

## 캡스톤 - 인공지능기반악성코드분석

### BERT를 이용한 악성파일 분류

2015004084 컴퓨터소프트웨어학부 김태훈

2015004148 컴퓨터소프트웨어학부 서성빈

## 1. 프로젝트 개요

IT분야의 발달로 생활 곳곳에 다양한 응용프로그램들이 활용되고 있다. 개발되는 응용프로그램이 증가함에 따라 악성파일의 위협도 같이 늘어나고 있다. 이에 맞서 악성코드에 대항하기 위한 다양한 악성코드 분류 방법이 제시되고 있다. 본 프로젝트는 주어진 파일이 악성코드를 실행하는 악성파일인지 분류하기 위해 동적 API calls sequence 기반으로 분석하는 방법을 소개한다. cuckoo sandbox open source tool 을 이용해 API calls sequence를 추출하고, NLP(자연어처리)관점에서 추출된 API calls sequence의 특징을 분석한다. 특징을 분석하는 방법으로는 NLP 분야에서 높은 성능을 보이는 딥러닝 모델 BERT를 이용한다.

## 2. 프로젝트 내용

프로젝트 전체적인 과정은 다음과 같다. 1) 동적 API calls sequence 추출, 2) BERT 구현 (pre-training), 3) 학습모델 설정(fine-tuning), 4) 학습 및 결과 확인, 5) 데이터 재구성 및 결과 확인

### 1) 동적 API calls sequence 추출

cuckoo sandbox open source tool은 가상환경에서 파일을 실행하고 실행파일을 분석한다. 분석된 내용 중에 실행파일이 호출한 API calls 함수 목록이 report.json 파일에 포함되어 있고, 이 목록을 추출하여 악성파일 분석에 사용한다. 추출된 결과는 아래의 그림과 같다.

Viewer Text

- JSON
  - info
  - signatures
  - target
  - extracted
  - network
  - static
  - dropped
  - behavior
    - generic
    - apistats
  - processes
    - 0
    - 1
      - process\_path : "C:\Documents and Settings\Administrator\Local Settings\Temp\0b0241135c226c4cf1732ec12e2fe287.vir"
      - calls
        - 0
          - category : "system"
          - status : 1
          - stacktrace
            - api : "NtQuerySystemInformation"
            - return\_value : 0
          - arguments
            - time : 1622278253.266073
            - tid : 3752
          - tags
        - 1
          - category : "process"
          - status : 1
          - stacktrace
            - api : "NtAllocateVirtualMemory"
            - return\_value : 0
          - arguments
            - time : 1622278253.266073
            - tid : 3752
          - flags
        - 2
          - category : "process"
          - status : 1
          - stacktrace
            - api : "NtAllocateVirtualMemory"
            - return\_value : 0
          - arguments
            - time : 1622278253.266073
            - tid : 3752
    - 2



## 2) BERT 구현 , pre-training

BERT는 빅데이터를 바탕으로 먼저 학습되고, 이후에 사용자가 구축한 학습모델에서 참조하는 백과사전 역할을 한다. 따라서 일반적으로 사람이 사용하는 문장구조라면 방대한 데이터로 학습을 완료하여 공개된 BERT 모델을 사용할 경우 학습모델의 성능을 높일 수 있을 것이다.

하지만 본 프로젝트를 NLP에 대입하여 생각하면 API calls sequence를 문장으로, API calls 함수의 이름을 하나의 단어로 인식해야하기 때문에 사람이 사용하는 문장의 구조와 단어의 의미가 다른 특수한 상황이라고 할 수 있다. 따라서 기존에 공개된 BERT모델을 사용할 수 없고, 따로 학습의 참조가 되는 BERT모델을 구축하는 pre-training이 필요하다.

