공학석사 학위논문

트리 탐색을 활용한 딥 러닝 비밀번호 생성 기법

Inference Method for Password Guessing Model with Tree Search

2021년 8월

서울대학교 대학원

컴퓨터공학부

임현재

트리 탐색을 활용한
딥 러닝 비밀번호 생성 기법

Inference Method for Password Guessing Model
with Tree Search

지도교수 이 재 진

이 논문을 공학석사 학위논문으로 제출함
2021년 7월

서울대학교 대학원
컴퓨터공학부
임현재

임현재의 석사 학위논문을 인준함
2021년 8월

위 원 장 김 진 수

부위원장 이 재 진

위 원 버나드 에거

초록

본 논문은 딥 러닝을 이용한 비밀번호 생성 기법의 추론 과정을 트리 탐색과 접목하여 비밀번호 재생성 문제를 해결하고 추론 성능을 크게 향상한다. 최근 관련 분야의 연구 동향이 딥 러닝 모델의 개선에 집중한 반면, 본 논문에서 제시한 추론 방법을 이용하면 모델의 개선 없이 추론 방법의 변경만으로 2~12배의 높은 효율을 달성할 수 있음을 보인다. 또한, 본 논문에서 제시한 추론 기법은 현존하는 다양한 딥 러닝 모델 구조 중 시계열 모델을 사용하는 모든 구조에 접목될 수 있다.

주요어 : 비밀번호 생성, 딥 러닝, 트리 탐색, 추론, LSTM
학 번 : 2019-27787

목 차

표 목 차

그 림 목 차

1장 서론

비밀번호를 이용한 사용자 인증은 현대에서 가장 많이 사용되는 방법이다. 비밀번호는 알파벳, 숫자, 특수문자로 이루어진 짧은 길이의 문자열으로, 길이가 10자만 되어도 10^{18}가지의 서로 다른 비밀번호가 존재할 수 있어 매우 강력한 사용자 인증 방법이다. 그러나 사용자들은 자신의 비밀번호를 의미가 담긴 영단어, 이름, 생일 등을 조합하여 설정하는 경우가 많고, 똑같은 비밀번호를 여러 사이트에서 재사용하는 경향이 있다. 이러한 경향은 사용자들이 비밀번호를 선택할 확률이 편향되게 만들고, 공격자로 하여금 비밀번호 추측을 쉽게 만들어 준다.

공격자들은 여러 가지 방법으로 사용자들의 비밀번호를 알아낼 수 있다. 그 중 한 가지 방법은 Offline guessing attack으로, 서버에서 탈취한 개인정보 데이터베이스를 이용하여 사용자들의 비밀번호를 알아낸다. 대부분의 서버에서는 비밀번호를 해쉬한 값을 데이터베이스에 저장하고, 해쉬된 비밀번호로부터 평문 비밀번호를 알아내는 방법은 비밀번호를 추측하고 이를 해쉬하여 그것과 비교하는 방법이 유일하다. 따라서 공격자들은 사용자들이 사용할 법한 비밀번호들을 생성하고, 최대한 적은 횟수로 최대한 많은 비밀번호를 추측하는 것을 목표로 한다. 이때 생성한 비밀번호 개수 대비 맞춘 비밀번호 개수를 hit rate라 한다.

대중적으로 많이 쓰이는 비밀번호를 추측하는 프로그램은 Hashcat[1]과 John the Ripper[2]가 있다. 이들은 Brute-force attack 뿐만 아니라 Word list(흔히 과거에 유출된 비밀번호를 사용한다)를 사용하는 Combinator attack, Rule-based attack 등을 지원한다.

최근 비밀번호 추측에 머신러닝을 접목하여 hit rate를 높이고자 하는 시도가 이루어지고 있다. Weir et al.[3]은 probabilistic context-free grammars(PCFG)를, Narayanan et al.[4]은 Markov model을 이용하여 비밀번호를 확률 모델로 표현하고자 하였다. 또한, Melicher et al.[5]을 시작으로 Neural Network를 이용하여 비밀번호를 딥 러닝으로 생성하려는 연구가 점차 늘고 있다. Meilcher et al.[5]은 Long-Short Term

Memory(LSTM)을 이용하였고, Hitaj et al.[6]은 Generative Adversarial Network(GAN)을 이용하여 비밀번호를 생성하는 모델을 제시하였다. 더 나아가서 Biesner et al.[7]은 Transformer[8] 및 Autoencoder[9] 모델을 사용하여 비밀번호를 생성하는 모델을 제시하였고, 이들의 추론 성능을 비교하였다.

LSTM, Transformer 등을 이용한 비밀번호 생성 모델은 비밀번호를 자연어 처리(NLP)에서 쓰이는 언어 모델(Language Model)로 모델링 한다는 점에 있어 공통점이 있다. 비밀번호는 token들의 sequence로 표현되고, 언어 모델은 token의 sequence가 주어질 때 다음으로 올 수 있는 token들의 확률을 예측한다. 즉, 언어 모델은 token sequence $x_{1:t} = [x_1, x_2, x_3, ..., x_t]$가 입력으로 주어졌을 때, 다음으로 올 수 있는 x_{t+1}의 확률 $P(x_{t+1}|x_1, x_2, ...x_t)$을 예측한다. 입력 $x_{1:t}$의 경우의 수는 무한히 많기 때문에, $P(x_{t+1}|x_1, x_2, ...x_t)$의 정확한 값은 계산이 불가능하므로 딥 러닝 모델이 이를 근사하는데 사용된다. 사용자는 모델이 근사한 다음 token들의 확률을 이용하여 비밀번호를 생성할 수 있다. 구체적으로, 사용자는 첫 토큰 x_1 =〈sos〉을 모델에 입력하면 모델은 다음 토큰의 확률 $P(x_2|x_1)$을 계산하고, 이를 토대로 다음 토큰 x_2를 sampling한다. 이제 사용자는 $x_{1:2}$를 가지고 있고, 이를 다시 모델에 입력하여 $P(x_3|x_1, x_2)$를 계산한다. 이는 〈eos〉토큰이 샘플링 될 때까지 지속된다.

그러나 위와 같은 샘플링을 통해 비밀번호를 생성할 경우, 이전에 이미 생성하였던 비밀번호를 재생성한다는 문제점이 있다. 비밀번호 추측 분야에서 발표되는 최신 연구들은 10^9개 가량의 비밀번호를 생성하는데, 생성된 10^9개의 비밀번호 중 중복된 비밀번호를 제거하면 개수는 50% 이하로 크게 떨어진다.[6,7] 이에 본 논문에서는 딥 러닝 모델을 이용한 비밀번호 생성에 나타나는 샘플링에 나타나는 비밀번호 재생성 문제를 해결하는 알고리즘을 제시한다. 비밀번호 생성 분야의 최신 연구 경향들이 딥 러닝 모델을 개선하는 방향으로 이루어지고 있으나, 본 논문에서는 동일한 모델을 사용하여도 샘플링 방법에 따라 추론 성능이 증가할 수 있음을 보인다.

