최종 정확도: 7518.0/10000 (**75.18000%**)

CIFAR10 인식 정확도 챌린지 리포트

201710780 성시원

<result 001>

<result 001> - Pytorch - GPU

GPU 환경을 다음과 같이 구축함.

- OS Windows 10
- anaconda3 4.8.3
- python 3.7.4
- NVIDIA 그래픽 드라이버 442.19
- NVIDIA CUDA Toolkit 10.0
- NVIDIA cuDNN 7.6.5.32
- Pytorch 1.2.0

0. Pytorch GPU 환경 구축

[참고1] https://pytorch.org/get-started/locally/
[참고2] https://lsjsj92.tistory.com/494
[참고3] PyTorch를 활용한 강화학습/심층강화학습 실전 입문 (218p ~ 219p)

```
conda install pytorch torchvision cudatoolkit=10.0 -c pytorch
```

```
Collecting package metadata (current_repodata.json): ...working... done
Solving environment: ...working... done

# All requested packages already installed.


Note: you may need to restart the kernel to use updated packages.
```

```
print(torch.__version__)
```

```
1.2.0
```

```
torch.cuda.is_available()
```

```
True
```

```
if torch.cuda.is_available():
    print(torch.cuda.get_device_name(0))
else:
    print("cpu")
```

```
GeForce GTX 1060 6GB
```

```
use_cuda = torch.cuda.is_available()
device = torch.device("cuda" if use_cuda else "cpu")
print(device)

# 사용방법: 명령어.to(device) → GPU환경, CPU환경에 맞춰서 동작
```

```
cuda
```

<result 001> - Network Model

네트워크 모델은 다음과 같이 구현하였으며, 실습 당시 사용하던 구조를 수정하지 않았다.

4. 모델 선언

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.layer = nn.Sequential(
            # 입력채널 수, 출력채널 수, 필터크기, 패딩
            nn.Conv2d(3, 16, 3, padding=1), # 32*32*3 → 32*32*16
            nn.ReLU(),
#             nn.Dropout2d(0, 2), # (1) Drop out
#             nn.BatchNorm2d(16), # (5) Batch normalization
            nn.Conv2d(16, 32, 3, padding=1), # 32*32*16 → 32*32*32
            nn.ReLU(),
#             nn.Dropout2d(0, 2),
#             nn.BatchNorm2d(32),
            nn.MaxPool2d(2, 2), # 32*32*32 → 16*16*32
            nn.Conv2d(32, 64, 3, padding=1), # 16*16*32 → 16*16*64
            nn.ReLU(),
#             nn.Dropout2d(0, 2),
#             nn.BatchNorm2d(64),
            nn.MaxPool2d(2, 2) # 16*16*64 → 8*8*64
        )
        self.fc_layer=nn.Sequential(
            nn.Linear(64*8*8, 100), # 8*8*64 = 4096 → 100
            nn.ReLU(),
#             nn.Dropout2d(0, 2),
#             nn.BatchNorm1d(100), # 1줄 → BatchNorm1d
            nn.Linear(100, 10) # 100 → 10
        )

    def forward(self, x):
        out=self.layer(x)
        out=out.view(batch_size, -1)
        out=self.fc_layer(out)

        return out
model = CNN().to(device)
```

<result 001> - Hyper Parameters

전체 학습 과정을 에러없이 구동할 수 있는지 확인하기위해 epoch의 크기를 3으로 조절함.

Hyper Parameters

```
1  batch_size=16
2  learning_rate=0.001
3  num_epoch=3
```

<result 001> - Result

- 실행시간: 에폭 당 약 15초
- Accuracy of Test Data: 3067.0/10000 (**30.67000%**)

- 결과 분석: Pytorch-GPU 환경 구축부터 pkl 파일 저장, 불러오기 까지 에러 없이 실행 성공

<result 002>

<result 002> - Network Model

네트워크 모델은 다음과 같이 구현하였으며, <result 001>에서 사용하던 구조를 수정하지 않았다.

4. 모델 선언

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.layer = nn.Sequential(
            # 입력채널 수, 출력채널 수, 필터크기, 패딩
            nn.Conv2d(3, 16, 3, padding=1), # 32*32*3 → 32*32*16
            nn.ReLU(),
#             nn.Dropout2d(0, 2), # (1) Drop out
#             nn.BatchNorm2d(16), # (5) Batch normalization
            nn.Conv2d(16, 32, 3, padding=1), # 32*32*16 → 32*32*32
            nn.ReLU(),
#             nn.Dropout2d(0, 2),
#             nn.BatchNorm2d(32),
            nn.MaxPool2d(2, 2), # 32*32*32 → 16*16*32
            nn.Conv2d(32, 64, 3, padding=1), # 16*16*32 → 16*16*64
            nn.ReLU(),
#             nn.Dropout2d(0, 2),
#             nn.BatchNorm2d(64),
            nn.MaxPool2d(2, 2) # 16*16*64 → 8*8*64
        )
        self.fc_layer=nn.Sequential(
            nn.Linear(64*8*8, 100), # 8*8*64 = 4096 → 100
            nn.ReLU(),
#             nn.Dropout2d(0, 2),
#             nn.BatchNorm1d(100), # 1줄 → BatchNorm1d
            nn.Linear(100, 10) # 100 → 10
        )

    def forward(self, x):
        out=self.layer(x)
        out=out.view(batch_size, -1)
        out=self.fc_layer(out)

        return out
model = CNN().to(device)
```

<result 002> - Hyper Parameters

본격적으로 학습을 시작. epoch의 크기를 100으로 조절함.

Hyper Parameters

```
1  batch_size=16
2  learning_rate=0.001
3  num_epoch=100
```

<result 002> - Result

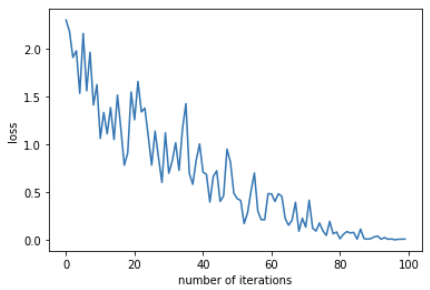
mlp_weight_002.pkl

- 실행시간: 1705.96초
- Accuracy of Test Data: 6494.0/10000 (**64.93999%**)
- 결과 분석:

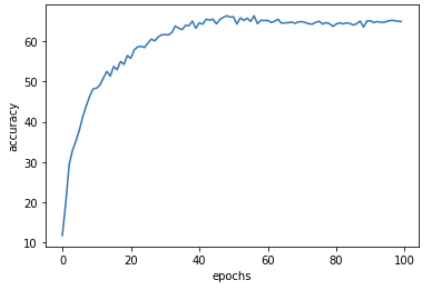
  Loss값의 변화를 보면, 감소하다가 중간에 증가하는 모습을 볼 수 있는데, 이는 Learning Rate가 너무 높기 때문이라고 추측할 수 있다.

```
Running Time: 2.934156 s

1000 tensor(0.0025, device='cuda:0', grad_fn=<NllLossBackward>)
Running Time: 4.884823 s

2000 tensor(0.0038, device='cuda:0', grad_fn=<NllLossBackward>)
Running Time: 4.840037 s

3000 tensor(0.0018, device='cuda:0', grad_fn=<NllLossBackward>)
Running Time: 4.891539 s

Accuracy of Test Data: 6494.0/10000 (64.93999%)

*Running Time Epoch 99: 15.252188 s


-------------------------

Total Running Time: 1705.964684 s
```

```
plt.plot(itr_arr, loss_arr) # 반복 횟수에 따른 손실함수
plt.xlabel("number of iterations")
plt.ylabel("loss")
plt.show()
```




```
plt.plot(itr_arr, accr_arr) # 반복 횟수에 따른 정확도
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.show()
```



<result 003>

<result 003> - Network Model

네트워크 모델은 다음과 같이 구현하였으며, <result 002>에서 사용하던 구조를 수정하지 않았다.

