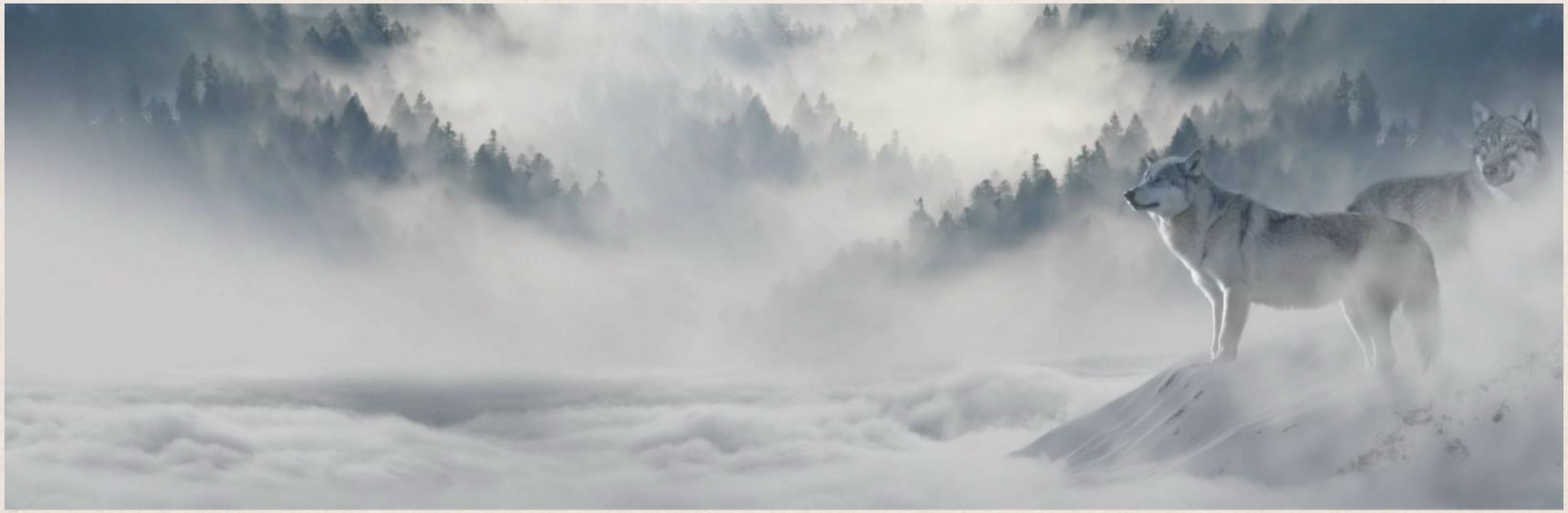
# CNN_pytorch_fashion100



경북대학교

# GPU 사용 설정

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
torch.manual_seed(777)
if device == 'cuda':
    torch.cuda.manual_seed_all(777)
print(device + " is available")
```

CUDA가 있으면 device = 'cuda'로 설정해서 언제든 gpu를 사용할 수 있게 만들어줌

# 데이터셋 로드

```
#  Fashion MNIST의 training 데이터를 다운로드합니다.
training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(), # ToTensor() - PIL 이미지 또는 nu
)

# Fashion MNIST의 test 데이터를 다운로드합니다.
test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor(), # ToTensor() - PIL 이미지 또는 nu
)
```

root = cifar 데이터셋을 저장할 경로

train : True로 설정 시 trainset, False로 설정시 test set을 불러옴

download = True로 설정시 인터넷에서 데이터셋 다운로드

transform = 데이터셋을 불러온 후 어떤 전처리를 할지 설정

여기서는 ToTensor()함수를 사용하여 0~255를 0~1로 변환

# import

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
```

torch : pytorch의 기본 패키지, tensor 연산 및 딥러닝 모델 구현에 필요한 함수들을 제공

torch.nn : 신경망 모델을 구현하기 위한 클래스와 함수들이 들어있는 모듈

torch.nn.functional : 신경망 모델에서 자주 사용하는 함수

torch.optim : 경사하강법등 최적화 알고리즘 제공

torchvision : 이미지 및 비디오 처리를 위한 패키지

torchvision.transforms : 이미지 변환을 위한 함수

# 데이터 불러오기

```
batch_size = 64

# 데이터 로더. 반복적으로 지정한 데이터셋에서 지정한 배치
# training 데이터셋과 test 데이터셋을 가져올 데이터 로더를
train_dataloader = DataLoader(training_data, # 데이터를 ㅎ
                              batch_size=batch_size) #

test_dataloader = DataLoader(test_data, # 데이터를 로드할
                             batch_size=batch_size) #
```

