

source code는 <http://yann.lecun.com/exdb/mnist/>에서 train-images-idx3-ubyte.gz, train-labels-idx1-ubyte.gz, t10k-images-idx3-ubyte.gz, t10k-labels-idx1-ubyte.gz 파일을 다운받아서 아래와 같이 Colab 환경 기준으로 파일을 올린 후, file_path에 해당 경로를 수정한 후 사용할 수 있습니다.



The screenshot shows a Google Colab notebook titled 'HW1_내가.ipynb'. The left sidebar displays the file explorer with a folder named 'sample_data' containing four files: 't10k-images-idx3-ubyte.gz', 't10k-labels-idx1-ubyte.gz', 'train-images-idx3-ubyte.gz', and 'train-labels-idx1-ubyte.gz'. The main code area contains the following Python code:

```
import urllib.request
import os.path
import gzip
import pickle
import os
import numpy as np
import sys

file_path = ['./train-images-idx3-ubyte.gz', './train-labels-idx1-ubyte.gz', './t10k-images-idx3-ubyte.gz', './t10k-labels-idx1-ubyte.gz']

def file_load(path):
    if path.find('images') != -1:
        with gzip.open(path, 'rb') as f:
            data = np.frombuffer(f.read(), np.uint8, offset=16)
            data = data.reshape(-1, 784)
            return data
    else:
        with gzip.open(path, 'rb') as f:
```

1.

