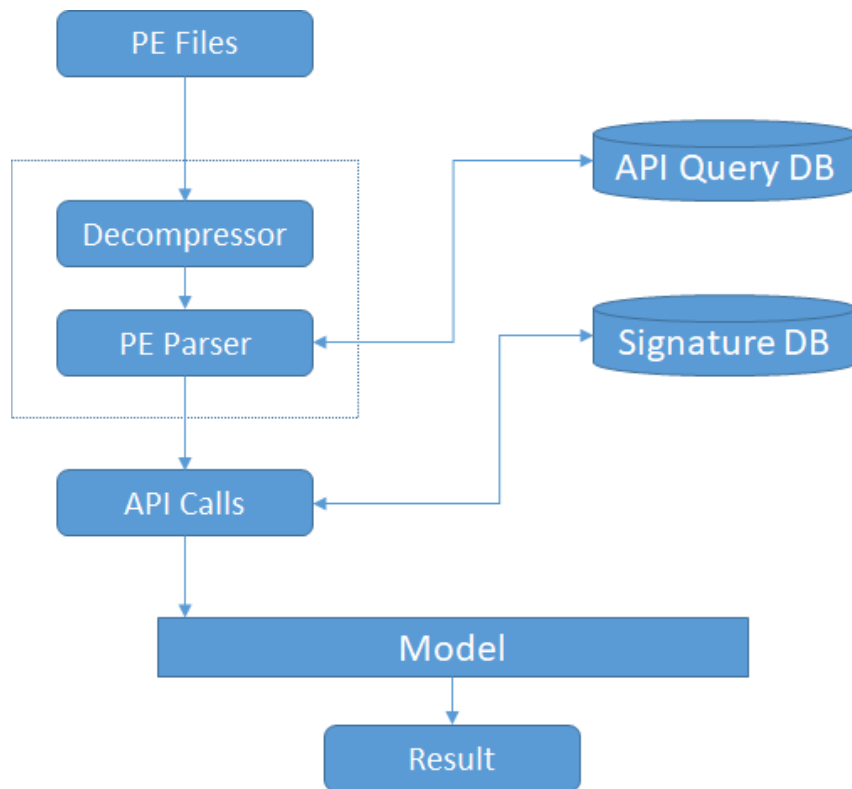


圖 2.30 於 DL4MD 論文中提出惡意程式資料前置處理方法[37]

另外為惡意程式圖像化，如下圖 2.31，將以二進位方式讀取每一個位元組(byte)，數值介於 0x00 至 0xFF 之間(0 到 255 之間)，每一個位元組表達一個灰階影像像素(pixel)，資料讀取完後以固定長度、寬度並以圖檔方式存檔[38][39]。

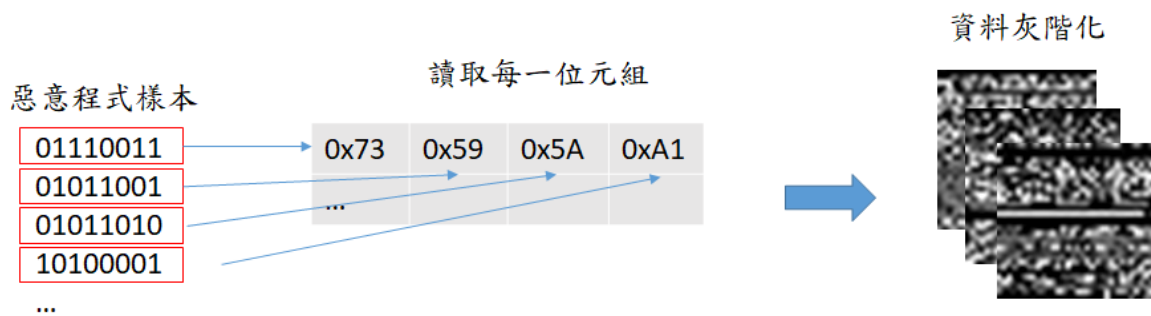


圖 2.31 惡意程式樣本圖像化

2.3.2 微軟惡意程式分類大賽第一名隊伍演算法

在 2015 年微軟藉由 Kaggle 平台舉辦惡意程式分類大賽中，第一名隊伍(NO to overfitting!)主要是利用 xgboost 函示庫，其演算法如下圖 2.34，使用隨機生成樹(random forest)方式進行分類，而在特徵處理過程中，針對 Opcode 部分有作 2-gram、

3-gram 及 4-gram，其次為區段(segment，此為 PE 檔頭資訊)，此比賽最具有亮點為將惡意程式轉換圖像技術(Lakshmanan Nataraj 於 2011 年提出)。但在此資料集中，樣本族群為「Simba」僅在訓練資料中全部樣本 10868 筆佔 42 筆，資料分佈並不平均(imbalanced data)[40]。

2.3.3 作者 Daniel Gibert 提出演算法

主要是將惡意程式之二進位檔案(binary file)轉換為灰階影像，該研究主要使用資料集為微軟於 2015 年舉辦惡意程式分類大賽。該研究將惡意程式轉換之灰階影像縮放(resize)至長、寬度皆為 32 像素，設計 CNN 類神經網路模型進行訓練。在此資料集中，總共區分兩部分，其中為組合語言檔(利用 IDA Pro 工具逆向產生程式碼)，另外為十六進位文字檔(代表惡意程式二進位檔案)。該資料集訓練資料計(training data)有 10868 筆樣本，測試資料(testing data)有 10873 筆樣本，共有 9 種標籤[41]。

該作者之類神經模型提出 3 種模型，第一種模型如下：

- (1) 資料輸入：32 * 32 * 1
- (2) Conv2D：11*11 size, 64 filters
- (3) Max-Pooling
- (4) FC：4096
- (5) FC：9

第二種模型如下：

- (1) 資料輸入：32 * 32 * 1
- (2) Conv2D：3*3 size, 64 filters
- (3) Max-Pooling
- (4) Conv2D：3*3 size, 128 filters
- (5) Max-Pooling
- (6) FC：512
- (7) FC：9

第三種模型如下：

- (1) 資料輸入：32 * 32 * 1
- (2) Conv2D：3*3 size, 64 filters
- (3) Max-Pooling
- (4) Conv2D：3*3 size, 128 filters
- (5) Max-Pooling
- (6) Conv2D：3*3 size, 256 filters
- (7) Max-Pooling

- (8) FC : 1024
- (9) FC : 512
- (10) FC : 9

其中以第二種模型，正確率為 99.76% 最高(Cross-entropy : 0.0257)。

2.3.4 作者 Abien Fred M. Agarap 提出演算法

該論文主要利用 CNN、GRU(gate recurrent unit)及 MLP 實作，針對各類神經網路輸出層在利用 SVM(support vector machine)作分類，依序作者實驗 CNN-SVM、GRU-SVM 及 MLP-SVM，該研究最後結果以 GRU-SVM 之正確率(accuracy)84.92% 最高 [42]。該研究所用資料集與本論文相同，採用 Maling 蠕蟲資料集。其中該篇研究類神經網路模型中，各神經元所使用激活函數為 LeakyReLU，其中 m 代表斜率，公式表示如下：

$$y = \begin{cases} x & (x > 0) \\ mx & (x \leq 0) \end{cases} \quad (12)$$



第三章 系統設計

本章節藉由 Malimg 惡意程式資料集，利用 CNN 與 RNN 系列類神經網路模型作結合運用。

3.1 資料集前置處理

該惡意程式資料集已運用樣本圖像化方法轉為二維陣列方式，如下圖 3.1，其中長寬皆為 32 像素，數值區間為 0 至 255 之間，將訓練資料進行各標籤分類，分類完後將數值進行規一化處理(normalization)，將調整為 0 至 1 之間。處理完後進行類神經網路模型訓練，即可達到惡意程式分類效果。其中該資料集具有 25 標籤，共計 9339 樣本數，分類項目如下表 3.1。

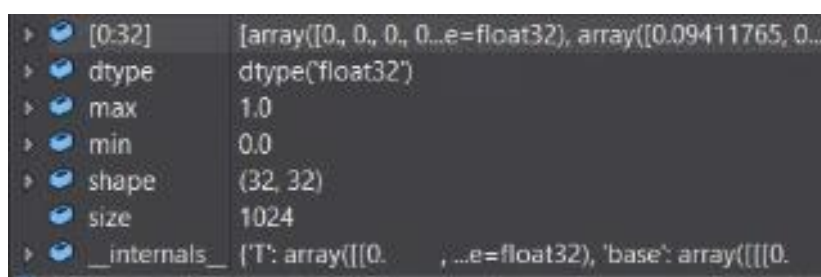


圖 3.1 資料集長寬大小(shape)與規一化數值範圍值(max 與 min)

表 3.1 惡意程式樣本 Malimg 詳細資訊

項次	族群	名稱	個數
1	Dialer	Adialer.C	122
2	Backdoor	Agent.FYI	116
3	Worm	Allapple.A	2949
4	Worm	Allapple.L	1591
5	Trojan	Alueron.gen!J	198
6	Worm:AutoIT	Autorun.K	106
7	Trojan	C2Lop.P	146
8	Trojan	C2Lop.gen!G	200
9	Dialer	Dialplatform.B	177
10	Trojan	Downloader Dontovo.A	162
11	Rogue	Fakerean	381
12	Dialer	Instantaccess	431
13	PWS	Lolyda.AA 1	213
14	PWS	Lolyda.AA 2	184
15	PWS	Lolyda.AA 3	123

16	PWS	Lolyda.AT	159
17	Trojan	Malex.gen!J	136
18	Trojan	Downloader Obfuscator.AD	142
19	Backdoor	Rbot!gen	158
20	Trojan	Skintrim.N	80
21	Trojan	Downloader Swizzor.gen!E	128
22	Trojan	Downloader Swizzor.gen!I	132
23	Worm	VB.AT	408
24	Trojan	Downloader Wintrim.BX	97
25	Worm	Yuner.A	800

