인공지능 학기말 프로젝트

과제 제목: 딥러닝 모델 학습

학번: B511157
이름: 이현수

1. 과제 개요

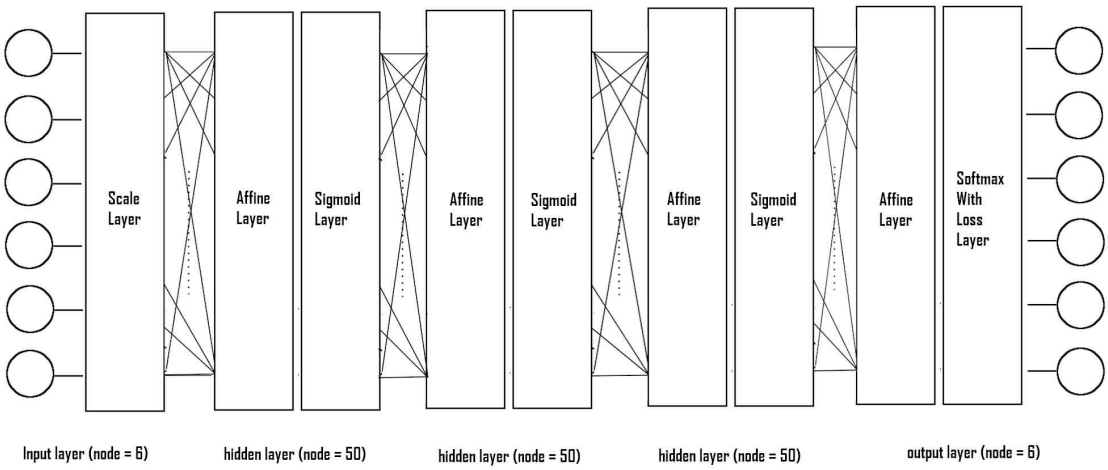
- Python library를 사용하는 것이 아닌 딥러닝 모델을 직접 구현하여 사용되는 여러 알고리즘과 activation, optimizer에 대한 이해를 높이는 것을 목표로 한다.
- 주어진 data set을 사용하여 학습시킨 후 보다 높은 정확도를 달성하는 것을 목표로 하여 실제 데이터 학습에 영향을 끼칠 수 있는 요소와 사용하는 activation, optimizer들의 차이를 이해하여 적절한 알고리즘을 적용시킬 수 있도록 분석해 본다.

2. 구현 환경

Python3 windows 10 jupyter notebook, idle

3. 알고리즘에 대한 설명

아래의 그림은 구현한 딥러닝 모델을 보기 쉽도록 시각화 한 것이다. input layer - hidden layer 3층 - output layer로 구성되어 있으며 hidden layer의 노드수는 [50,50,50]이다. input layer에는 scaler layer가 있으며 이후 affine layer, activation layer(sigmoid) 순으로 구성되어 진행된다. 마지막 output layer에는 softmax with loss layer가 존재하여 classification의 loss를 구함과 동시에 output y를 계산해낸다.



각 layer 에는 input x를 이용하여 다음 layer로 보낼 output을 계산하는 forward method가 있으며 이후 계산한 y에 대한 loss를 이전 layer로 전달하는 backward propagation이 존재한다. 따라서 모델은 한번 학습할 때 forward 를 따라 y값을 계산해낸 뒤 바로 그 계산한 y를 이용하여 loss를 전달하며 W값과 bias값을 수정한다. 이러한 과정을 여러 번 진행하여 적절한 w와 bias 값을 찾아간다.