torch.utils.data.DataLoader = 미리 정의된 batch size로 train_set과 test_set을 load함

root = cifar 데이터셋을 저장할 경로

train : True로 설정 시 trainset, False로 설정시 test set을 불러옴

download = True로 설정시 인터넷에서 데이터셋 다운로드

transform = 데이터셋을 불러온 후 어떤 전처리를 할지 설정

여기서는 ToTensor()함수를 사용하여 0~255를 0~1로 변환

# ConvNet class __init__

```
class NeuralNetwork(nn.Module):

    # 사용하려는 모든 레이어를 선언합니다.
    def __init__(self):
        super(NeuralNetwork, self).__init__()

        # 1차원 배열로 변환합니다.
        self.flatten = nn.Flatten()

        # 레이어를 쌓아서 모델을 생성합니다.
        self.linear_relu_stack = nn.Sequential(
            # y = Wx + b 선형 변환을 위한 입력과 출력 개수를 설정
            nn.Linear(28*28, 512),
            # 활성화 함수로 비선형 함수인 ReLU를 사용합니다.
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10)
        )

    # 미리 선언해놓은 레이어들을 사용하여 입력에서 출력까지 모델을
    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

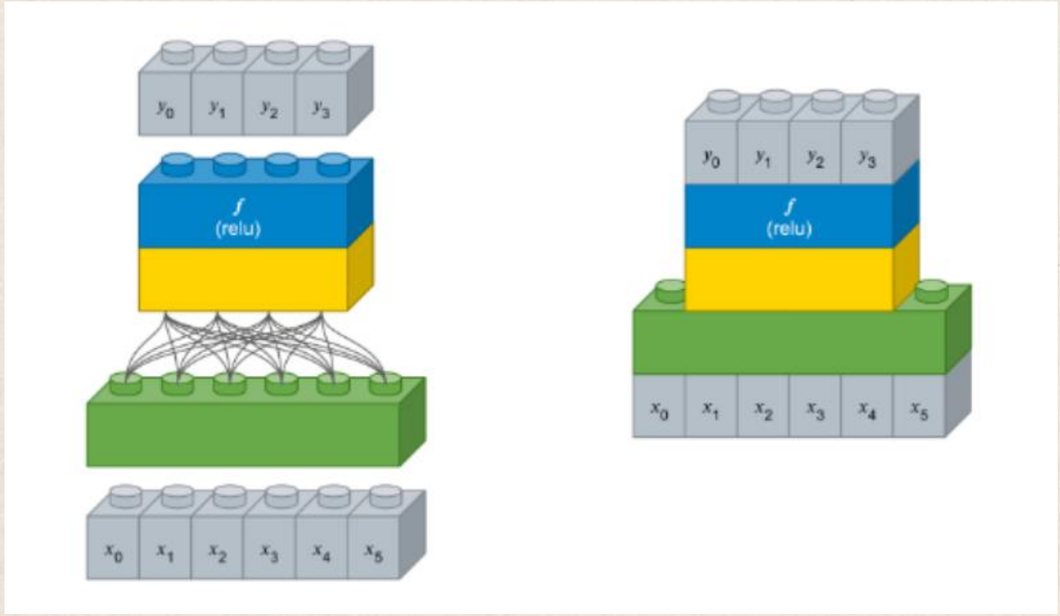
1. `in_channels` (int): 입력 이미지의 채널 수
2. `out_channels` (int): 출력 이미지의 채널 수 (필터의 개수)
3. `kernel_size` (int 또는 Tuple[int, int]): 컨볼루션 커널의 크기
4. `stride` (int 또는 Tuple[int, int]): 컨볼루션 연산 시 필터의 이동 간격
5. `padding` (int 또는 Tuple[int, int]): 입력 이미지 주변에 추가할 패딩 크기
6. `dilation` (int 또는 Tuple[int, int]): 컨볼루션 커널 내부의 간격
7. `groups` (int): 입력 채널을 나누어 여러 그룹으로 컨볼루션 연산을 수행할 때 그룹 수
8. `bias` (bool): 편향(bias) 사용 여부

class ConvNet은 nn.Module 클래스를 상속받아 정의됨, nn.Module 클래스는 pytorch에서 제공하는 모든 신경망 모듈의 기본 클래스

ConV2d 함수를 다섯번 작성, maxpool 작성, Linear 3개작성, dropout작

# Dense>>Linear

