2024 암호분석경진대회 - 비밀분쇄기팀 1번 문제 풀이

1 세부 문제 1: Feed Polynomial 및 초기값이 주어졌을 때 LFSR 구현.

세부 문제 1을 해결하기 위해 Python 언어를 사용했다.

먼저 LFSR 의 feedback polynomial 을 관리하기 위해 `Polynomial` 클래스를 구현했다. $GF(2)$ 위에서 정의된 polynomial 이므로 1차원 binary 리스트를 coefficient 로 가진다. 문제 설명의 그림과 일치하도록 coefficient 는 가장 높은 차수부터 낮아지는 순서대로 저장하도록 했다. 구현한 코드의 일부분은 아래와 같다.

```python
# lfsr.py
# GF(2) polynomial
class Polynomial:
    # coeff is a list of coefficients, starting from the highest degree
    # coeff = [p_m, p_m-1, p_m-2, ..., p_0]
    # ex: x^3 + x + 1 -> Polynomial([1, 0, 1, 1])
    def __init__(self, coeff):
        assert coeff[0] == 1
        self.deg = len(coeff) - 1
        self.coeff = coeff
```

이후 `Polynomial` 클래스와 initial state (binary 리스트)를 초기화 입력으로 받는 `LFSR` 클래스를 구현했다. 문제 설명의 그림과 일치하도록 initial state 및 state 는 feedback 입력부터 출력 순으로 저장하도록 했다. `step()` 멤버 함수를 호출해 LSFR 의 다음 비트를 생성할 수 있고, `generate_n_bits()` 함수로 한번에 n 개의 bitstream 을 생성할 수 있다. 구현한 구현 코드의 일부분은 아래와 같다.

```python
# lfsr.py
class LFSR:
    # Initial states are stored in a list, from the highest degree to the lowest
    # Like: [s_m-1, s_m-2, ..., s_0]
    # At each step, the rightmost bit is returned, and the states are updated according to the polynomial
    def __init__(self, poly, initial_states):
        assert len(initial_states) == poly.deg
        self.poly = poly
        self.initial_states = initial_states
        self.states = initial_states
        self.len = len(initial_states)

    def step(self):
        ret = self.states[-1]
        next_bit = 0
        for i in range(self.len):
            next_bit ^= self.poly.coeff[self.len - i] \
                * self.states[self.len - 1 - i]
        self.states = [next_bit] + self.states[:-1]
        return ret

    def generate_n_bits(self, n):
        self.states = self.initial_states
        return [self.step() for _ in range(n)]
```

`Polynomial` 및 `LSFR` 클래스를 이용해 feedback polynomial이 (0,1,2,7,128)인 LFSR을 구현한 코드가 `prob1.py` 에 있다. 초기값을 바꾸려면 `line 11` 의 `initial_states` 를 설정하는 부분을 변경하면 된다.

```python
# prob1.py
from lfsr import *
import random

deg = 128
poly_poplist = [0, 1, 2, 7, 128]
coeff = [1 if deg-i in poly_poplist else 0 for i in range(deg+1)]
poly = Polynomial(coeff)  # x^128 + x^7 + x^2 + x + 1
print("Polynomial: ", poly)

initial_states = [random.randint(0, 1) for _ in range(deg)]
```

```
initial_states_str = ''.join([str(b) for b in initial_states])
print("Initial states: ", initial_states_str)

lfsr = LFSR(poly, initial_states)
bitstream = lfsr.generate_n_bits(1024)
bitstream_str = ''.join([str(b) for b in bitstream])
print("First 1024 bitstream: ", bitstream_str)
```

실행 결과는 다음과 같다.

```
$ python prob1.py
Polynomial:  x^128 + x^7 + x^2 + x + 1
Initial states:  110100100110100 (...) 11011111111000100110110110100100
First 1024 bitstream:  1110001000000110110001 0 (...) 1001111000111101110000010111110 1011
```

2 세부 문제 2: LFSR 의 출력 bitstream 으로부터 주기를 찾아내는 DNN 학습.

2.1 학습 데이터셋 생성

다양한 polynomial 및 bitstream 길이에 대해 모델이 잘 작동하도록 하기 위해 무작위로 데이터를 생성했다. Feedback polynomial 의 최소 차수는 3, 최대 차수는 9 로 설정했다. 데이터셋을 생성할 때 bitstream 의 최소 길이는 32, 최대 길이는 1024로 설정했다.

하나의 데이터를 생성하는 과정은 다음과 같다.

1. Feedback polynomial 의 차수 d 를 [3, 9] 범위에서 무작위로 선택해 결정한다.

2. Feedback polynomial 의 계수들을 결정한다. $x^d + p_{d-1}x^{d-1} + \ldots + p_1 x + 1$ 에서 $p_{d-1}, \ldots, p_1$ 의 값을 각각 0 또는 1 중 무작위로 선택한다.