pre-training은 unsupervised 학습으로 이뤄지기 때문에 라벨링 없는 문장 형태의 데이터를 embedding한 값이 있으면 학습이 가능하다. embedding 전 데이터의 모습은 다음과 같다.

```

1  for i in range(5):
2  | print(train_data[i])

D  NtAllocateVirtualMemory NtFreeVirtualMemory NtAllocateVirtualMemory GetFileType GetFileType
GetSystemTimeAsFileTime SetUnhandledExceptionFilter SetErrorMode GetFileAttributesW GetFileAttributesA
GetSystemTimeAsFileTime LdrGetDIHandle LdrGetProcedureAddress LdrGetProcedureAddress LdrGet
SetUnhandledExceptionFilter NtAllocateVirtualMemory NtAllocateVirtualMemory NtClose LdrLoadD
GetSystemTimeAsFileTime LdrLoadDII LdrGetProcedureAddress LdrLoadDII LdrLoadDII LdrGetProced

```

BERT는 입력 sequence 길이를 최대 512단어까지만 인식한다. 하지만 본 프로젝트에서 사용되는 API calls sequence는 API calls함수의 개수가 100만이 넘어가는 경우도 있다. 따라서 pre-training을 위해서는 입력 데이터의 차원을 줄여줄 필요가 있다.

조사 결과 매우 긴 문장의 경우에도 앞쪽 512 단어만 있으면 성능이 보장된다고 한다. 이에따라 본 프로젝트에서도 API calls sequence의 앞쪽 512개의 API calls 함수를 입력 데이터로 사용한다.

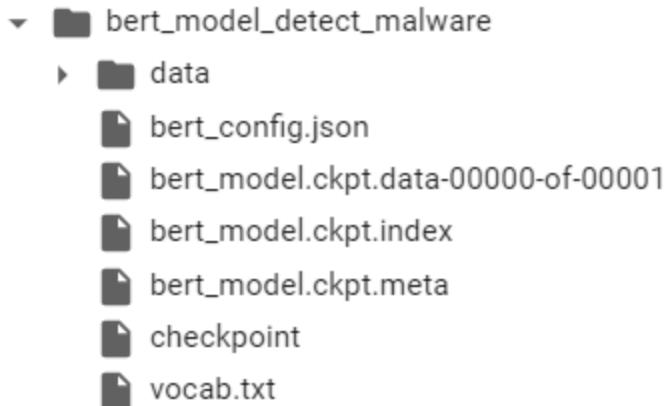
입력데이터가 준비되면 구글에서 제공하는 bert 모듈을 이용하여 pretraining 한다.

자세한 코드 및 과정은 아래의 링크에서 확인할 수 있다.

<https://colab.research.google.com/drive/141KY6Ba5DAFmEvOMgjVHmJY299Ixk0tc?usp=sharing>

학습이 완료되면 모델은 ckpt 형식으로 저장되며 step별로 저장된 모델 중 50000 step(약 10시간 소요)의 모델을 학습에 이용한다.

pre-training 모델은 아래와 같이 나타난다.



checkpoint를 제외한 5개의 파일이 모델을 대표한다 bert\_config.json은 환경설정을 나타내고, bert\_model 파일들은 BERT 모델을 나타낸다. 마지막으로 vocab.txt는 학습 시 데이터를 전처리할 때 참고하는 vocabulary로 모든 API calls 함수 목록과 sequence의 시작, 끝, mask, padding, unknown symbol을 포함하고 있다.

### 3) 학습모델 설정, fine-tuning

이전 과정에서 구축된 BERT를 적용하여 학습모델을 만드는 것을 fine-tuning이라고 한다. 본 프로젝트는 API calls sequence 를 분석하여 악성파일인지 정상파일인지 분류하는 binary classification을 하는 것이 목표이기 때문에 BERT에 sigmoid layer를 붙인 모델을 구성한다.