該資料集將陣列數值輸出圖像，依據蠕蟲族群名稱如圖 3.2 至 3.26。



圖 3.2 類別 1- Adialer.C



圖 3.3 類別 2- Agent.FYI



圖 3.4 類別 3- Allapple.A



圖 3.5 類別 4- Allapple.L

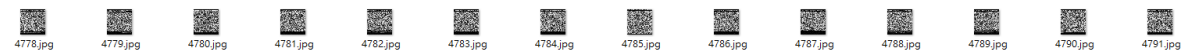


圖 3.6 類別 5- Alueron.gen!J



圖 3.7 類別 6- Autorun.K

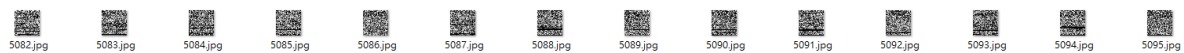


圖 3.8 類別 7- C2Lop.P

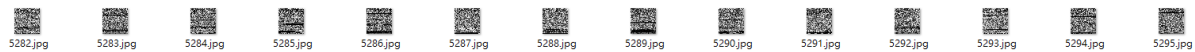


圖 3.9 類別 8- C2Lop.gen!G



圖 3.10 類別 9- Dialplatform.B

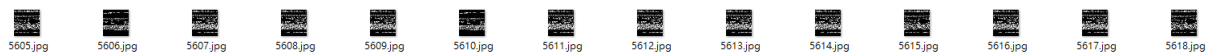


圖 3.11 類別 10- Downloader Dontovo.A

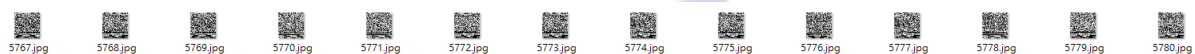


圖 3.12 類別 11- Fakerean

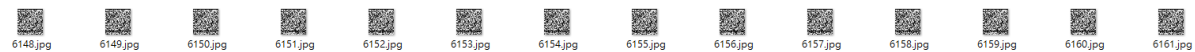


圖 3.13 類別 12- Instantaccess



圖 3.14 類別 13- Lolyda.AA 1

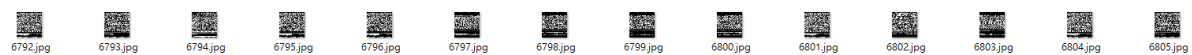


圖 3.15 類別 14- Lolyda.AA 2



圖 3.16 類別 15- Lolyda.AA 3



圖 3.17 類別 16- Lolyda.AT

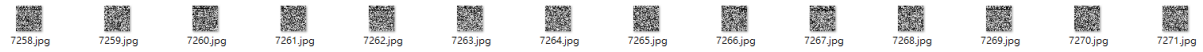


圖 3.18 類別 17- Malex.gen!J

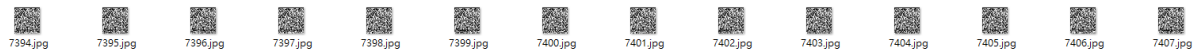


圖 3.19 類別 18- Downloader Obfuscator.AD



圖 3.20 類別 19- Rbot!gen

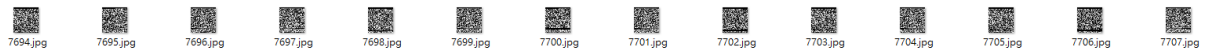


圖 3.21 類別 20- Skintrim.N

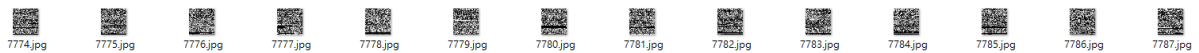


圖 3.22 類別 21- Downloader Swizzor.gen!E

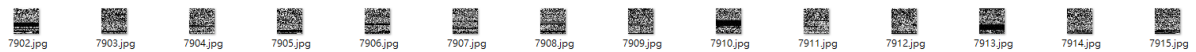


圖 3.23 類別 22- Downloader Swizzor.gen!I

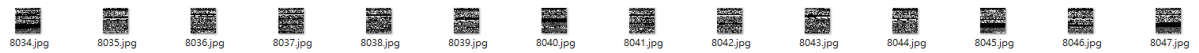


圖 3.24 類別 23- VB.AT

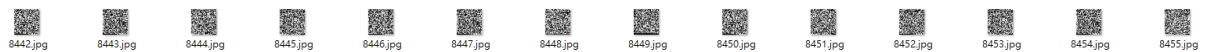


圖 3.25 類別 24- Downloader Wintrim.BX

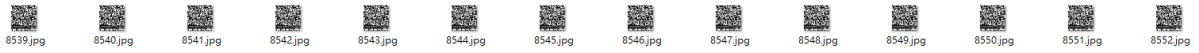


圖 3.26 類別 25- Yuner.A

3.2 類神經網路設計

資料處理完後，後續實驗將以 CNN、RNN、LSTM、GRU、CNN+RNN、CNN+LSTM、CNN+GRU 等 7 種類神經網路模型，與 Abien Fred M. Agarap 之模型比較。

如下圖 3.27，其中會將 CNN 與 RNN 系列類神經網路模型結合，主要是將特徵擷取與時間順序性特點之模型結合，旨在將關鍵惡意程式特徵不因免殺技巧、程式邏輯改變等其他方式調換而改變分類正確率。

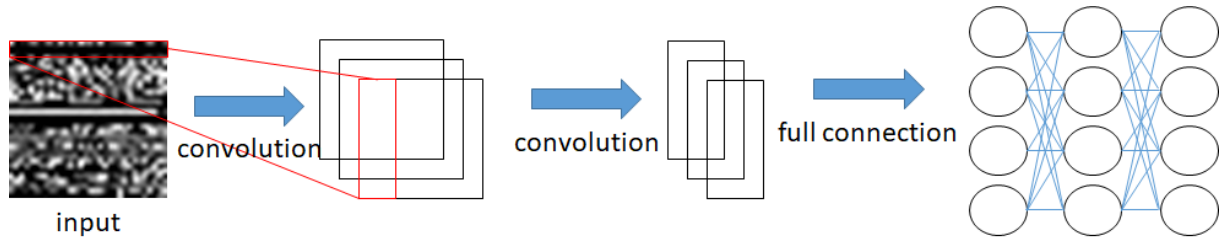


圖 3.27 運用 CNN 與 RNN 類神經網路模型結合示意圖[43]

最後結果比較，使用混淆矩陣(confusion matrix)檢查以監督式學習演算法結果，藉此可看出該模型是否混淆兩個或多個類別；另外將會以正確率(accuracy)、F1 score、精確率(precision)、召回率(recall)，以預測正確與方向作設定，令 TP 為 True Positive、FP 為 False Positive、FN 為 False Negative 及 TN 為 True Negative，其表示如下：

$$\text{Accurcay} = \frac{TP+TN}{TP+TN+FP+FN} \quad , \quad (13.1)$$

$$\text{Precision} = \frac{TP}{TP+FP} \quad , \quad (13.2)$$

$$\text{Recall} = \frac{TP}{TP+FN} \quad , \quad (13.3)$$

$$\text{F1 Score} = \frac{2TP}{2TP+FP+FN} \quad . \quad (13.4)$$

第四章 實驗結果分析與討論

4.1 實驗環境

訓練模型之硬體及系統環境如下表 4.1。

表 4.1 硬體及系統環境說明

項目	名稱
作業系統	Microsoft Windows 10 x64 bits
處理器	AMD Ryzen Threadripper 2950X 16-Core Processors @ 3.42 GHz
記憶體	8GB * 3 DDR4
硬碟	ADATA SX930 SSD 256GB
圖形顯示卡	NVIDIA GeForce GTX 1080 Ti 11GB DDR5

在實驗惡意程式分類中，以 Python 程式語言實作，運用函示庫計有 Google Tensorflow-GPU[44]版本、Keras、Numpy、Mateplotlib[45]及 Scikit-learn[46]。其中函示庫 Tensorflow 為 GPU 版本，利用圖形顯示卡即可大幅度加速類神經網路訓練[47]。

4.2 實驗結果與比較

本文將資料集進行切割，以 80% 資料作為訓練資料(training data)及 20% 資料最為測試資料(testing data)，訓練批次(batch size)資料為 256，共執行 100 與 200 次訓練回合(epoch)。令 Malimg 之資料為 x ，分類為 y 個類別，即可表達為 $f: x \mapsto y$ 。

以下實驗會以 CNN、RNN、LSTM、GRU、CNN+RNN、CNN+GRU 及 CNN+LSTM 共 7 種類神經模型進行比較，其中各模型架構中隱藏層之神經元數及深度皆相同，因避免在模型訓練中產生過度擬合(overfitting)，若發現誤差值比上一個回合(epoch)訓練沒有下降將會停止訓練(early stop)。最後運用 Adam 作為優化器(optimizer)[48]，此方法是將 Momentum 與 Adagrad 融合，組合兩手法優點[49]，進行參數校正，其學習率(learning rate)為 10^{-3} ， β_1 為 0.9， β_2 為 0.999。

4.2.1 CNN 模型

CNN 模型建構之類神經網路模型如下，其正確率、對數損失函數(logarithmic loss function)與混淆矩陣結果如下圖 4.1 至圖 4.3：

- (1) Conv1D：7 size, 32 filters, 初始值以 normal distribution 初始化
- (2) Dropout：0.5 rate
- (3) Conv1D：9 size, 32 filters, 初始值以 normal distribution 初始化
- (4) Dropout：0.5 rate
- (5) FC：512, 初始值以 normal distribution 初始化
- (6) Dropout：0.25 rate
- (7) FC：512, 初始值以 normal distribution 初始化
- (8) Dropout：0.25 rate
- (9) FC：512, 初始值以 normal distribution 初始化
- (10) Dropout：0.25 rate
- (11) FC：25, 初始值以 normal distribution 初始化