본 논문이 기여하는 바는 다음과 같다.

- 비밀번호의 언어 모델을 근사하는 딥 러닝 모델이 주어졌을 때, 기존의 샘플링 방법보다 효율적인 추론 알고리즘을 제시한다.

이 논문의 나머지는 다음과 같이 구성되어 있다. 2장에서는 본 논문에서 다루는 주제와 관련된 주제를 다룬 최근 연구들에 대해 설명한다. 3장은 자연어 처리를 이용한 비밀번호 생성 딥 러닝 모델에 대해 설명하고, 본 논문에서 사용되는 모델을 정의한다. 4장은 트리 탐색을 적용한 추론 기법을 제시한다. 5장은 기존의 샘플링 방법과 제시된 추론 기법을 비교하는 실험과 분석을 진행한다. 6장은 본 논문을 결론짓는다.

제 2장 관련 연구

비밀번호를 효과적으로 생성하기 위한 연구는 딥 러닝이 유행하기 이전부터 존재하였다. 그 중 가장 유명한 프로그램은 Hashcat[1]과 John the Ripper[2]일 것이다. 이들은 유출된 비밀번호 데이터셋으로부터 사용자의 Heuristic이 반영된 Rule을 사용하여 새로운 비밀번호를 생성한다.

반면 사용자의 Heuristic이 필요 없이, 오로지 데이터셋으로부터 비밀번호 패턴을 학습하여 새로운 비밀번호를 학습하는 딥 러닝 모델들이 2016년 이후로 등장하기 시작하였다. Melicher et al.[5]이 비밀번호 생성을 위해 LSTM[16] 모델을 이용한 이후로 Hitaj et al.[6]은 GAN[22]모델을 사용하였고, Li et al.은 BERT[8]로부터 Knowledge Distillation을 적용한 Bidirectional LSTM 모델을 사용하였다. 또한, Biesner et al.[7]은 GAN[22], Autoencoder[9], GPT2[10] 등의 모델을 사용하고, 이들을 비교하였다.

제 3장 자연어 처리를 이용한 비밀번호 생성 모델

3.1 학습을 위한 비밀번호 데이터셋

딥 러닝을 이용하여 비밀번호 생성 모델을 학습시키기 위해서는 다양한 비밀번호들이 필요하다. 이를 위해 과거 공격자들에 의해 유출된 비밀번호 데이터셋들을 이용하였다. 적합한 모델 구조를 찾기 위한 데이터셋으로써 본 논문에서는 다음 4가지 비밀번호 데이터셋에서 각각 25만 개 씩 무작위로 추출한 100만개의 비밀번호를 사용하였다.(이후 MixedMillion 데이터셋이라 표기)

- ‘Rockyou’[14](1434만 개), Rockyou 웹사이트에서 유출된 평문 비밀번호 데이터셋
- ‘pwned’(3.2억 개), 여러 가지 웹사이트에서 유출된 암호화된 비밀번호들을 모은 데이터셋. 대부분의 비밀번호는 해독되었음.
- ‘Linkedin’[13](6057만 개), Linkedin 웹사이트에서 유출된 암호화된 비밀번호 데이터셋. 대부분의 비밀번호는 해독되었음.
- ‘Yahoo’[15](538만 개), Yahoo 웹사이트에서 유출된 암호화된 비밀번호 데이터셋. 대부분의 비밀번호는 해독되었음.

추론 기법을 평가하기 위한 딥 러닝 모델은 Rockyou[14] 데이터셋 만을 학습에 사용하였다.

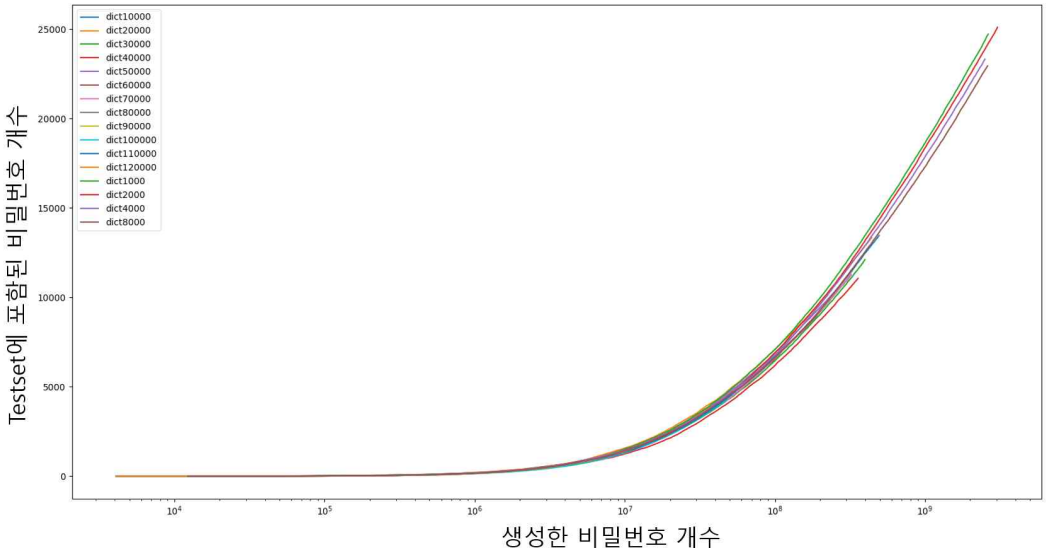
3.2 토큰화(Tokenization)

자연어 처리에서 토큰화는 텍스트 데이터를 토큰들로 분절하는 작업을 말한다. 이 때 분절된 토큰의 종류에 따라 Character tokenizer, Subword tokenizer, Word tokenizer 등이 존재한다.

