

# 인공지능 과제 리포트

과제 제목:

Deep Neural Network

학번: B411001

이름: 강도연

## 1. 과제 개요

과제의 주 목적은 **Deep Neural Network**를 라이브러리의 도움을 받지 않고 직접 모델을 구현하는 것이다. **Hidden layer 개수 및 unit 수, optimizer, weight 초기화, loss function** 등 필요한 요소들을 직접 선택하고 설계한다. 그 후, 이로 인해 일어나는 **overfitting** 등 여러 문제들을 분석하고 **hyperparameter**들 또한 알맞게 수정해 가며 높은 예측 정확도를 내는 Network를 만든다. 전체적으로, 직접 설계한 Deep Neural Network를 이용하여 주어진 여섯 가지 동작에 대한 train data를 이용하여 학습시킨 후, 최종적으로 test data로 예측 진행 시 높은 accuracy를 만들어 낸다.

## 2. 구현 환경

Windows OS를 사용하였으며, IDE는 Pycharm(ver.2019.3.5.), 언어는 python(ver.3.7), Package는 numpy, AReM data를 사용하였다.

## 3. 알고리즘에 대한 설명

**Deep Neural Network**는 **hidden layer가 최소 1개** 이상인 network를 말하며, Layer들 사이에는 각 Weight와 bias가 존재한다.

### 3.1) 이번과제에서 설계한 Network 구성 (figure 1):

Layer = 6개

- Input layer 1개 -> units = 6
- Output layer 1개 -> units = 6
- Hidden layer 4개 -> units = 13, 27, 55, 111