Hidden layer의 개수와 노드수는 실험적으로 여러 번의 train을 거친 후 결정하였으며 자세한 실험 과정은 결과 및 분석에서 볼 수 있다. Activation function은 뒤의 layer에게 input값을 보낼 강도를 결정하게 되며 Activation으로 무슨 함수를 사용하느냐에 따라 정확

도가 달라질 수 있다. 딥러닝으로 많이 사용하는 activation은 Relu함수이다. 하지만 이 과제에서는 Sigmoid를 사용하였으며 그 이유는 vanishing gradient가 생길만큼 model의 층이 깊지 않고 무엇보다 Sigmoid 와 Relu로 각각 실험을 진행해 본 결과 Sigmoid의 정확도가 더 높게 나왔기 때문이다. optimizer로는 Adam optimizer를 사용하였다. Adam optimizer는 momentum 방식과 Rmsprop방식을 혼합한 방식으로 momentum의 이전의 gradient의 일정 %만큼 반영하여 관성 모멘트와 같은 효과를 주는 개념과 Rmsprop의 learning rage를 조절해준다는 개념을 같이 적용시킨 알고리즘이다.

Scaler layer는 standard scaling을 진행하며 다른 layer들과의 형식을 맞춰주기 위해 forward backward를 구현하였다. forward는 scaling 진행, backward는 아무것도 하지 않는다. scaling을 해준 이유는 input 변수들 간의 범위 차이가 학습에 영향을 끼칠 수 있기 때문에 이러한 변수들 간의 범위를 맞춰주기 위해 진행하였다. scaling에는 standard scaling과 min max scaling을 사용하였는데 결과 및 분석에서 보다시피 standard scaling의 결과가 더 좋게 나왔기에 이를 사용하였다.

위의 hidden layer 수와 node 수, activation과 optimizer 및 scaling 방법 차이에 대한 자세한 결과는 아래의 결과 및 분석에서 볼 수 있다.

4. 데이터에 대한 설명

AReM data

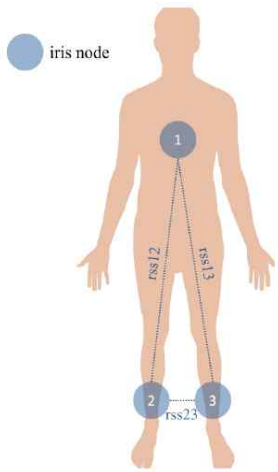
총 데이터 수 : 33406

walking	standing	sitting	lying	cycling	bending
4291	4301	4311	4342	4408	3402

Train 데이터 수 : 25055

Test 데이터 수 : 8351

4.1 Input Feature



Input class 수 : 6

Data type : Float

Input data : 왼쪽 사진의 센서값을 나타내며 각각 avg_rss12, var_rss12, avg_rss 13, avg_rss23, var_rss23 으로 센서값의 평균과 분산을 나타낸다.

4.2 Target Output

Output class 수 : 6

Data type : numeric

Output data : {0: walking, 1: standing, 2: sitting. 3: lying, 4: cycling. 5: bending}

5. 소스코드에 대한 설명

```
class MOpt: #momentum optimizer입니다.
    def __init__(self,lr = 0.01,momentum = 0.9):
        self.lr = lr
        self.v = None
        self.momentum = momentum
        self.saved = {}
    def update(self,params,grads):
        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.v[key] = self.momentum*self.v[key] - self.lr*grads[key]
            params[key] += self.v[key] #momentum 정도는 일반적으로 많이 ㅅ
```