4. 모델 선언

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.layer = nn.Sequential(
            # 입력채널 수, 출력채널 수, 필터크기, 패딩
            nn.Conv2d(3, 16, 3, padding=1), # 32*32*3 → 32*32*16
            nn.ReLU(),
#             nn.Dropout2d(0, 2), # (1) Drop out
#             nn.BatchNorm2d(16), # (5) Batch normalization
            nn.Conv2d(16, 32, 3, padding=1), # 32*32*16 → 32*32*32
            nn.ReLU(),
#             nn.Dropout2d(0, 2),
#             nn.BatchNorm2d(32),
            nn.MaxPool2d(2, 2), # 32*32*32 → 16*16*32
            nn.Conv2d(32, 64, 3, padding=1), # 16*16*32 → 16*16*64
            nn.ReLU(),
#             nn.Dropout2d(0, 2),
#             nn.BatchNorm2d(64),
            nn.MaxPool2d(2, 2) # 16*16*64 → 8*8*64
        )
        self.fc_layer=nn.Sequential(
            nn.Linear(64*8*8, 100), # 8*8*64 = 4096 → 100
            nn.ReLU(),
#             nn.Dropout2d(0, 2),
#             nn.BatchNorm1d(100), # 1줄 → BatchNorm1d
            nn.Linear(100, 10) # 100 → 10
        )

    def forward(self, x):
        out=self.layer(x)
        out=out.view(batch_size, -1)
        out=self.fc_layer(out)

        return out
model = CNN().to(device)
```

<result 003> - Hyper Parameters

적절한 learning rate를 찾는 것이 가장 먼저 할 일이라고 생각하였다.

Learning rate를 더 작은 값으로 설정했을 때의 loss 변화를 확인하기위해, 0.0001로 낮추고, 같은 이유로 loss의 변화에만 주목하기위해 epoch의 크기를 30으로 줄임.

Hyper Parameters

```
1  batch_size=16
2  learning_rate=0.0001
3  num_epoch=30
```

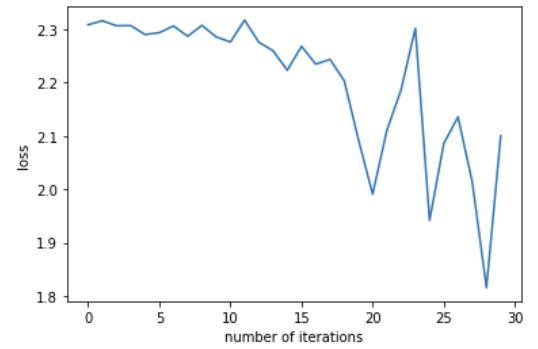
<result 003> - Result

- 실행시간: 485.42초
- Accuracy of Test Data: 2930.0/10000 (**29.63000%**)
- 결과 분석:

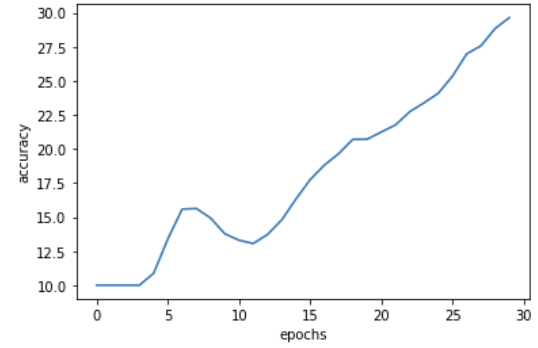
  Learning Rate가 여전히 높다고 추측할 수 있으며, epoch 값이 너무 작아 loss의 변화를 관찰하기엔 부족한 반복 횟수라고 볼 수 있다.

```
Running Time: 2.735681 s

1000 tensor(2.0531, device='cuda:0', grad_fn=<NllLossBackward>)
Running Time: 4.429163 s

2000 tensor(1.9606, device='cuda:0', grad_fn=<NllLossBackward>)
Running Time: 4.609690 s

3000 tensor(2.1368, device='cuda:0', grad_fn=<NllLossBackward>)
Running Time: 4.470582 s

Accuracy of Test Data: 2963.0/10000 (29.63000%)

*Running Time Epoch 29: 14.043754 s


------------------------

Total Running Time: 485.418658 s
```

```
plt.plot(itr_arr, loss_arr) # 반복 횟수에 따른 손실함수
plt.xlabel("number of iterations")
plt.ylabel("loss")
plt.show()
```




```
plt.plot(itr_arr, accr_arr) # 반복 횟수에 따른 정확도
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.show()
```



<result 004>

<result 004> - Network Model

네트워크 모델은 다음과 같이 구현하였으며, <result 003>에서 사용하던 구조를 수정하지 않았다.

4. 모델 선언

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.layer = nn.Sequential(
            # 입력채널 수, 출력채널 수, 필터크기, 패딩
            nn.Conv2d(3, 16, 3, padding=1), # 32*32*3 → 32*32*16
            nn.ReLU(),
#             nn.Dropout2d(0, 2), # (1) Drop out
#             nn.BatchNorm2d(16), # (5) Batch normalization
            nn.Conv2d(16, 32, 3, padding=1), # 32*32*16 → 32*32*32
            nn.ReLU(),
#             nn.Dropout2d(0, 2),
#             nn.BatchNorm2d(32),
            nn.MaxPool2d(2, 2), # 32*32*32 → 16*16*32
            nn.Conv2d(32, 64, 3, padding=1), # 16*16*32 → 16*16*64
            nn.ReLU(),
#             nn.Dropout2d(0, 2),
#             nn.BatchNorm2d(64),
            nn.MaxPool2d(2, 2) # 16*16*64 → 8*8*64
        )
        self.fc_layer=nn.Sequential(
            nn.Linear(64*8*8, 100), # 8*8*64 = 4096 → 100
            nn.ReLU(),
#             nn.Dropout2d(0, 2),
#             nn.BatchNorm1d(100), # 1줄 → BatchNorm1d
            nn.Linear(100, 10) # 100 → 10
        )

    def forward(self, x):
        out=self.layer(x)
        out=out.view(batch_size, -1)
        out=self.fc_layer(out)

        return out
model = CNN().to(device)
```

<result 004> - Hyper Parameters

Learning rate를 더 작은 값으로 설정했을 때의 loss 변화를 확인하기위해, 0.00001(10e-5)로 낮추고, loss의 변화를 더 오래 관찰하기 위해 epoch의 크기를 60으로 늘림.