학습을 위해서는 악성파일인지 나타내기 위해 라벨링이 필요하다. 따라서 입력데이터의 형식은 아래의 그림과 같이 API calls sequence 배열과 label을 담는 데이터로 가공하여 사용한다. (label 1 : 악성파일, label 0 : 정상파일)

fine-tuning 의 BERT layer부분은 transformer 모델의 encoder에 해당하기 때문에 BERT를 사용하려면 transformer의 encoder 환경설정이 필요하다. BERT를 제작한 구글에서 제안하는 BERT\_base 환경설정은 Self-attention head layer를 12개 총 encoder layer를 12개 총이지만, 그대로 적용하게 되면 본 프로젝트에서 사용하는 학습환경 google colab이 제공하는 시간과 용량을 초과하기 때문에 적절히 타협하여 Self-attention head layer를 4개 총, encoder layer를 4개 총으로 설정했다.

학습에 이용하는 API calls sequence의 길이 또한 BERT의 최대입력 길이에 해당하는 512를 사용하면 학습환경의 가용자원을 초과하기 때문에 최대길이 64로 설정했다.

모델의 환경설정을 정리하면 다음과 같다.

Number of Self-attention head layer : 4

Number of encoder layer : 4

max sequence length : 64

자세한 fine-tuning 과정 및 코드는 아래의 링크에서 확인할 수 있다.

[https://colab.research.google.com/drive/1iDr7fbK8qxoQL92-Vd-c6K\\_XVpPf0HkI?usp=sharing](https://colab.research.google.com/drive/1iDr7fbK8qxoQL92-Vd-c6K_XVpPf0HkI?usp=sharing)

#### 4) 학습 및 결과

학습데이터는 train data : 5032 (정상파일 1277, 악성파일 3755), test data : 1678 (정상파일 439, 악성파일 1239) 로 train data와 test data를 3: 1비율로 무작위 추출했다.

bert모듈에서 제공하는 tokenizer를 이용하여 pre-training에서 만들어 둔 vocab.txt를 참고하여 데이터를 전처리하고 아래의 설정으로 학습을 실행한다.

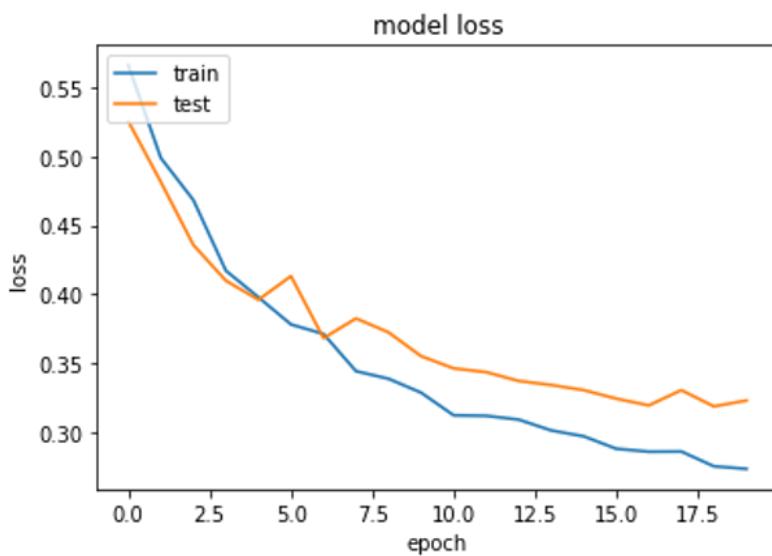
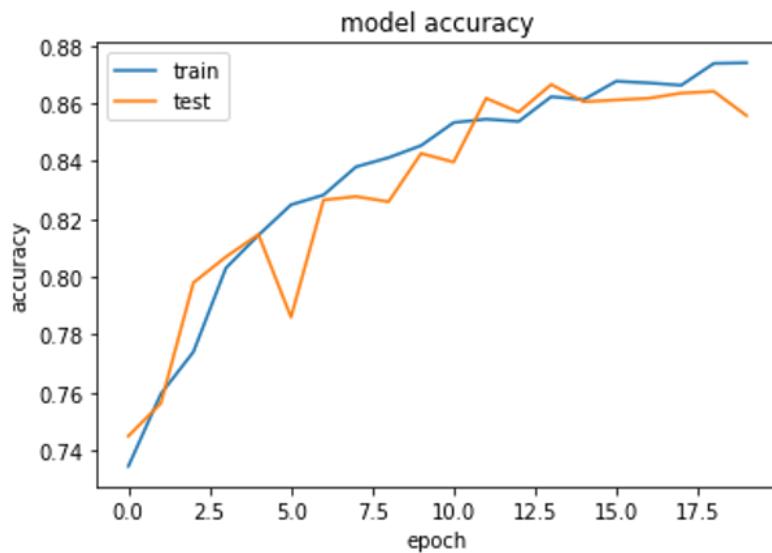
Epoch : 20