3. Initial state 를 결정한다. $s_{d-1}, s_{d-2}, \ldots, s_1, s_0$ 의 값을 각각 0 또는 1 중 무작위로 선택한다.

4. 1.-3. 과정에서 얻은 LFSR 의 주기 τ 를 계산한다. 이 과정은 brute-force 방식으로 같은 state 가 두 번 이상 반복될 때까지 bitstream 을 생성해 주기를 찾아낸다. LFSR 의 주기가 3보다 작거나, 512를 초과하면 (주기가 너무 길어 bitstream 이 최대 길이임에도 한 주기가 반복되지 못할 경우) 1. 로 돌아간다.

5. 생성할 bitstream 길이를 $[2\tau, 1024]$ 사이에서 무작위로 결정한 뒤 해당 길이만큼 bitstream 을 생성한다.

위 과정을 반복해 총 32768 개의 데이터를 생성한 뒤, train, validation, test 데이터셋을 8:1:1 의 비율로 나누어 사용했다.

2.2 DNN 모델 구조 및 학습

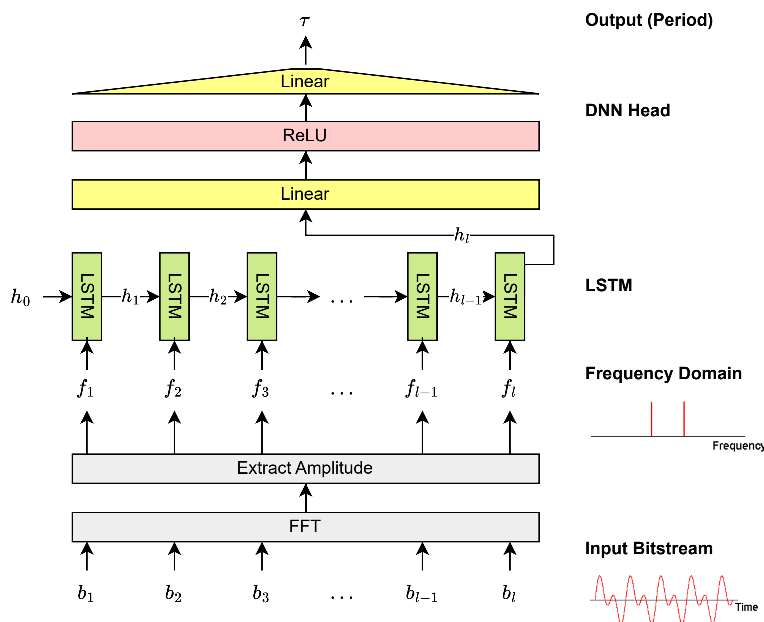

Figure 1: DNN 모델 구조

모델 구조 Figure 1 에 사용한 DNN 모델 구조가 도식화되어 있다. 먼저, 입력 bitstream 에 Fast Fourier Transform (FFT) 를 적용해서 시계열 데이터를 time domain 에서 frequency domain 으로 변환한 뒤, amplitude 값을 취한다. 이렇게 변환된 데이터를 LSTM [1] 모델의 입력으로 넣은 다음 최종적인 hidden state h_l 을 취한다. LSTM 의 마지막 hidden state h_l 을 두 개의 Linear 레이어로 구성된 DNN head 에 통과시켜 scalar output 을 얻어낸다. Scalar output 이 실수 형태로 주어지므로, 이를 반올림해서 최종적인 bitstream 의 주기 값으로 사용한다.

LSTM 모델의 hidden vector 의 크기는 128로 설정했다. DNN head 의 첫 Linear 레이어의 크기는 128×128 이고, 두번째 Linear 레이어의 크기는 128×1 이다.

Table 1: DNN 학습을 수행한 시스템 정보.

Motherboard	ASRockRack GENOAD8X-2T/BCM
CPU	2 x AMD EPYC 9124 16-Core Processor
Main Memory	8 x DDR4-2666 32GB
GPU	1 x NVIDIA Geforce RTX 4090
PCIe	16 x Gen3 lanes per GPU
OS	Ubuntu 20.04.6 LTS (kernel 5.4.0-100-generic)
GPU Driver	550.54.15
CUDA Version	12.4

모델 학습 환경 및 파라미터 모델 학습은 총 150 epoch 을 진행했다. Optimizer 는 Adam 을 사용했다. Learning rate 기본값은 0.005, $\beta_1 = 0.9, \beta_2 = 0.95, \epsilon = 10^{-5}$ 으로 설정했다. Learning rate scheduler 는 LambdaLR 을 사용했다. Loss function 으로는 MSE (Mean Squared Error) 을 사용했다.