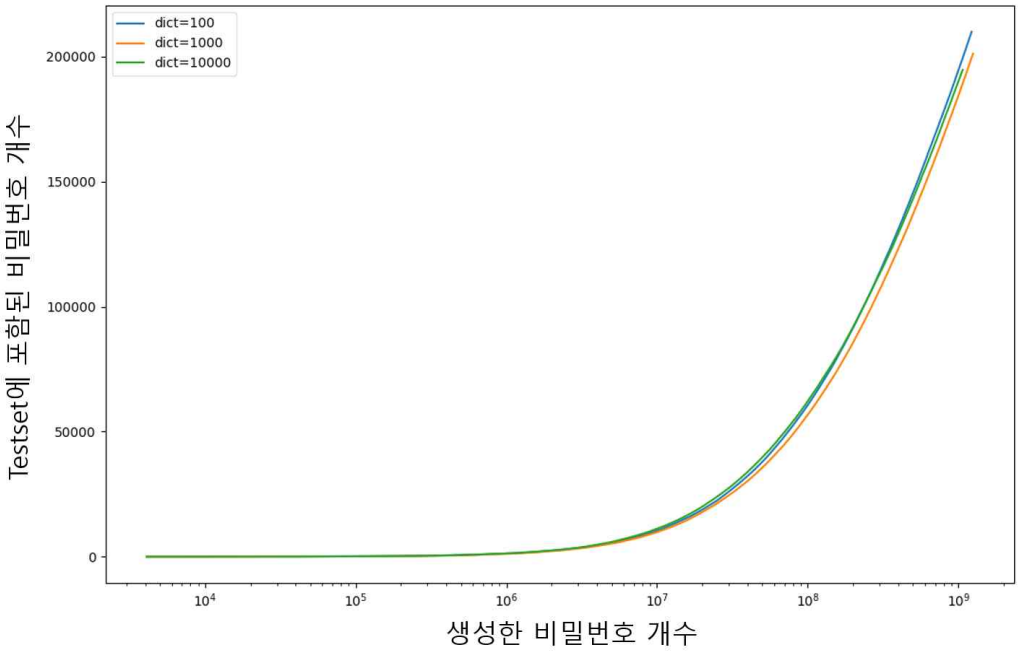
최근 자연어처리에서 인기있는 모델인 BERT[8]는 WordPiece tokenizer를 사용하고, GPT2[10]는 Byte-Pair Encoding(BPE)[11]를 사용한다. WordPiece와 BPE 모두 subword tokenizer의 일종으로, subword tokenizer은 문장을 단어(word)가 아닌 형태소(subword)로 분절함으로써 word tokenizer에서 등장하는 Out of Vocabulary 문제를 해결할 수 있다. 또한, word들의 집합인 dictionary를 사용자가 준비할 필요 없이, corpus에서 직접 dictionary를 제작할 수 있다는 장점이 있다. 본 논문에서는 토큰화로 BPE를 사용하였다.

GPT2[10] 논문에 의하면, BPE를 학습시키기 전에 먼저 Raw text를 알파벳, 숫자, 특수문자, 어퍼스트로피가 포함된 형태소(〈’re〉, 〈’ll〉, 〈’am〉) 등으로 먼저 분절하고, 그 후에 BPE를 학습하였다고 한다. 그 이유는 자주 등장하는 단어(e.g. dog) 뒤에 문장기호가 붙어 있는 경우,(e.g. dog., dog!, dog?) BPE는 문장기호까지 포함하여 하나의 형태소로 판단할 경우가 있기 때문이다. 하지만 실험 결과 비밀번호 생성 모델에서는 BPE를 적용하기 전에 미리 Raw text를 분절하는 것이 추론 성능 증가에 전혀 영향이 없었다. 본 논문에서는 Raw text를 미리 분절하지 않고 BPE를 적용하였다.

BPE의 유일한 Hyperparameter는 BPE로 만들 Dictionary 크기 N이다. 흔히 자연어처리 분야에서 N은 32,000~64,000사이 값을 사용하는 것이 일반적이고, GPT2에서는 $N=50,257$의 값을 사용하고 있다.[10] 그러나 N의 크기에 따른 Hyperparameter tuning을 진행해본 결과, N의 크기도 추론 성능에 큰 영향을 미치지 않았다고 판단하였다. [그림 1]과 [그림 2]는 각각 MixedMillion 및 Rockyou 데이터셋을 사용하여 학습된 모델의 N에 따른 추론 성능의 변화를 나타낸다. 서로 다른 N에 대해서 학습을 진행하여도 추론 성능의 변화가 거의 없음을 알 수 있다. N이 커지면 커질수록 모델



[그림 2] MixedMillion dataset에서 Dictionary size에 따른 추론 성능 변화.



[그림 3] Rockyou dataset에서 Dictionary size에 따른 추론 성능 변화.

크기와 계산량이 증가하므로, 실험에 사용한 N들 중 제일 작은 크기인 $N=100$을 사용하였다. 즉, 기본적으로 dictionary에 포함되어야 하는 97개

의 기본 토큰(알파벳, 숫자, 특수문자, 〈sos〉, 〈pad〉)을 제외한 단 3개의 subword만 dictionary에 포함하였다.

또한, BPE를 통한 subword 토큰화가 끝난 후, subword sequence의 맨 앞에 〈sos〉 토큰을 추가하였고, 학습의 배치(batch)화를 위해 모든 비밀번호의 길이를 16으로 정규화할 필요가 있어서 길이 16미만 subword sequence의 뒤에 〈pad〉 토큰을 추가하였다.

즉, 예를 들어 “password123!”이라는 비밀번호를 BPE를 사용하여 subword 토큰화를 진행하면 다음과 같은 subword sequence가 출력된다.

$x_{1:16}$ =[〈sos〉,〈p〉,〈a〉,〈s〉,〈s〉,〈w〉,〈o〉,〈r〉,〈d〉,〈12〉,〈3〉,〈!〉,〈pad〉,〈pad〉,〈pad〉]

3.3 워드 임베딩(Word Embedding)

임베딩(Embedding)은 토큰을 딥 러닝 모델이 이해할 수 있도록 벡터로 매핑해주는 일을 말한다. 매핑한 벡터의 종류에 따라 정수 인코딩, 원-핫 인코딩(One-Hot Encoding), 워드 임베딩(Word Embedding)등의 방법으로 분류할 수 있다. 워드 임베딩은 토큰을 밀집 벡터(dense vector)로 표현함으로써 토큰들의 관계에 대한 정보를 표현할 수 있다는 점에서 다른 임베딩에 비해 월등하다. 자연어처리 분야에서는 Pre-trained Word embedding 모델들(e.g. GloVe, word2vec 등)이 존재하나, 이는 비밀번호 생성을 위한 언어 모델에 사용하기에는 부적합하다고 판단하여, 본 논문에서는 모델과 함께 Embedding table도 무작위로 초기화하여 모델과 함께 학습하였다.