```
self.drop2D = nn.Dropout2d(p=0.25, inplace=False) # 랜덤하
self.mp = nn.MaxPool2d(2)  # 오버피팅을 방지하고, 연산에
self.fc1 = nn.Linear(320,100) # 4x4x20 vector로 flat한 것
self.fc2 = nn.Linear(100,10) # 100개의 출력을 10개의 출력
```




nn.Linear` 함수는 PyTorch에서 완전 연결(fully connected) 레이어를 정의하는데 사용됩니다. 이 함수는 입력과 출력의 차원을 인자로 받아 가중치 행렬을 만들고, 입력에 가중치를 곱하고 편향을 더하여 출력을 계산합니다.

예를 들어, `nn.Linear(10, 5)`는 10차원의 입력을 5차원의 출력으로 변환하는 완전 연결 레이어를 정의합니다. 이 레이어의 가중치 행렬은 크기가 (5, 10)이며, 편향 벡터의 크기는 (5,)입니다.

==`Dense(5, input_shape=(10,))

# def forward

```
def forward(self, x):
    x = self.flatten(x)
    logits = self.linear_relu_stack(x)
    return logits
```

x = F.relu(self.mp(self.conv1(x)))`: 입력 데이터 `x`를 첫 번째 합성곱 레이어 `self.conv1`을 통해 통과시키고, 그 결과에 ReLU 함수를 적용하고, 맥스 풀링 레이어 `self.mp`를 적용합니다

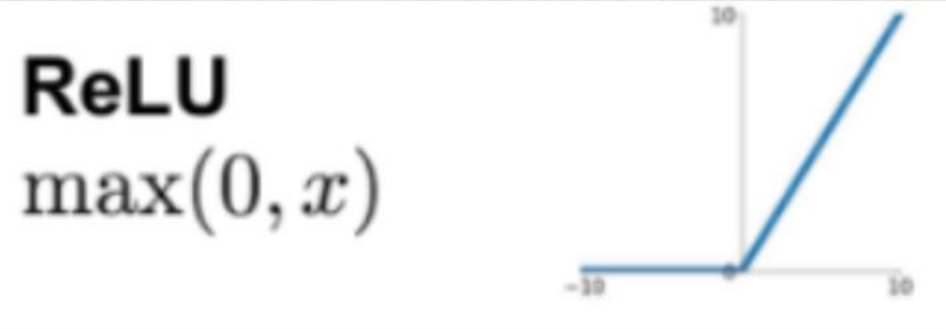
`x = self.drop2D(x)`: 출력값 `x`에 2D 드롭아웃 레이어 `self.drop2D`를 적용합니다.

x = x.view(x.size(0), -1)`: 출력값 `x`를 2차원으로 변환합니다. 이는 fully connected 레이어에 입력으로 넣기 위한 과정(Flatten)

`x = self.fc1(x)`: 2차원으로 변환된 출력값 `x`를 첫 번째 fully connected 레이어 `self.fc1`에 입력으로 넣습니다
return F.log_softmax(x)`: 마지막으로 fully connected 레이어를 통과한 결과값 `x`에 로그 소프트맥스 함수를 적용한 결과를 반환합니다.

# Conv2D (activation(relu)

```
def forward(self, x):
      x = F.relu(self.mp(self.conv1(x))) # convolution layer 1번에 relu를 씌우고 maxpool, 결과값은 12x12x10
      x = F.relu(self.mp(self.conv2(x))) # convolution layer 2번에 relu를 씌우고 maxpool, 결과값은 4x4x20
      x = self.drop2D(x)
      x = x.view(x.size(0), -1) # flat
      x = self.fc1(x) # fc1 레이어에 삽입
      x = self.fc2(x) # fc2 레이어에 삽입
      return F.log_softmax(x) # fully-connected layer에 넣고 logsoftmax 적용
```




ReLU 함수는 입력값이 0보다 클 때는 선형 함수처럼 작동하지만, 입력값이 0 이하일 때는 항상 0을 출력함

ReLU 함수는 계산 비용이 매우 적게  듬

죽은 뉴런(dead neuron) 현상이 발생할 수 있습니다. 이러한 현상이 발생하면 해당 뉴런은 학습 과정에서 어떠한 역할도 하지 못하게 됨

# 모델 학습 설정

