인공지능 과제 리포트

과제 제목: Deep neural network를 이용한 AReM data 분류

학번: B611155
이름: 이유진

1. 과제 개요

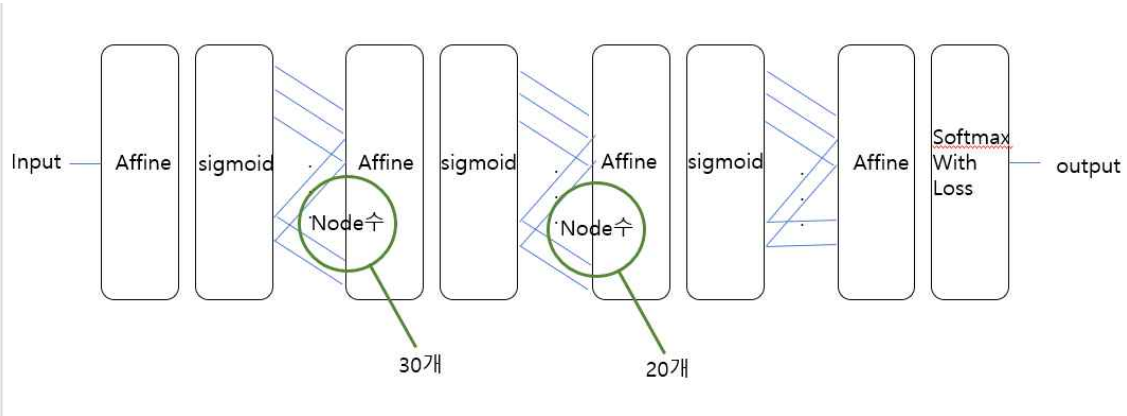
라이브러리의 도움을 받지않고 딥러닝 모델을 구현하여 AReM 데이터를 분리한다.

2. 구현 환경

Window, Anaconda Jupyter notebook

3. 알고리즘에 대한 설명

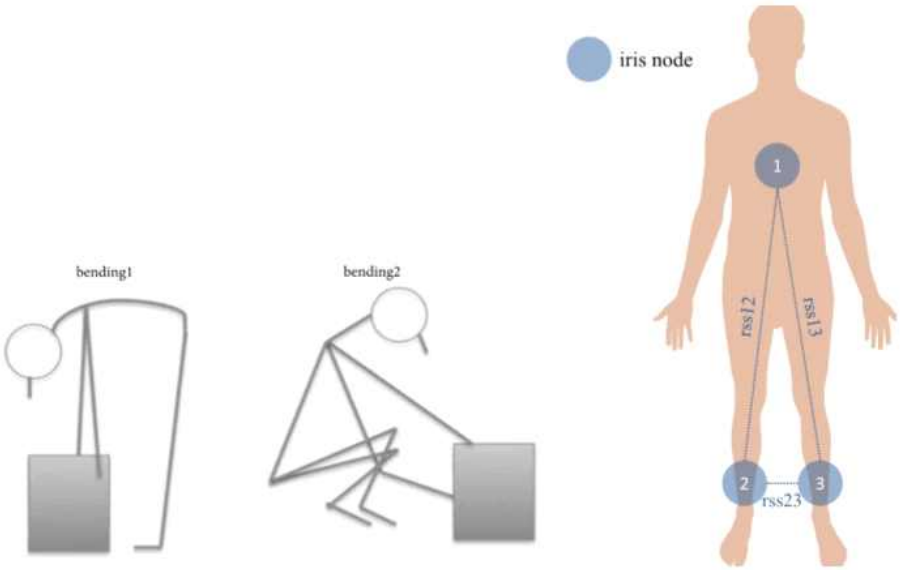
이번 과제에서는 3층짜리 딥러닝 모델를 만들었으며 각 hidden 레이어 당 노드수는 30, 20 개다.



