Problem 1

1. Describe the architecture & implementation details of your model

```
I referenced the github (https://github.com/orobix/Prototypical-Networks-
for-Few-shot-Learning-PyTorch).

Implementation details:
a. 10-way 1-shot for training, 5-way 1-shot for testing
b. epochs: 100
c. optimizer: Adam with learning rate 0.001, for both model and parametric
function
d. scheduler: StepLR(optimizer, step_size=20, gamma=0.5)
e. transforms:
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406],
                             std=[0.229, 0.224, 0.225])
f. training episodes: 100

Accuracy on validation set and 95% confidence interval under 5-way 1-shot
(Euclidean distance) setting: 42.91 +- 0.79 %
```

2. Euclidean distance, cosine similarity and parametric function of 5-way 1-shot

```
eu:  42.91 +- 0.79 %

cos: 40.59 +- 0.79 %

param: 45.89 +- 0.83 % (The version I choose)

parametric function design:
```

```python
class Parametric_func(nn.Module):
    def __init__(self):
        super().__init__()
        self.enc = nn.Sequential(
            nn.Linear(3200, 1600),
            nn.ReLU(),
            nn.Linear(1600, 100),
            nn.ReLU(),
            nn.Linear(100,1)
        )


    def forward(self, a, b): # a: torch.Size([150, 1600]), b: torch.Size([10, 1600])
        n = a.shape[0]
        m = b.shape[0]
        a = a.unsqueeze(1).expand(n, m, -1) # torch.Size([150, 10, 1600])
        b = b.unsqueeze(0).expand(n, m, -1)
        c = torch.cat((a,b),2)
        # print(c.shape)
        ans = self.enc(c)
        # print(ans.shape)
        return ans.squeeze()
```

3. Compare the accuracy with different 5-way K-shot (K=1, 5, 10)

```
The results are shown below. It seems that more shot is better, and
Euclidean distance will be the best when increasing the number of shot.
```

distance / k-shot	Euclidean	cosine similarity	parametric function
1 shot	0.4351	0.4109	0.4697
5 shot	0.6452	0.5220	0.6091
10 shot	0.6973	0.5441	0.6280

Problem 2

1. Description of implementation of SSL

```
a. Self-pretrained stats

1. backbone = models.resnet50(pretrained=False)
2. The batch size of the mini dataset: 64
3. hyperparameters under the library:

    learner = BYOL(
        backbone,
        image_size = 128,
```

```
        hidden_layer = 'avgpool'
    )
    opt = torch.optim.Adam(learner.parameters(), lr=3e-4)

4. The loss at the last epoch: 0.2288914283675452
```

2. Table

setting	pretrained	finetuning	accuracy
A	None	full	0.2783
B	TA	full	0.3645
C	Self	full	0.3817 (to be tested)
D	TA	classifier only	0.2635
E	Self	classifier only	0.204

3. Analysis

```
a. The possible upper-bound

For the first three settings, the training-accuracy and loss have almost
reached 9X% and 0.01XX in 50 epochs. I think this implies accuracy on
validation set would be the best performance under this setting. While for
the D, E settings, the loss and accuracy on train-set in 50 epochs 3.XX and
0.22. I guess the possibility of the higher performance might exist, but
the time needed to be trained is unknown, and TAs seem not to ask us to do
as high as possible.

The first trial didn't pass the requirements, so I added some transforms
and let the training time longer (20 -> 50).

The results show that the pretrained performance would be the best, while
it's supriesed the classifier-only might worse that the absent of
pretrained module. As discussion above, the classifer-only would be better
if I train longer or finetune some part. But the result shows that under
the light, simple setting, A is better than D, E.

b. Downstream task setting

1. Transforms

    transforms.Resize((128,128)),

    transforms.RandomHorizontalFlip(),
```

```
    transforms.RandomGrayscale(p=0.1),

    transforms.ColorJitter(),

    transforms.ToTensor(),

    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])

2. Optimizers & Scheduler

    optimizer = torch.optim.Adam(params_to_update, lr=2e-4, betas=(0.5,
0.999))

    scheduler =
torch.optim.lr_scheduler.CosineAnnealingLR(optimizer,args.n_epochs(50) *
len(train_loader))
```