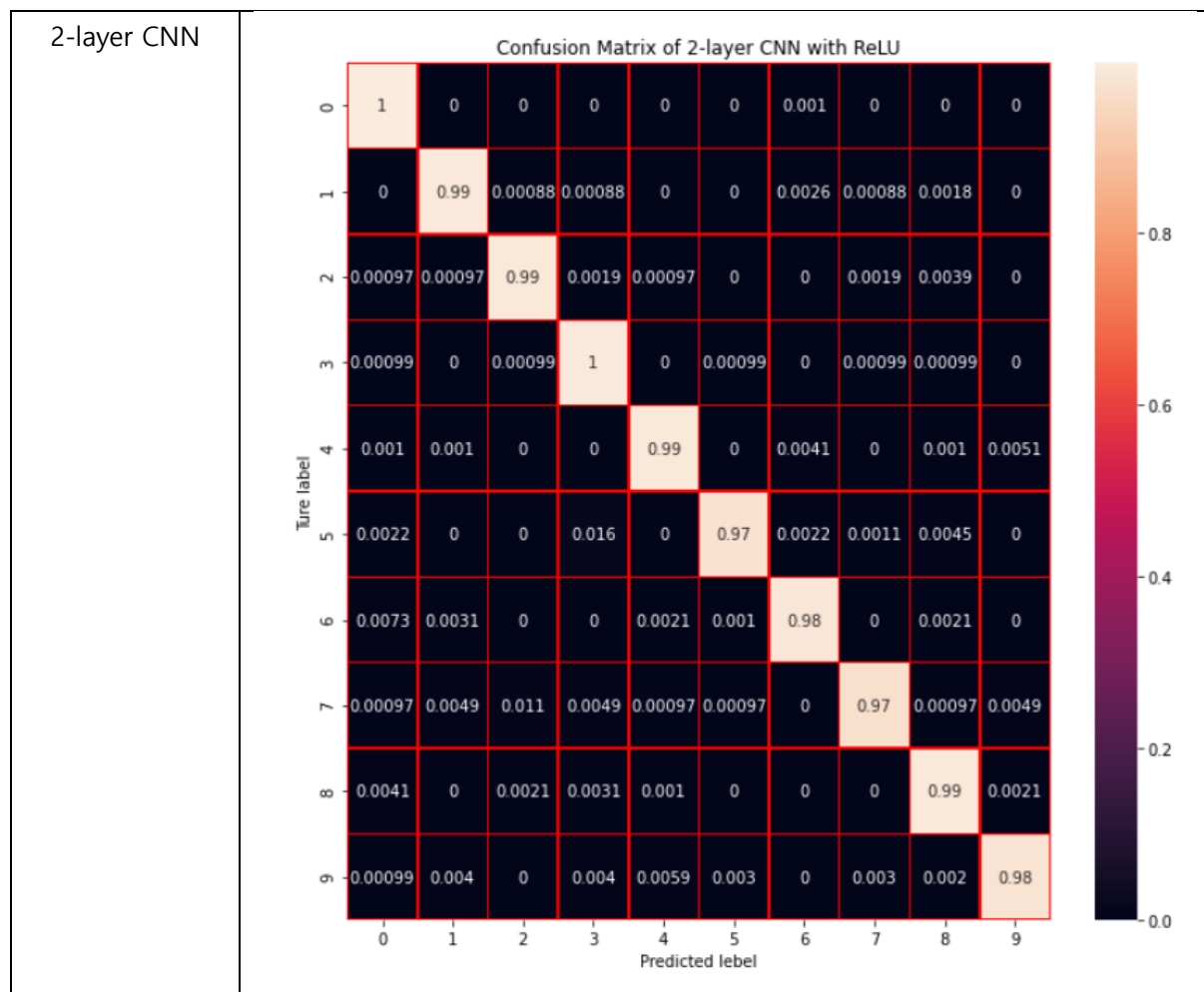
- 10 x 10 Confusion Matrix

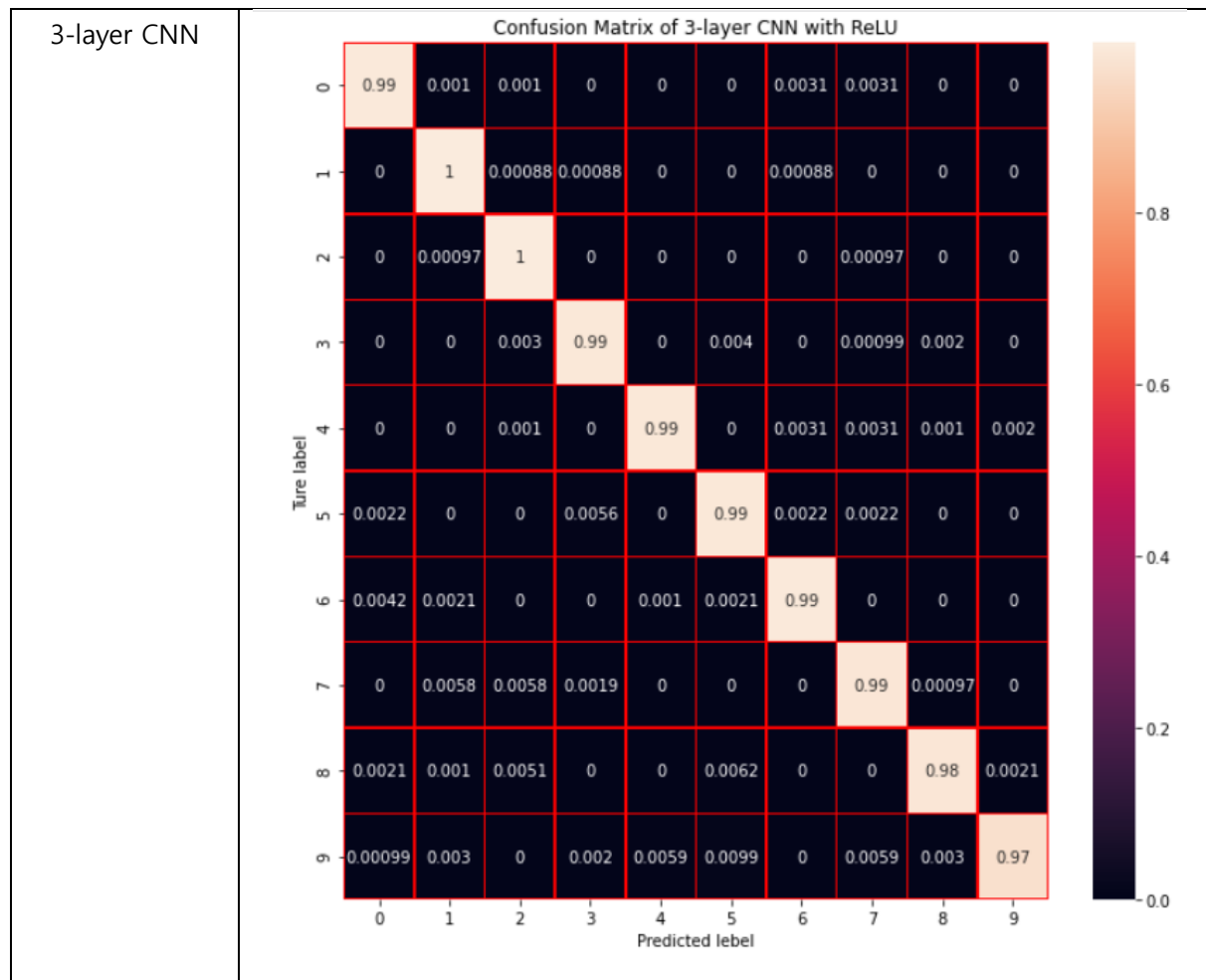
Learning rate = 0.1

Batch_size = 100, epoch = 3

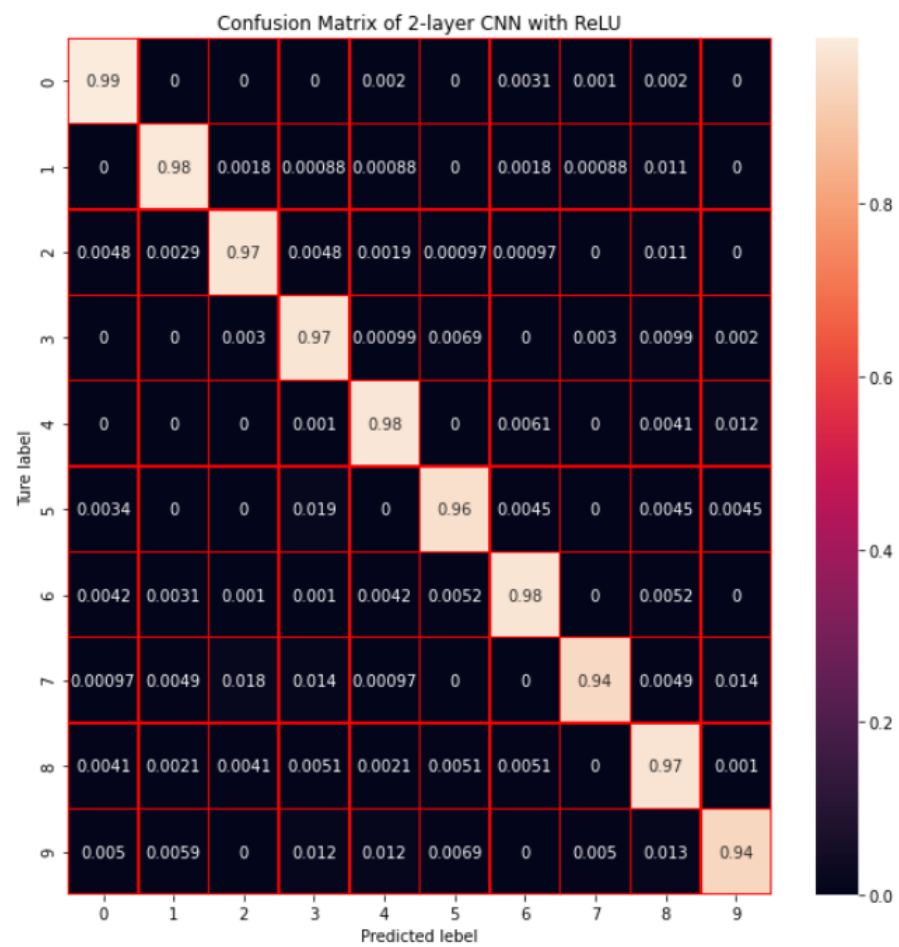
2-layer model: filters = 20, filter_size=(5,5) → filters = 40, filter_size=(10,10)

3-layer model: filters = 20, filter_size=(5,5) → filters = 40, filter_size=(10,10) → filters = 80, filter_size=(10,10)