위는 코드안의 momentum optimizer를 구현해놓은 부분이다.
momentum은 이전 gradient 값의 일정 %를 반영하여 gradient를 적용하는 방식으로 관성 모멘트와 같은 효과를 주는 알고리즘이다.
self.v 변수가 이전까지의 gradient값들의 일정 %를 반영해 놓은 값을 저장하고 있다.
Self.momentum = 0.9로 이전 gradient의 90%를 반영하겠다는 의미이며 매 iteration마다 v값에 momentum을 곱한값에 현재 gradient * learnning rate를 곱하여 갱신한다. 이후 갱신한 v를 이용하여 params의 W,b를 update 해준다.

```
class Adam: #adam optimizer입니다. momentum 과 rmsprop를 합쳐논 개념입니다.
    def  __init__(self,lr = 0.01):
        self.lr = lr
        self.v = None
        self.h = None
        self.momentum = 0.9 #momentum 과 decay 입니다.
        self.decay = 0.999
        self.iter = 1
        self.saved = {}
    def update(self,params,grads): #adam 구현은 논문의 psudo 코드를 보고 구현하였으며 ㄴ
        if self.v is None:
            self.v = {} #속도 v와 이전 기울기값 h를 저장하기 위한 변수입니다.
            self.h = {}
            for key,val in params.items():
                self.v[key] = np.zeros_like(val)
                self.h[key] = np.zeros_like(val)
                self.saved[key] = []
        self.iter += 1 #초반에 w값이 작게증가하는 adam의 단점을 해결하기 위해 epoch 정ㄴ
        for key in params.keys():
            if key == 'mean' or key == 'std': #params들 중 mean과 std는 건너 뜁니다.
                continue
            self.v[key] = (self.momentum*self.v[key] + (1-self.momentum) * grads[key])
            tmpv = self.v[key]/(1-self.momentum**self.iter) #epoch 정보를 이용하여 초반
            self.h[key] = (self.decay*self.h[key] + (1-self.decay)* (grads[key]**2))
            tmph = self.h[key]/(1-self.decay**self.iter) #위의 v와 마찬가지로 초반 속도
            params[key] -= (self.lr*tmpv)/(np.sqrt(tmph) + 1e-7) #분모가 0이 되는것을 ㅁ
```

코드상의 Adam optimizer를 구현해놓은 부분이다.
Adam optimizer는 2개의 momentum을 가지며 제 1 모멘텀, 제 2 모멘텀이라 불린다. 위에서 v가 제 1모멘텀에 해당하는 값이며 h가 제 2 모멘텀에 해당하는 값이다. momentum과 Rmsprop의 개념을 합쳤다는 것을 쉽게 이해하기 위해 v와 h로 사용하였다.
v를 갱신하는 부분인 {self.v[key] = (self.momentum*slef ….} 부분은 앞의 momentum과

비슷하게 이전 gradient값의 일정 %(momentum)을 반영하겠다는 의미이며 뒤의 (1-self.momentum) * grad[key]) 부분은 현재 gradient 값의 1-momentum 만큼의 %를 반영하겠다는 의미이다. 따라서 이전의 v에는 계속 momentum이 곱해지며 점점 작아지게 된다. 이는 Rmsprop에 모멘텀의 개념을 포함시킨 것이 된다. 이후 self.h에 해당하는 갱신식인 {self.h[key] = (self.decay*self.h[key] + …} 부분은 Rmsprop의 이전 기울기 값을 이용하여 learning rate를 조절하는 개념으로 이전 기울기의 제곱값들을 decay 만큼 곱해주며 점점 작게 만들며 현재 gradient의 제곱값을 1-decay의 값으로 반영하겠다는 의미이다.

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

위의 사진은 Adam optimizer 논문에 나와있는 pseudo-code 이다. 각각의 수식은 위에서 설명하였으며 설명하지 않았던 $(1 - \beta_1^t)$ 부분은 위의 코드에서 tmpv와 tmph 계산의 1-momentum(or decay)**self.iter에 해당한다. 이는 momentum과 Rmsprop에서 각각의 v,h가 최초 0으로 설정되어 학습 초반에 0으로 biased 문제를 해결하기 위함이다.

6. 학습 과정에 대한 설명

학습은 input data에 대해 model 내부에서 각 layer들의 forward를 진행한 후 마지막 layer에서 나온 softmax_with_loss 에서의 backward 로부터 이전 layer들로 loss를 전달하며 진행되는 back propagation으로 진행된다. back propagation으로 전달받은 dout들을 이용하여 W와 b의 gradient값을 구하고 이후 optimizer가 gradient값을 이용하여 W,b를 갱신해 주게 된다. optimzer로 Adam을 사용하였기 때문에 최저점으로 도달하는데 SGD처럼 local minimum에 빠지거나 평평한 곳을 만나 학습이 진행되기 어려운 부분이 적어져 빠르게 최저점을 향해 나아갈 수 있다. loss 함수의 진행은 결과 및 분석을 보면 알 수 있다.