Batch size : 64

```

Epoch 10/20
79/79 [=====] - 44s 556ms/step - loss: 0.3286 - accuracy: 0.8454 - val_loss: 0.3551 - val_accuracy: 0.8427
Epoch 11/20
79/79 [=====] - 44s 556ms/step - loss: 0.3121 - accuracy: 0.8533 - val_loss: 0.3462 - val_accuracy: 0.8397
Epoch 12/20
79/79 [=====] - 44s 554ms/step - loss: 0.3118 - accuracy: 0.8545 - val_loss: 0.3435 - val_accuracy: 0.8617
Epoch 13/20
79/79 [=====] - 44s 555ms/step - loss: 0.3090 - accuracy: 0.8537 - val_loss: 0.3371 - val_accuracy: 0.8570
Epoch 14/20
79/79 [=====] - 44s 556ms/step - loss: 0.3012 - accuracy: 0.8623 - val_loss: 0.3341 - val_accuracy: 0.8665
Epoch 15/20
79/79 [=====] - 44s 556ms/step - loss: 0.2969 - accuracy: 0.8613 - val_loss: 0.3304 - val_accuracy: 0.8605
Epoch 16/20
79/79 [=====] - 44s 555ms/step - loss: 0.2879 - accuracy: 0.8676 - val_loss: 0.3242 - val_accuracy: 0.8611
Epoch 17/20
79/79 [=====] - 44s 555ms/step - loss: 0.2858 - accuracy: 0.8671 - val_loss: 0.3193 - val_accuracy: 0.8617
Epoch 18/20
79/79 [=====] - 44s 558ms/step - loss: 0.2859 - accuracy: 0.8663 - val_loss: 0.3304 - val_accuracy: 0.8635
Epoch 19/20
79/79 [=====] - 44s 556ms/step - loss: 0.2751 - accuracy: 0.8738 - val_loss: 0.3186 - val_accuracy: 0.8641
Epoch 20/20
79/79 [=====] - 44s 554ms/step - loss: 0.2733 - accuracy: 0.8740 - val_loss: 0.3229 - val_accuracy: 0.8558
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])

```



결과를 보면 정답률은 약 86%인 것을 확인할 수 있다. **loss** 그래프에서 **train**의 기울기 변화 크기보다 **test**의 기울기 변화 크기가 더 작은 것을 보면 5 epoch 이후부터 약간의 오버피팅이 생기는 것을 알 수 있다. 꾸준히 감소하는 경향을 유지하기 때문에 눈에 띠는 오버피팅은 아니라고 할 수 있겠으나 20epoch 이후에는 오버피팅이 점점 심해지는 경향이 나타날 것으로 보인다.

##### 5) 데이터 재구성 및 결과확인

위의 모델의 결과 그래프를 보면 **accuracy**가 꾸준히 증가하고, **loss**는 꾸준히 감소하기 때문에 눈에 띠는 오버피팅은 없다고 할 수 있다. 따라서 성능이 다소 아쉬운 이유가 모델 때문은 아니라고 생각했고, 아래와 같은 이유로 데이터를 재구성하면 성능을 향상시킬 수 있다고 생각했다.