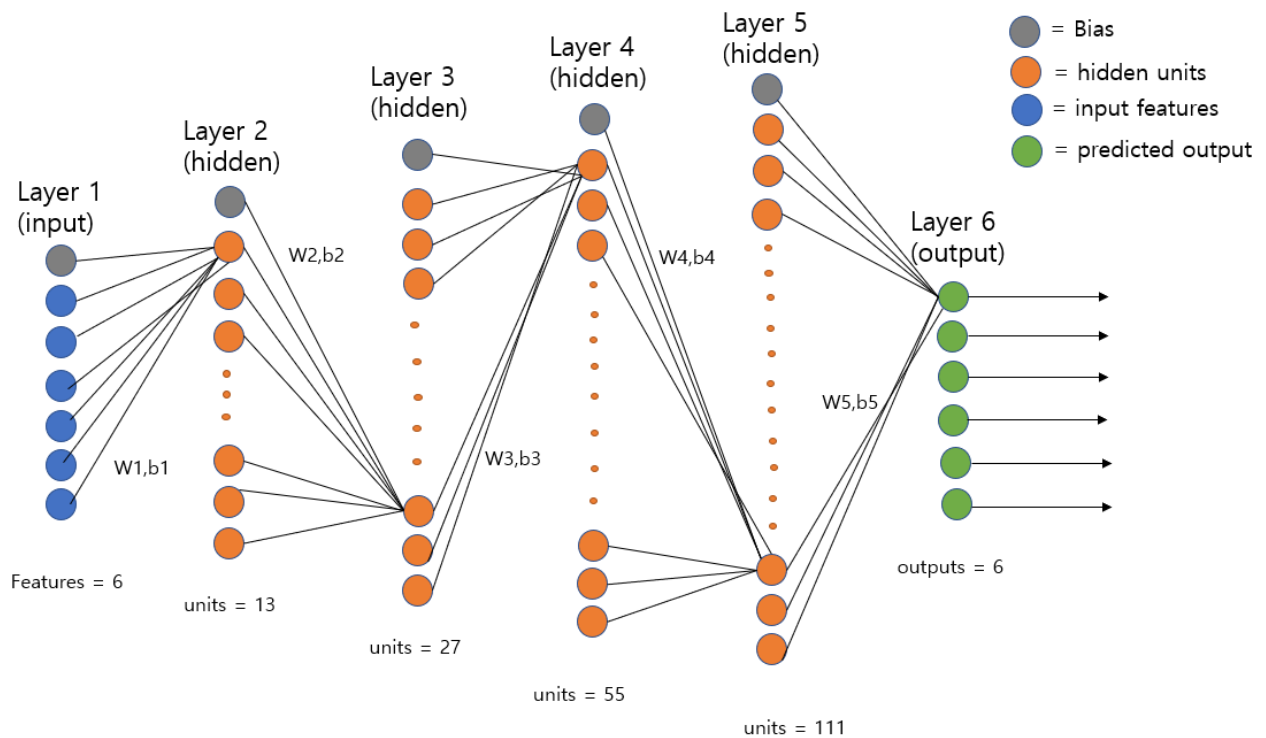
Weight 초기화 - He(카이밍 히) 초깃값

Activation function - ReLU, Softmax

Loss function - Cross-Entropy-error

Optimization - Adamax

Overfitting 방지 - L2 regularization (weight decay)



**Figure 1. Deep Neural Network Architecture**

- Activation function으로는 vanishing gradient현상을 해결하기 위해 ReLU 그리고 classification에서 output값들을 정규화 시키고 확률 값으로 해석하기 위해 softmax를 사용하였다.
- Weight 초기화는 ReLU를 activation으로 사용하였을 때 초기값의 정규분포가 좋게 평가된 He 초기값을 이용하였다. (He 초기값 - 표준편차가  $\sqrt{2 / \text{이전 layer의 노드 수}}$ )
- Loss function으로는, Mean squared error보다 빠른 속도로 학습이 가능한 Cross entropy error를 사용하였다.
- Optimization으로는 Adamax를 사용하였다. Adam의 장점은, 학습을 진행하면서, learning rate를 L2 norm 기반으로 조절한다는 점인데, 이를 Lp norm으로 확장시켰으며, 이때 p가 너무 커지면 문제가 될 것을 고려하여 infinity norm을 사용하여 만들어 진 것이 Adamax이다.

참고:<https://arxiv.org/pdf/1412.6980.pdf>,

[https://en.wikipedia.org/wiki/Norm\\_\(mathematics\)#p-norm](https://en.wikipedia.org/wiki/Norm_(mathematics)#p-norm)

## 4. 데이터에 대한 설명

### 4.1) AReM data

#### 4.1.1) Data set

- 여섯가지 동작(bending, cycling, lying, sitting, standing, walking)을 수행했을 때 센서 값.
- Walking: 4291, Standing: 4301, Sitting: 4311, Lying: 4342, Cycling: 4408, Bending: 3402
- Train data set 25000개, 배치 사이즈=100. 즉, 250개의 배치로 매 epoch마다 진행.

#### 4.1.2) Input feature

- Input feature는

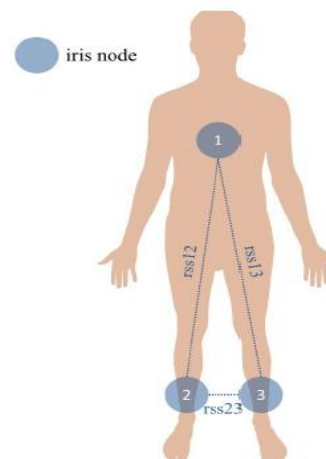
센서 값 평균, 센서 값 분산으로

avg\_rss12, var\_rss12,

avg\_rss13, var\_rss13,

avg\_rss23, var\_rss23

**총 6개이다.**



#### 4.1.3) Target output

- Output class는 walking, standing, sitting, lying, cycling, bending으로 **총 6개이다.**

## 5. 소스코드에 대한 설명

### 5.1) model.py (Deep Neural Network class)

#### 5.1.1) def softmax

- 예측 값 계산을 위한 마지막 layer에서에서의 activation function

#### 5.1.2) def cross\_entropy\_error

- 예측 값과, one-hot-encoding된 target값으로 loss값을 계산하기 위한 loss function

### 5.1.3) class ReLU

- Hidden layer에서의 activation function. Forward와 backward method를 작성하였다.

### 5.1.4) class Affine

- Input에 가중치 값과 bias를 적용. Forward와 backward 메소드를 작성하였다.