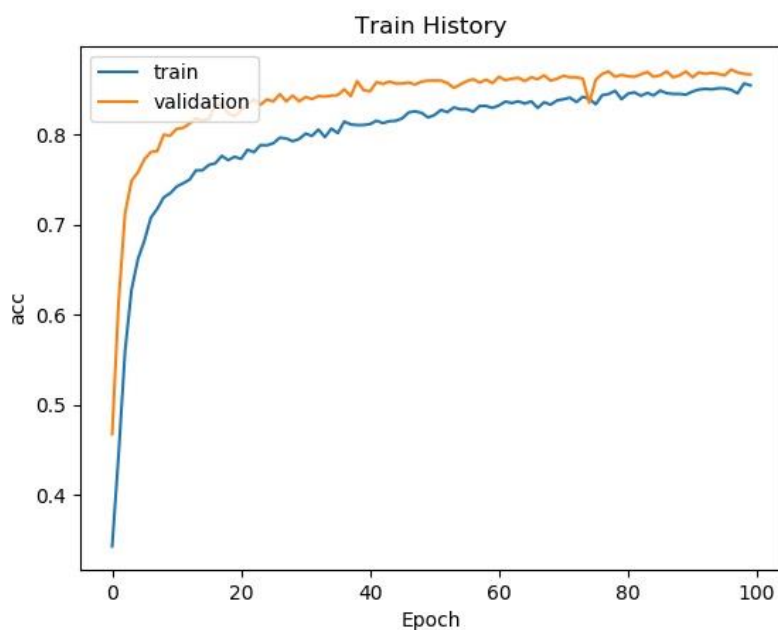


圖 4.1 利用 matplotlib 繪製 CNN 模組訓練之正確率(y)與訓練週期(x)

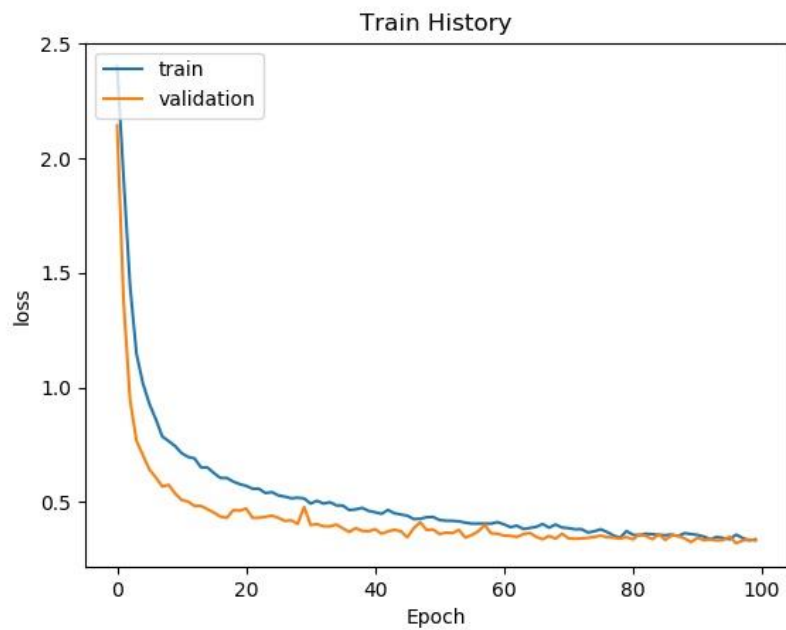


圖 4.2 利用 matplotlib 繪製 CNN 模組訓練之 $\log \text{loss}(y)$ 與訓練週期(x)

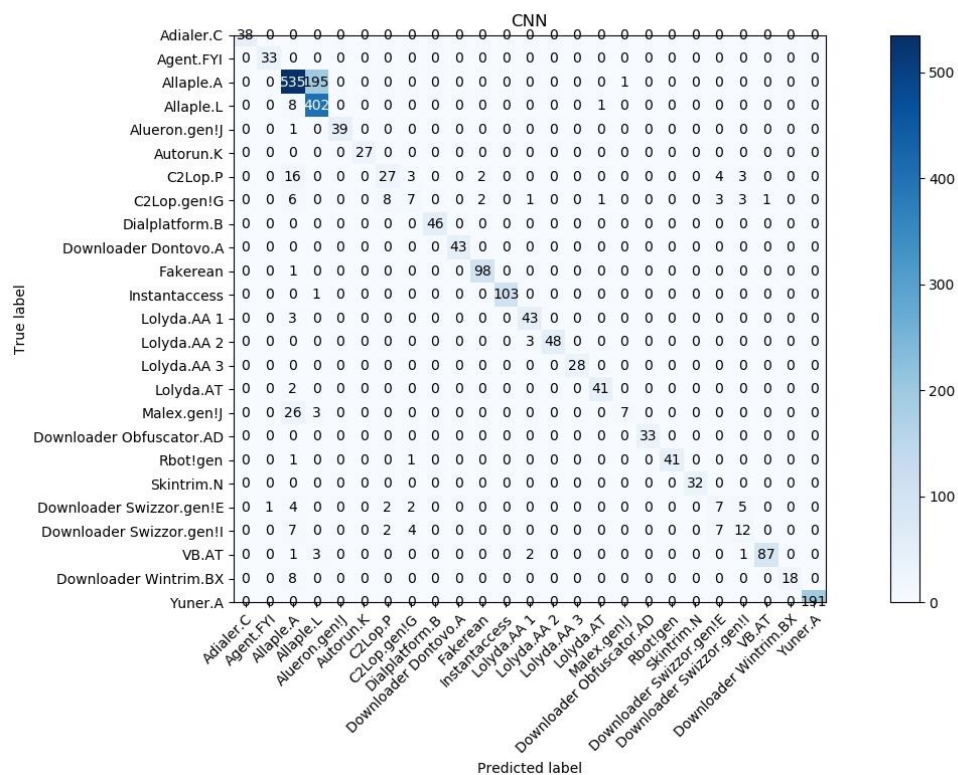


圖 4.3 利用 matplotlib 繪製 CNN 模組訓練之混淆矩陣

4.2.2 RNN 模型

RNN 模型建構之類神經網路模型如下，其正確率、對數損失函數與混淆矩陣結果如下圖 4.4 至圖 4.6：

- (1) RNN : 128
- (2) Dropout : 0.5 rate
- (3) FC : 512, normal distribution 初始化
- (4) Dropout : 0.25 rate
- (5) FC : 512, normal distribution 初始化
- (6) Dropout : 0.25 rate
- (7) FC : 512, normal distribution 初始化
- (8) Dropout : 0.25 rate
- (9) FC : 25, normal distribution 初始化

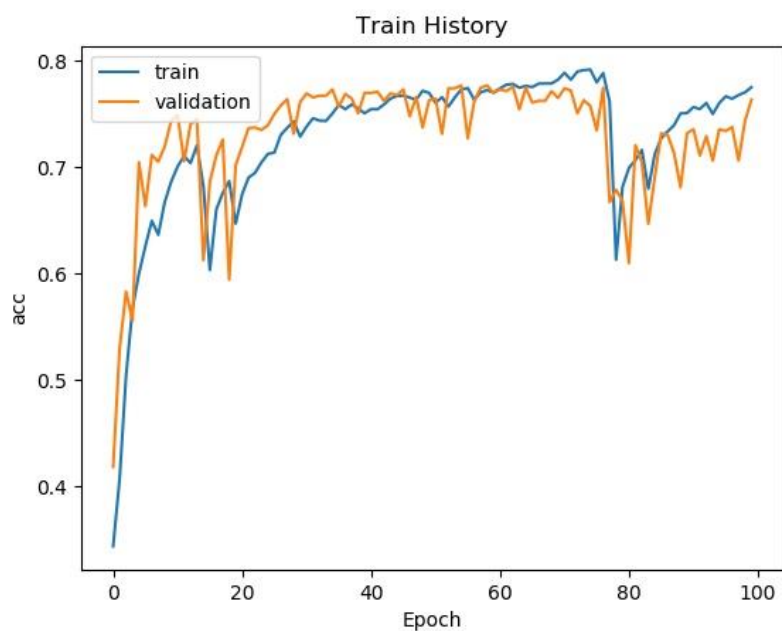


圖 4.4 利用 matplotlib 繪製 RNN 模組訓練之正確率(y)與訓練週期(x)

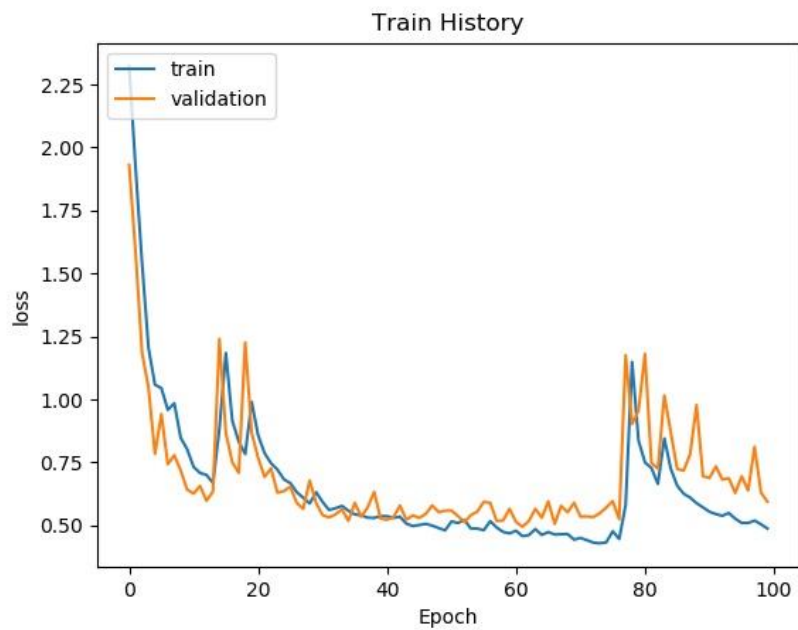


圖 4.5 利用 matplotlib 繪製 RNN 模組訓練之 $\log \text{loss}(y)$ 與訓練週期(x)