Hyper Parameters

```
1  batch_size=16
2  learning_rate=0.00001
3  num_epoch=60
```

<result 004> - Result

- 실행시간: **학습중단**
- Accuracy of Test Data: 1000.0/10000 (**10.00000%**)
- 결과 분석:

  정확도 향상이 전혀 이루어지지 않는데, learning rate값이 너무 작은 것이라 추정.

```
Running Time: 2.629532 s

1000 tensor(2.2862, device='cuda:0', grad_fn=<NllLossBackward>)
Running Time: 4.690615 s


2000 tensor(2.2902, device='cuda:0', grad_fn=<NllLossBackward>)
Running Time: 4.706765 s

3000 tensor(2.2977, device='cuda:0', grad_fn=<NllLossBackward>)
Running Time: 4.723964 s

Accuracy of Test Data: 1000.0/10000 (10.00000%)

*Running Time Epoch 36: 14.720352 s
```

<result 005>

<result 005> - Network Model

네트워크 모델은 다음과 같이 구현하였으며, <result 004>에서 사용하던 구조를 수정하지 않았다.

4. 모델 선언

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.layer = nn.Sequential(
            # 입력채널 수, 출력채널 수, 필터크기, 패딩
            nn.Conv2d(3, 16, 3, padding=1), # 32*32*3 → 32*32*16
            nn.ReLU(),
#             nn.Dropout2d(0, 2), # (1) Drop out
#             nn.BatchNorm2d(16), # (5) Batch normalization
            nn.Conv2d(16, 32, 3, padding=1), # 32*32*16 → 32*32*32
            nn.ReLU(),
#             nn.Dropout2d(0, 2),
#             nn.BatchNorm2d(32),
            nn.MaxPool2d(2, 2), # 32*32*32 → 16*16*32
            nn.Conv2d(32, 64, 3, padding=1), # 16*16*32 → 16*16*64
            nn.ReLU(),
#             nn.Dropout2d(0, 2),
#             nn.BatchNorm2d(64),
            nn.MaxPool2d(2, 2) # 16*16*64 → 8*8*64
        )
        self.fc_layer=nn.Sequential(
            nn.Linear(64*8*8, 100), # 8*8*64 = 4096 → 100
            nn.ReLU(),
#             nn.Dropout2d(0, 2),
#             nn.BatchNorm1d(100), # 1줄 → BatchNorm1d
            nn.Linear(100, 10) # 100 → 10
        )

    def forward(self, x):
        out=self.layer(x)
        out=out.view(batch_size, -1)
        out=self.fc_layer(out)

        return out
model = CNN().to(device)
```

<result 005> - Hyper Parameters

<result 003>, <result 004>의 사이 값으로 설정해보고자 Learning rate를 0.00005로 설정하였다.

Hyper Parameters

```
batch_size=16
learning_rate=0.00005
num_epoch=60
```

<result 005> - Result

- 실행시간: **학습중단**
- Accuracy of Test Data: 1000.0/10000 (**10.00000%**)
- 결과 분석:

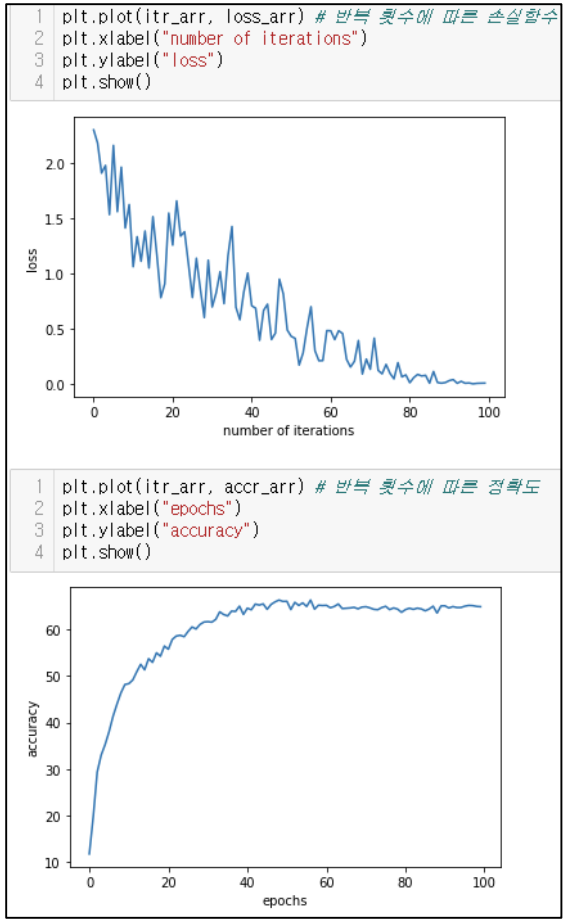
  마찬가지로 정확도 향상이 전혀 이루어지지 않았는데,
  learning rate값이 너무 작은 것이라 추정.

```
1000 tensor(2.3009, device='cuda:0', grad_fn=<NllLossBackward>)
Running Time: 4.568338 s

2000 tensor(2.2926, device='cuda:0', grad_fn=<NllLossBackward>)
Running Time: 4.773825 s

3000 tensor(2.3068, device='cuda:0', grad_fn=<NllLossBackward>)
Running Time: 4.697783 s

Accuracy of Test Data: 1000.0/10000 (10.00000%)