첫 번째로, 사용되는 **sequence**의 길이가 너무 짧다. **max sequence length**를 64로 설정했다는 것은 API **calls sequence**의 앞쪽 64개만 보고 판단을 한다는 뜻이기 때문에 **sequence** 전체의 특징을 대표하기에는 부족하다.

두 번째로, 데이터의 양이 너무 적다. 훈련 데이터는 약 5032개로 딥러닝 학습을 하기에는 부족하다.

이와 같은 이유로 **max sequence length**와 데이터의 양을 증가시켜서 데이터를 재구성했다. **max sequence length**는 학습시간을 고려하여 적절한 값인 150으로 증가시켰고, 데이터의 양을 증가시키기 위한 방법으로는 다음과 같은 방법을 사용했다.

각 파일의 API **calls sequence**마다 무작위로 150개를 추출한다. 이때 150개는 **index**를 같이 저장하여 **index** 정렬을 통해 순서를 유지하도록 한다. 이런 방식으로 각 파일당 30번 추출하게 되면 데이터의 개수는 약 30배 증가한다.

**train data : 150975 (악성 112237, 정상파일 38738 )**

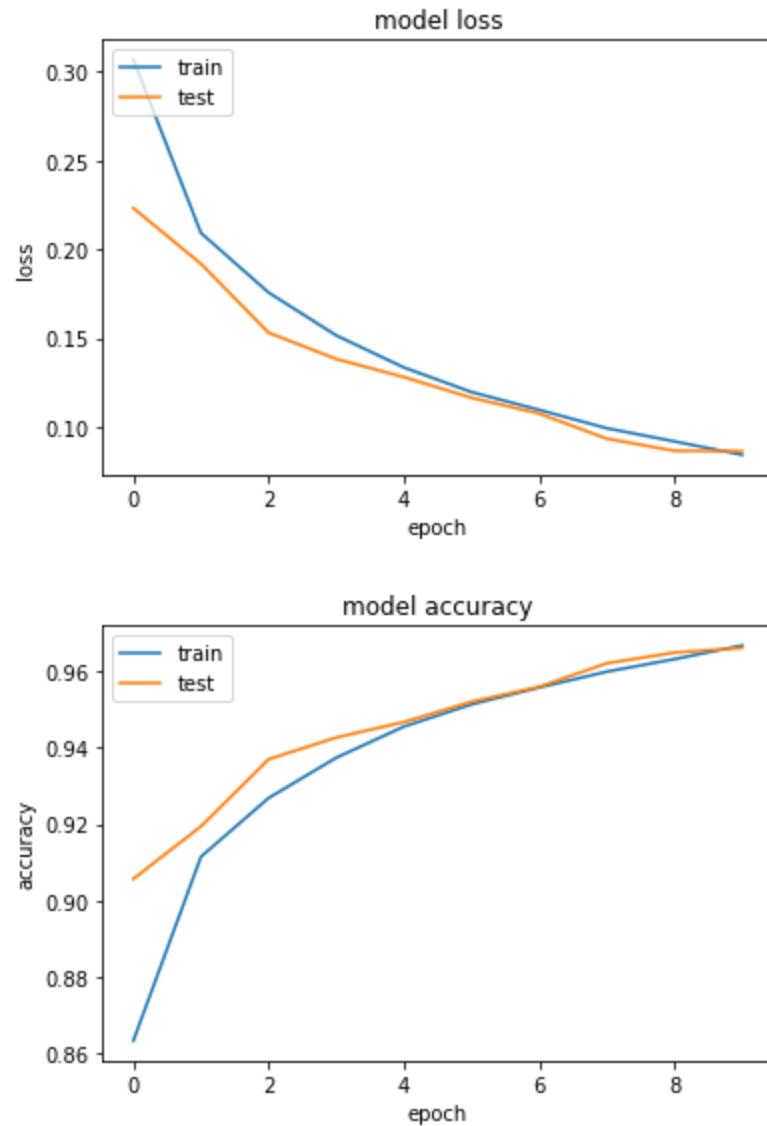
**test data : 50325 (악성파일 37583, 정상파일 12742)**

이렇게 순서를 유지하면서 무작위로 데이터를 추출하게 되면 API **calls sequence**의 전체적인 특징을 파악할 수 있다.