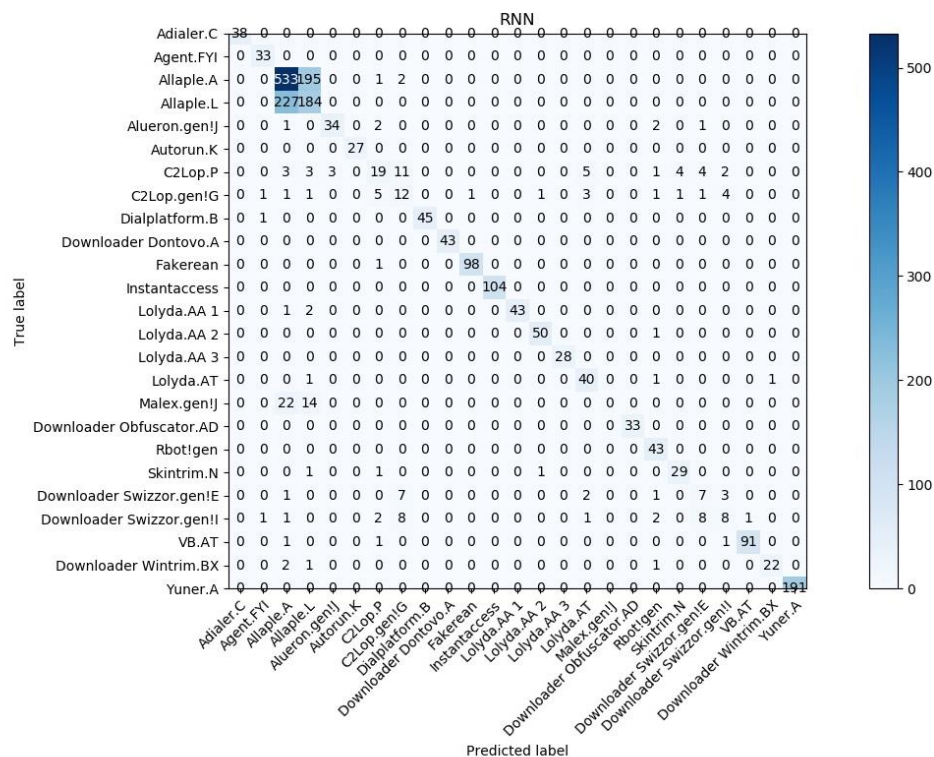


圖 4.6 利用 matplotlib 繪製 RNN 模組訓練之混淆矩陣

4.2.3 LSTM 模型

CNN 模型建構之類神經網路模型如下，其正確率、對數損失函數與混淆矩陣結果如下圖 4.7 至圖 4.9：

- (1) LSTM：128
- (2) Dropout：0.5 rate
- (3) FC：512, normal distribution 初始化
- (4) Dropout：0.25 rate
- (5) FC：512, normal distribution 初始化
- (6) Dropout：0.25 rate
- (7) FC：512, normal distribution 初始化
- (8) Dropout：0.25 rate
- (9) FC：25, normal distribution 初始化

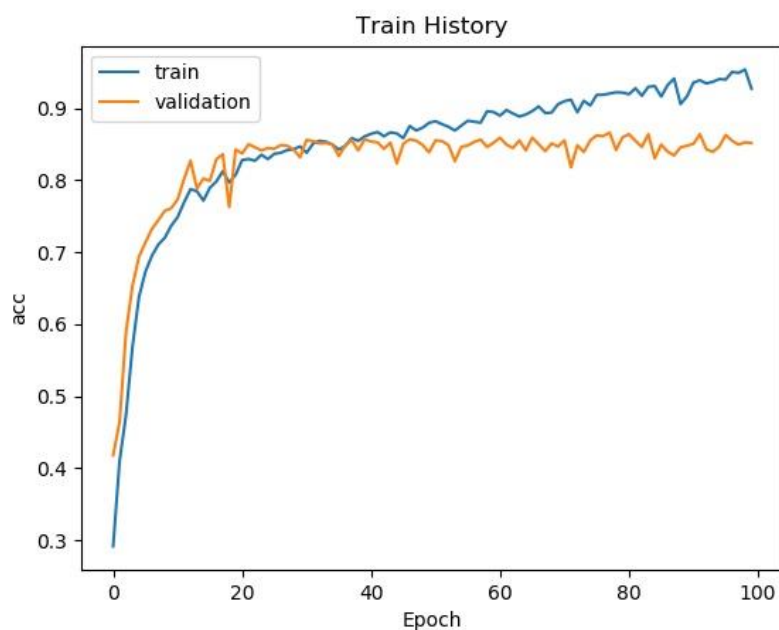


圖 4.7 利用 matplotlib 繪製 LSTM 模組訓練之正確率(y)與訓練週期(x)

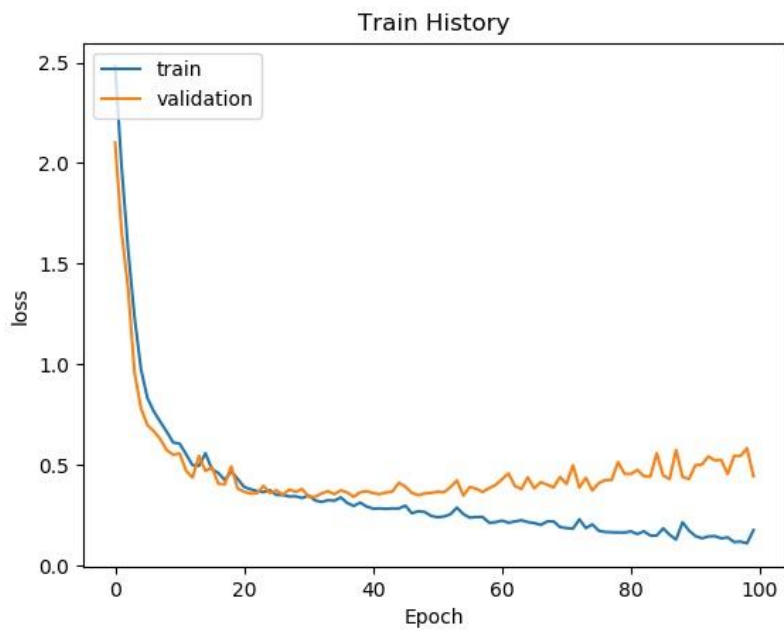


圖 4.8 利用 matplotlib 繪製 LSTM 模組訓練之 $\log \text{loss}(y)$ 與訓練週期(x)

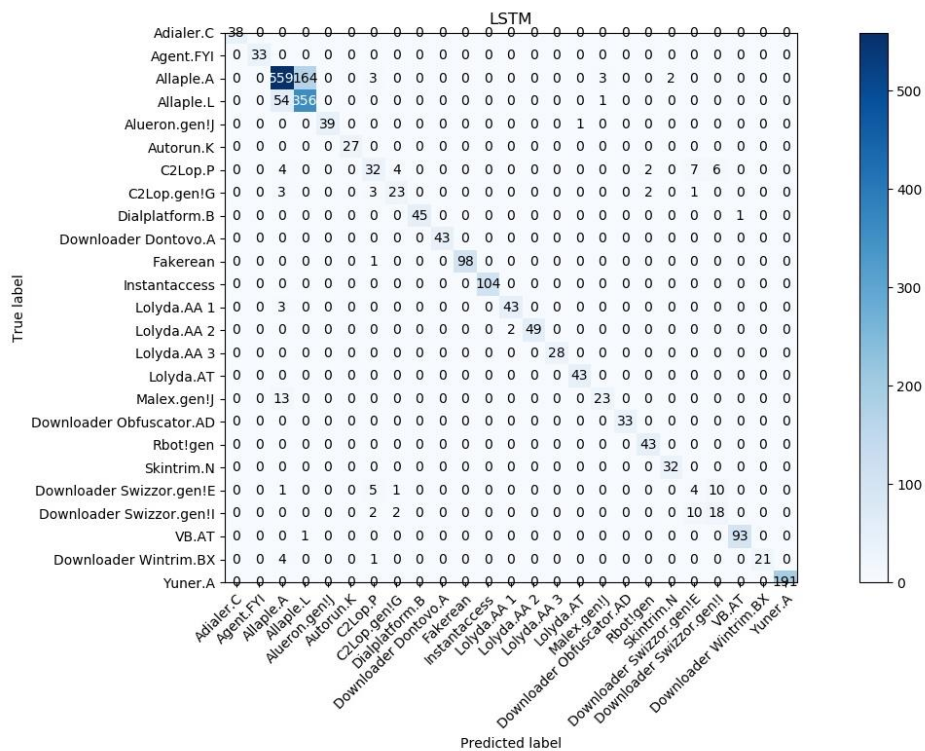


圖 4.9 利用 matplotlib 繪製 LSTM 模組訓練之混淆矩陣

4.2.4 GRU 模型

GRU 模型建構之類神經網路模型如下，其正確率、對數損失函數與混淆矩陣結果如下圖 4.10 至圖 4.12：

- (1) GRU : 128
- (2) Dropout : 0.5 rate
- (3) FC : 512, normal distribution 初始化
- (4) Dropout : 0.25 rate
- (5) FC : 512, normal distribution 初始化
- (6) Dropout : 0.25 rate
- (7) FC : 512, normal distribution 初始化
- (8) Dropout : 0.25 rate
- (9) FC : 25, normal distribution 初始化