```
criterion = nn.CrossEntropyLoss()
```

```
optimizer = torch.optim.SGD(params = model.parameters(), lr=1e-3)
```

model = ConvNet().to(device)`: `ConvNet` 클래스를 인스턴스화하여 `model` 변수에 저장합니다. 이때 `to(device)` 메서드를 사용하여 모델이 GPU에서 실행되도록 설정합니다.

criterion = nn.CrossEntropyLoss().to(device)`: 크로스 엔트로피 손실 함수 `nn.CrossEntropyLoss()`를 인스턴스화하여 `criterion` 변수에 저장

optimizer = torch.optim.Adam(model.parameters(), lr = learning_rate)`: Adam 옵티마이저를 인스턴스화하여 `optimizer` 변수에 저장합니다. model.parameters()`를 사용하여 모델의 학습 가능한 매개변수들을 전달하고, `lr` 매개변수를 통해 학습률을 설정합니다.

# 모델 학습 설정(test)

```python
def test(dataloader, model, criterion):
  size = len(dataloader.dataset)
  num_batches = len(dataloader)
  model.eval()
  test_loss, test_acc = 0, 0
  with torch.no_grad():
    for X, y in dataloader:
      X, y = X.to(device), y.to(device)
      pred = model(X)
      test_loss += criterion(pred, y).item()
      test_acc += (pred.argmax(1) == y).type(torch.float).sum().item()

  test_loss /= num_batches
  test_acc /= size

  print(f"TEST: \n Accuracy: {(100*test_acc):>.1f}%, Avg loss: {test_loss:>8f} \n")

  return test_loss, test_acc
```

dataset읽어오는 모듈,모델,손실함수
모델을 평가모드로 설정한 후 no_grad()를 사용하여 그래디언트 계산을 비활성화 합니다. (모델의 파라미터를 업데이트 하는 과정에서 계산 생략 (평가할때는 필요 X))

그 후 dataloader에서 배치데이터를 읽어오면서
손실과 정확도를 반환합니다.

X = 입력데이터(batch), y = 정답데이터

# 모델 학습 설정(train)

```
def train(dataloader, model, criterion, optimizer):
  size = len(dataloader.dataset)
  train_accuracy, train_loss = 0, 0
  model.train()
  for batch, (X, y) in enumerate(dataloader, 0):

    X, y = X.to(device), y.to(device)
    optimizer.zero_grad()
    pred = model(X)
    loss = criterion(pred, y)
    loss.backward()
    optimizer.step()

    train_accuracy += (pred.argmax(1) == y).type(torch.float).sum().item()
    train_loss += loss.item()

    if batch % 100 == 0:
      current = (batch+1) * len(X)

      print(f"loss: {train_loss/current:>7f} accuracy: {train_accuracy/curre

  return train_loss/len(dataloader), train_accuracy/size
```

dataset읽어오는 모듈,모델,손실함수,최적화 함

size변수에 전체 데이터셋의 크기 저장. 그 후 초기화, 그리고 모델을 학습모드로 설정.

입력데이터와 정답데이터를 분배한 후 gpu로 설정, optimizer 기울기 초기화. 그 후 예측 시작 순전파, 역전파 그 후 파라미터 업데이트,

정확도와 손실도 계산 후 배치수가 100의 배수일때마다 손실과 정확도 출력

# 모델 훈련 후 저장