**pre-training**에서는 연속된 데이터 512개를 사용했기 때문에 연속된 데이터의 특징을 파악할 수 있다고 생각할 수 있으므로 결국 **pre-training**과 **fine-tuning**을 거치면서 API **calls sequence**의

연속적인 특징과 전체적인 특징을 모두 파악하게 되고 이에 따라 데이터를 다각도로 해석하게 되어  
높은 성능을 기대할 수 있다고 생각했다.

```
Epoch 1/10
2359/2359 [=====] - 1135s 478ms/step - loss: 0.3069 - accuracy: 0.8633 - val_loss: 0.2232 - val_accuracy: 0.9057
Epoch 2/10
2359/2359 [=====] - 1128s 478ms/step - loss: 0.2092 - accuracy: 0.9115 - val_loss: 0.1919 - val_accuracy: 0.9194
Epoch 3/10
2359/2359 [=====] - 1128s 478ms/step - loss: 0.1756 - accuracy: 0.9268 - val_loss: 0.1530 - val_accuracy: 0.9369
Epoch 4/10
2359/2359 [=====] - 1127s 478ms/step - loss: 0.1515 - accuracy: 0.9373 - val_loss: 0.1382 - val_accuracy: 0.9426
Epoch 5/10
2359/2359 [=====] - 1128s 478ms/step - loss: 0.1334 - accuracy: 0.9455 - val_loss: 0.1280 - val_accuracy: 0.9467
Epoch 6/10
2359/2359 [=====] - 1128s 478ms/step - loss: 0.1196 - accuracy: 0.9513 - val_loss: 0.1163 - val_accuracy: 0.9520
Epoch 7/10
2359/2359 [=====] - 1128s 478ms/step - loss: 0.1095 - accuracy: 0.9557 - val_loss: 0.1075 - val_accuracy: 0.9559
Epoch 8/10
2359/2359 [=====] - 1128s 478ms/step - loss: 0.0992 - accuracy: 0.9598 - val_loss: 0.0933 - val_accuracy: 0.9620
Epoch 9/10
2359/2359 [=====] - 1128s 478ms/step - loss: 0.0918 - accuracy: 0.9631 - val_loss: 0.0865 - val_accuracy: 0.9648
Epoch 10/10
2359/2359 [=====] - 1127s 478ms/step - loss: 0.0842 - accuracy: 0.9667 - val_loss: 0.0863 - val_accuracy: 0.9661
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```



결과를 보면 10 epochs에서 96.6% 정도의 정답률을 보이는 것을 확인할 수 있다. 데이터 재구성 전의 정답률 86%와 비교했을 때 12.3%정도의 성능이 향상되었다고 할 수 있고, 96.6%라는 수치만으로 봤을 때는 꽤 높은 정답률을 보인다고 할 수 있다.  
또한 감소하는 경향을 봤을 때 train과 거의 비슷한 수준으로 감소하는 것을 알 수 있으므로 데이터 재구성 전에 있었던 약간의 오버피팅도 해결됐다고 할 수 있다.

새로운 파일을 테스트하고 싶다면 **api calls sequence**를 추출하고, 순서를 유지하여 무작위 150개의 **API calls** 함수 추출을 N번 시행하여 N개의 결과 중 다수결에 따라 악성파일인지 결정하는 방식을 이용하면 높은 정답률을 보일 것으로 예상된다.

여기까지가 최종발표 때 발표한 학습 및 결과에 대한 내용입니다. 저희가 사용한 학습환경 **google colab**에서 제공하는 시간이 12시간이고, **ram**은 12GB를 제공합니다. 그리고 **CPU와 GPU**의 성능이 랜덤하게 할당되기 때문에 학습을 진행할때마다 시간적 한계가 있습니다. 하지만 발표가 끝나고 운이 좋게도 **google colab pro**를 사용하는 학우에게 구글계정을 빌릴 수 있었고 학습에 필요한 가용자원의 제한이 어느정도 풀리게 되었습니다. 그래서 저희는 **sequence**의 길이와 **epoch**을 증가시켜서 한 번 더 학습을 진행해보기로 했고 학습 데이터와 파라미터는 다음과 같습니다.  
(데이터 추출방식은 동일하고 길이만 늘렸습니다)

**train data : 150975 (악성 112237, 정상파일 38738 )**

**test data : 50325 (악성파일 37583, 정상파일 12742)**

**max sequence length : 300 [150에서 300으로 증가]**

**batch : 64**

**epoch : 20 [10에서 20으로 증가]**

결과는 아래와 같습니다.

Epoch 9/20