### 5.1.5) class SoftmaxWithLoss

- softmax적용 및 loss값 계산을 동시에 진행. Forward와 backward 메소드를 작성하였다.

### 5.1.6) class Adamax

- 정확도 향상을 위해 직접 찾아보고 작성한 Adam의 변형된 optimizer.

**for key in params.keys():**

**self.t += 1**                    #epoch count를 위한 변수

**self.m[key] = (self.b1 \* self.m[key]) + ((1 - self.b1) \* grads[key])**                    #moment vector

**self.v[key] = np.maximum((self.b2 \* self.v[key]), np.abs(grads[key]))**                    #infinity norm 적용

**params[key] -= (self.lr / (1 - self.b1 \*\* self.t)) \* (self.m[key] / (self.v[key] + 1e-8))**

### 5.1.6) class Model

- 전체적인 network구성 및 학습 시 필요한 method들 작성.

5.1.6.1) def \_\_init\_layer() - 전체적인 layer 뼈대 생성 및 필요한 class들 선언.

Training전, network에 weight, bias값 적용.

5.1.6.2) def \_\_init\_weight() - layer사이마다 필요한 Weight, bias값 초기화

5.1.6.3) def update() - Optimizer를 통해 gradient를 가지고 W, b값들 update.

5.1.6.4) def predict() - softmax를 적용하여 각 class에 속할 확률 값 계산

5.1.6.5) def loss() - 5.1.6.4)의 값을 가지고 loss값 계산

5.1.6.6) def gradient() - Forward를 진행하고, backward propagation을 진행함으로써, 각 layer들 사이에서의 gradient값을 구한다. 이에 더하여, overfitting 방지를 위해 weight decay를 적용하였다.

**Forward 예)**

**forward\_L1=self.layer['L1\_ReLU'].forward(self.layer['L1\_Affine'].forward(x))**

- input(x)을 L1의 Affine(  $x * W + b$ )에 적용, 그 후, ReLU를 적용

Backward 예)

```
backprop_L4 = self.layer['L4_Affine'].backward(self.layer['L4_ReLU'].backward(backprop_L5))
grads['L4_W'] = self.layer['L4_Affine'].dW
grads['L4_b'] = self.layer['L4_Affine'].db
```

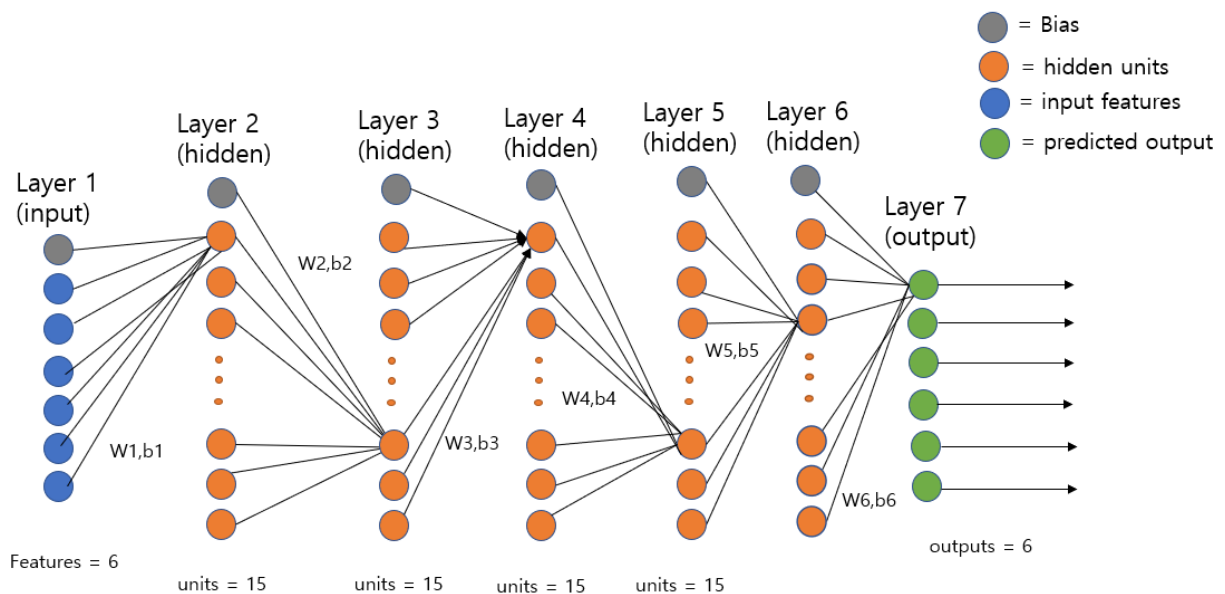
- ReLU backward를 우선 진행 후, Affine backward를 통해 계산된 그 layer W, b(gradient) 저장.