2.3 모델 학습 결과

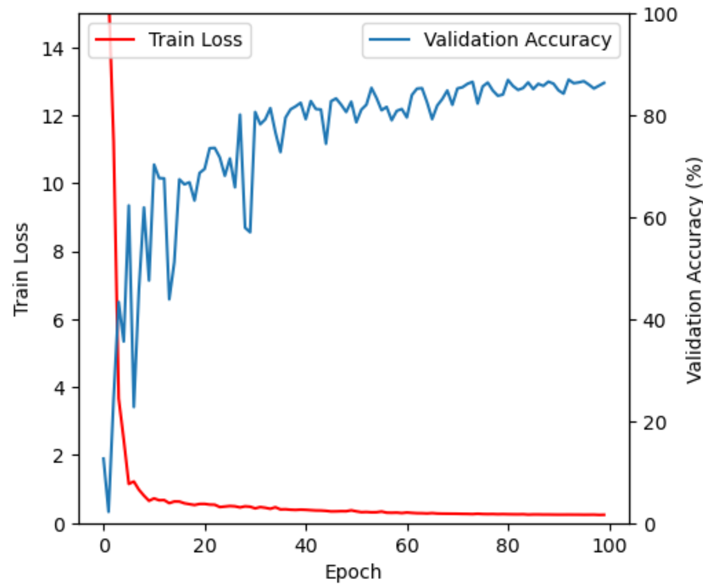

Figure 2: 모델 학습 결과

Figure 2 에 모델 학습 결과가 나타나 있다. Training loss 가 떨어짐에 따라 validation accuracy 가 증가하는 경향이 잘 나타나 있다. Validation accuracy 가 가장 높은 모델을 최종 모델로 선정하고, 이를 이용해 test dataset 에 성능을 평가한 결과, 최종적인 test accuracy 는 87.25% 를 달성할 수 있었다. DNN 모델이 3278 개의 bitstream 데이터 중 2860 개의 주기를 정확히 찾아낸 것이다.

3 소스 코드 설명

사용한 파이썬 패키지 버전 개발 및 모델 학습에 Python 3.10.14, PyTorch 2.4.0, Pandas 2.2.2 를 사용했다.

Polynomial 및 LFSR 구현 코드 `lfsr.py` 는 $GF(2)$ polynomial 및 LFSR 을 구현한 소스 코드이다. 별도의 외부 패키지를 사용하지 않고 구현하였다. 구현을 테스트하는 코드는 `prob1.py` 에 있다.

데이터셋 생성 코드 `generate_dataset.py` 는 LFSR 데이터셋을 생성해 csv 파일로 저장하는 소스 코드이다.

`python generate_dataset.py` 와 같이 실행 가능하며, 프로그램 실행 결과로 `train.csv`, `val.csv`, `test.csv` 의 세 파일이 생성된다.

모델 학습 코드 `train.py` 에 모델 구조 정의 및 학습 코드가 구현되어 있다.

`python train.py` 와 같이 실행 가능하며, 학습 결과는 csv 형태로 stdout 에 출력되고, 학습된 모델 파라미터는 `model.pth` 에 저장된다. 실행 예시는 아래와 같다.

```
$ python -u train.py | tee train_log.csv
epoch,train_loss,val_acc
0,15.647392200842136,0.2268009768009768
1,9.17580356365297,0.28357753357753357
2,14.240020184400604,0.10531135531135531
3,4.716839269312417,0.3324175824175824
(...)
96,0.2460476072823129,0.8601953601953602
97,0.24704658643501562,0.8525641025641025
98,0.240397003365726,0.858058608058608
99,0.2431693112704812,0.8635531135531136
Training done
Best validation accuracy: 87.00%
```

결과 csv 파일을 받아 도식화하는 프로그램은 `plot.ipynb` 에 구현되어 있다.

4 학습된 모델 평가 코드

`test.py` 에 학습된 모델을 테스트 데이터셋으로 평가하는 코드가 구현되어 있다. 학습된 모델 파라미터를 `model.pth` 에서 읽어오고, 테스트 데이터를 `test.csv` 에서 읽어온 뒤, 전체 bitstream 데이터 중 정확히 주기를 추측한 개수를 stdout 으로 출력한다. `test.csv` 파일 내용으로 bitstream 데이터를 `seq` 열에, 주기를 `period` 열에 제공해 주면 된다. 본 보고서 작성에 사용된 예시 `test.csv` 파일을 함께 제출한다. 실행 예시는 다음과 같다.

```
$ python -u test.py
Test accuracy:  87.25 %, 2860 / 3278
```

References

[1] Alex Graves and Alex Graves. Long short-term memory. *Supervised sequence labelling with recurrent neural networks*, pages 37–45, 2012.