*Running Time Epoch 5: 14.656293 s


<num_epoch: 6>
0 tensor(2.2891, device='cuda:0', grad_fn=<NllLossBackward>)
Running Time: 2.739132 s
```

<result 006>

<result 006> - 이전 결과 재분석

- <result 002>는 epoch 40까지는 빠르게 accuracy가 증가하는 것을 확인 할 수 있으며, 이후로 정체되는 모습을 확인할 수 있다.
- 실제 loss 함수는 굉장히 복잡할 것이기 때문에, 중간 중간에 loss 값이 증가하는 건 충분히 일어날 수 있는 일이라고 생각하였다.
- 따라서 learning rate 0.001 자체는 크게 문제 될 것이 아니라고 판단하였다. 다만, 정체되는 구간부터는 learning 감소시켜야 한다고 판단하였다. (learning rate decay)



<result 002>

<result 006> - Network Model

네트워크 모델은 다음과 같이 구현하였으며, <result 005>에서 사용하던 구조를 수정하지 않았다.

4. 모델 선언

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.layer = nn.Sequential(
            # 입력채널 수, 출력채널 수, 필터크기, 패딩
            nn.Conv2d(3, 16, 3, padding=1), # 32*32*3 → 32*32*16
            nn.ReLU(),
#             nn.Dropout2d(0, 2), # (1) Drop out
#             nn.BatchNorm2d(16), # (5) Batch normalization
            nn.Conv2d(16, 32, 3, padding=1), # 32*32*16 → 32*32*32
            nn.ReLU(),
#             nn.Dropout2d(0, 2),
#             nn.BatchNorm2d(32),
            nn.MaxPool2d(2, 2), # 32*32*32 → 16*16*32
            nn.Conv2d(32, 64, 3, padding=1), # 16*16*32 → 16*16*64
            nn.ReLU(),
#             nn.Dropout2d(0, 2),
#             nn.BatchNorm2d(64),
            nn.MaxPool2d(2, 2) # 16*16*64 → 8*8*64
        )
        self.fc_layer=nn.Sequential(
            nn.Linear(64*8*8, 100), # 8*8*64 = 4096 → 100
            nn.ReLU(),
#             nn.Dropout2d(0, 2),
#             nn.BatchNorm1d(100), # 1줄 → BatchNorm1d
            nn.Linear(100, 10) # 100 → 10
        )

    def forward(self, x):
        out=self.layer(x)
        out=out.view(batch_size, -1)
        out=self.fc_layer(out)

        return out
model = CNN().to(device)
```

<result 006> - Hyper Parameters

<result 002>에서 사용한 값으로 설정해보고자 Learning rate를 0.001로 설정하였으며, learning rate decay가 효과적인지 확인하기 위해 epoch을 80으로 설정하였다.

Hyper Parameters

```
1  batch_size=16
2  learning_rate=0.001
3  num_epoch=80
```

learning rate decay

```
1  scheduler = lr_scheduler.StepLR(optimizer, step_size=20, gamma=0.1)
2  # gamma (float) - Multiplicative factor of learning rate decay. Default: 0.1.
```

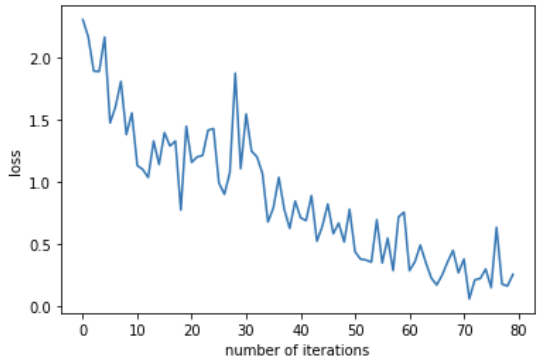
<result 006> - Result

- 실행시간: 1354.96초
- Accuracy of Test Data: 6282.0/10000 (62.82000%)
- 결과 분석:

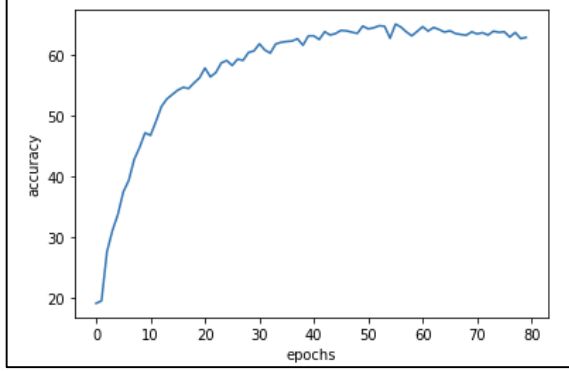
  정체구간이 여전히 존재, 최적화 알고리즘을 바꾸고, 감마 값(감소량)을 더 크게 설정하고자 하였다.

```
Running Time: 2.680827 s

1000 tensor(0.2446, device='cuda:0', grad_fn=<NllLossBackward>)
Running Time: 4.711749 s

2000 tensor(0.1445, device='cuda:0', grad_fn=<NllLossBackward>)
Running Time: 4.708110 s

3000 tensor(0.0312, device='cuda:0', grad_fn=<NllLossBackward>)
Running Time: 4.695331 s

Accuracy of Test Data: 6282.0/10000 (62.82000%)

*Running Time Epoch 79: 14.722586 s


------------------------

Total Running Time: 1354.955061 s
```

```
1 plt.plot(itr_arr, loss_arr) # 반복 횟수에 따른 손실함수
2 plt.xlabel("number of iterations")
3 plt.ylabel("loss")
4 plt.show()
```




```
1 plt.plot(itr_arr, accr_arr) # 반복 횟수에 따른 정확도
2 plt.xlabel("epochs")
3 plt.ylabel("accuracy")
4 plt.show()
```



<result 007>

<result 007> - Network Model

네트워크 모델은 다음과 같이 구현하였으며, <result 006>에서 사용하던 구조를 수정하지 않았다.

4. 모델 선언

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.layer = nn.Sequential(
            # 입력채널 수, 출력채널 수, 필터크기, 패딩
            nn.Conv2d(3, 16, 3, padding=1), # 32*32*3 → 32*32*16
            nn.ReLU(),
#             nn.Dropout2d(0, 2), # (1) Drop out
#             nn.BatchNorm2d(16), # (5) Batch normalization
            nn.Conv2d(16, 32, 3, padding=1), # 32*32*16 → 32*32*32
            nn.ReLU(),
#             nn.Dropout2d(0, 2),
#             nn.BatchNorm2d(32),
            nn.MaxPool2d(2, 2), # 32*32*32 → 16*16*32
            nn.Conv2d(32, 64, 3, padding=1), # 16*16*32 → 16*16*64
            nn.ReLU(),
#             nn.Dropout2d(0, 2),
#             nn.BatchNorm2d(64),
            nn.MaxPool2d(2, 2) # 16*16*64 → 8*8*64
        )
        self.fc_layer=nn.Sequential(
            nn.Linear(64*8*8, 100), # 8*8*64 = 4096 → 100
            nn.ReLU(),
#             nn.Dropout2d(0, 2),
#             nn.BatchNorm1d(100), # 1줄 → BatchNorm1d
            nn.Linear(100, 10) # 100 → 10
        )

    def forward(self, x):
        out=self.layer(x)
        out=out.view(batch_size, -1)
        out=self.fc_layer(out)

        return out
model = CNN().to(device)
```

<result 007> - Hyper Parameters

learning rate decay가 효과적인지 확인하기 위해 gamma를 0.2로, epoch을 60으로 설정하였으며, 최적화 방법을 SGD에서 AdamOptimizer로 변경하였다.

Hyper Parameters