7. 결과 및 분석

우선 hidden layer 30,30으로 진행하였다. Adam과 sigmoid Activation function을 사용하였으며 결과는 다음과 같다.

```
=== epoch: 1 , iteration: 0 , train acc:0.039 , test acc:0.037 , train loss:11.81 ===
=== epoch: 2 , iteration: 250 , train acc:0.627 , test acc:0.619 , train loss:0.791 ===
=== epoch: 3 , iteration: 500 , train acc:0.615 , test acc:0.615 , train loss:0.981 ===
=== epoch: 4 , iteration: 750 , train acc:0.627 , test acc:0.621 , train loss:0.723 ===
=== epoch: 5 , iteration: 1000 , train acc:0.633 , test acc:0.626 , train loss:0.83 ===

=== epoch: 495 , iteration: 123500 , train acc:0.772 , test acc:0.749 , train loss:0.524 ===
=== epoch: 496 , iteration: 123750 , train acc:0.761 , test acc:0.742 , train loss:0.509 ===
=== epoch: 497 , iteration: 124000 , train acc:0.772 , test acc:0.75 , train loss:0.507 ===
=== epoch: 498 , iteration: 124250 , train acc:0.772 , test acc:0.744 , train loss:0.439 ===
=== epoch: 499 , iteration: 124500 , train acc:0.766 , test acc:0.749 , train loss:0.449 ===
=== epoch: 500 , iteration: 124750 , train acc:0.771 , test acc:0.751 , train loss:0.547 ===
============== Final Test Accuracy ==============
test acc:0.7493713327745181, inference_time:9.553872986534172e-07
```

(hidden layer 2 [30,30] adam optimizer)

정확도가 만족스럽지 못란 결과가 나왔다. 이에 input으로 들어오는 x변수들 사이의 범위의 차이가 있어 학습이 제대로 진행되지 않는것이 아닐까라는 생각을 하게되어 머신러닝에서 주로 사용하는 scaling을 진행하였다. scaling에는 standard scaling과 min max scailing을 진행하였으며 다음과 같다.

```
=== epoch: 1 , iteration: 0 , train acc:0.172 , test acc:0.17 , train loss:1.909 ===
=== epoch: 2 , iteration: 250 , train acc:0.604 , test acc:0.609 , train loss:0.945 ===
=== epoch: 3 , iteration: 500 , train acc:0.649 , test acc:0.649 , train loss:0.792 ===
=== epoch: 4 , iteration: 750 , train acc:0.662 , test acc:0.658 , train loss:0.907 ===
=== epoch: 5 , iteration: 1000 , train acc:0.67 , test acc:0.664 , train loss:0.789 ===

=== epoch: 495 , iteration: 123500 , train acc:0.801 , test acc:0.776 , train loss:0.357 ===
=== epoch: 496 , iteration: 123750 , train acc:0.801 , test acc:0.778 , train loss:0.542 ===
=== epoch: 497 , iteration: 124000 , train acc:0.802 , test acc:0.776 , train loss:0.4 ===
=== epoch: 498 , iteration: 124250 , train acc:0.802 , test acc:0.78 , train loss:0.543 ===
=== epoch: 499 , iteration: 124500 , train acc:0.802 , test acc:0.777 , train loss:0.419 ===
=== epoch: 500 , iteration: 124750 , train acc:0.801 , test acc:0.778 , train loss:0.468 ===
============== Final Test Accuracy ==============
test acc:0.7790683750449048, inference_time:1.194262673020891e-06
```

(hidden layer 2 [30,30] adam optimizer with standard scaling)

```
=== epoch: 1 , iteration: 0 , train acc:0.171 , test acc:0.177 , train loss:1.965 ===
=== epoch: 2 , iteration: 250 , train acc:0.332 , test acc:0.313 , train loss:1.772 ===
=== epoch: 3 , iteration: 500 , train acc:0.275 , test acc:0.229 , train loss:1.743 ===

=== epoch: 499 , iteration: 124500 , train acc:0.722 , test acc:0.692 , train loss:0.638 ===
=== epoch: 500 , iteration: 124750 , train acc:0.72 , test acc:0.693 , train loss:0.518
=== =============== Final Test Accuracy ===============
test acc:0.6936893785175428, inference_time:2.5880592281326476e-06
```

(hidden layer 2 [30,30] adam optimizer with min-max scaling)

결과에서 보다시피 확실히 scailing을 추가한 model의 정확도가 더 높았으며 min max scailing보단 standard scailing이 더 좋은 결과가 나옴을 확인하였다. 따라서 이후의 시험엔 standard scailing을 추가하여 진행하였다.