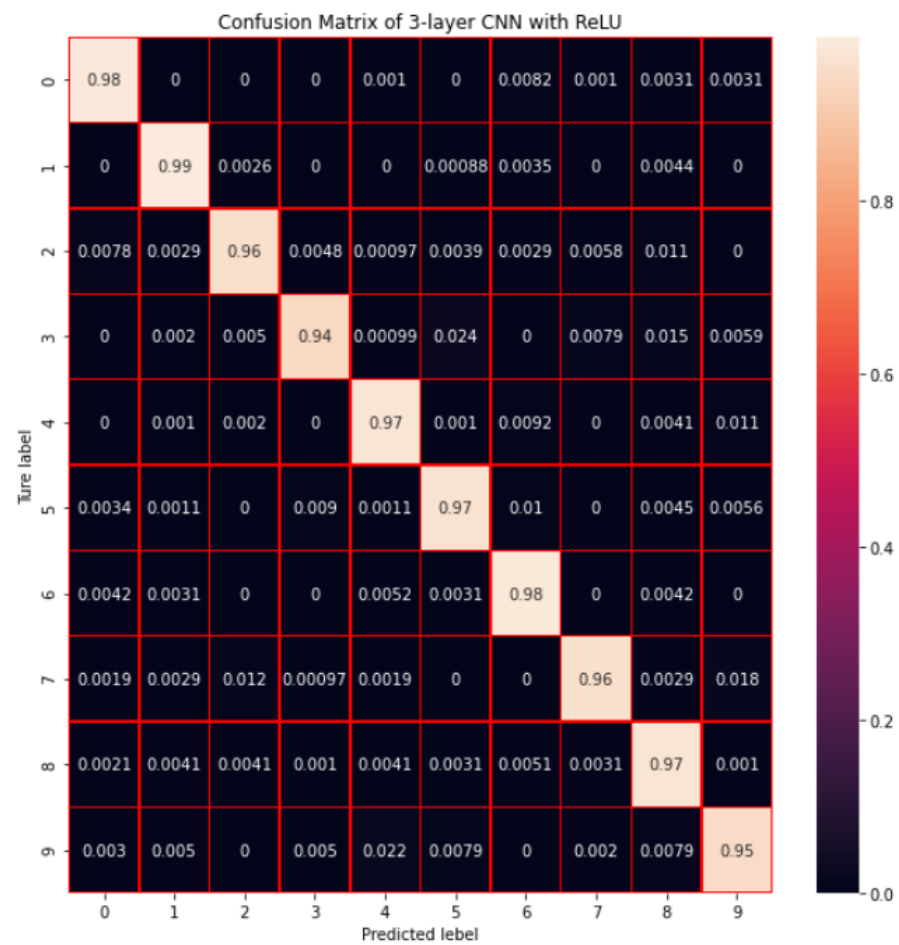


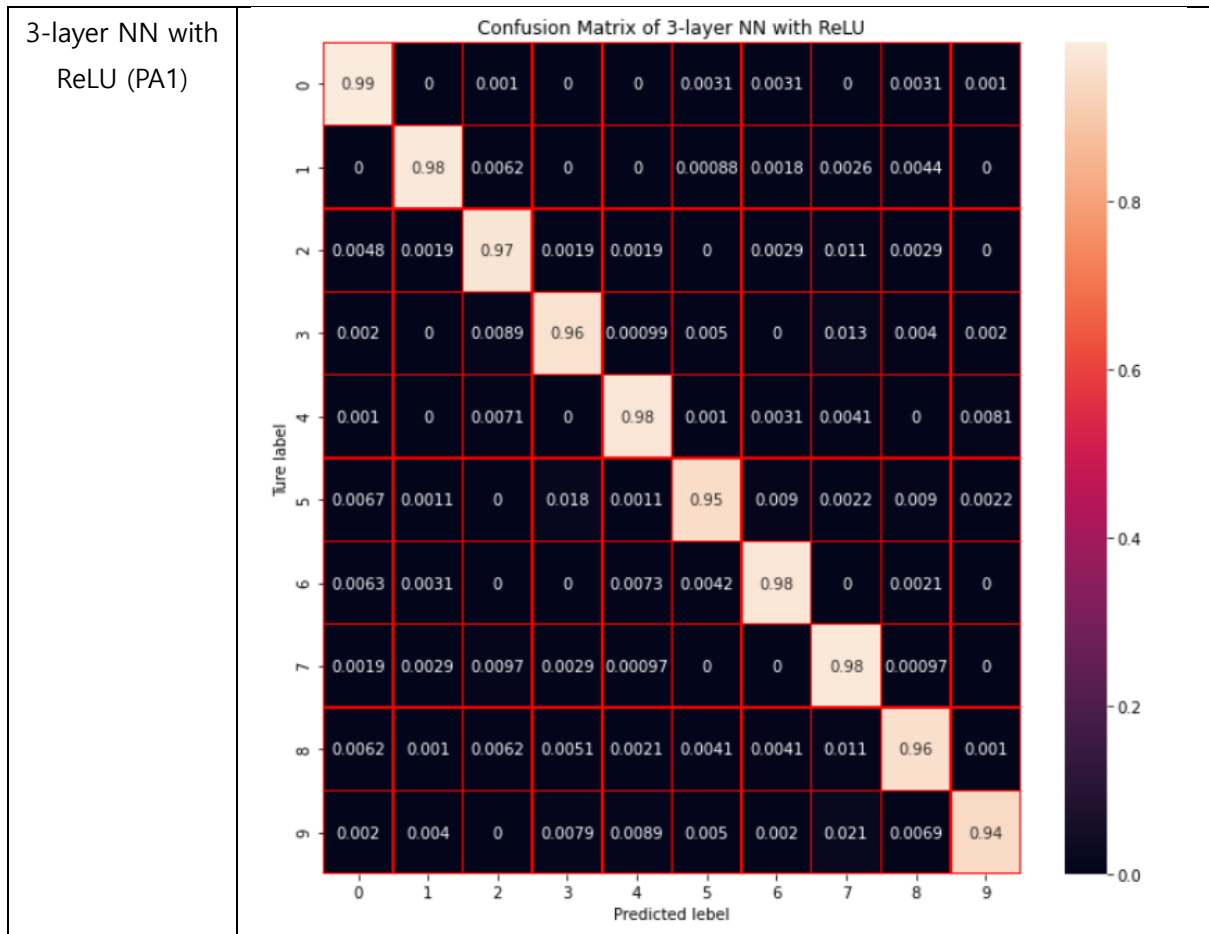


2-layer CNN
using DL-
framework



3-layer CNN
using DL-
framework





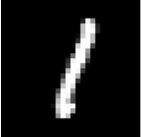
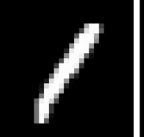