```
# 5 Epoch 반복하도록 합니다. 1 Epoch는 전체 Train 데이터셋을 학습에 한번 사용하는 것을 의미합니다.
# 여기에선 5 Epoch를 사용하지만 실제론 더 많은 Epoch를 사용해야 합니다.
epochs = 5
training_loss, training_accuracy = [], []
testing_loss, testing_accuracy = [], []
for t in range(epochs):

    print(f"Epoch {t+1}₩n-------------------------------")

    train_loss, train_accuracy = train(train_dataloader, model, criterion, optimizer)
    test_loss, test_accuracy = test(test_dataloader, model, criterion)

    training_loss.append(train_loss)
    training_accuracy.append(train_accuracy)
    testing_loss.append(test_loss)
    testing_accuracy.append(test_accuracy)

print("Done!")
history = [training_loss,training_accuracy,testing_loss, testing_accuracy]
PATH_TO_TRAINED_MODEL = 'model.pth'
torch.save(model.state_dict(), PATH_TO_TRAINED_MODEL)
```

epochs 변수설정
train_loss와 train_accuracy
val_loss와 val_accuracy list에 저장 후
history로 묶음

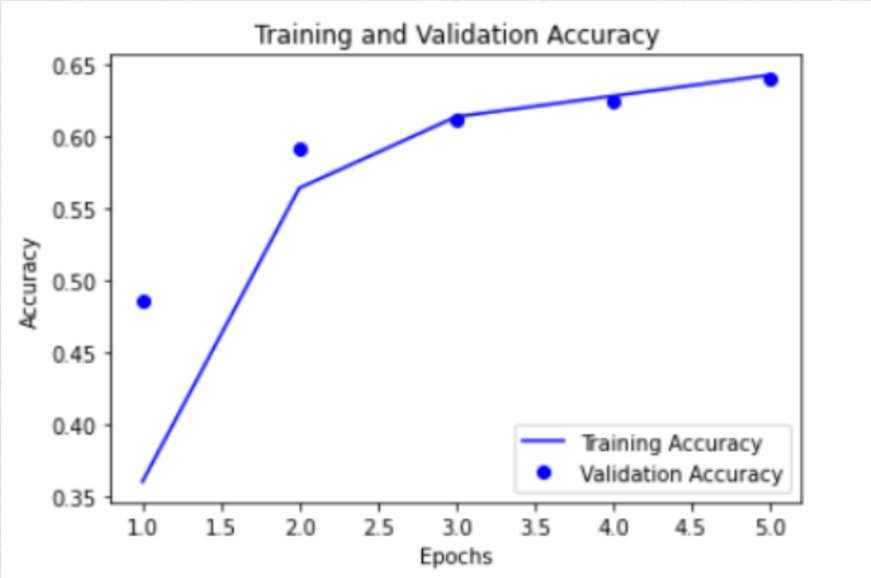
model 저장!!

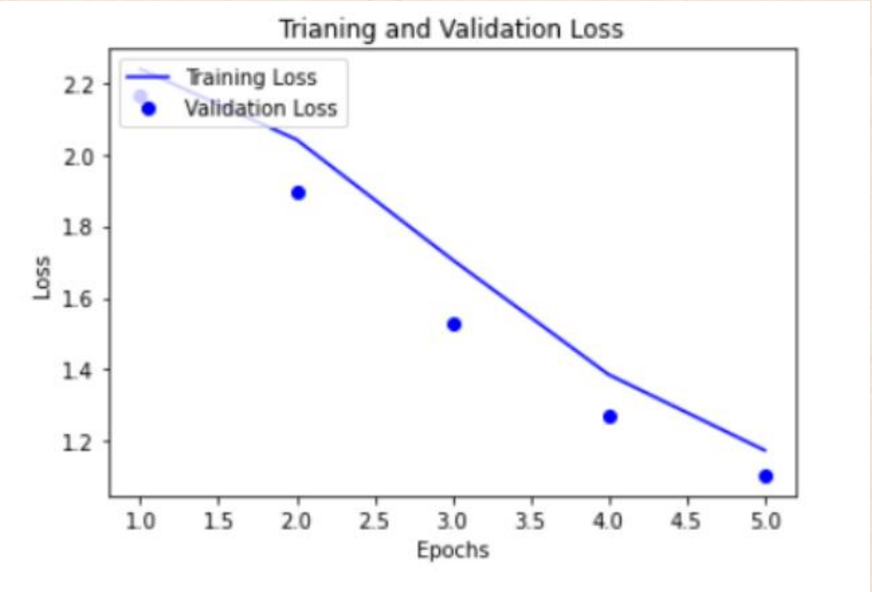
# epoch