Model의 hidden layer 수와 node수를 결정하기 위해 3개의 서로다른 경우를 시도해보았다. 각각 hidden layer 2층 [30,30], hidden layer 2층 [50,50], hidden layer 3층 [50,50,50] 이며 []안의 수는 노드수를 의미한다. 또한 layer수와 node수에 이어 Activation function또한 정확도에 영향을 끼칠수 있으므로 위의 경우들을 Momentum, Rmsprop, Adam optimizer으로 실험해 보았으며 결과는 다음과 같으며 활성화 함수로는 Sigmoid를 사용했다.

=== epoch: 1 , iteration: 0 , train acc:0.171 , test acc:0.177 , train loss:2.027 ===
=== epoch: 2 , iteration: 250 , train acc:0.246 , test acc:0.25 , train loss:1.758 ===
=== epoch: 3 , iteration: 500 , train acc:0.351 , test acc:0.352 , train loss:1.755 ===
=== epoch: 4 , iteration: 750 , train acc:0.411 , test acc:0.409 , train loss:1.725 ===
=== epoch: 5 , iteration: 1000 , train acc:0.413 , test acc:0.413 , train loss:1.656 ===

=== epoch: 495 , iteration: 123500 , train acc:0.72 , test acc:0.713 , train loss:0.568 ===
=== epoch: 496 , iteration: 123750 , train acc:0.721 , test acc:0.714 , train loss:0.631 ===
=== epoch: 497 , iteration: 124000 , train acc:0.72 , test acc:0.715 , train loss:0.728 ===
=== epoch: 498 , iteration: 124250 , train acc:0.72 , test acc:0.712 , train loss:0.582 ===
=== epoch: 499 , iteration: 124500 , train acc:0.72 , test acc:0.712 , train loss:0.625 ===
=== epoch: 500 , iteration: 124750 , train acc:0.72 , test acc:0.715 , train loss:0.777 ===
=============== Final Test Accuracy ===============
test acc:0.7111723146928511, inference_time:1.194205573612652e-06

(hidden layer 2 [30,30] Momentum optimizer)

=== epoch: 1 , iteration: 0 , train acc:0.172 , test acc:0.17 , train loss:1.755 ===
=== epoch: 2 , iteration: 250 , train acc:0.62 , test acc:0.616 , train loss:0.98 ===
=== epoch: 3 , iteration: 500 , train acc:0.648 , test acc:0.649 , train loss:0.835 ===
=== epoch: 4 , iteration: 750 , train acc:0.666 , test acc:0.664 , train loss:0.831 ===
=== epoch: 5 , iteration: 1000 , train acc:0.668 , test acc:0.661 , train loss:0.693 ===

=== epoch: 495 , iteration: 123500 , train acc:0.796 , test acc:0.772 , train loss:0.438 ===
=== epoch: 496 , iteration: 123750 , train acc:0.794 , test acc:0.768 , train loss:0.613 ===
=== epoch: 497 , iteration: 124000 , train acc:0.793 , test acc:0.768 , train loss:0.504 ===
=== epoch: 498 , iteration: 124250 , train acc:0.792 , test acc:0.765 , train loss:0.339 ===
=== epoch: 499 , iteration: 124500 , train acc:0.795 , test acc:0.772 , train loss:0.62 ===
=== epoch: 500 , iteration: 124750 , train acc:0.792 , test acc:0.765 , train loss:0.544 ===
=============== Final Test Accuracy ===============
test acc:0.7654173152915819, inference_time:1.194262673020891e-06

(hidden layer 2 [30,30] Rmsprop optimizer)

위의 adam with scaling과 동일합니다.