5.1.6.7) `load_params()` - 저장된 `params.pkl`값을 가져온다. 이때, 5.1.6.1)을 한번

더 사용하여 현재 layer에 적용시켜준다.

## 6. 학습 과정에 대한 설명

### 6.1) 초기 Architecture:



**Loss:** Cross entropy, **Optimizer:** Adagrad, **Activation:** ReLU, softmax, **Weight init:** He 초기값,

**Overfitting:** Dropout

결과:

```
=== epoch: 1081, iteration: 270000, train acc:0.61, test acc:0.602, train loss:1.451 ===
=== epoch: 1082, iteration: 270250, train acc:0.638, test acc:0.633, train loss:1.405 ===
=== epoch: 1083, iteration: 270500, train acc:0.636, test acc:0.634, train loss:1.466 ===
=== epoch: 1084, iteration: 270750, train acc:0.643, test acc:0.643, train loss:1.428 ===
=== epoch: 1085, iteration: 271000, train acc:0.651, test acc:0.644, train loss:1.474 ===
=== epoch: 1086, iteration: 271250, train acc:0.664, test acc:0.663, train loss:1.456 ===
=== epoch: 1087, iteration: 271500, train acc:0.639, test acc:0.63, train loss:1.455 ===
=== epoch: 1088, iteration: 271750, train acc:0.641, test acc:0.637, train loss:1.451 ===
```

## 6.2) 학습 과정:

### 6.2.1) Hidden layer 수, layer unit 수 test.

- hidden layer를 5개 더 늘렸을 때, test 정확도는 0.70로서 증가하였고, 10개 늘렸을 때, 정확도에는 변화가 없었다.
- hidden layer unit수를 5개 줄였더니 정확도가 0.61로서 저하되었고, 10개 더 늘렸더니 정확도에는 변화가 없었다.
- hidden layer마다 unit 수를 각각 다르게 세팅하였을때. Train accuracy랑 test accuracy의 차이가 0.1로서, overfitting이 심했다.
- 위의 Test를 통해 필요 없는 layer 수와, unit 수들을 없앴다.

### 6.2.2) Optimizer test.

#### 6.2.3.1) Epoch 수, Learning rate test for

##### Adagrad:

- 기본적으로, 여러 learning rate로 하여도, accuracy를 올리기 위해 많은 수의 epoch가 필요했다. Train accuracy = 0.754에서 0.812로 올리는데 약 3000번 정도의 epoch가 필요하였다.
- 이는, h값이 어느순간 0에 가까워지기 때문에 lr 갱신강도가 약해지는 Adagrad의 약점이었고, 이를 개선하기 위해 설계된 3개의 optimizer를 구현 후, 이용하여 다시 test해보았다.