3-layer NN with ReLU의 경우, 3 epoch보다 많은 에폭 수로 학습이 이루어지면 2-layer CNN과 3-layer CNN 모두와 유사한 각 클래스별 높은 정답을 보였다. 본 문서에 포함한 3-layer NN with ReLU의 경우 약 20 epoch으로 학습한 결과이다. 또한, 직접 구현한 CNN과 Deep learning Framework를 활용한 CNN의 경우, 직접 구현한 CNN에서 각 클래스별로 올바르게 정답 클래스를 말할 확률이 1이 되는 클래스가 존재했다. 이는 DL-framework를 이용한 CNN은 input과 output의 size가 같도록 padding을 넣어 모델을 구현한 차이 때문에 발생한다고 할 수 있다. 또한 각 class별 정답을 말할 확률이 직접 구현한 CNN이 더 높은 경향을 보이는 것을 알 수 있다. 그리고 2-layer CNN과 3-layer CNN 사이의 성능은 직접 구현과 DL-framework를 이용한 구현 둘 다 큰 차이가 없는 것을 알 수 있다. 어떤 클래스에서는 2-layer CNN이 더 높은 확률로 정답을 맞추지만 어떤 클래스에서는 3-layer CNN이 더 높은 확률로 정답을 맞추고 있다. 하지만 이 둘의 차이는 미미하기 때문에 2-layer CNN과 3-layer CNN 둘 다 높은 확률로 epoch =3 만에 좋은 성능을 발휘한다는 것을 알 수 있다. NN의 경우 많은 epoch수를 학습시켜야 CNN과 비슷하거나 못한 성능을 내지만 CNN은 epoch수가 3이상만 되어도 좋은 성능을 낸다는 것을 알 수 있다.

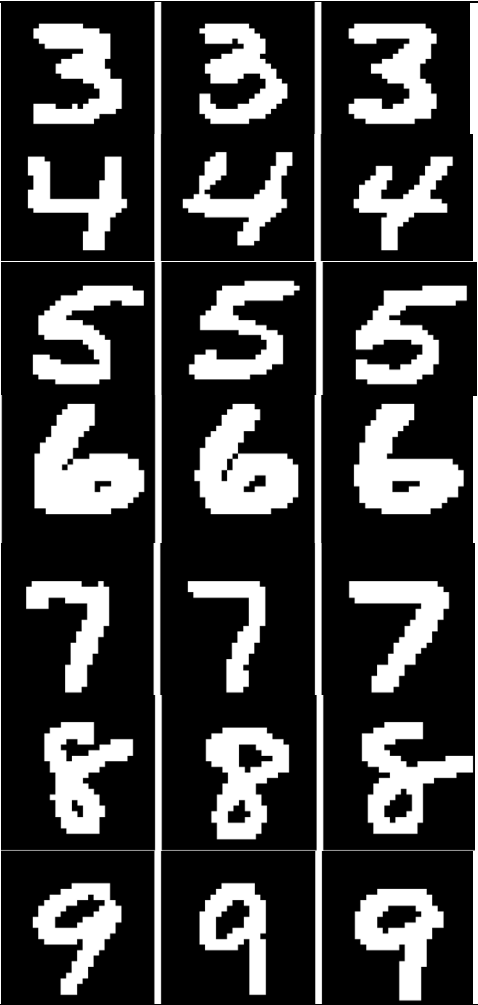
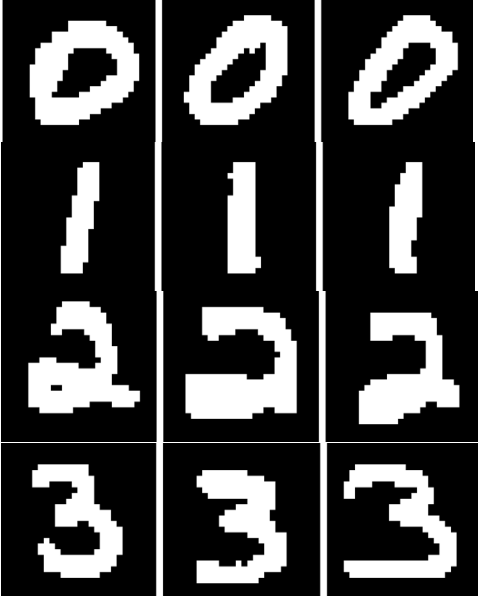
- Top 3 score images (all classes), using Tensorboard

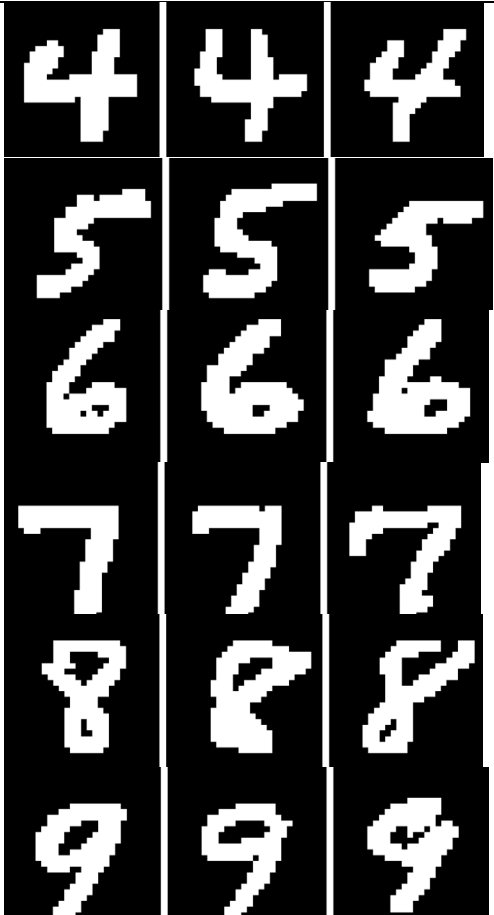
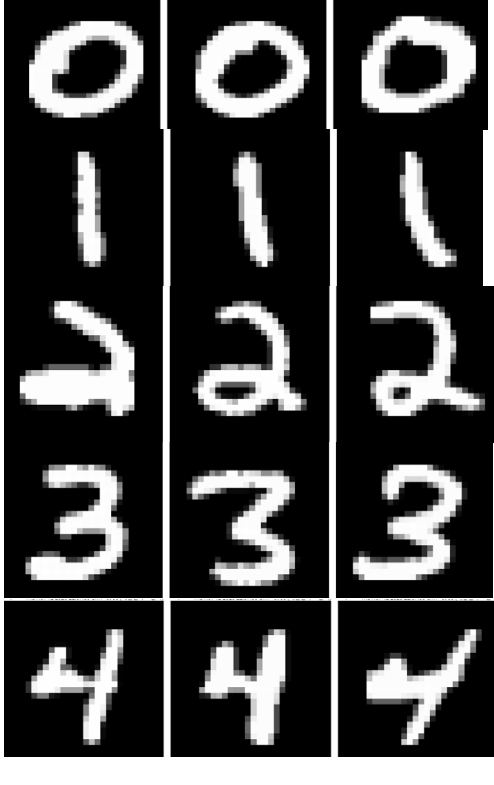
Model 학습 조건은 위와 동일.