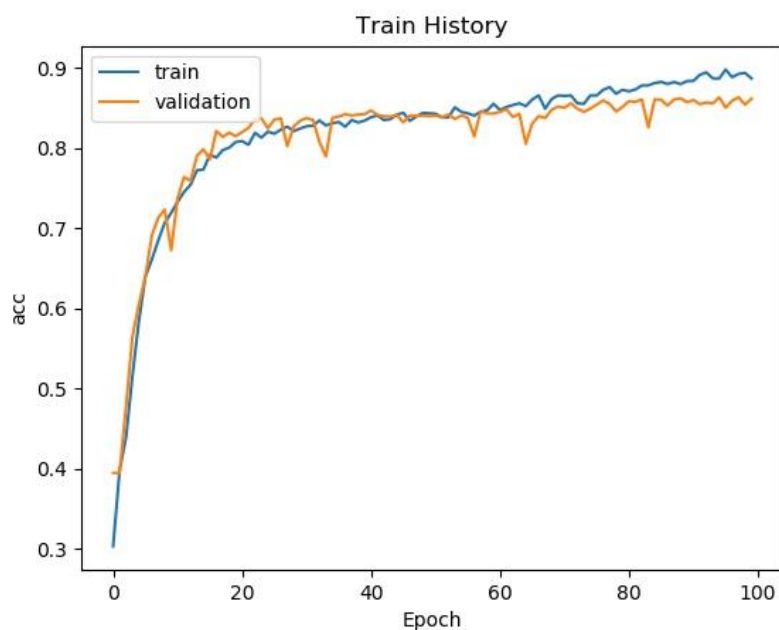


圖 4.10 利用 matplotlib 繪製 GRU 模組訓練之正確率(y)與訓練週期(x)

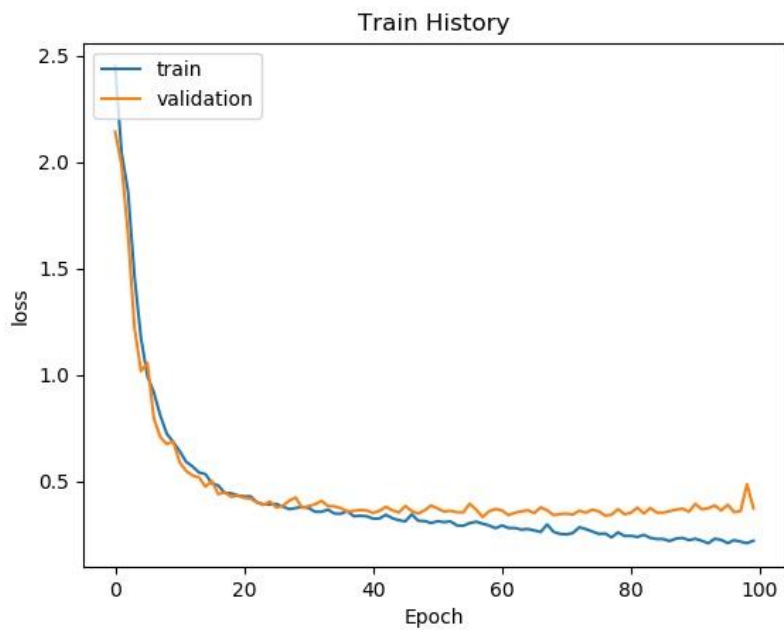


圖 4.11 利用 matplotlib 繪製 GRU 模組訓練之 $\log \text{loss}(y)$ 與訓練週期(x)

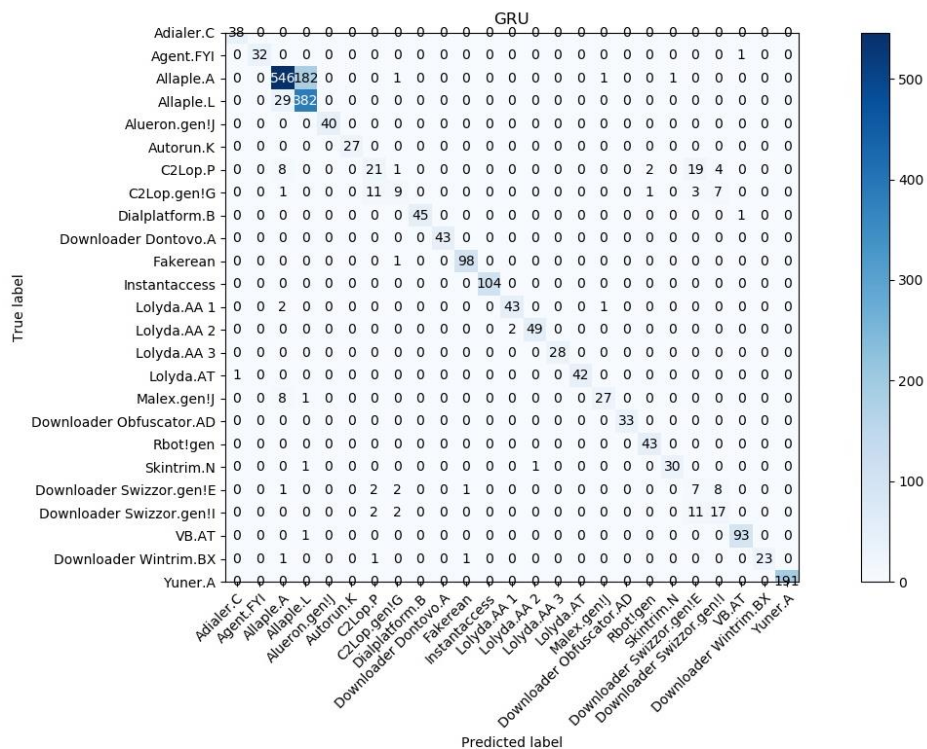


圖 4.12 利用 matplotlib 繪製 GRU 模組訓練之混淆矩陣

4.2.5 CNN+RNN 模型

CNN+RNN 模型建構之類神經網路模型如下，其正確率與對數損失函數結果如下圖 4.13 至圖 4.15：

- (1) Conv1D：7 size, 32 filters, normal distribution 初始化
- (2) Dropout：0.25 rate
- (3) Conv1D：9 size, 32 filters, normal distribution 初始化
- (4) Dropout：0.25 rate
- (5) RNN：128, normal distribution 初始化
- (6) Dropout：0.25 rate
- (7) FC：512, normal distribution 初始化
- (8) Dropout：0.25 rate
- (9) FC：512, normal distribution 初始化
- (10) Dropout：0.25 rate
- (11) FC：512, normal distribution 初始化
- (12) Dropout：0.25 rate
- (13) FC：25, normal distribution 初始化

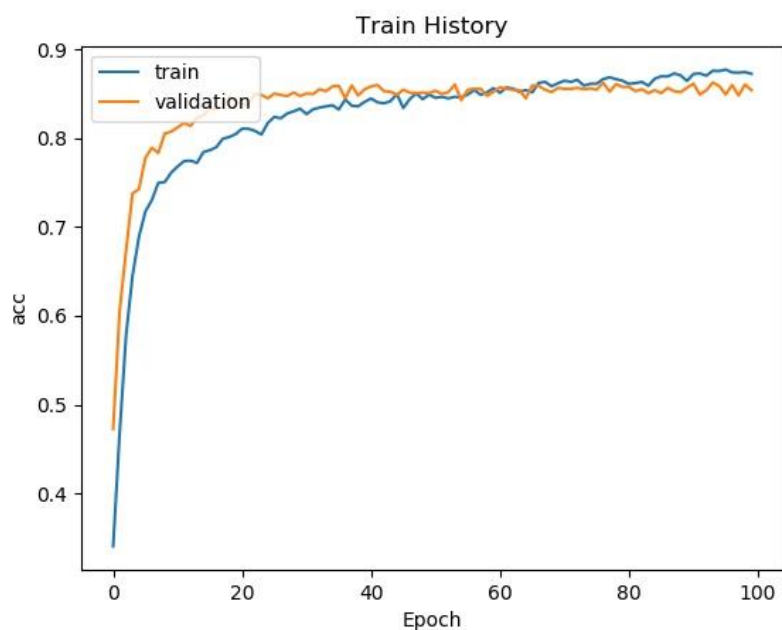


圖 4.13 利用 matplotlib 繪製 CNN+RNN 模組訓練之正確率(y)與訓練週期(x)