##### Adadelata, Adam:

-Test 결과, Adadelata, Adam의 공통점:

- 10의 거듭제곱 단위로 learning rate test시, 아예 학습이 안되거나 accuracy에 변화가 없었다.
- 빠른 epoch (400)안에 어느정도 accuracy에 도달함을 볼 수 있었지만 어느 순간 accuracy가 오르지 않았다. Test accuracy = 0.752

##### -Adamax:

- 초기 설정 시 learning rate가 0.001 단위로 결과에 영향을 많이 주는 것을 확인할 수 있었다.
- epoch 500 이하시에도, test accuracy 0.758로 평균적으로 앞의 두개의 optimizer보다 좋은 성과를 내었다.

```

class Adadelta:
    def __init__(self, lr):
        self.lr = lr
        self.g = None
        self.s = None
        self.d = None

    def update(self, params, grads):
        if self.g is None:
            self.g = {}
            for key, val in params.items():
                self.g[key] = np.zeros_like(val)
        if self.s is None:
            self.s = {}
            for key, val in params.items():
                self.s[key] = np.zeros_like(val)
        if self.d is None:
            self.d = {}
            for key, val in params.items():
                self.s[key] = np.zeros_like(val)

        for key in params.keys():
            self.g[key] = (0.99) * self.g[key] + (1 - 0.99) * (grads[key] * grads[key])
            self.d[key] = -1 * np.sqrt(self.s[key] + 1e-7) * grads[key] / np.sqrt(self.g[key] + 1e-7)
            self.s[key] = (0.99) * self.s[key] + (1 - 0.99) * (self.d[key] * self.d[key])
            params[key] += self.d[key]

```

Figure 2. Adadelta

```

class Adam:
    def __init__(self, lr):
        self.lr = lr
        self.m = None
        self.v = None
        self.b1 = 0.9
        self.b2 = 0.999
        self.t = 0

    def update(self, params, grads):
        if self.m is None:
            self.m = {}
            for key, val in params.items():
                self.m[key] = np.zeros_like(val)
        if self.v is None:
            self.v = {}
            for key, val in params.items():
                self.v[key] = np.zeros_like(val)

        for key in params.keys():
            self.t += 1
            self.m[key] = (self.b1 * self.m[key]) + ((1 - self.b1) * grads[key])
            self.v[key] = (self.b2 * self.v[key]) + ((1 - self.b2) * grads[key] * grads[key])
            self.m[key] = self.m[key] / (1 - (self.b1 ** self.t))
            self.v[key] = self.v[key] / (1 - (self.b2 ** self.t))
            params[key] -= self.lr * self.m[key] / (np.sqrt(self.v[key] + 1e-8))

```

Figure 3. Adam

#### 6.2.4) Overfitting: Dropout ratio test

- Dropout ratio 0.5 test시 train accuracy = 0.83, test accuracy = 0.754
- Dropout ratio 0.2 test시 train accuracy = 0.625, test accuracy = 0.622
- Dropout ratio가 적을수록 학습이 잘 안되었고, 올릴수록 overfitting이 완화되지 않았다.
- Dropout을 모든 hidden layer에 적용하지 말고, unit수가 큰 layer에만 적용시, overfitting은 완화되었지만, training accuracy가 더 이상 증가하지 않았다.
- Epoch 2000번 동안 train accuracy: 0.773, test accuracy: 0.754

#### 6.2.5) Overfitting: L2 regularization(weight decay) parameter test

- 6.2.4)의 문제를 해결하기 위해, 다른 overfitting 해결 방법인 Weight decay를 적용하였으며, 10의 거듭제곱단위로 test를 진행하였다.
- 결과적으로, overfitting도 줄면서, test accuracy 또한 0.768로 증가하였다.

## 7. 결과 및 분석

### 7.1) Hyperparameter controlled:

1. learning rate, 2. Adamax(parameter b1, b2), 3. Weight decay parameter 4. hidden layer units.