Class별 image의 Predicted Probability

2-layer CNN		<p>0:</p> <p>[0.999999971578912, 0.9999999841731422, 0.9999999914354786]</p> <p>1:</p> <p>[0.9998073029447844, 0.9998309501997764, 0.9998745139968646]</p> <p>2:</p> <p>[0.9999999943527536, 0.9999999948787168, 0.9999999979659373]</p> <p>3:</p> <p>[0.9999999997845281, 0.999999999824321, 0.999999999854245]</p> <p>4:</p> <p>[0.9999999692056366, 0.9999999693650081, 0.9999999871868505]</p> <p>5:</p> <p>[0.9999997380845405, 0.9999997558482104, 0.9999998579965437]</p> <p>6:</p> <p>[0.9999999939794035, 0.9999999972316984, 0.9999999974251268]</p> <p>7:</p> <p>[0.9999998382219406, 0.999999872245216, 0.999999938066147]</p> <p>8:</p> <p>[0.9999999151351239, 0.9999999306226622, 0.9999999719396541]</p> <p>9:</p> <p>[0.9999999123936728, 0.9999999195911391, 0.9999999795253294]</p>
3-layer CNN		<p>0:</p> <p>[0.999999941224533, 0.9999999430435026, 0.9999999557289965]</p>

	  	1: [0.9998626184272575, 0.9998780736415516, 0.999890065858422] 2: [0.9999999832270289, 0.9999999846467922, 0.9999999903447848] 3: [0.9999999963641022, 0.9999999985172063, 0.9999999996661173] 4: [0.9999999889127839, 0.9999999915169507, 0.9999999982852683] 5: [0.9999999848029499, 0.9999999848345029, 0.9999999871671325] 6: [0.99999999286678, 0.9999999944027957, 0.9999999977925351] 7: [0.9999720256069099, 0.9999759536011434, 0.9999869107686082] 8: [0.9999999999168987, 0.9999999999779039, 0.99999999997862] 9: [0.9999999627094932, 0.999999978316036, 0.9999999849639736]
2-layer CNN using DL- framework	  	0: [0.9999827, 0.9999907, 0.99999225] 1: [0.9994531, 0.9995216, 0.9995278] 2: [0.9999999, 0.9999999, 1.0] 3: [0.99999976, 0.9999999, 0.9999999]

		<p>4: [0.99999297, 0.9999933, 0.9999968]</p> <p>5: [0.99999964, 0.99999976, 0.9999999]</p> <p>6: [0.99999857, 0.9999989, 0.9999995]</p> <p>7: [0.99999964, 0.99999976, 0.99999976]</p> <p>8: [0.9999329, 0.9999429, 0.99995315]</p> <p>9: [0.9999112, 0.9999113, 0.999941]</p>
3-layer CNN using DL- framework		<p>0: [0.99997926, 0.99998, 0.9999907]</p> <p>1: [0.99992, 0.9999207, 0.99992466]</p> <p>2: [0.99998856, 0.9999888, 0.99999404]</p> <p>3: [0.9999988, 0.9999988, 0.9999994]</p> <p>4: [0.9999989, 0.9999993, 0.9999994]</p> <p>5: [0.99999726, 0.9999988, 0.99999905]</p> <p>6: [0.99999297, 0.9999944, 0.99999523]</p> <p>7:</p>

		<p>[0.9999796, 0.99998105, 0.9999827]</p> <p>8: [0.9999989, 0.99999917, 0.9999993]</p> <p>9: [0.99994993, 0.9999535, 0.999979]</p>
<p>3-layer NN with ReLU</p>		<p>0: [0.9999999982201826, 0.9999999984500103, 0.9999999993399187]</p> <p>1: [0.9999854362636408, 0.9999917413581879, 0.9999963211089911]</p> <p>2: [0.9999999997504616, 0.9999999998524896, 0.9999999998887183]</p> <p>3: [0.9999999999246152, 0.9999999999509106, 0.9999999999889333]</p> <p>4: [0.9999998968392132, 0.9999999320785986, 0.9999999770658999]</p> <p>5: [0.999999999996487, 0.9999999999964893, 0.999999999997742]</p> <p>6:</p>

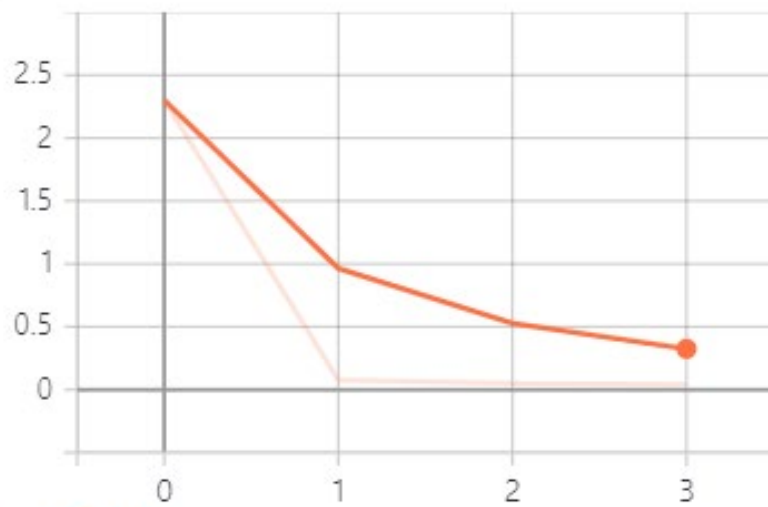
	5	5	5	[0.9999999984761649, 0.9999999996125151, 0.9999999996193614]
	6	6	6	7: [0.9999998226429875, 0.9999998588171173, 0.9999999527638577]
	7	7	7	8: [0.9999999863718099, 0.9999999872974402, 0.9999999932824548]
	8	8	8	9: [0.9999996447270504, 0.9999996840154376, 0.9999998755664109]
	9	9	9	

Epoch = 20으로 학습한 3-layer NN with ReLU의 경우, 학습을 많이 시킬수록 CNN과 유사하게 각 클래스를 높은 확률로 잘 분류하는 것을 다시한번 확인할 수 있다. Tensorboard에 출력된 image 상의 차이를 비교하자면, DL-framework를 이용한 2-layer CNN과 3-layer CNN의 경우, 글씨체가 두껍고 뭉툭한 image를 높은 확률로 해당 클래스로 잘 분류하는 반면, DL-framework를 이용하지 않고 직접 구현한 2-layer CNN과 3-layer CNN의 경우, 상대적으로 얇은 글씨체의 image를 잘 분류하는 것을 알 수 있다. 3-layer NN with ReLU의 경우, 마치 이 둘을 적절하게 섞은 듯이, 굵은 글씨체와 얇은 글씨체의 image에서도 높은 확률로 해당 클래스를 잘 분류하는 것을 볼 수 있다. 또한, 5가지의 모델 모두 각 image별 정답 확률이 99%로 나오기 때문에 5가지의 모델 모두 좋은 성능을 나타낸다고 할 수 있다.