(hidden layer 2 [30,30] Adam optimizer)

이후 Adam optimizer가 가장 정확도가 높아 adam 의 결과만 표시하였습니다.

=== epoch: 1 , iteration: 0 , train acc:0.173 , test acc:0.169 , train loss:1.799 ===
=== epoch: 2 , iteration: 250 , train acc:0.638 , test acc:0.635 , train loss:0.957 ===
=== epoch: 3 , iteration: 500 , train acc:0.66 , test acc:0.658 , train loss:0.72 ===
=== epoch: 4 , iteration: 750 , train acc:0.669 , test acc:0.662 , train loss:0.829 ===
=== epoch: 5 , iteration: 1000 , train acc:0.669 , test acc:0.664 , train loss:0.753 ===

=== epoch: 495 , iteration: 123500 , train acc:0.817 , test acc:0.779 , train loss:0.489 ===
=== epoch: 496 , iteration: 123750 , train acc:0.819 , test acc:0.781 , train loss:0.504 ===
=== epoch: 497 , iteration: 124000 , train acc:0.82 , test acc:0.78 , train loss:0.523 ===
=== epoch: 498 , iteration: 124250 , train acc:0.82 , test acc:0.778 , train loss:0.336 ===
=== epoch: 499 , iteration: 124500 , train acc:0.816 , test acc:0.776 , train loss:0.337 ===
=== epoch: 500 , iteration: 124750 , train acc:0.818 , test acc:0.779 , train loss:0.521 ===
=============== Final Test Accuracy ===============
test acc:0.77966710573584, inference_time:1.6719563223475997e-06

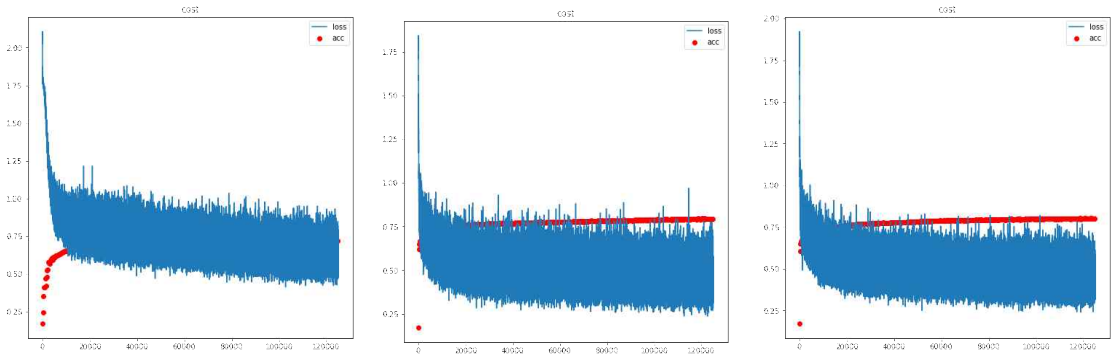
(hidden layer 2 [50,50] Adam optimizer)

=== epoch: 1 , iteration: 0 , train acc:0.173 , test acc:0.169 , train loss:1.843 ===
=== epoch: 2 , iteration: 250 , train acc:0.608 , test acc:0.607 , train loss:0.887 ===
=== epoch: 3 , iteration: 500 , train acc:0.645 , test acc:0.65 , train loss:0.933 ===
=== epoch: 4 , iteration: 750 , train acc:0.665 , test acc:0.664 , train loss:0.794 ===
=== epoch: 5 , iteration: 1000 , train acc:0.674 , test acc:0.669 , train loss:0.775 ===

=== epoch: 495 , iteration: 123500 , train acc:0.834 , test acc:0.779 , train loss:0.352 ===
=== epoch: 496 , iteration: 123750 , train acc:0.836 , test acc:0.778 , train loss:0.371 ===
=== epoch: 497 , iteration: 124000 , train acc:0.834 , test acc:0.778 , train loss:0.397 ===
=== epoch: 498 , iteration: 124250 , train acc:0.836 , test acc:0.784 , train loss:0.482 ===
=== epoch: 499 , iteration: 124500 , train acc:0.834 , test acc:0.784 , train loss:0.508 ===
=== epoch: 500 , iteration: 124750 , train acc:0.834 , test acc:0.78 , train loss:0.444 ===
=============== Final Test Accuracy ===============
test acc:0.7844569512633217, inference_time:2.3885538957459015e-06