```
2359/2359  [=====] - 1270s 538ms/step - loss: 0.0584 -
accuracy: 0.9773 - val_loss: 0.0602 - val_accuracy: 0.9768
```

Epoch 10/20

```
2359/2359  [=====] - 1273s 539ms/step - loss: 0.0518 -
accuracy: 0.9797 - val_loss: 0.0550 - val_accuracy: 0.9788
```

Epoch 11/20

```
2359/2359  [=====] - 1273s 539ms/step - loss: 0.0480 -
accuracy: 0.9812 - val_loss: 0.0516 - val_accuracy: 0.9808
```

Epoch 12/20

```
2359/2359 [=====] - 1272s 539ms/step - loss: 0.0446 -  
accuracy: 0.9827 - val_loss: 0.0494 - val_accuracy: 0.9808
```

Epoch 13/20

```
2359/2359 [=====] - 1272s 539ms/step - loss: 0.0401 -  
accuracy: 0.9845 - val_loss: 0.0464 - val_accuracy: 0.9818
```

Epoch 14/20

```
2359/2359 [=====] - 1270s 539ms/step - loss: 0.0368 -  
accuracy: 0.9856 - val_loss: 0.0447 - val_accuracy: 0.9834
```

Epoch 15/20

```
2359/2359 [=====] - 1270s 538ms/step - loss: 0.0338 -  
accuracy: 0.9867 - val_loss: 0.0440 - val_accuracy: 0.9840
```

Epoch 16/20

```
2359/2359 [=====] - 1270s 538ms/step - loss: 0.0315 -  
accuracy: 0.9880 - val_loss: 0.0517 - val_accuracy: 0.9818
```

Epoch 17/20