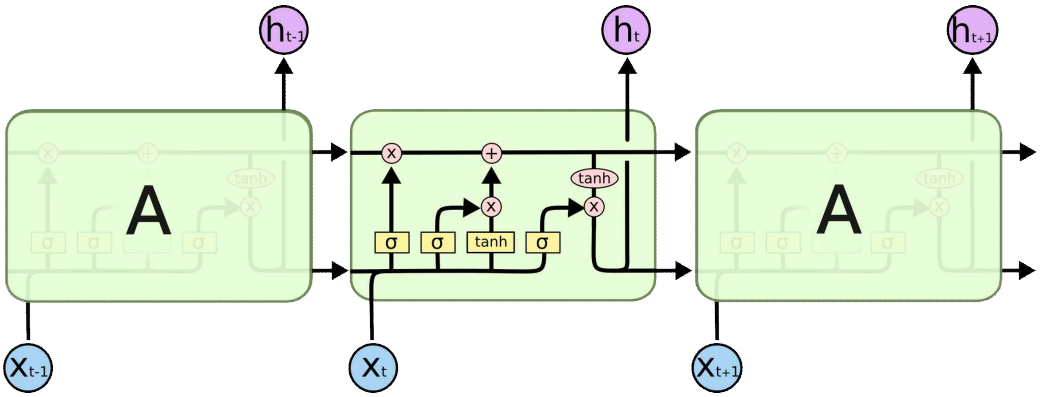
또한, GPT2[10]의 경우 토큰을 벡터로 매핑할 때, 토큰에 대응되는 벡터에 Positional encoding을 추가로 더한다. Positional encoding이란 토큰이 텍스트의 몇 번째에 위치하는지 표현하는 벡터이다. 본 논문에서는 마찬가지로 무작위로 초기화하여 모델과 함께 학습하였다.

3.4 모델 구조

3.4.1 LSTM

Long Short-Term Memory(LSTM)[16]은 Recurrent Neural Network의 일종이다. RNN은 번역, 문자, 음성 인식 등의 sequential data를 학습하기 위해 만들어진 신경망 모델인데, Vanilla RNN의 경우 데이터의 길이가 길수록 Backpropagation 중 Gradient가 소실되어 최적의 모델을 찾기 어렵다는 문제점이 있다. LSTM은 이를 개선하기 위해 등장한 구조로, RNN 구조의 hidden state에 cell state를 추가하여 장기 기억을 가질 수 있다.

LSTM의 구조는 다음과 같다.[17]



[그림 4] LSTM의 구조

LSTM은 시간 t에서 입력 x_t를 받고, 직전 시간 $t-1$에서 넘겨받은 hidden state h_{t-1}과 cell state c_{t-1}를 통해 h_t와 c_t를 계산한다. 이 때 h_t가 LSTM의 출력이 된다. h_{t-1}, c_{t-1}, x_t를 이용하여 h_t와 c_t를 계산하는 수식은 다음과 같다.

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$$

$$i_t = \sigma(W_i[h_{t-1,} x_t] + b_i)$$

$$\widetilde{C_t} = \tanh(W_C[h_{t-1}, x_t] + b_C)$$

$$C_t = f_t * C_{t-1} + i_t * \widetilde{C_t}$$

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * \tanh(C_t)$$

이 때 $[a,b]$는 a와 b의 concatenate, $\sigma()$는 sigmoid, $*$는 벡터의 내적을 의미한다. 또한, W와 b는 모델의 가중치이다.

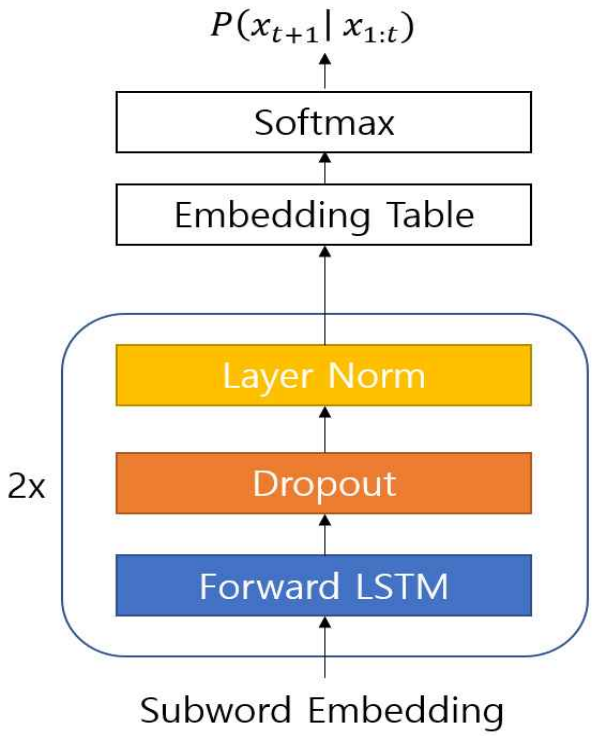
3.4.2 Layer Normalization

뉴럴 네트워크 구조에서 네트워크가 깊어질수록 각 레이어의 입력 분포가 이전 레이어의 파라미터에 의존하기 때문에 학습이 어려워진다. loffe et al.[18]은 이를 Internal covariate shift 문제라 명명하여, Batch normalization을 통해 이 문제를 완화시켰다. 그러나 Batch normalization의 효과는 mini batch 크기에 따라 달라지며, RNN 네트워크에 적용하기는 어렵다. Ba et al.[19]은 mini batch 대신 각 레이어의 입력을 정규화하여 RNN에도 normalization을 적용하였다. 본 논문에서 사용한 모델에도 마찬가지로 RNN의 레이어 사이에 Layer normalization을 적용하였다. 이를 통해 학습을 안정화, 가속화하는 효과를 꾀할 수 있다.

3.4.3 Dropout

뉴럴 네트워크의 파라미터가 많아질수록 학습 데이터셋에 모델이 overfit 될 수 있는 확률이 커진다. Srivastava et al.[20]은 모델이 학습하는 도중 확률적으로 네트워크의 일부분만 동작하고, 일부분은 동작하지 않게 하는 Dropout 기법을 통해 overfit 문제를 해결하였다. 본 논문에서도 모델에 Dropout을 적용함으로써 학습의 품질을 향상하는 효과를 기대하였다.