(hidden layer 3 [50,50,50] Adam optimizer)

Momentum rmsprop adam



위의 결과에서 보다시피 30,30 이 확실히 node수가 적어 가장 빠르지만 정확도는 보다

node수가 많은 50,50 과 50,50,50 보다 떨어진다. 3가지의 Activation 중 Momentum의 경우 Adam, Rmsprop보다 떨어지는 결과가 보이며 Adam과 rmsprop는 서로 거의 비슷한 결과가 나왔다. 하지만 50,50,50에서는 Adam이 약 1퍼트정도 안정적으로 보여 최종적으로 hidden layer 3층 [50,50,50], Adam optimizer를 선택하게 되었다.
Activation function를 무엇을 사용하냐에 따라 결과가 달라질 수 있다.따라서 Activation function으로 Relu 와 Sigmoid를 사용하여 비교해 보았다.(LRelu는 Relu와 결과가 거의 비슷하여 생략하였다) 결과는 다음과 같다.

```
=== epoch: 1 , iteration: 0 , train acc:0.168 , test acc:0.162 , train loss:1.752 ===
=== epoch: 2 , iteration: 250 , train acc:0.682 , test acc:0.671 , train loss:0.615 ===
=== epoch: 3 , iteration: 500 , train acc:0.713 , test acc:0.705 , train loss:0.667 ===
=== epoch: 4 , iteration: 750 , train acc:0.72 , test acc:0.712 , train loss:0.678 ===
=== epoch: 5 , iteration: 1000 , train acc:0.722 , test acc:0.711 , train loss:0.742 ===

=== epoch: 495 , iteration: 123500 , train acc:0.791 , test acc:0.762 , train loss:0.474 ===
=== epoch: 496 , iteration: 123750 , train acc:0.794 , test acc:0.77 , train loss:0.501 ===
=== epoch: 497 , iteration: 124000 , train acc:0.794 , test acc:0.764 , train loss:0.54 ===
=== epoch: 498 , iteration: 124250 , train acc:0.791 , test acc:0.763 , train loss:0.481 ===
=== epoch: 499 , iteration: 124500 , train acc:0.795 , test acc:0.765 , train loss:0.436 ===
=== epoch: 500 , iteration: 124750 , train acc:0.793 , test acc:0.764 , train loss:0.48 ===
=============== Final Test Accuracy ===============
test acc:0.7606274697641001, inference_time:2.1533043338015982e-06
```

(hidden layer 3 [50,50,50] Adam optimizer with Relu)

#sigmoid의 결과 값은 위의 layer node수 실험에서의 결과와 같습니다.

Sigmoid 함수가 Relu함수에 비해 더 높은 정확도를 보였다. 따라서 Sigmoid 함수를 사용하기로 하였다.
위의 layer와 옵티마이저 실험을 보면 epoch가 진행될수록 train acc가 test acc에 비해 너무 높게 올라가는 overfitting이 발생하게 되었다. epoch가 진행됨에 따라 overfitting의 정도가 심해지며 오히려 train acc가 떨어지게 되는 현상을 보았으며 이에 drop out layer를 추가하여 overfitting을 막고, 안정적으로 학습을 진행하려 하였다. 결과는 다음과 같다.

```
=== epoch: 1 , iteration: 0 , train acc:0.171 , test acc:0.171 , train loss:1.937 ===
=== epoch: 2 , iteration: 250 , train acc:0.526 , test acc:0.535 , train loss:1.033 ===
=== epoch: 3 , iteration: 500 , train acc:0.602 , test acc:0.608 , train loss:0.844 ===
=== epoch: 4 , iteration: 750 , train acc:0.626 , test acc:0.627 , train loss:0.965 ===
=== epoch: 5 , iteration: 1000 , train acc:0.641 , test acc:0.642 , train loss:0.883 ===

=== epoch: 495 , iteration: 123500 , train acc:0.777 , test acc:0.757 , train loss:0.53 ===
=== epoch: 496 , iteration: 123750 , train acc:0.777 , test acc:0.755 , train loss:0.508 ===
=== epoch: 497 , iteration: 124000 , train acc:0.776 , test acc:0.754 , train loss:0.585 ===
=== epoch: 498 , iteration: 124250 , train acc:0.776 , test acc:0.753 , train loss:0.508 ===
=== epoch: 499 , iteration: 124500 , train acc:0.775 , test acc:0.756 , train loss:0.529 ===
=== epoch: 500 , iteration: 124750 , train acc:0.775 , test acc:0.755 , train loss:0.423 ===
=============== Final Test Accuracy ===============
test acc:0.7555981319602443, inference_time:5.276413566836938e-06
```