```
batch_size=16
learning_rate=0.001
num_epoch=60
```

5. Loss, Optimizer

```
# loss_func = nn.CrossEntropyLoss() # 분류 → 크로스엔트로피 → logit(# of classe
# optimizer = optim.SGD(model.parameters(), lr=learning_rate) # 최적화 방법: Sto

loss_func = nn.CrossEntropyLoss() # 분류 → 크로스엔트로피 → logit(# of classes,
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate) # Adam optimi
```

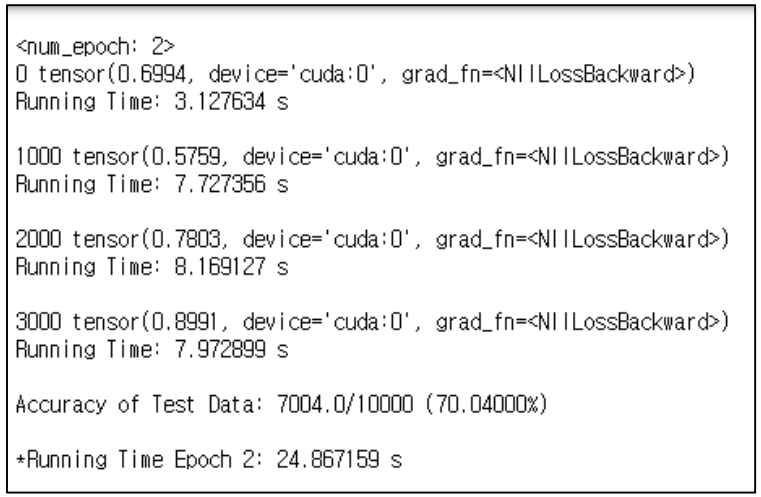
learning rate decay

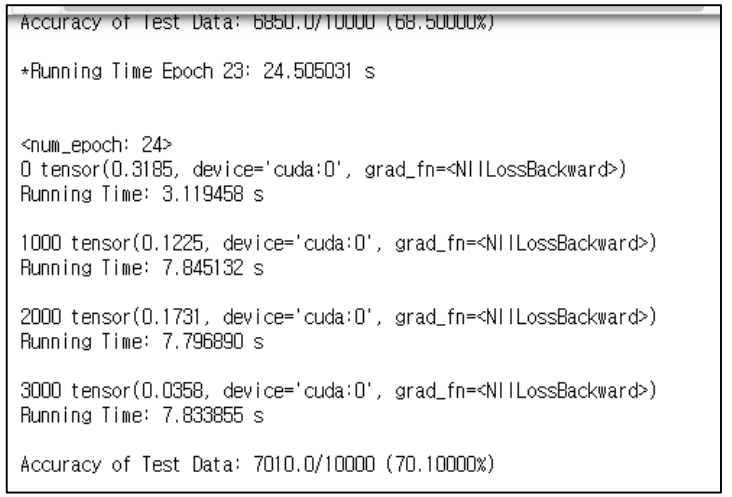
```
scheduler = lr_scheduler.StepLR(optimizer, step_size=20, gamma=0.2)
# gamma (float) - Multiplicative factor of learning rate decay. Default: 0.1.
```

<result 007> - Result

- 실행시간: **학습중단**
- Accuracy of Test Data: 7010.0/10000 (**70.10000%**)
- 결과 분석:

  Epoch 2에서 70%의 정확도를 달성하였으나, 이후 loss값과 accuracy값이 이 수렴을 하지않고 진동함. 따라서 learning rate값의 재조정, LeakyReLU, batch normalization, dropout을 추가로 설정함





<result 008>

<result 008> - Network Model

네트워크 모델은 다음과 같이 구현하였으며, 활성화 함수를 LeakyReLU로, Regularization을 위해 dropout을 추가하였고, 모든 레이어 값들의 분포를 가우시안 분포를 따르게 하기 위해, batch normalization을 추가하였다.

4. 모델 선언 ¶

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.layer = nn.Sequential(
            # 입력채널 수, 출력채널 수, 필터크기, 패딩
            nn.Conv2d(3, 16, 3, padding=1), # 32*32*3 → 32*32*16
            nn.LeakyReLU(),
            nn.Dropout2d(0, 2), # Drop out
            nn.BatchNorm2d(16), # Batch normalization

            nn.Conv2d(16, 32, 3, padding=1), # 32*32*16 → 32*32*32
            nn.LeakyReLU(), # Activation function: LeakyReLU
            nn.Dropout2d(0, 2),
            nn.BatchNorm2d(32),

            nn.MaxPool2d(2, 2), # 32*32*32 → 16*16*32

            nn.Conv2d(32, 64, 3, padding=1), # 16*16*32 → 16*16*64
            nn.LeakyReLU(),
            nn.Dropout2d(0, 2),
            nn.BatchNorm2d(64),

            nn.MaxPool2d(2, 2) # 16*16*64 → 8*8*64
        )
        self.fc_layer=nn.Sequential(
            nn.Linear(64*8*8, 100), # 8*8*64 = 4096 → 100
            nn.LeakyReLU(),
            nn.Dropout2d(0, 2),
            nn.BatchNorm1d(100), # 1줄 → BatchNorm1d

            nn.Linear(100, 10) # 100 → 10
        )

    def forward(self, x):
        out=self.layer(x)
        out=out.view(batch_size, -1)
        out=self.fc_layer(out)

        return out
model = CNN().to(device)
```

<result 008> - Hyper Parameters

진동하는 것을 방지하고자, Learning rate를 0.0001로, 학습 시간이 오래 걸릴 것을 고려하여 epoch을 30으로 설정하였다.

Hyper Parameters

```
1  batch_size=16
2  learning_rate=0.0001
3  num_epoch=30
```

5. Loss, Optimizer

```
1  # loss_func = nn.CrossEntropyLoss() # 분류 → 크로스엔트로피 → logit(# of classe
2  # optimizer = optim.SGD(model.parameters(), lr=learning_rate) # 최적화 방법: Sto
3
4  loss_func = nn.CrossEntropyLoss() # 분류 → 크로스엔트로피 → logit(# of classes,
5  optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate) # Adam optimi
```

learning rate decay

```
1  scheduler = lr_scheduler.StepLR(optimizer, step_size=20, gamma=0.2)
2  # gamma (float) - Multiplicative factor of learning rate decay. Default: 0.1.
```

<result 008> - Result

- 실행시간: 약 20분
- Accuracy of Test Data: 약 70%
- 결과 분석:

  loss값과 accuracy값이 이 수렴을 하지않고 진동함.

  nn.Dropout2d(0, 2) → 오타로 인해 드랍 아웃 비율이 0%로 학습된 듯

```
nn.Conv2d(3, 16, 3, padding=1),
nn.LeakyReLU(),
nn.Dropout2d(0, 2), # Drop out
nn.BatchNorm2d(16), # Batch nor
```

<result 009>

<result 009> - Network Model

<result 008>에서의 네트워크에서 Dropout2d()의 파라미터들만 0.2로 전부 수정하였다.