3.4.4 모델의 구조



[그림 5] 비밀번호 생성을 위한 딥 러닝 모델 구조

3.4.1~3절의 기술들을 종합하여 본 논문에서 사용한 모델의 구조는 [그림 4]와 같다. 모델은 2개의 LSTM 레이어로 이루어져 있고, t번째 토큰 x_t와 이전 시간의 hidden state h_t를 입력받아서 다음 시간의 hidden state h_{t+1}을 출력한다. h_{t+1}으로부터 토큰 x_{t+1}의 확률의 추정은 다음 식을 따른다.

$$h_{t+1} = \mathrm{LSTM}(\boldsymbol{e}_{x_t}, h_t)$$

$$\boldsymbol{g}_{t+1} = \mathrm{softmax}(E\, h_{t+1})$$

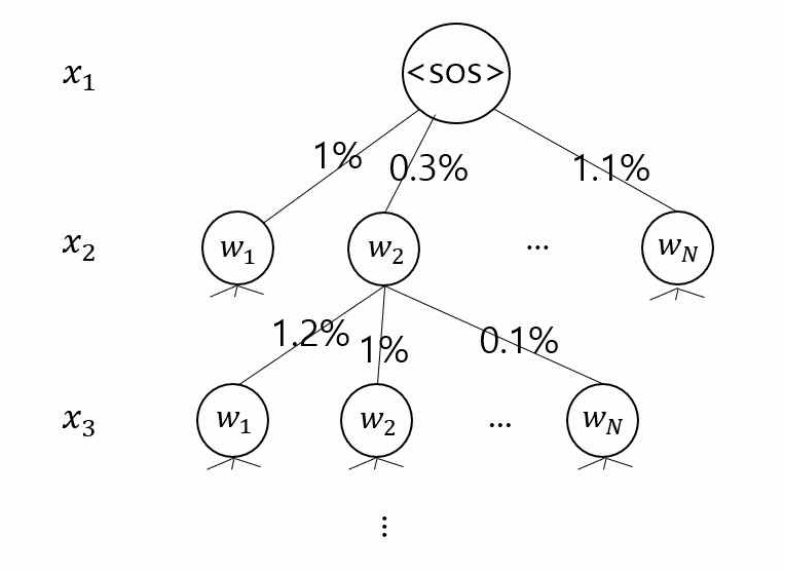
$$P(x_{t+1}) \sim \boldsymbol{g}_{t+1}$$

이 때 $\boldsymbol{e}_{x_t}$는 x_t에 대응되는 Embedding vector를 의미하고, LSTM은 Dropout과 Layer normalization이 적용된 2 레이어 LSTM, E는 Embedding table, $\boldsymbol{g}_{t+1}$은 $t+1$번째 토큰 x_{t+1}의 확률 분포를 나타낸다.

제 4장 트리 탐색을 적용한 비밀번호 추론

3장에서 설명한 비밀번호 생성을 위한 딥 러닝 모델은 $P(x_{t+1}|x_{1:t})$를 근사하기 위해 사용된다. 4장에서는 $P(x_{t+1}|x_{1:t})$ 확률 분포대로 x_{t+1}을 샘플링하여 비밀번호를 생성하는 대신, 트리 탐색 기법을 사용하여 비밀번호 재생성 문제를 해결한다.

4.1 Search space



[그림 5] 트리 형태로 표현한 비밀번호 탐색 공간

비밀번호 $x_{1:16}$은 길이 16의 subword token x_t들의 sequence로 나타낼 수 있다. 이 때 각 subword token는 BPE Dictionary Size N종류가 있고, 시작 토큰 x_1은 〈sos〉로 고정되어 있다. 즉, 비밀번호 $x_{1:16}$를 생성하는 것은 Branch factor=N, Maximum depth≦16으로 이루어진 트리에서 노드를 탐색하는 것과 동일하다. 이 때 각 edge는 딥 러닝 모델이 근사한 확률 $P(x_{t+1}|x_{1:t})$가 할당된다. [그림 5]는 이를 나타낸다.

모델에서 사용한 Tokenization은 비밀번호의 길이가 16보다 짧을 경우, 〈pad〉토큰을 뒤에 추가함으로써 길이를 16으로 고정한다. 즉, 트리에서 〈pad〉토큰은 Terminal node(Leaf node)와 동일한 역할을 하며, 〈pad〉 토큰이 등장한 후의 토큰들은 의미가 없어진다. 즉, 어떠한 비밀번호 $x_{1:16}$의 확률은 다음과 같다.

$$P(x_{1:16}) = P(x_{1:l}) = P(x_1)P(x_2|x_1)...P(x_l|x_{1:l-1})$$

이 때 l은 〈pad〉 토큰이 처음으로 등장한 index이다.

4.2 탐색 기법

공격자는 한정된 계산 자원과 시간으로 최대한 많은 비밀번호를 알아내려고 할 것이다. 최근 발표된 딥 러닝을 이용한 비밀번호 생성 관련 연구[6,7]는 최대 10^9개의 비밀번호를 생성한 후, 이들 중 test set에서 몇 개가 포함되었는지를 평가함으로써 모델의 성능을 평가한다.

공격자가 한정된 수의 비밀번호를 생성하여 비밀번호를 최대한 많이 맞추기 위해서는 모델링된 패스워드 확률 $P(x_{1:16})$이 제일 높은 순서대로 비밀번호 생성을 시도하는 것이 가장 유리하다. 이와 비슷한 접근이 자연어 처리 분야의 Beam search이다. Beam search는 GPT2 등의 문장 생성 모델이 있을 때, 가장 높은 확률의 문장을 검색하는 알고리즘이다. 이는 각 단계에서 탐색의 영역을 k개의 가장 확률이 높은 노드들로 유지하며 다음 단계를 탐색한다. k가 클수록 더 넓은 영역을 탐색하기 때문에 더 높은 확률을 가진 문장을 생성할 수 있지만, 속도가 느려지고 공간복잡도가 커진다. 하나의 문장을 만들기 위해 일반적으로 $k=5$에서 10 사이의 수를 선택한다.

그러나 비밀번호 생성을 위해 Beam search를 적용할 경우, 10^9개의 비밀번호를 생성해야 하므로 k를 이보다 큰 숫자로 적용해야 한다. 하지만 각 노드는 LSTM의 hidden state를 저장하고 있어야 하므로, 최소한 1KB의 저장공간이 필요하여 이를 모두 저장하는 데만 최소 1TB의 메모리가 필요하고, 정렬에도 큰 어려움이 있을 것이다.

많은 트리 탐색 기법들 중 가장 메모리를 적게 사용하는 탐색은 Depth First Search(DFS)이다. DFS는 재귀적으로 호출되며, 현재 노드에서 자신의 자식 노드 중 하나에 방문함으로써 탐색이 진행된다. 그러나 DFS를 그대로 비밀번호 생성에 적용하면, 탐색 도중 Terminal 노드를 방문하여 비밀번호 1개를 생성한다고 하더라도 이 비밀번호가 상위 10^9개의 비밀번호에 속하는지 알 수 없다.

길이 16 이하의 모든 가능한 비밀번호(약 98^{15}가지)를 모델링한 확률에 따라 내림차순으로 정렬한다고 가정하자. 공격자는 이들 중 상위 10^9개를 생성하는 것이 최상의 공격일 것이다. 만약 상위 10^9번째 패스워드의 확률을 정확하게 알 수만 있다면, DFS를 변형한 알고리즘 [Alg.1]를 이용하면 정확히 10^9개의 비밀번호를 생성할 수 있다. 본 논문에서는 이 확률을 Threshold Probability P_{th}라 정의하였다.