(hidden layer 3 [50,50,50] Adam optimizer with Dropout)

=== epoch: 1 , iteration: 0 , train acc:0.171 , test acc:0.171 , train loss:1.967 ===
=== epoch: 2 , iteration: 250 , train acc:0.451 , test acc:0.454 , train loss:1.1 ===
=== epoch: 3 , iteration: 500 , train acc:0.586 , test acc:0.587 , train loss:0.924 ===
=== epoch: 4 , iteration: 750 , train acc:0.618 , test acc:0.619 , train loss:0.851 ===
=== epoch: 5 , iteration: 1000 , train acc:0.639 , test acc:0.641 , train loss:0.744 ===

=== epoch: 495 , iteration: 123500 , train acc:0.761 , test acc:0.743 , train loss:0.671 ===
=== epoch: 496 , iteration: 123750 , train acc:0.759 , test acc:0.74 , train loss:0.559 ===
=== epoch: 497 , iteration: 124000 , train acc:0.76 , test acc:0.737 , train loss:0.468 ===
=== epoch: 498 , iteration: 124250 , train acc:0.757 , test acc:0.743 , train loss:0.509 ===
=== epoch: 499 , iteration: 124500 , train acc:0.762 , test acc:0.743 , train loss:0.579 ===
=== epoch: 500 , iteration: 124750 , train acc:0.764 , test acc:0.745 , train loss:0.398 ===
=============== Final Test Accuracy ===============
test acc:0.7354807807448209, inference_time:2.5079773080775787e-06

(hidden layer 3 [50,50,50] Rmsprop optimizer with Dropout)

드롭아웃을 진행 후 train acc도 이전처럼 많이 올라가진 않았지만 test acc또한 떨어지는 것을 보았다. 드롭아웃의 비율을 변경하며 실험을 진행하여도 test acc가 이전처럼 높게 나오지는 않았다.따라서 overfitting을 막기위해 드롭아웃을 추가하기 보단 어느 정도의 overfitting을 감수하고 높은 test acc를 유지하는 방향으로 진행하기로 하였다. 따라서 train이 진행됨에 따라 loss값과 acc값을 보며 overfitting이 심하게 일어나며 test acc가 역전되는 부분을 찾은뒤 그전까지만 epoch를 수행하는 방식으로 하였다.

이후 최적의 파라미터(lr)을 찾기 위한 하이퍼 파라미터 탐색을 진행하였다. 초기에 10^(-9) ~ 10^(-1) 범위에서 20회를 시도하였으며 이후 좋게 나왔던 부분으로 범위를 좁혀 좁혀 10^(-5)~10^(-1)으로 진행하였다.

하이퍼 파라미터 탐색 결과 가장 정확도가 높았던 lr=0.0017019024512130883 을 사용하였으며 여기서 찾은 lr을 이용하여 위의 실럼을 다시 반복하였다. 결과파일이 너무 많아져 위의 실험결과들은 모두 여기서 찾은 lr을 사용한 실험 결과들만 나타내었다.

따라서 최종적으로 hidden layer 3 [50,50,50], Adam Optimizer, Sigmoid, standard scaling, lr=0.0017019024512130883 의 모델을 사용하였다.