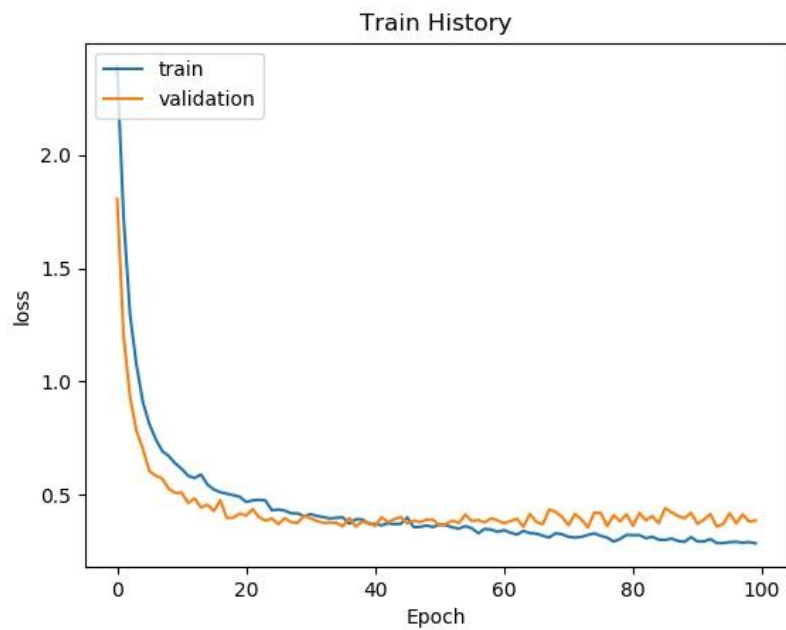


圖 4.14 利用 matplotlib 繪製 CNN+RNN 模組訓練之 log loss(y)與訓練週期(x)

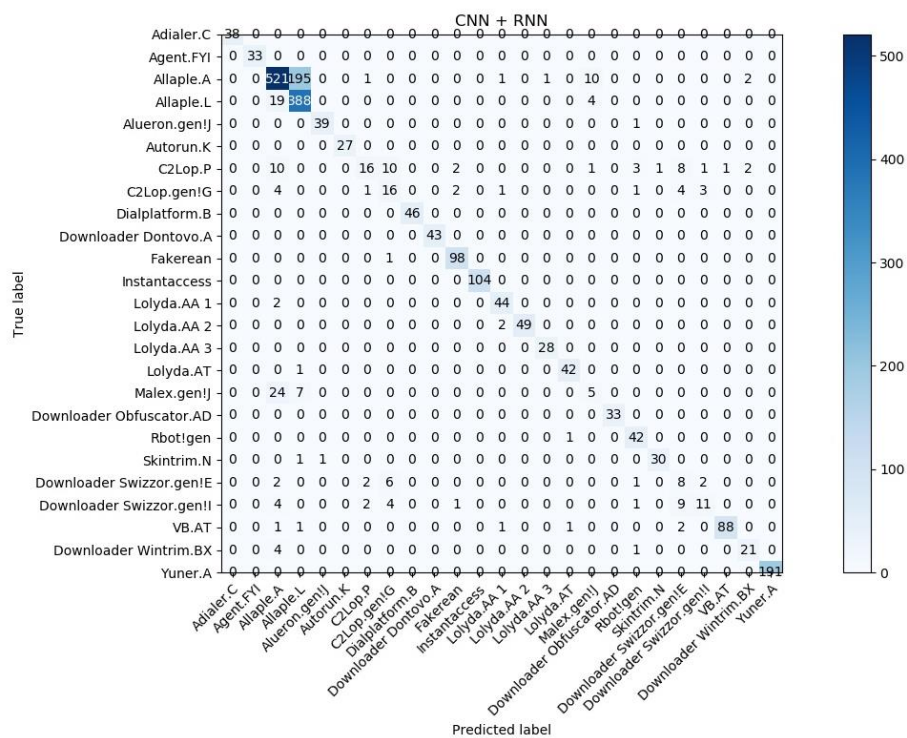


圖 4.15 利用 matplotlib 繪製 CNN+RNN 模組訓練之混淆矩陣

4.2.6 CNN+LSTM 模型

CNN+LSTM 模型建構之類神經網路模型如下，其正確率、對數損失函數與混淆矩陣結果如下圖 4.16 至圖 4.18：

- (1) Conv1D：7 size, 32 filters, normal distribution 初始化
- (2) Dropout：0.25 rate
- (3) Conv1D：9 size, 32 filters, normal distribution 初始化
- (4) Dropout：0.25 rate
- (5) LSTM：128, normal distribution 初始化
- (6) FC：512, normal distribution 初始化
- (7) Dropout：0.25 rate
- (8) FC：512, normal distribution 初始化
- (9) Dropout：0.25 rate
- (10) FC：512, normal distribution 初始化
- (11) Dropout：0.25 rate
- (12) FC：25, normal distribution 初始化

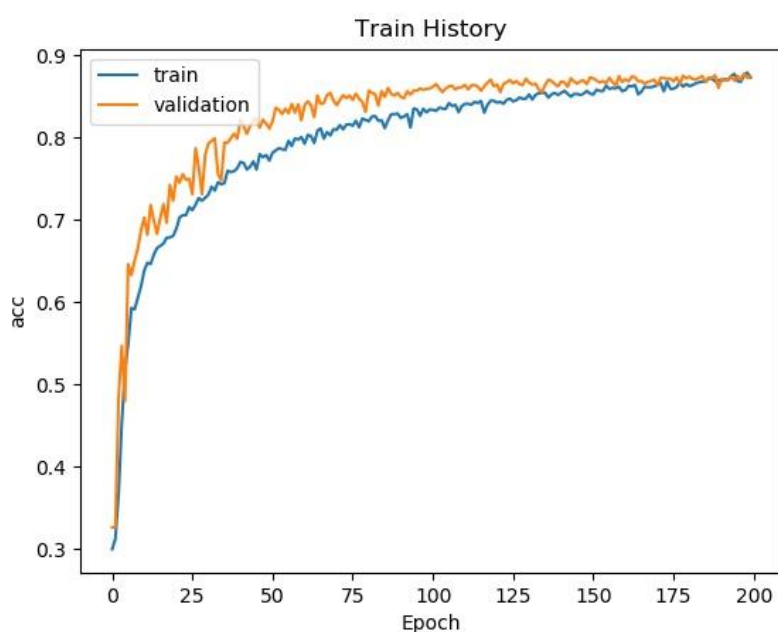


圖 4.16 利用 matplotlib 繪製 CNN+LSTM 模組訓練之正確率(y)與訓練週期(x)

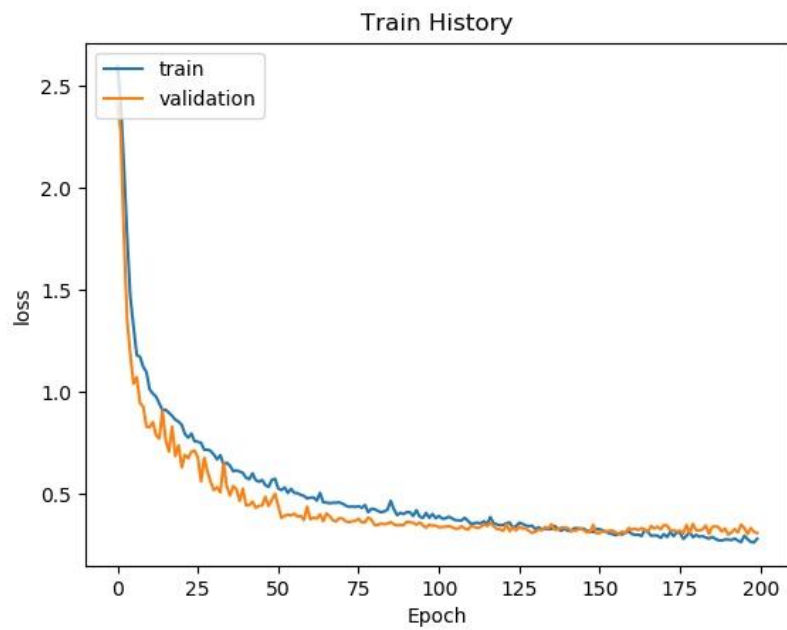


圖 4.17 利用 matplotlib 繪製 CNN+LSTM 模組訓練之 $\log \text{loss}(y)$ 與訓練週期(x)

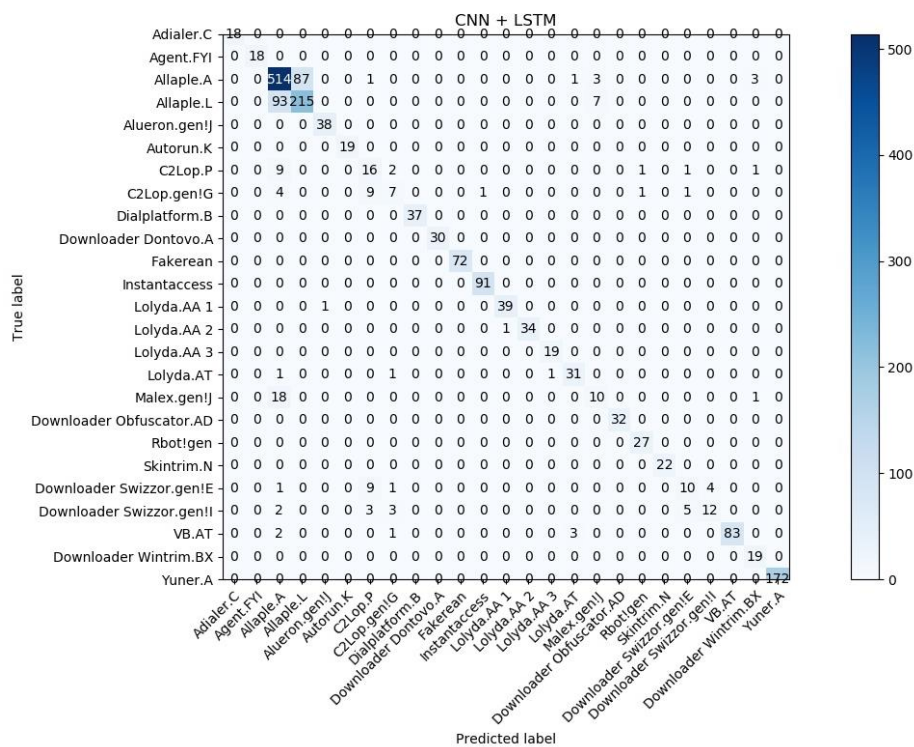


圖 4.18 利用 matplotlib 繪製 CNN+LSTM 模組訓練之混淆矩陣

4.2.7 CNN+GRU 模型

CNN+GRU 模型建構之類神經網路模型如下，其正確率與對數損失函數結果如下圖 4.19 至圖 4.21：

- (1) Conv1D：7 size, 32 filters, normal distribution 初始化
- (2) Dropout：0.25 rate
- (3) Conv1D：9 size, 32 filters, normal distribution 初始化
- (4) Dropout：0.25 rate
- (5) GRU：128, normal distribution 初始化
- (6) Dropout：0.25 rate
- (7) FC：512, normal distribution 初始化
- (8) Dropout：0.25 rate
- (9) FC：512, normal distribution 初始化
- (10) Dropout：0.25 rate
- (11) FC：512, normal distribution 初始化
- (12) Dropout：0.25 rate
- (13) FC：25, normal distribution 初始化