간단한 알고리즘 설명은 input train data 가 들어오면 각 class 당 평균과 분산을 구해 스케일링을 한 후, model에 넘겨준다.
model의 layer 구성은 위의 그림과 같다.
Model 안에서는 먼저 forward, back propagation 을 이용하여 각 layer에서의 기울기를 구한 후 optimizer를 사용하여 기울기가 작아지는 방향 (loss값이 작아지는 방향)으로 가중치 파라미터를 update 한다.
학습데이터에 맞춘 파라미터를 통해 test 데이터를 분류한다.

4. 데이터에 대한 설명

a.



예시)	avg_rss12	var_rss12	avg_rss13	var_rss13	avg_rss23	var_rss23
bending	39.25	0.43	22.75	0.43	33.75	1.3
cycling	32	4.85	17.5	3.35	22.5	3.2
lying	29	0	9	0.71	8.5	0.5
sitting	42	0	19.2	0.98	15.5	2.06
standing	46.5	0.5	11.5	0.5	20.33	0.94
walking	35	3.67	16.5	3.77	14	1.63

4.1 Input Feature

Input data는 AReM 데이터이며, 여섯가지 동작을 수행할 때 센서의 값을 나태내며 input feature는 각 동작에 대한 센서데이터의 평균과 분산이다.즉 feaure 개수는 6개다.
레이블 값은 원핫 인코딩하여 넣는다.

4.2 Target Output

Output data는 해당 input이 6개의 동작 중 해당 동작일 확률값을 나타낸다.
가장 큰값으로 분류한다.

5. 소스코드에 대한 설명

(1) 활성화 함수

활성화 함수는 sigmoid와 ReLU로 돌린 결과 sigmoid의 정확도가 더 높아 sigmoid를 사용한다.
가중치 초기값은 이에 맞게 xavier 초기값 사용한다,

(2) optimizer

optimizer는 Nadam을 사용한다. Nadam은 NAG(nesterov acceletated gradient)와 Adam의 개념을 합친 것이다. 즉 현재위치에서 다음위치로 이동할 gradient와 momentum 값을 구하는 것이 아니라 momentum 값으로 이동한 뒤에 gradient 값을 구하는 것이다.
Nadam 은 Adam보다 조금 더빠르게 global minimun을 찾아낼 수 있다고 하여 구현해보았는데 시간이 큰차이는 없었다.

(3) feature scailing
전체 학습데이터에 대한 평균과 표준편차를 구해 모델에 인자로 전달하고 그 인자로 predict에서 들어오는 x를 scailing한다. 이 인자들을 피클에 저장해놓고 테스트 데이터도 스케일링한다.

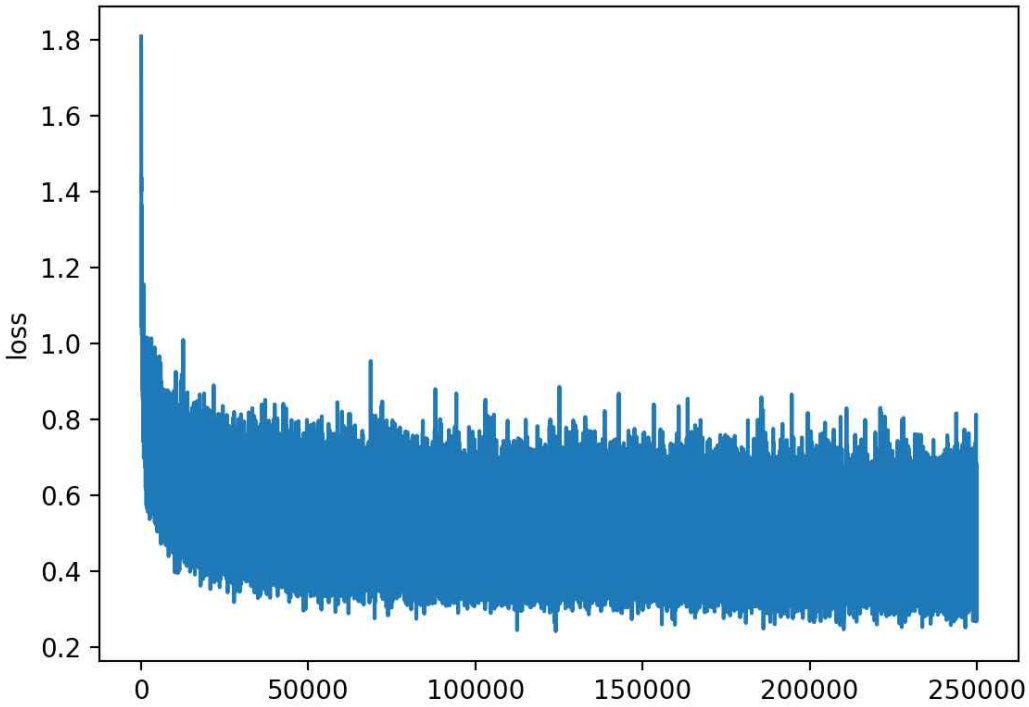
(4) 하이퍼 파라미터
Epoch은 200, 500, 1000 / learning rate는 0.01, 0.005, 0.001 중
Epoch = 1000, learning rate = 0.001 을 선택하였을 때 결과 정확도와 loss값이 가장 괜찮게 나와서 이 값들을 선택했다.