Algorithm 1 DFS with P_{th}

```
procedure DFS(P, s_t, x_{1:t})
    if x_t == ⟨pad⟩ then
        Add x_{1:t} into Set of Generated Passwords
        return
    end if
    if t ≥ 16 then
        return
    end if
    P(x_{t+1}), s_{t+1} ← model(s_t, x_t)
    for x_{t+1} in set of all tokens do
        if x_{t+1} != ⟨sos⟩ and P_th ≤ P × P(x_{t+1}) then
            DFS(P × P(x_{t+1}), s_{t+1}, x_{1:t+1})
        end if
    end for
end procedure
```

이 때 P는 현재 노드 $x_{1:t}$의 확률 $P(x_{1:t})$를 의미한다.

Threshold Probability P_{th}는 비밀번호 생성 모델에 따라 달라지며, 참값은 실험적으로만 찾을 수 있다. 다만 P_{th}에 따라 찾을 수 있는 비밀번호의 개수는 $1/P_{th}$ 이하임을 귀류법으로 증명 가능하다.

4.3 병렬화 기법

대부분의 Neural Network 모델들은 CPU가 아닌 GPU에서 연산하는 것이 속도 측면에서 훨씬 유리한데, 그 이유는 Neural Network는 본질적으로 행렬 곱셈(GEMM)연산으로 나타낼 수 있고, 이는 CPU보다 GPU가 연산하는 것이 훨씬 빠르기 때문이다. GPU가 CPU보다 빠르게 GEMM 연산을 할 수 있는 이유는 GEMM이 병렬화가 수월하여, 수 천개의 core에 연산을 균등하게 분배할 수 있기 때문이다.

본 논문에서 사용하는 LSTM 모델의 경우, GEMM 연산에 사용되는 행렬의 크기는 Batch Size, Hidden size 등에 의해 결정된다. 즉, Batch size가 작으면 병렬화가 불가능하여 GPU에서 연산을 할지라도 속도 향상을 기대할 수 없게 된다. [Alg.1]처럼 Recursive하게 구현된 DFS는 Batch size가 1이 될 수밖에 없어서, 기존의 추론 방법 대비 알고리즘 수행 시간이 100배 이상 느려진다. 이러한 속도 차이는 추론 방법 개선에 의해 동일 생성 개수 대비 Hit rate를 증가시키더라도 이를 무의미하게 만든다.

DFS를 병렬화하기 위해서는 하나의 Search tree를 여러 개의 instance가 탐색하여야 한다. 이 때 instance들의 탐색 범위들이 서로 겹치지 않게 하기 위해서 각 instance들의 시작 node를 Root node가 아닌, 같은 Level에 있는 서로 다른 자식 노드들을 시작 node로 한다. [Alg.2.1]은 이를 위해 DFS를 시작하기 전, 모든 시작 노드들을 계산하는 알고리즘이다. [Alg.2.2]는 [Alg.2.1]에서 계산된 모든 시작 노드들을 Batch 단위로 묶은 후, Iterative DFS를 수행하는 알고리즘이다. 알고리즘 수행 결과는 5장에 나타내었다.

Algorithm 2.1 Calculate starting nodes

```
procedure PREPROCESS(l)
    for all possible x_{1:l} do
        Calculate P(x_{1:l})
        if P_{x_{1:l}} ≥ P_{th} then
            S ← S + {x_{1:4}}
        end if
    end for
    return S
end procedure
```

Algorithm 2.2 Batch DFS

```
procedure BATCH-DFS(S)
    Select the frontiers from S as many as the number of batch size.
    Concatenate hidden, cell states of frontiers.
    while true do
        Expand next nodes using model.
        For each batch items, decide next node using P_{th}.
        If next node doesn't exist for all batch items, return .
    end while
end procedure
```

이 때 l은 시작 노드의 Level, S는 시작 노드의 집합을 의미한다.

4.4 Batch DFS의 개량

[Alg.2.1]과 [Alg.2.2]는 [Alg.1]을 Batch화 함으로써 속도가 향상되었지만, 여전히 그 구현체의 수행 속도가 느렸다. 수행에 있어서 [Alg.2.2]의 line 6이 병목인 지점이었는데, Batch 크기만큼 존재하는 instance들의 서로 다른 의사결정(자식으로 내려갈 지, 부모로 올라갈지)으로 인해 Branch divergence가 발생하여, 성능 저하가 발생한다.

이에 본 논문은 각 instance들의 의사결정을 통일화하도록 [Alg.2.2]를 개량하여 [Alg.3]을 사용하였다. 모든 instance들에 대해 expand가 끝났으면, 자식 노드의 개수를 세어 평균을 내어 반올림한다. 이 숫자는 각 instance가 다음에 탐색할 자식 노드의 개수가 된다(이 때 자식 노드들은 P_{th}와 관계없이 확률이 큰 순서대로 선정된다). 즉, Batch size의 절반 이상의 instance가 자식 노드로 이동하길 희망한다면, 자식노드가 없는 instance도 자식노드로 이동하고, 절반 이상의 instance가 부모 노드로 이동하길 희망한다면, 자식 노드가 있는 instance도 탐색을 중단하고 부모 노드로 이동한다. 이는 [Alg.1] DFS 알고리즘이 가지고 있는 두 가지 특색 ① Uniqueness constraint, ② Threshold Probability constraint에서 Threshold Probability constraint를 완화한 것이라고 볼 수도 있다.

Start node들을 Batch 단위로 묶을 때, 어떻게 묶는 지에 따라 그 결과는 상이할 것이다. 자식 노드가 많은 start node가 비교적 적은 start node와 묶일 경우 탐색을 방해받아 더 많은 노드로 내려갈 수 있는 경우에도 불구하고 탐색을 더 이상 진행하지 못한다. 이에 첫 번째 실험은 [Alg.2.1]에서 만들어진 start node 집합 S를 확률에 따라 정렬한 후, 확률이 높은 순서대로 Batch 단위로 묶는 시도를 하였다. 이 Batch들을 여러 대의 GPU에 분산하여 실행하였더니, load balancing이 저열하여 Batch마다 굉장히 실행 시간에 차이가 있었다. 두 번째 실험은 GPU가 n대일 때 start node를 확률 순으로 $\{k, n+k, n+2k, ...\}$씩 묶었다. 이 방법으로 Batch화를 한 것이 첫 번째보다 load balancing이 좋고, 성능 차이가 그다지 나지 않았다.