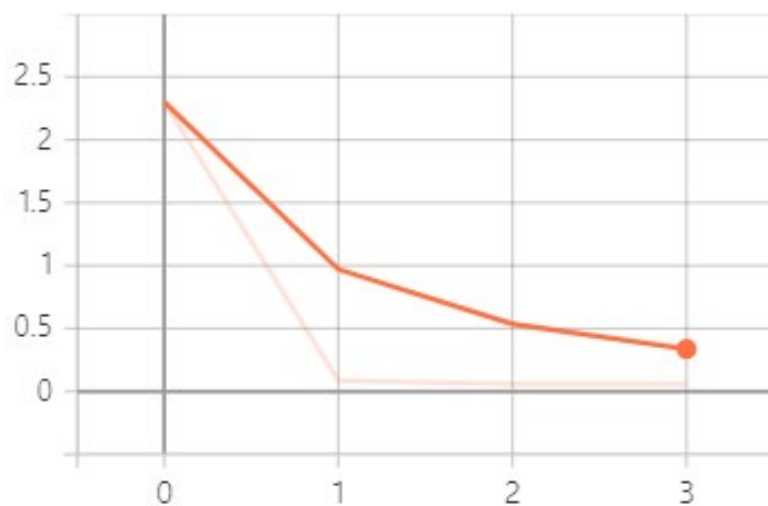
- Training Loss graph

Model 학습 조건은 위와 동일.

2-layer CNN

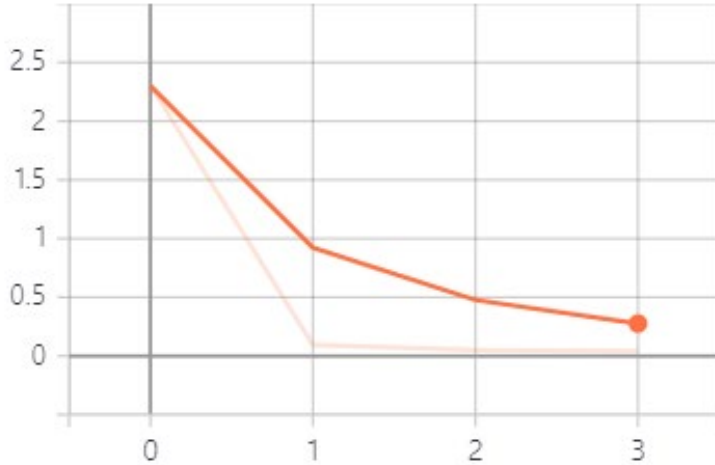



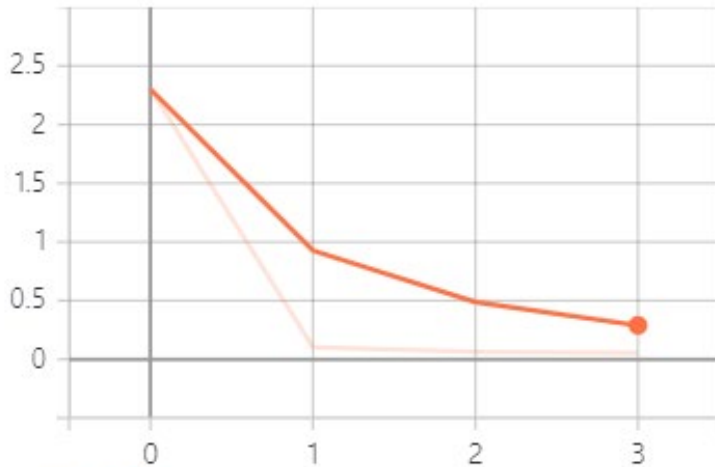



train_loss
tag: train_loss

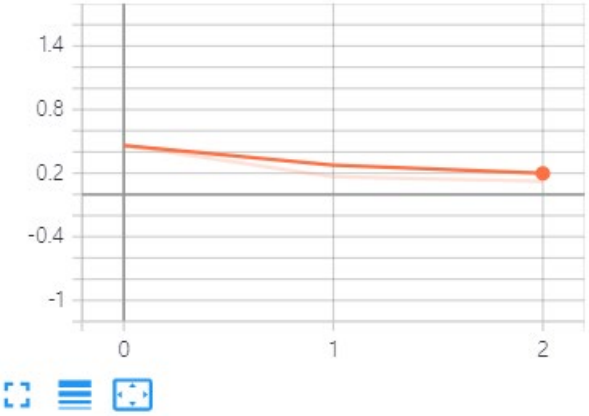
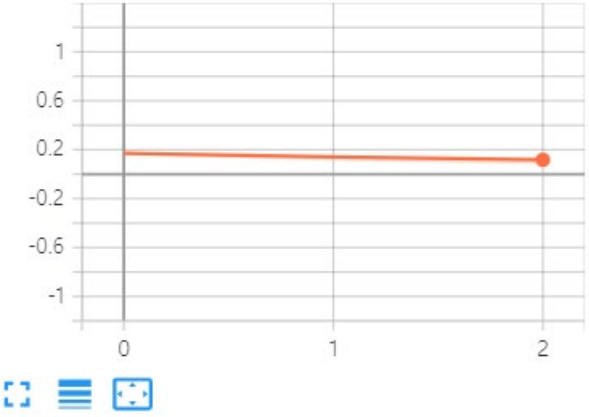
valid_loss