```
Epoch 5
-------------------------------
loss:0.021024 accuracy: 0.593750 [   64/60000]
loss:0.019453 accuracy: 0.637531 [ 6464/60000]
loss:0.019398 accuracy: 0.634562 [12864/60000]
loss:0.019293 accuracy: 0.634136 [19264/60000]
loss:0.019153 accuracy: 0.636651 [25664/60000]
loss:0.019008 accuracy: 0.635791 [32064/60000]
loss:0.018853 accuracy: 0.638571 [38464/60000]
loss:0.018691 accuracy: 0.640536 [44864/60000]
loss:0.018553 accuracy: 0.640605 [51264/60000]
loss:0.018410 accuracy: 0.642533 [57664/60000]
TEST:
 Accuracy: 64.0%, Avg loss: 1.104790
```

\epoch = 10, Accuracy : 37%, Avg loss : 2.477

# accuracy, loss graph



# Conv2D (가중치)

```
def forward(self, x):
      x = F.relu(self.mp(self.conv1(x))) # convolution layer 1번에 relu를 씌우고 maxpool, 결과값은 12x12x10
      x = F.relu(self.mp(self.conv2(x))) # convolution layer 2번에 relu를 씌우고 maxpool, 결과값은 4x4x20
      x = self.drop2D(x)
      x = x.view(x.size(0), -1) # flat
      x = self.fc1(x) # fc1 레이어에 삽입
      x = self.fc2(x) # fc2 레이어에 삽입
      return F.log_softmax(x) # fully-connected layer에 넣고 logsoftmax 적용
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 10, 24, 24]             260
         MaxPool2d-2          [-1, 10, 12, 12]               0
            Conv2d-3           [-1, 20, 8, 8]           5,020
         MaxPool2d-4           [-1, 20, 4, 4]               0
         Dropout2d-5           [-1, 20, 4, 4]               0
            Linear-6                [-1, 100]          32,100
            Linear-7                 [-1, 10]           1,010
================================================================
Total params: 38,390
Trainable params: 38,390
Non-trainable params: 0
----------------------------------------------------------------
```

필터 크기 5*5개
출력 필터 개수 10개
따라서 10*5*5 + 10(편향) = 260

입력체널의 수는 10, 출력 채널의 수는 20
10*20*5*5 + 20(편향) = 5020

입력채널의 수 20*4*4 = 320, 출력 채널 = 100
320*100 +100(편향) = 32100

# 테스트 진행(실제 데터)

```
model = NeuralNetwork()
model.load_state_dict(torch.load(PATH_TO_TRAINED_MODEL))
model.eval()
```

```
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

저장했던 모듈 불러온 후 평가모드로 전환

# 테스트 진행(실제 데터)

```
classes = [
    "T-shirt/top",
    "Trouser",
    "Pullover",
    "Dress",
    "Coat",
    "Sandal",
    "Shirt",
    "Sneaker",
    "Bag",
    "Ankle boot",
]
```

class 설정

```
a=[3,3,3,3,0,0,0,0,1,1,1,1,2,2,2,2,4,4,4,4,5,5,5,5,6,6,6,6,7,7,7,7,8,8,8,8,9,9,9,9]
```

# 테스트 진행(실제 데터)

```
import cv2
import numpy as np
from torchvision import transforms
import os
img_path_list = []
count = 0
false_number=[]

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

for i in range(40):
  path = '/content/drive/MyDrive/Colab Notebooks/GITHUB/colab_ML/ML_CN
  if os.path.exists(path + str(i+1) + '.PNG'):
    img = cv2.imread(path + str(i+1) + '.PNG')
  else:
    img = cv2.imread(path + str(i+1) + '.png')
  img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) # Convert the image to g
  img = cv2.resize(img, (28, 28)) # Resize the image to 28x28
  img = transform(img).unsqueeze(0) # Add a batch dimension
  with torch.no_grad():
    output = model(img)
    predicted_class = torch.argmax(output).item()

  print("Predicted class:", predicted_class, a[i])

  if(predicted_class == a[i]):
    count+=1
print(print(f'accuracy = {count/30}'))
```

```
Predicted class: 8 6
Predicted class: 8 6
Predicted class: 7 7
Predicted class: 7 7
Predicted class: 7 7
Predicted class: 7 7
Predicted class: 4 8
Predicted class: 3 8
Predicted class: 3 8
Predicted class: 3 8
Predicted class: 9 9
Predicted class: 9 9
Predicted class: 8 9
Predicted class: 5 9
accuracy = 0.43333333333333335
```

예상값과 정답, 그리고 output출력
정확도 : 약 43%정도