4. 모델 선언

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.layer = nn.Sequential(
            # 입력채널 수, 출력채널 수, 필터크기, 패딩
            nn.Conv2d(3, 16, 3, padding=1), # 32*32*3 → 32*32*16
            nn.LeakyReLU(),
            nn.Dropout2d(0.2), # Drop out
            nn.BatchNorm2d(16), # Batch normalization

            nn.Conv2d(16, 32, 3, padding=1), # 32*32*16 → 32*32*32
            nn.LeakyReLU(), # Activation function: LeakyReLU
            nn.Dropout2d(0.2),
            nn.BatchNorm2d(32),

            nn.MaxPool2d(2, 2), # 32*32*32 → 16*16*32

            nn.Conv2d(32, 64, 3, padding=1), # 16*16*32 → 16*16*64
            nn.LeakyReLU(),
            nn.Dropout2d(0.2),
            nn.BatchNorm2d(64),

            nn.MaxPool2d(2, 2) # 16*16*64 → 8*8*64
        )
        self.fc_layer=nn.Sequential(
            nn.Linear(64*8*8, 100), # 8*8*64 = 4096 → 100
            nn.LeakyReLU(),
            nn.Dropout2d(0.2),
            nn.BatchNorm1d(100), # 1줄 → BatchNorm1d

            nn.Linear(100, 10) # 100 → 10
        )

    def forward(self, x):
        out=self.layer(x)
        out=out.view(batch_size, -1)
        out=self.fc_layer(out)

        return out
model = CNN().to(device)
```

<result 009> - Hyper Parameters

<result 008>과 동일하게 설정하였다.

Hyper Parameters

```
1  batch_size=16
2  learning_rate=0.0001
3  num_epoch=30
```

5. Loss, Optimizer

```
1  # loss_func = nn.CrossEntropyLoss() # 분류 → 크로스엔트로피 → logit(# of classe
2  # optimizer = optim.SGD(model.parameters(), lr=learning_rate) # 최적화 방법: Sto
3
4  loss_func = nn.CrossEntropyLoss() # 분류 → 크로스엔트로피 → logit(# of classes,
5  optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate) # Adam optimi
```

learning rate decay

```
1  scheduler = lr_scheduler.StepLR(optimizer, step_size=20, gamma=0.2)
2  # gamma (float) - Multiplicative factor of learning rate decay. Default: 0.1.
```

<result 009> - Result

mlp_weight_009.pkl

- 실행시간: 1343.91초
- Accuracy of Test Data: 7496.0/10000 (**74.96000%**)
- 결과 분석:
  Loss값은 dropout의 영향으로 크게 진동하는 것으로 보임.
  Accuracy값은 학습 결과보다, 테스트 결과가 더 높게 나옴. (이 역시 dropout의 영향으로 보임)

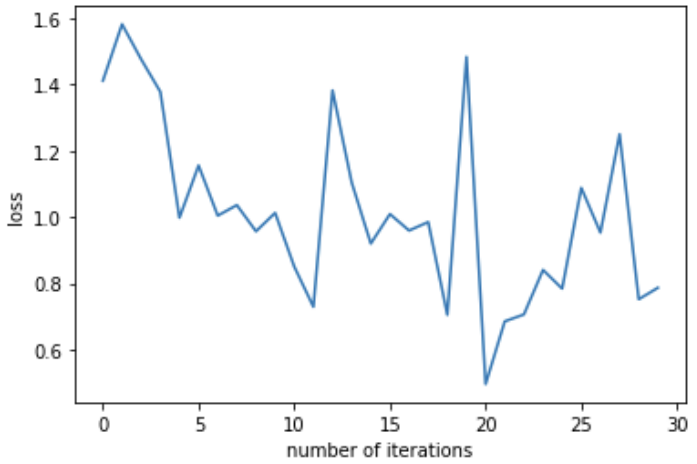
7. 테스트

```
model.eval()
ComputeAccr(test_loader, model)
```

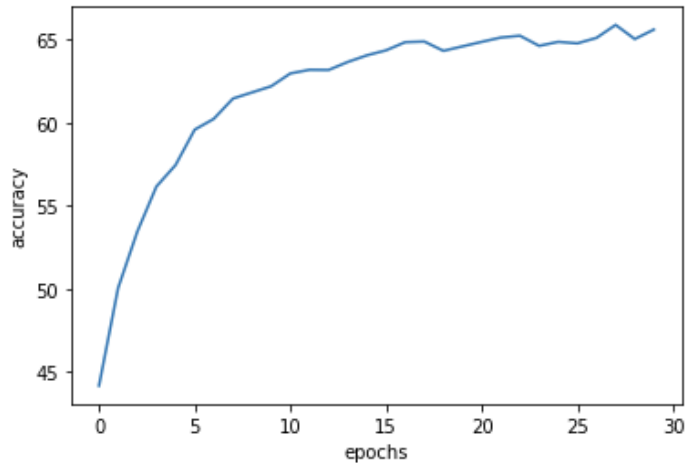
```
C:₩Users₩KIMA₩Anaconda3₩lib₩site-packages₩ipyker
tile was removed and now has no effect. Use `wit

Accuracy of Test Data: 7496.0/10000 (74.96000%)


tensor(74.9600, device='cuda:0')
```

```
Running Time: 4.923829 s

1000 tensor(1.0323, device='cuda:0', grad_fn=<NllLossBackward>)
Running Time: 13.389433 s

2000 tensor(0.3378, device='cuda:0', grad_fn=<NllLossBackward>)
Running Time: 13.221134 s

3000 tensor(0.8268, device='cuda:0', grad_fn=<NllLossBackward>)
Running Time: 13.376904 s

Accuracy of Test Data: 6560.0/10000 (65.60000%)

*Running Time Epoch 29: 41.640268 s


------------------------

Total Running Time: 1343.911493 s
```

```
plt.plot(itr_arr, loss_arr) # 반복 횟수에 따른 손실함수
plt.xlabel("number of iterations")
plt.ylabel("loss")
plt.show()
```




```
plt.plot(itr_arr, accr_arr) # 반복 횟수에 따른 정확도
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.show()
```



<result 010>

<result 010> - Network Model

<result 009>에서의 네트워크를 그대로 사용하였다.

4. 모델 선언

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.layer = nn.Sequential(
            # 입력채널 수, 출력채널 수, 필터크기, 패딩
            nn.Conv2d(3, 16, 3, padding=1), # 32*32*3 → 32*32*16
            nn.LeakyReLU(),
            nn.Dropout2d(0.2), # Drop out
            nn.BatchNorm2d(16), # Batch normalization

            nn.Conv2d(16, 32, 3, padding=1), # 32*32*16 → 32*32*32
            nn.LeakyReLU(), # Activation function: LeakyReLU
            nn.Dropout2d(0.2),
            nn.BatchNorm2d(32),

            nn.MaxPool2d(2, 2), # 32*32*32 → 16*16*32

            nn.Conv2d(32, 64, 3, padding=1), # 16*16*32 → 16*16*64
            nn.LeakyReLU(),
            nn.Dropout2d(0.2),
            nn.BatchNorm2d(64),

            nn.MaxPool2d(2, 2) # 16*16*64 → 8*8*64
        )
        self.fc_layer=nn.Sequential(
            nn.Linear(64*8*8, 100), # 8*8*64 = 4096 → 100
            nn.LeakyReLU(),
            nn.Dropout2d(0.2),
            nn.BatchNorm1d(100), # 1줄 → BatchNorm1d

            nn.Linear(100, 10) # 100 → 10
        )

    def forward(self, x):
        out=self.layer(x)
        out=out.view(batch_size, -1)
        out=self.fc_layer(out)

        return out
model = CNN().to(device)
```

<result 010> - Hyper Parameters

Learning rate decay를 고려하여, 초기 learning rate를 0.1로 설정하고, epoch을 100으로 설정하였다. Decay의 gamma가 0.2여서 학습이 진행되면, 충분히 작은 learning rate로 학습이 이루어 질 것이라 판단하였다.