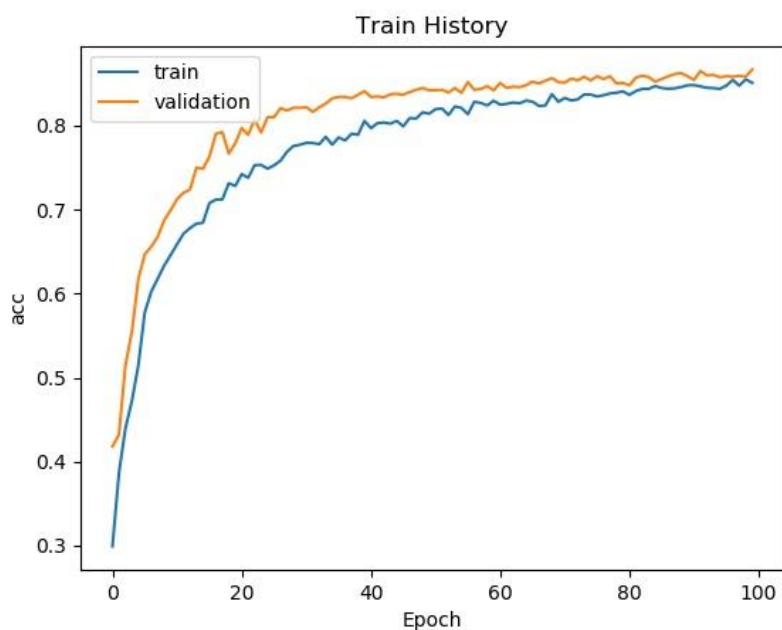


圖 4.19 利用 matplotlib 繪製 CNN+GRU 模組訓練之正確率(y)與訓練週期(x)

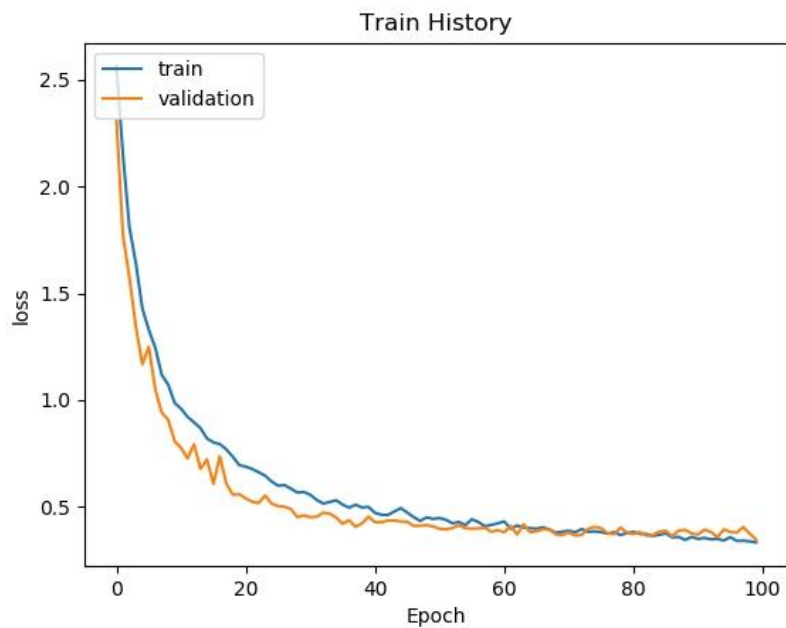


圖 4.20 利用 matplotlib 繪製 CNN+GRU 模組訓練之 log loss(y)與訓練週期(x)

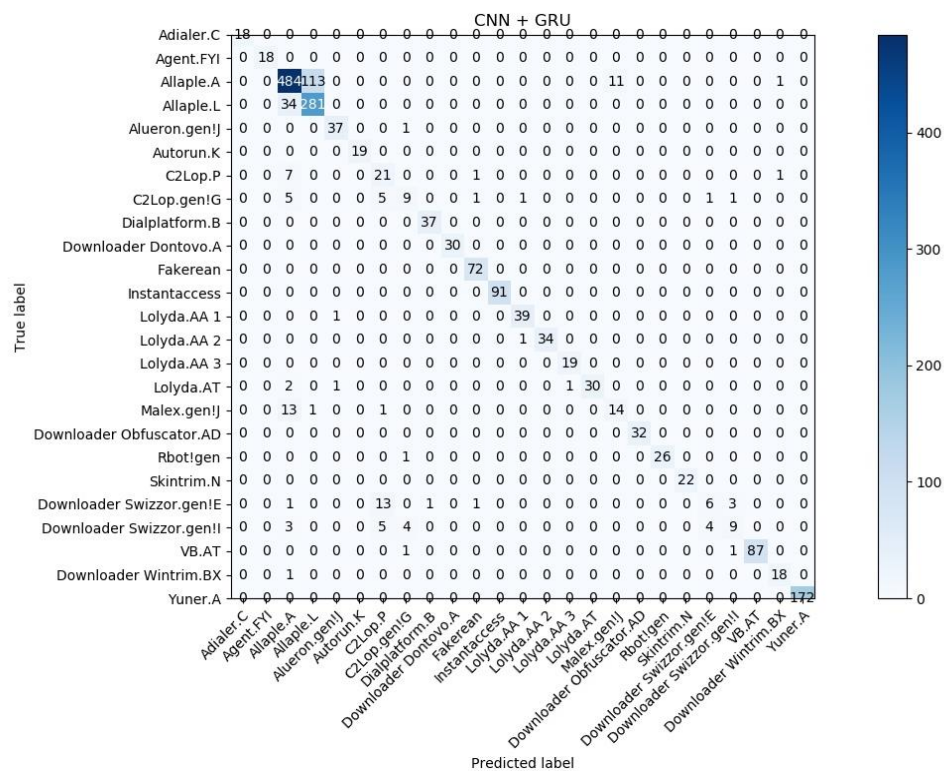


圖 4.21 利用 matplotlib 繪製 CNN+GRU 模組訓練之混淆矩陣

4.3 實驗綜合比較

其本文參數及正確率與 AF Agarap(2019)所實驗之 CNN-SVM、GRU-SVM 及 MLP-SVM 作比較，如下表 4.2、4.3。

表 4.2 使用參數比較

模型	Batch	FC	Dropout	Epoch	LR
CNN-SVM	256	2	0.85	100	10^{-3}
GRU-SVM	256	5	0.85	100	10^{-3}
MLP-SVM	256	5	None	100	10^{-3}
CNN	256	3	0.5/0.25	100	10^{-3}
RNN	256	3	0.5/0.25	100	10^{-3}
LSTM	256	3	0.5/0.25	100	10^{-3}
GRU	256	3	0.5/0.25	100	10^{-3}
CNN+RNN	256	3	0.25	200	10^{-3}
CNN+LSTM	256	3	0.25	200	10^{-3}
CNN+GRU	256	3	0.25	200	10^{-3}

表 4.3 結果比較

模型	Accuracy	F1	Precision	Recall
CNN-SVM	77.23%	0.79	0.84	0.77
GRU-SVM	84.92%	0.85	0.85	0.85
MLP-SVM	80.47%	0.81	0.83	0.80
CNN	86.40%	0.87	0.88	0.86
RNN	76.34%	0.73	0.87	0.63
LSTM	85.17%	0.85	0.86	0.85
GRU	86.12%	0.86	0.86	0.86
CNN+RNN	86.67%	0.87	0.88	0.86
CNN+LSTM	87.79%	0.88	0.88	0.87
CNN+GRU	87.58%	0.88	0.88	0.87

經實驗結果，處理此資料集進行惡意程式分類利用單一模型已達到 84.92% 準確率效果 (GRU-SVM)，若將 CNN 及 LSTM 模型可提高至 87.79% 準確率，另外發現當 CNN 結合 RNN 類型模型時可提高正確率。此模型結合方法除將惡意程式之特徵擷取，也解決程式碼順序性問題。

第五章 結論與未來工作

由於目前以公開情資之惡意程式資料庫數量有限及僅限針對 Windows 作業系統惡意程式，後續將利用此演算法結合目前工作上運用，目前已像國家高速網路與計算中心之惡意程式資料庫(malware knowledge base)[50]，並進一步針對其他作業系統平台(如：Android、Linux 等)之惡意程式分類。

在 Malimg 研究後，觀察各類神經網路模型之混淆矩陣發現，在分類「Allapple.A」與「Allapple.L」較易混淆，由於該資料集已經過灰階圖像化處理，後續將利用國網中心惡意程式資料庫中所有樣本，從逆向完組合語言碼進行與圖像化方式進行比對。