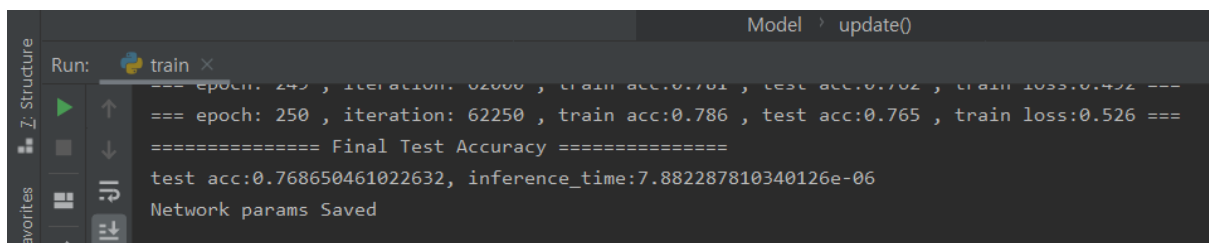


5. Layer 수, 6. Dropout ratio

## 7.2) 분석

- Network에 있어서, hidden layer 수와 그에 따른 unit수는 충분히 있어야 학습 정확도가 나오며, 너무 많게 되면 무의미하게 network를 키우고 계산시간만 늘릴 수도 있다.
- 설계한 network에서 learning rate는 상당한 시간을 들여서 test해보야하고, 그 network에 어울리는 특정 값이 있기도 하다.
- Optimizer 종류에 따라서 불필요한 epoch값들을 줄일 수 있으며, 각 optimizer들의 성능은, 어느 것이 더 좋다고 말하기 어렵다.

## 7.3) 최종 결과



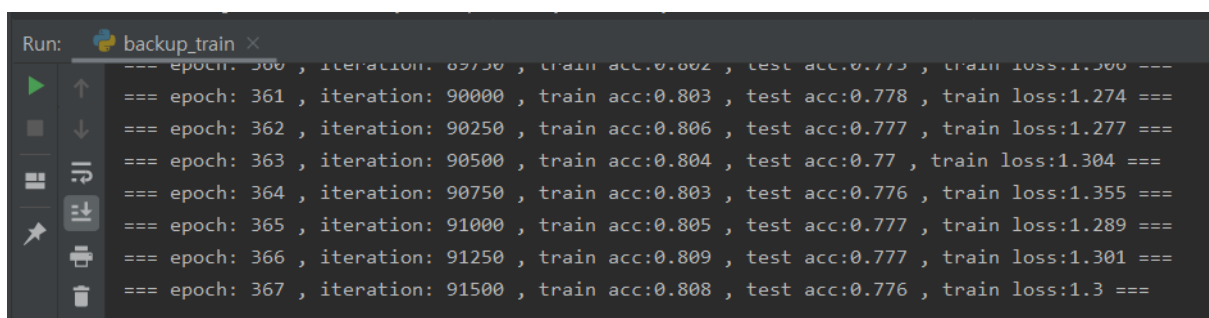
```
Run: train x
Model > update()
==== epoch: 249 , iteration: 62000 , train acc:0.781 , test acc:0.782 , train loss:0.472 ===
==== epoch: 250 , iteration: 62250 , train acc:0.786 , test acc:0.765 , train loss:0.526 ===
===== Final Test Accuracy =====
test acc:0.768650461022632, inference_time:7.882287810340126e-06
Network params Saved
```

-최종적으로, 초기 Epoch 1000번의 test accuracy 0.64에서,

Epoch 250번의 test accuracy = 0.768을 이뤄냈다.

-추가)

Data를 import할때, standardize = True로 하여 가져온 data로 training 진행 시, test accuracy가 0.01이 증가한 0.777을 기록할 수 있었다.



```
Run: backup_train x
==== epoch: 360 , iteration: 89750 , train acc:0.802 , test acc:0.775 , train loss:1.300 ===
==== epoch: 361 , iteration: 90000 , train acc:0.803 , test acc:0.778 , train loss:1.274 ===
==== epoch: 362 , iteration: 90250 , train acc:0.806 , test acc:0.777 , train loss:1.277 ===
==== epoch: 363 , iteration: 90500 , train acc:0.804 , test acc:0.77 , train loss:1.304 ===
==== epoch: 364 , iteration: 90750 , train acc:0.803 , test acc:0.776 , train loss:1.355 ===
==== epoch: 365 , iteration: 91000 , train acc:0.805 , test acc:0.777 , train loss:1.289 ===
==== epoch: 366 , iteration: 91250 , train acc:0.809 , test acc:0.777 , train loss:1.301 ===
==== epoch: 367 , iteration: 91500 , train acc:0.808 , test acc:0.776 , train loss:1.3 ===
```