Hyper Parameters

```
batch_size=16
learning_rate=0.1
num_epoch=100
```

5. Loss, Optimizer

```
# loss_func = nn.CrossEntropyLoss() # 분류 → 크로스엔트로피 → logit(# of classe
# optimizer = optim.SGD(model.parameters(), lr=learning_rate) # 최적화 방법: Sto

loss_func = nn.CrossEntropyLoss() # 분류 → 크로스엔트로피 → logit(# of classes,
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate) # Adam optimi
```

learning rate decay

```
scheduler = lr_scheduler.StepLR(optimizer, step_size=20, gamma=0.2)
# gamma (float) - Multiplicative factor of learning rate decay. Default: 0.1.
```

<result 010> - Result

- 실행시간: 4642.30초
- Accuracy of Test Data: 6620.0/10000 (66.20000%)
- 결과 분석:

  Loss값과 Accuracy값은 매우 크게 진동하는 것으로 보임.

  Learning rate가 너무 큰 것으로 추정, 그래프와 테스트 결과가 상이하여 측정하기 어려움

mlp_weight_010.pkl

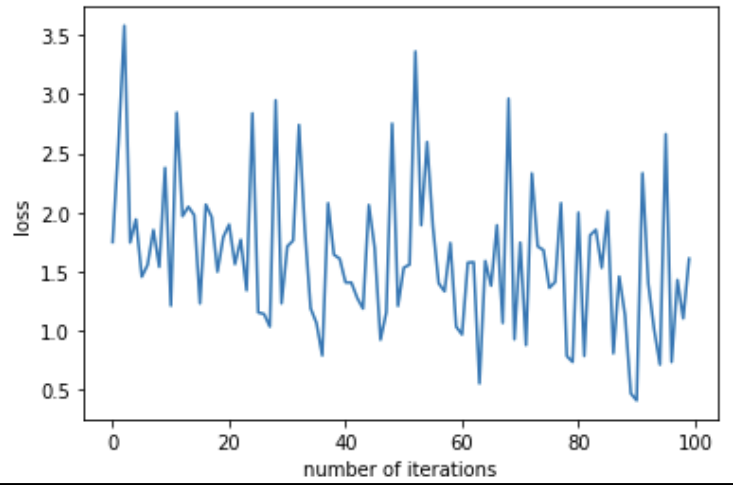
7. 테스트

```
model.eval()
ComputeAccr(test_loader, model)
```

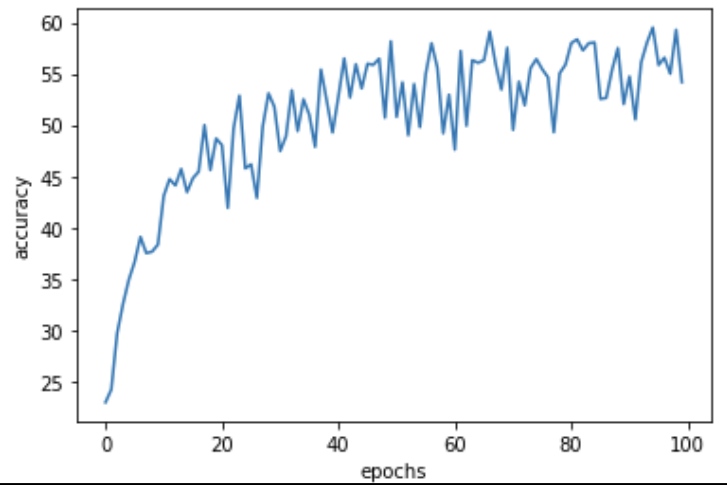
```
C:₩Users₩KIMA₩Anaconda3₩lib₩site-packages₩ipyker
has no effect. Use `with torch.no_grad():` inste

Accuracy of Test Data: 6620.0/10000 (66.20000%)


tensor(66.2000, device='cuda:0')
```

```
Running Time: 5.210109 s

1000 tensor(1.3361, device='cuda:0', grad_fn=<NllLossBacky
Running Time: 13.957690 s

2000 tensor(1.2393, device='cuda:0', grad_fn=<NllLossBacky
Running Time: 13.948722 s

3000 tensor(1.8137, device='cuda:0', grad_fn=<NllLossBacky
Running Time: 13.814047 s

Accuracy of Test Data: 5418.0/10000 (54.18000%)

*Running Time Epoch 99: 43.434868 s


------------------------


Total Running Time: 4642.304471 s
```

```
plt.plot(itr_arr, loss_arr) # 반복 횟수에 따른 손실함수
plt.xlabel("number of iterations")
plt.ylabel("loss")
plt.show()
```




```
plt.plot(itr_arr, accr_arr) # 반복 횟수에 따른 정확도
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.show()
```



<result 011>

<result 011> - Network Model

<result 010>에서의 네트워크를 그대로 사용하였다.

4. 모델 선언

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.layer = nn.Sequential(
            # 입력채널 수, 출력채널 수, 필터크기, 패딩
            nn.Conv2d(3, 16, 3, padding=1), # 32*32*3 → 32*32*16
            nn.LeakyReLU(),
            nn.Dropout2d(0.2), # Drop out
            nn.BatchNorm2d(16), # Batch normalization

            nn.Conv2d(16, 32, 3, padding=1), # 32*32*16 → 32*32*32
            nn.LeakyReLU(), # Activation function: LeakyReLU
            nn.Dropout2d(0.2),
            nn.BatchNorm2d(32),

            nn.MaxPool2d(2, 2), # 32*32*32 → 16*16*32

            nn.Conv2d(32, 64, 3, padding=1), # 16*16*32 → 16*16*64
            nn.LeakyReLU(),
            nn.Dropout2d(0.2),
            nn.BatchNorm2d(64),

            nn.MaxPool2d(2, 2) # 16*16*64 → 8*8*64
        )
        self.fc_layer=nn.Sequential(
            nn.Linear(64*8*8, 100), # 8*8*64 = 4096 → 100
            nn.LeakyReLU(),
            nn.Dropout2d(0.2),
            nn.BatchNorm1d(100), # 1줄 → BatchNorm1d

            nn.Linear(100, 10) # 100 → 10
        )

    def forward(self, x):
        out=self.layer(x)
        out=out.view(batch_size, -1)
        out=self.fc_layer(out)

        return out
model = CNN().to(device)
```

<result 011> - Network Model

그래프와 테스트 결과가 상이한 것을 없애기 위해, 정확도를 측정할 때, 임시로 model.eval()로 변경함