valid_loss
tag: valid_loss

```
print(train_loss)
print(valid_loss)
```

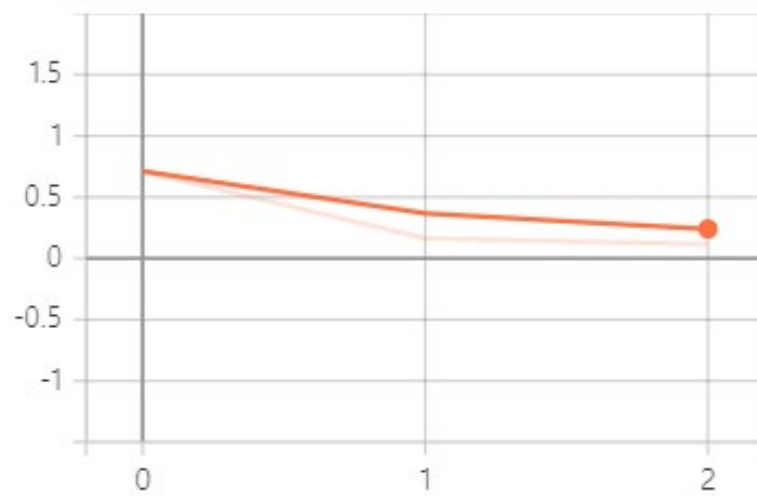
```
[2.300891668836569, 0.07754266616879284,
0.04818194085447217, 0.03996298718089977]
```

	<pre>[2.3009217167926272, 0.08655219554861356, 0.06041587591855102, 0.057751415921657646]</pre>
3-layer CNN	<div><div>train_loss tag: train_loss</div><div></div></div> <hr/> <div><div>valid_loss tag: valid_loss</div><div></div></div> <pre>print(train_loss) print(valid_loss)</pre>

	<pre>[2.3024553104171575, 0.09507419991408973, 0.04847764341246823, 0.03961645634930468] [2.302508180507089, 0.10320231074168384, 0.06477457632208879, 0.056316496803868514]</pre>
2-layer CNN using DL- framework	<p>train_loss tag: train_loss</p>  <p>valid_loss</p> <p>valid_loss tag: valid_loss</p>  <pre>y_vloss = run.history['val_loss'] y_loss = run.history['loss'] print(y_vloss) print(y_loss)</pre> <pre>[0.16910044848918915, 0.12512506544589996, 0.09389950335025787] [0.46201395988464355, 0.1642458438873291, 0.12289094924926758]</pre>

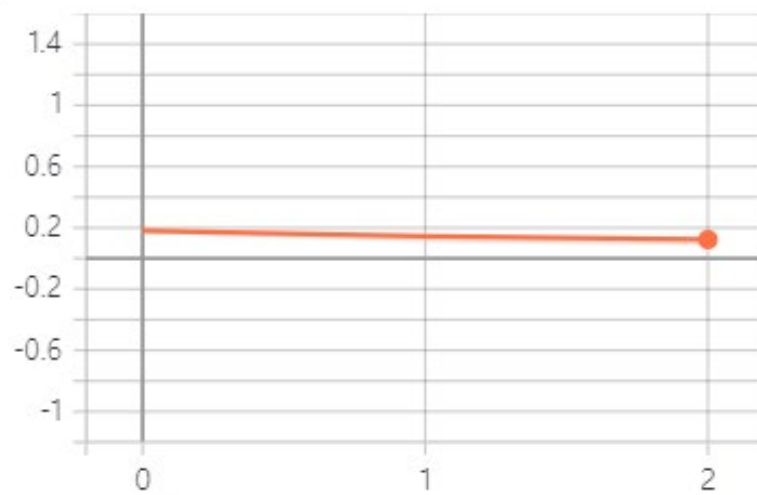
3-layer CNN
using DL-
framework

train_loss
tag: train_loss



valid_loss

valid_loss
tag: valid_loss



```
y_vloss = run.history['val_loss']  
y_loss = run.history['loss']  
print(y_vloss)  
print(y_loss)
```

	<pre>[0.18105103075504303, 0.12388487905263901, 0.1020936518907547] [0.7121426463127136, 0.11696659028530121] 0.1630721092224121,</pre>
3-Layer NN	<div> <div> train_loss tag: train_loss </div> </div> <div> <div> valid_loss tag: valid_loss </div> </div> <pre>print(train_loss)</pre> <pre>[2.302499864730141, 1.08075405405345, 0.4427186856179334, 0.3037999693038388, 0.24103046771174946, 0.1870552683936071, 0.15126654492568575, 0.1520891449877983, 0.11955924575343674, 0.10570147023275156, 0.09618448082264587, 0.09298680865201944, 0.08338123511481071, 0.07166709780142144, 0.0761639432235458, 0.062469553102384924, 0.0589123999101079,</pre>

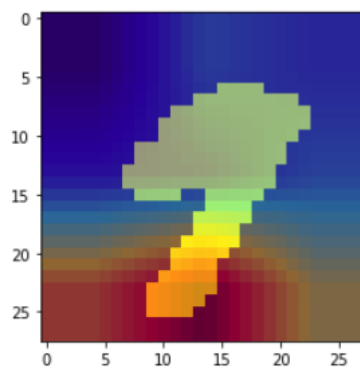
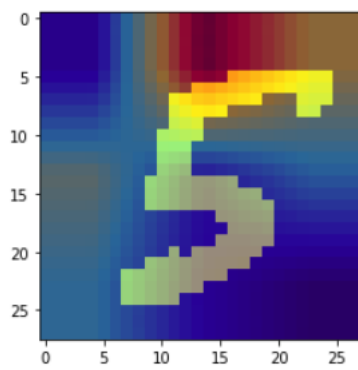
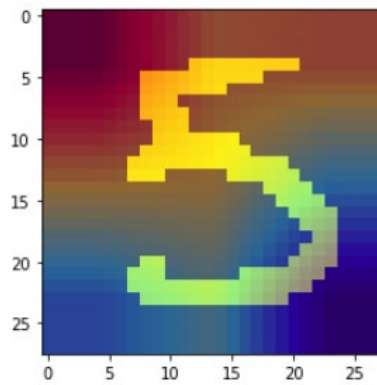
	<pre> 0.051861536821506946, 0.05036254903388184, 0.04841806592763099, 0.04369398464024996] print(valid_loss) [2.3024919471091643, 0.4510271849650023, 0.24478576665062535, 0.16251330318375024, 0.14413360743628292, 0.12633323486328601, 0.11939685550181425, 0.1179757951048675, 0.11172685018716662, 0.10527114030227269, 0.11092706282774216] </pre>
--	---