또한, [Alg.1]에는 마지막 토큰이 〈sos〉 토큰이어야만 $x_{1:n}$이 패스워드로

취급되지만, DFS 탐색 과정 중 나타나는 모든 internal node들도 패스워드로 취급할 수도 있다. 이에 대한 실험 결과는 5장에 나타내었다.

Algorithm 3 Improved Batch DFS

```
procedure IMPROVED-BATCH-DFS(S)
    Select the frontiers from S as many as the number of batch size.
    Concatenate hidden, cell states of frontiers.
    while true do
        Expand child nodes using model.
        Calculate n = round(Avg num. of child nodes with P ≤ P_th)
        Select n child nodes per batch items with highest probability .
        Decide next nodes.
        If next node doesn't exist, return .
    end while
end procedure
```

제 5장 실험 결과

5.1 모델의 하이퍼파라미터

[표 1]는 본 논문에서 사용된 비밀번호 생성을 위한 딥 러닝 모델의 하이퍼파라미터를 나타낸다.

Tokenizer	BPE Dictionary size	100
Word Embedding	Embedding vector size	256
LSTM	Layers	2
	Input, Hidden size	256
	Dropout	0.3
Training	Batch	1024
	Learning Rate	0.001
	Optimizer	Adam
	Epoch	150

[표 1] 모델의 하이퍼파라미터

[표 2]는 실험에 사용한 데이터셋(Rockyou)의 크기를 나타낸다.

Train set	12909895
Test set	1434433
Total	14344328

[표 2] 학습에 사용한 데이터셋의 크기

Rockyou 데이터셋을 Train, Test set은 9:1의 비율로 나누었으며, Train set의 90%는 학습에, 10%는 Validation에 사용되었다.

5.2 DFS 탐색 기법의 효과

모든 실험은 NVIDIA RTX3090 4대가 탑재된 노드에서 진행하였다.

	생성한 비밀번호	Test set에 포함된 비밀번호	Coverage Rate	Hit Rate
기존 기법	1.0×10^{9}	5.9×10^{5}	41%	1700:1
Alg.2 ($P_{th}=e^{-23}$)	8.8×10^{8}	8.0×10^{5}	56%	1100:1

[표 3] 추론 기법에 따른 Coverage rate, Hit rate 비교

	실행 시간	Test set에 포함된 비밀번호	시간당 Hit
기존 기법	2h 40m	5.9×10^{5}	60개/초
Alg.2 ($P_{th}=e^{-23}$)	10d	8.0×10^{5}	0.16개/초

[표 4] 추론 기법에 따른 시간 당 Hit 비교

[표 3]는 기존 추론 기법과 [Alg.2]를 생성한 비밀번호와 Test set에 포함된 비밀번호의 개수를 나타낸다. Coverage rate는 Test set의 비밀번호 중 맞춘 비밀번호의 비율을 말하며, Hit rate는 생성한 비밀번호 중 Test set에 포함된 비밀번호의 비율을 말한다. [표 4]는 실행시간을 고려한 비교를 나타낸다. [표 3]의 결과에 의하면 기존 기법 대비 [Alg.2]가 생성 개수 대비 맞춘 개수가 1.55배가량 좋지만, 시간을 고려한다면 같은 시간이 주어졌을 때 기존 기법이 375배 많은 비밀번호를 맞출 수 있다. 즉, [Alg.2]는 Hash algorithm이 굉장히 느려서 병목현상이 패스워드 생성이 아닌 hashing에 있을 때 기존 추론 기법보다 유리하다고 할 수 있다.

5.3 개량된 Batch DFS의 효과

위와 같이 실행 시간이 오래 걸리는 점을 개선하기 위해 앞서 4.4절에 [Alg.3]을 소개한 바 있다. 4.4절에서 소개한 [Alg.3]의 두 가지 옵션에 대한 실험 결과는 다음과 같다.

5.3.1 Start node의 batch화 방법

batch화 방법	실행 시간	생성한 비밀번호 개수	Test set에 포함된 비밀번호	시간당 Hit
1	2070초	1.5×10^9	659032	318개/초
2	974초	1.4×10^9	610327	627개/초

[표 5] batch화 방법에 따른 시간 당 Hit 비교($P_{th}=e^{-23}$)

[표 5]에서 1번 방법은 start node를 확률 순서대로 {1,2,3,⋯,1024}, {1025,1026,⋯,2048}, ⋯ 로 묶고, 각 batch를 GPU에 Round Robin으로 분배한 방법이다. 2번 방법은 start node를 확률 순서대로 {1,5,9,⋯,4093}, {2,6,10,⋯,4094}, ⋯ 로 묶고, 각 batch를 GPU에 Round Robin으로 분배한 방법이다. 실험 결과, 1번 방법은 각 GPU당 실행시간이 가장 짧은 것은 528초에서 가장 긴 것은 2070초로 상이하였지만, 2번 방법은 모든 GPU가 974초 근처로 균등하게 실행되었다. 1번 방법이 더 많은 비밀번호를 생성하고 맞춘 비밀번호도 많지만, 시간을 고려한다면 2번 방법이 1번 방법보다 약 2배정도 유리함을 알 수 있다.

5.3.2 Internal node의 비밀번호 취급

Internal node	실행 시간	생성한 비밀번호 개수	Test set에 포함된 비밀번호	시간당 Hit
O	2124초	1.5×10^{9}	659032	318개/초
X	2470초	4.1×10^{8}	622143	251개/초

[표 6] Internal node의 패스워드 취급 여부에 따른 시간 당 Hit 비교($P_{th}=e^{-23}$)

[표 6]은 Internal node를 비밀번호로 취급하는 지 여부에 따라 실험 결과이다. Internal node를 비밀번호로 취급하지 않는다면, Leaf node에 대한 검사가 필요하므로 추가적인 overhead가 들어 실행 시간이 16%정도 더 늘었음을 볼 수 있다. 또한, 생성된 비밀번호 개수는 1/4수준으로 감소하였고, Test set에 포함된 비밀번호도 마찬가지로 감소하였다. 즉, Internal node를 패스워드로 취급하는 것이 모든 면에서 효율적이다.

5.3.3 종합 분석

공격자가 최종적으로 목표하는 것은 단위 시간당 더 많은 비밀번호를 맞추는 것이다. 맞춘 개수가 많아질수록 시간당 맞추는 비밀번호의 개수는 당연히 감소한다. 본 논문에 등장한 [Alg.2], [Alg.3]의 공평한 비교를 위해 최종 개수를 10^9개로 고정한 후 이를 비교하면 [표 7]과 같다.