6. 학습 과정에 대한 설명
알고리즘에 대한 설명과 동일

7. 결과 및 분석

```
ss:0.527 ===
=== epoch: 983 , iteration: 245500 , train acc:0.793 , test acc:0.77 , train los
s:0.407 ===
=== epoch: 984 , iteration: 245750 , train acc:0.793 , test acc:0.766 , train lo
ss:0.415 ===
=== epoch: 985 , iteration: 246000 , train acc:0.793 , test acc:0.769 , train lo
ss:0.478 ===
=== epoch: 986 , iteration: 246250 , train acc:0.794 , test acc:0.766 , train lo
ss:0.403 ===
=== epoch: 987 , iteration: 246500 , train acc:0.793 , test acc:0.767 , train lo
ss:0.545 ===
=== epoch: 988 , iteration: 246750 , train acc:0.795 , test acc:0.77 , train los
s:0.606 ===
=== epoch: 989 , iteration: 247000 , train acc:0.792 , test acc:0.768 , train lo
ss:0.394 ===
=== epoch: 990 , iteration: 247250 , train acc:0.796 , test acc:0.768 , train lo
ss:0.486 ===
=== epoch: 991 , iteration: 247500 , train acc:0.793 , test acc:0.769 , train lo
ss:0.335 ===
=== epoch: 992 , iteration: 247750 , train acc:0.793 , test acc:0.771 , train lo
ss:0.451 ===
=== epoch: 993 , iteration: 248000 , train acc:0.794 , test acc:0.769 , train lo
ss:0.498 ===
=== epoch: 994 , iteration: 248250 , train acc:0.793 , test acc:0.769 , train lo
ss:0.467 ===
=== epoch: 995 , iteration: 248500 , train acc:0.795 , test acc:0.769 , train lo
ss:0.569 ===
=== epoch: 996 , iteration: 248750 , train acc:0.795 , test acc:0.771 , train lo
ss:0.342 ===
=== epoch: 997 , iteration: 249000 , train acc:0.793 , test acc:0.767 , train lo
ss:0.434 ===
=== epoch: 998 , iteration: 249250 , train acc:0.794 , test acc:0.767 , train lo
ss:0.498 ===
=== epoch: 999 , iteration: 249500 , train acc:0.793 , test acc:0.767 , train lo
ss:0.438 ===
=== epoch: 1000 , iteration: 249750 , train acc:0.793 , test acc:0.767 , train l
oss:0.444 ===
=============== Final Test Accuracy ===============
test acc:0.7645790923242726, inference_time:2.970453965108618e-06
```

Ln: 3664 Col: 81



결과를 보면 loss값은 0.4 정도가 나오지만 plot 이미지를 보면 굉장히 진동하는 것을 볼수 있다. 이유를 생각해봤을 때, 데이터 자체가 피쳐수는 적은데 굉장히 겹쳐져있어서 바운더리를 제대로 찾을수 없어 학습때마다 loss값이 변하고, 또는 배치 데이터를 뽑아서 학습시키는데 그 배치 데이터들마다 하나의 레이블에 치중되어있어서 loss값이 계속 진동하는 것 같다.