```
for i in range(num_epoch):
    print("<num_epoch: %d>" %i)
    for j, [image, label] in enumerate(train_loader): # batch_size 만큼
        x = Variable(image).to(device)
        y_ = Variable(label).to(device)

        optimizer.zero_grad()
        output = model.forward(x)
        loss = loss_func(output, y_)
        loss.backward() # back prop, - Gradient를 계산해서 각 노드에 저장
        optimizer.step() # weight 조정 - 저장된 Gradient를 이용해서 weight

        if j%1000==0:
            print(j, loss)
            end = time.time()
            print("Running Time: %f s₩n" %(end - start))
            start = time.time()

    model.eval() # 학습 진행 상황 확인용

    itr_arr.append(i) # 반복 횟수 저장
    loss_arr.append(loss) # 손실함수 값 저장

    epoch_end = time.time()
    accr_arr.append(ComputeAccr(test_loader, model)) # 정확도 값 저장, 원리
    print("*Running Time Epoch %d: %f s₩n₩n" %(i, epoch_end-epoch_start))
    epoch_start = time.time()

    model.train() # 학습 재개
```

<result 011> - Hyper Parameters

Learning rate decay를 고려하여, 초기 learning rate를 0.01로 설정하고, 충분한 학습이 이루어 지도록 epoch을 300으로 설정하였다.

Hyper Parameters

```
batch_size=16
learning_rate=0.01
num_epoch=300
```

5. Loss, Optimizer

```
# loss_func = nn.CrossEntropyLoss() # 분류 → 크로스엔트로피 → logit(# of classe
# optimizer = optim.SGD(model.parameters(), lr=learning_rate) # 최적화 방법: Sto

loss_func = nn.CrossEntropyLoss() # 분류 → 크로스엔트로피 → logit(# of classes,
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate) # Adam optimi
```

learning rate decay

```
scheduler = lr_scheduler.StepLR(optimizer, step_size=20, gamma=0.2)
# gamma (float) - Multiplicative factor of learning rate decay. Default: 0.1.
```

<result 011> - Result

mlp_weight_011.pkl

- 실행시간: 13358.17초
- Accuracy of Test Data: 7518.0/10000 (**75.18000%**)
- 결과 분석:
  Loss값이 매우 크게 진동, Accuracy값은 70% 부터 심하게 정체됨.

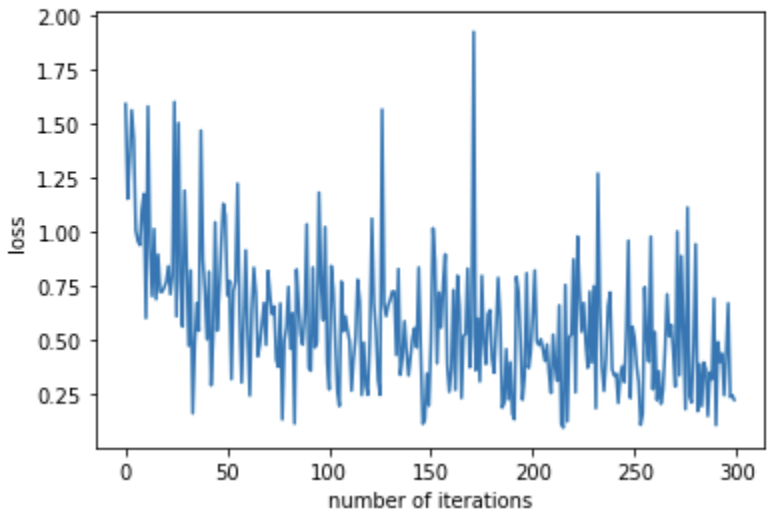
7. 테스트

```
model.eval()
ComputeAccr(test_loader, model)
```

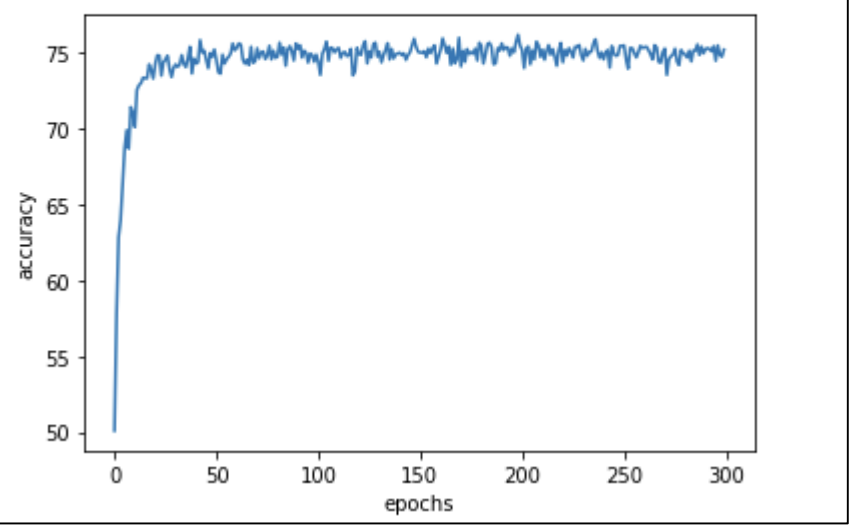
```
C:₩Users₩KIMA₩Anaconda3₩lib₩site-packages₩ipyker
tile was removed and now has no effect. Use `wit

Accuracy of Test Data: 7518.0/10000 (75.18000%)


tensor(75.1800, device='cuda:0')
```

```
Running Time: 4.302380 s

1000 tensor(0.1784, device='cuda:0', grad_fn=<NI
Running Time: 13.770845 s

2000 tensor(0.2607, device='cuda:0', grad_fn=<NI
Running Time: 13.448566 s

3000 tensor(0.3802, device='cuda:0', grad_fn=<NI
Running Time: 13.430997 s

Accuracy of Test Data: 7518.0/10000 (75.18000%)

*Running Time Epoch 299: 42.284800 s


------------------------

Total Running Time: 13358.171084 s
```

```
plt.plot(itr_arr, loss_arr) # 반복 횟수에 따른 손실함수
plt.xlabel("number of iterations")
plt.ylabel("loss")
plt.show()
```




```
plt.plot(itr_arr, accr_arr) # 반복 횟수에 따른 정확도
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.show()
```



Summary

<result 011>

batch_size=16

learning_rate=0.01

num_epoch=300

learning rate decay step=20, gamma=0.2

AdamOptimizer

LeakyReLU

batch normalization

dropout=0.2

13358.17초

Accuracy of Test Data: 7518.0/10000 (**75.18000%**)