	P_{th}	Start node batch화 방법	Internal node 비밀번호 취급	맞춘 개수	실행 시간	시간당 Hit
기존 기법	-	-	-	588055	9700초	61개/초
Alg.2	e^{-23}	-	-	802612	10일	0.16개/초
Alg.3	$e^{-23.5}$	1	O	384852	720초	535개/초
	e^{-23}	2	O	546574	720초	759개/초
	e^{-24}	1	X	714714	5564초	128개/초

[표 7] 만든 개수를 10^9로 고정한 후 각 알고리즘의 비교

즉, Hash algorithm이 빠르다는 가정 하에 단위 시간당 맞추는 개수를 최대화 하고 싶다면 초당 759개로 기존 알고리즘보다 약 12.4배 빠른 속도로 비밀번호를 맞출 수 있고, 이보다 느린 Hash algorithm의 경우, 같은 개수 대비 조금 더 Hit rate가 높은 알고리즘을 선택하여 초당 128개로 기존 알고리즘보다 2배가량 높은 속도로 비밀번호를 맞출 수 있다.

5.4 타 모델과 비교

비밀번호 생성 분야의 State-Of-The-Art 모델은 Biesner et al.[7]이 제시한 GPT2 구조이다. 그들은 Rockyou 데이터셋의 20%를 Test set으로 설정하였고, 10^9개의 패스워드를 생성하였을 때 45.1%의 Coverage rate를 달성하였다. 본 논문에서는 10%를 Test set으로 설정하여 완벽한 비교는 불가능 하지만, Coverage rate가 41%으로 [7]에 비해 성능이 떨어진다고 할 수 있다. 하지만 추론 기법을 통해 모델 변경 없이 [7]보다 높은 51%의 Coverage rate를 달성하면서, 기존 기법보다 약 2배가량 빠르게 비밀번호를 맞출 수 있다.

제 6장 결론

딥 러닝을 이용한 비밀번호 생성 분야의 최근 경향은 모델 구조를 바꾸고, Dropout이나 Normalization 처럼 일반적인 딥 러닝 모델들의 성능 향상 기술들을 이용하여 Hit rate 또는 Coverage rate를 향상시키는 것을 목표로 한다. 비밀번호를 Token sequence로 바꾸어 시계열 모델을 적용한 논문[5, 7, 21]등은 모두 모델이 출력한 다음 토큰의 확률 분포로부터 토큰을 샘플링하는 방식으로 비밀번호를 생성한다. 이러한 추론 방법은 생성할 수 있는 비밀번호 개수에 제한이 없으나, 이전에 생성하였던 비밀번호를 중복하여 생성할 수 있으므로 생성을 많이 할 수록 효율이 감소하게 된다. 본 논문은 모델 구조 대신 추론 방법을 제시함으로써 높은 수준의 Coverage rate를 달성하였다. 또한, 이 추론 방법은 비밀번호 생성을 위한 모든 시계열 딥 러닝 모델에 적용할 수 있다.

후속 연구로는 P_{th}에 따른 최대 생성 가능한 비밀번호 개수 및 Coverage rate를 통해 여러 데이터셋의 특성을 연구할 것이다. 본 논문에서 사용된 모델을 선정하기 위한 여러 시행착오 중, Myspace와 Rockyou 데이터셋에 비해 Phpbb 데이터셋에서는 특출나게 Hit rate가 떨어지는 경향이 있었다. 이러한 실험 결과로 부터 각 데이터셋이 저마다 가지고 있는 특성이 있을 것이라고 추정할 수 있고, 관련된 Metric을 정의할 것이다.

참 고 문 헌

[1] Hashcat. https://hashcat.net/hashcat/

[2] John the Ripper. https://www.openwall.com/john/

[3] M. Weir, S. Aggarwal, B. Medeiros, B. Glodek, "Password Cracking Using Probabilistic Context-Free Grammers", In *30th IEEE Symposium on Security and Privacy,* 2009, pp. 391-405

[4] A. Narayanan, V. shmatikov. "Fast dictionary attacks on passwords using time-space tradeoff." In *Proceedings of the 12th ACM conference on Computer and communications security*, 2005, pp.175-191

[5] W. Melicher et al. "Fast, Lean, and Accurate: Modeling Password Guessability Using Neural Networks", In P*roceedings of the 25th USENIX Security Symposium,* 2016, pp. 175-191

[6] B. Hitaj et al. "PassGAN: A Deep Learning Approach for Password Guessing" In *International Conference on Applied Cryptography and Network Security.* 2019. pp. 217-237.

[7] D. Biensner et al. "Generative Deep Learning Techniques for Password Generation", arXiv preprint arXiv:2012.05685, 2020

[8] A. Vaswani et al. "Attention is all you need" In *Advances in Neural Information Processing Systems*, 2017. pp. 5998-6008

[9] D. Kingma, M. Welling. "Auto-Encoding Variational Bayes", arXiv preprint arXiv:1312.6114, 2013

[10] A. Radford, J. Wu et al. "Language Models are Unsupervised Multitask Learniners", 2018

[11] R. Sennrich et al. "Neural Machine Translation of Rare Words with Subword Units", arXiv preprint arXiv:1508.07909, 2015

[12] Myspace. https://www.myspace.com/

[13] Linkedin, https://www.Linkedin.com/

[14] Yahoo, https://www.yahoo.com/

[15] Rockyou, https://www.rockyou.com/

[16] S. Hochreiter et al. “Long Short-Term Memory”, Neural Computation 9(8), 1997, pp. 1735-1780
[17] https://colah.github.io/posts/2015-08-Understanding-LSTMs/
[18] S. Ioffe, C. Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift.” arXiv preprint arXiv:1502.03167, 2015
[19] J. Ba et al. “Layer Normalization” arXiv preprint arXiv:1607.06450, 2016
[20] N. Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”, The Journal of Machine Learning Research 15(1), 2014, pp.1929-1958
[21] H. Li et al. “Password guessing via neural language modeling.” In *Machine Learning for Cyber Security*, 2019, pp.78-93. Springer International Publishing.
[22] I. Goodfellow et al. “Generative Adversarial Networks”, In *Advances in neural information processing systems*, 2014, pp.2672-2680

Abstract

Inference Method for Password Guessing Model with Tree Search

Lim Hyun Jae
Department of Computer Science & Engineering
The Graduate School
Seoul National University

This paper solves the password regeneration problem and significantly improves inference performance by combining the inference process of the password generation technique using deep learning with tree search. While recent works have focused on the improvement of deep learning models, the inference method presented in this paper shows that 2 to 12 times higher efficiency can be achieved only by changing the inference method without improving the model. In addition, the reasoning technique presented in this paper can be applied to any structure using a time series model among various existing deep learning model structures.

keywords : Password Generation, Password Guessing, Deep Learning, LSTM, Tree Search, Inference
Student Number : 2019-27787