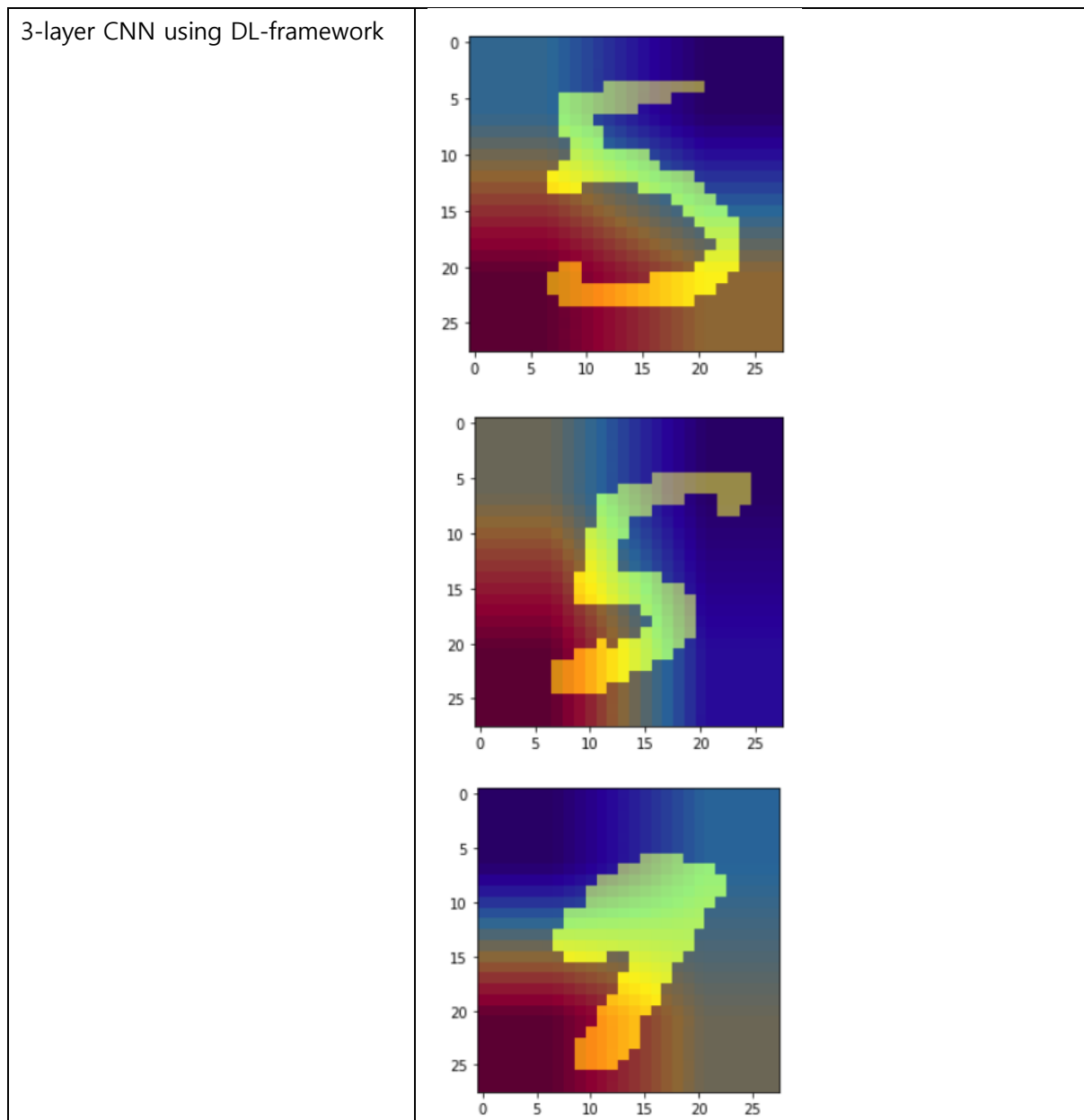
Loss 값으로 NN과 CNN의 차이를 확연히 볼 수 있다. CNN은 epoch = 3만 되어도 loss값이 0.5보다 훨씬 작아지며 성능이 안정화 되는데 반해, NN은 epoch = 3일 때에도 loss값이 0.5보다 높으며, epoch = 10은 되어야 loss값이 0.2정도의 값이 나오는 것을 확인할 수 있다. 또한, 직접 구현한 CNN과 DL-framework을 사용하여 구현한 CNN은 차이점도 확인할 수 있다. 직접 구현한 CNN의 경우, train과 validation에 대한 loss값이 2-layer, 3-layer 둘 다 2이상에서 시작하지만, DL-framework을 사용한 CNN은 2-layer, 3-layer 둘 다 1이하의 값에서 시작하는 것을 알 수 있다. 즉, 비교적 시작부터 DL-framework을 사용한 CNN이 loss값이 적으며 성능이 높게 시작함을 알 수 있다. 이는 초기화 되는 Convolution layer의 weight값의 차이로 인해 발생할 수 있다. 하지만 이후 epoch에서는 직접 구현한 것이든, DL-framework을 사용한 CNN이든 모두 안정적으로 loss값이 확연히 줄어드는 것을 알 수 있다. 따라서 본 Task에서는 NN보다 CNN이 적은 학습으로 더 좋은 성능을 낼 수 있는 것을 알 수 있다.

2. (Optional)

- CAM

2-layer CNN using DL-framework





Test set에서 예시 image 3개를 2-layer CNN using DL-framework과 3-layer CNN using DL-framework에 돌려보고 그 결과를 CAM으로 나타내어 각 CNN 모델이 image의 어느 부분을 중점적으로 보았는지 확인하였다.


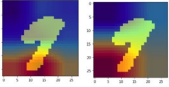
동일한 5 image의 첫 번째 image의 경우, 2-layer CNN using DL-framework은 5의 머리 부분



을 집중하여 본 반면, 3-layer CNN using DL-framework은 5의 받침 부분을 중점적으로 보았다. 숫자 5에 대한 이러한 경향은 두 번째 5 image에서도 유사한 패턴을 보였다. 2-layer



CNN using DL-framework은 마찬가지로 5의 머리 부분

using DL-framework은  받침 부분을 중점적으로 보았다. 숫자 9 image의 경우, 둘 다 비

슷하게 9의 꼬리 부분 을 중점적으로 본 것을 알 수 있다. 이 외에 첨부한 코드에
서 다른 image들의 CAM 출력결과들을 확인할 수 있다.