```
2359/2359 [=====] - 1270s 538ms/step - loss: 0.0293 -  
accuracy: 0.9888 - val_loss: 0.0422 - val_accuracy: 0.9853
```

Epoch 18/20

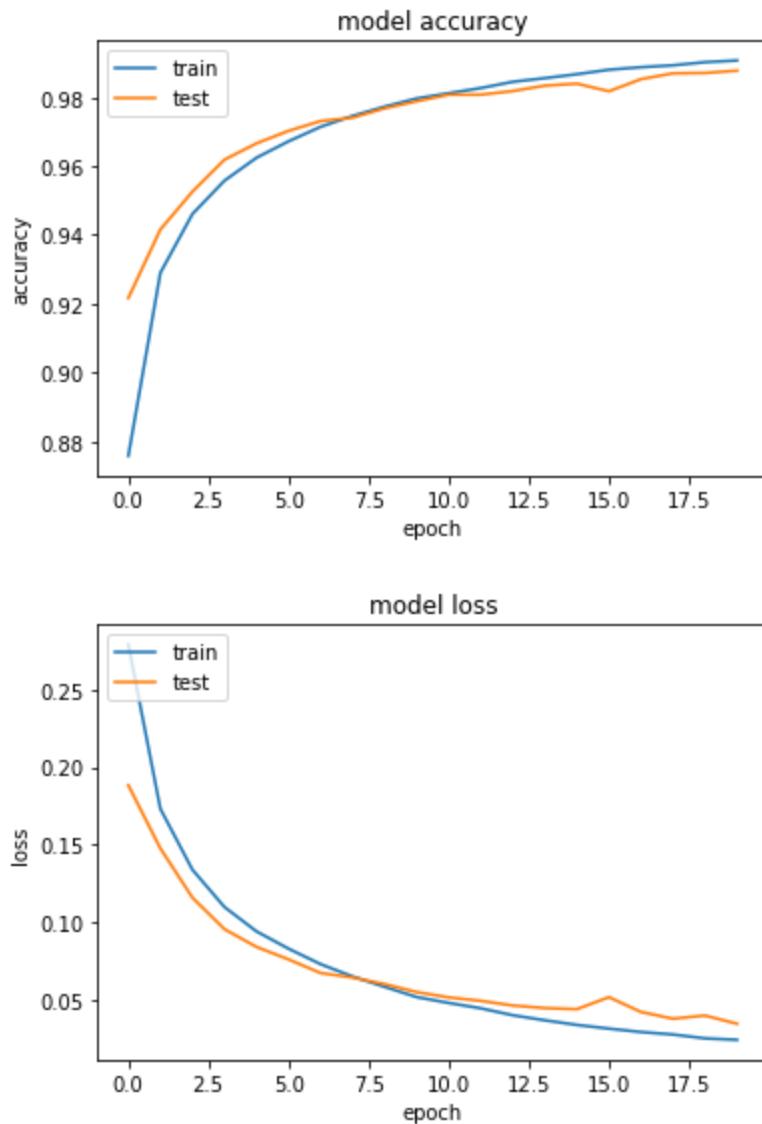
```
2359/2359 [=====] - 1270s 538ms/step - loss: 0.0277 -  
accuracy: 0.9893 - val_loss: 0.0378 - val_accuracy: 0.9870
```

Epoch 19/20

```
2359/2359 [=====] - 1270s 538ms/step - loss: 0.0252 -  
accuracy: 0.9902 - val_loss: 0.0399 - val_accuracy: 0.9871
```

Epoch 20/20

```
2359/2359 [=====] - 1270s 538ms/step - loss: 0.0242 -  
accuracy: 0.9907 - val_loss: 0.0346 - val_accuracy: 0.9877  
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```



98.7%정도의 정답률을 보이는 것으로 확인되고, 오버피팅이 적게 나타나는 것을 알 수 있습니다.

### 3. 프로젝트 결론

본 프로젝트에서는 학습에 필요한 데이터인 동적 API calls sequence를 추출하는 것부터 BERT를 구축하고 학습모델에 적용하여 의미있는 결과를 내는 것까지 BERT를 이용해 악성파일을 분류하는 과정을 바닥부터 진행했다.

BERT는 방대한 데이터를 바탕으로 할 때 높은 성능을 보이는 모델인데 적은 데이터로 pre-training을 시켰음에도 좋은 결과를 얻을 수 있었던 것을 보면 데이터의 악성파일 분류기준이 되는 패턴이 일반적인 문장의 분류 패턴보다 단순하다고 할 수 있다.

모델에서 사용되는 파라미터의 사이즈를 키우고, 데이터의 양을 늘리면서 결과가 더 좋아졌기 때문에 시간, 메모리측면에서의 가용자원이 충분하다면 더 좋은 결과를 얻을 수 있을 것으로 예상된다.

악성파일을 이미지화하여 CNN을 사용할 경우 99%정도의 정확도를 가진다는 것을 고려해볼 때 본 프로젝트의 최종 정답률 98.7%는 비슷한 성능을 낸다고 할 수 있으므로 악성파일 분류를 자연어처리 분야로 접근하여 attention을 활용한 transformer모델에 적용하는 것도 충분히 생각해볼만한 방법이라고 할 수 있다.