毛敬豪所長提出針對惡意程式轉成圖像化(bitmap)時，於研究中發現類神經網路在訓練過程中，因為追求高正確率，僅學習編譯器(compiler)編譯過後之特定特徵，即使撰寫簡單的「Hello World!」程式，利用特殊工具進行編譯可能被分類為惡意程式，由於部分工具會預先載入駭客可能常用之動態函示庫與 API 函數，導致分類錯誤。未來仍需針對此類問題進行進一步研究，針對惡意程式資料前置處理是否有較好方式，或調整類神經網路模型避免此類問題發生。

參考文獻

1. R. Ronen, M. Radu, C. Feuerstein, E. Yom-Tov, and M. Ahmadi, "Microsoft malware classification challenge," *arXiv preprint arXiv:1802.10135*, 2018.
2. Chilheb Chebbi, *Mastering Machine Learning for Penetration Testing*, U.K.: Packt Publishing Ltd., 2018, pp. 87-109.
3. G. E. Dahl, J. W. Stokes, L. Deng, D. Yu, "Large-scale malware classification using random projections and neural networks", *IEEE International Conference on Acoustics Speech and Signal Processing*, 2013, pp. 3422-3426.
4. Websecurity.symantec.com. *What are malware, viruses, Spyware, and cookies, and what differentiates them ?* Retrieved Nov. 4, 2019, from <https://www.websecurity.symantec.com/security-topics/what-are-malware-viruses-spyware-and-cookies-and-what-differentiates-them>.
5. 2019 TWNIC/CC 電子報 6 月份，惡意程式。Retrieved Nov. 4, 2019, from <https://blog.twNIC.net.tw/2019/06/13/3927/>.
6. 劉培文，「網際攻擊狙殺鍊」，資安文章，台北，財團法人資訊工業策進會與行政院國家資通安全會報技術服務中心，2017，第 2 頁。
7. O'Reilly | Safari. *Practical Cyber Intelligence*. Retrieved Nov. 4, 2019, from <https://www.oreilly.com/library/view/practical-cyber-intelligence/9781788625562/37a5852b-ef31-4b1e-a184-93ea7cf5cd75.xhtml>.
8. iSecurity. 讓你"想哭" (WannaCry) 的勒索病毒。Retrieved Nov. 4, 2019, from <https://www.isecurity.com.tw/news-and-events/wannacry-make-you-cry/>.
9. BBC News. *Clock ticking on worm attack code*, Retrieved Nov. 4, 2019, from <http://news.bbc.co.uk/2/hi/technology/7832652.stm>.
10. 網易新聞中心. 熊猫烧香电脑病毒案告破_网易新闻中心. Retrieved Nov. 4, 2019, from <http://news.163.com/special/0001273M/panda070213.html>.
11. AFAGarap/malware-classification. Retrieved Nov. 4, 2019, from <https://github.com/AFAGarap/malware-classification>.
12. 任曉琿，「黑客免殺攻防」，北京：機械工業出版社，2013，第 80-107 頁。
13. 武傳海譯，「逆向工程核心原理」，北京：人民郵電出版社，2012，第 90-123 頁。
14. Docs.microsoft.com. *PE Format - Windows applications*. Retrieved Nov. 4, 2019, from <https://docs.microsoft.com/en-us/windows/desktop/Debug/pe-format>.
15. Hex-Rays. *IDA Pro*. Retrieved Nov. 4, 2019, from <https://www.hex-rays.com/products/ida/>.
16. OllyDbg. *OllyDbg v1.10*. Retrieved Nov. 4, 2019, from <http://www.ollydbg.de/>.
17. UPX. *UPX-the Ultimate Packer for eXecutables*. Retrieved Nov. 4, 2019, from

- <https://upx.github.io/>.
18. ASpack. *ASpack Software – Application for compression, packing and protection of softare*. Retrieved Nov. 4, 2019, from <http://www.aspack.com/>.
 19. 冀云著,「C++黑客編程揭密與防範(第二版)」,北京:人民郵電出版社,2015,第151-201頁。
 20. 邵堅磊著,「天書夜讀--從匯編語言到 Windows 內核編程」,北京:電子工業出版社,2008,第237-256頁。
 21. X-Ways. *WinHex: Computer Forensics & Data Recovery Software, Hex Editor & Disk Editor*. Retrieved Nov. 4, 2019, from <https://www.x-ways.net/winhex/>.
 22. Boost.org. The BOOST_PP_SEQ_ENUM macro enumerates the elements in a seq. Retrieved Nov. 4, 2019, from https://www.boost.org/doc/libs/1_70_0/libs/preprocessor/doc/ref/seq_enum.html.
 23. FireEye. Threat Research Using Precalculated String Hashes when Reverse Engineering Shellcode. Retrieved Nov. 4, 2019, from <https://www.fireeye.com/blog/threat-research/2012/11/precaculated-string-hashes-reverse-engineering-shellcode.html>.
 24. Docs.microsoft.com. `__cdecl`. Retrieved Nov. 4, 2019, from <https://docs.microsoft.com/zh-tw/cpp/cpp/cdecl?view=vs-2019&view=vs-2019>.
 25. Docs.microsoft.com. `__stdcall`. Retrieved Nov. 4, 2019, from <https://docs.microsoft.com/zh-tw/cpp/cpp/stdcall?view=vs-2019&viewFallbackFrom=vs-2019,vs-2019>.
 26. Docs.microsoft.com. `naked` (C++). Retrieved Nov. 4, 2019, from <https://docs.microsoft.com/zh-tw/cpp/cpp/naked-cpp?view=vs-2019>.
 27. Vas3k.com. *Machine Learning for Everyone*. Retrieved Nov. 4, 2019, from https://vas3k.com/blog/machine_learning/.
 28. Deeplearning.net. *Multilayer Perceptron — DeepLearning 0.1 documentation*. Retrieved Nov. 4, 2019, from <http://deeplearning.net/tutorial/mlp.html>.
 29. Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *The Journal of Machine Learning Research*, 2014, pp. 1929 - 1958.
 30. TikZ. *Dropout*. Retrieved Nov. 4, 2019, from <https://github.com/PetarV-/TikZ/tree/master/Dropout>.
 31. Sheldon M Ross, *Introduction to probability and statistics for engineers and scientists*, US: Academic Press, 2009, pp. 141
 32. freeCodeCamp.org News. *An intuitive guide to Convolutional Neural Networks*. Retrieved Nov. 4, 2019, from <https://www.freecodecamp.org/news/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050/>.
 33. Brandon Rohrer. *How do Convolutional Neural Networks work?* Retrieved Nov. 4, 2019, from http://brohrer.github.io/how_convolutional_neural_networks_work.html.
 34. 林大貴著,「TensorFlow+Keras 深度學錫人工智慧實務應用」,新北:博碩文化股

份有限公司，2017，第 201-227 頁。

35. Sepp Hochreiter and Jurgen Schmidhuber. "Long short-term memory," *Neural computation*, 1997, pp. 1735–1780.
36. David C. Plaut, Geoffrey E. Hinton, "Learning sets of filters using back-propagation," *Computer Speech and Language*, 1987, vol. 2, pp. 35-61.
37. William Hardy, Lingwei Chen, Shifu Hou, Yanfang Ye, and Xin Li, "DL4MD: A deep learning framework for intelligent malware detection," *In Proceedings of the International Conference on Data Mining (DMIN)*, 2016, pp. 61-67.
38. Lakshmanan Nataraj, Shanmugavadivel Karthikeyan, Grégoire Jacob, B. S. Manjunath, "Malware Images: Visualization and Automatic Classification," *Proceedings of the 8th International Symposium on Visualization for Cyber Security*, 2011, Article No. 4.
39. Shun-Wen Hsiao, "Using dynamic analysis data for malware family classification by convolution neural network," *Communications of the CCISA*, vol. 24, no. 1, 2018, pp. 41-60.
40. No Free Hunch. *Microsoft Malware Winners' Interview: 1st place, "NO to overfitting!"*. Retrieved Nov. 4, 2019, from <http://blog.kaggle.com/2015/05/26/microsoft-malware-winners-interview-1st-place-no-to-overfitting/>.
41. Daniel Gibert, *Convolutional Neural Networks for Malware Classification*, Master's Thesis, Department of Computer Science, Escola Politecnica de Catalunya, UPC, Barcelon, Spain, 2016.
42. Agarap, A. F., and Pepito, F. J. H., "Towards Building an Intelligent Anti-Malware System: A Deep Learning Approach using Support Vector Machine (SVM) for Malware Classification," arXiv preprint arXiv:1801.00318, Feb. 2019.
43. 簡書. 基于 DeepConvLSTM 的感測器信號分類. Retrieved Nov. 4, 2019, from <https://www.jianshu.com/p/ecbd6b4f54d2>.
44. Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. (2015). Retrieved Nov. 4, 2019, from <http://tensorflow.org/>.
45. J. D. Hunter. "Matplotlib: A 2D graphics environment," *Computing In Science & Engineering*, 2007, pp. 90-95.
46. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M.

- Brucher, M. Perrot, and E. Duchesnay. “Scikit-learn: Machine Learning in Python”, *Journal of Machine Learning Research* 12, 2011, pp. 2825-2830
47. Puget Systems. *The Best Way to Install TensorFlow with GPU Support on Windows 10 (Without Installing CUDA)*. Retrieved Nov. 4, 2019, from <https://www.pugetsystems.com/labs/hpc/The-Best-Way-to-Install-TensorFlow-with-GPU-Support-on-Windows-10-Without-Installing-CUDA-1187/>.
48. Diederik Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.
49. 吳嘉芳譯，「**Deep Learning 用 Python 進行深度學習的基礎理論實作**」，台北：基峰資訊股份有限公司，2017，第 158 頁。
50. National Center for High-performance Computing. *Malware Knowledge Base*. Retrieved Nov. 4, 2019, from <https://owl.nchc.org.tw/>.

