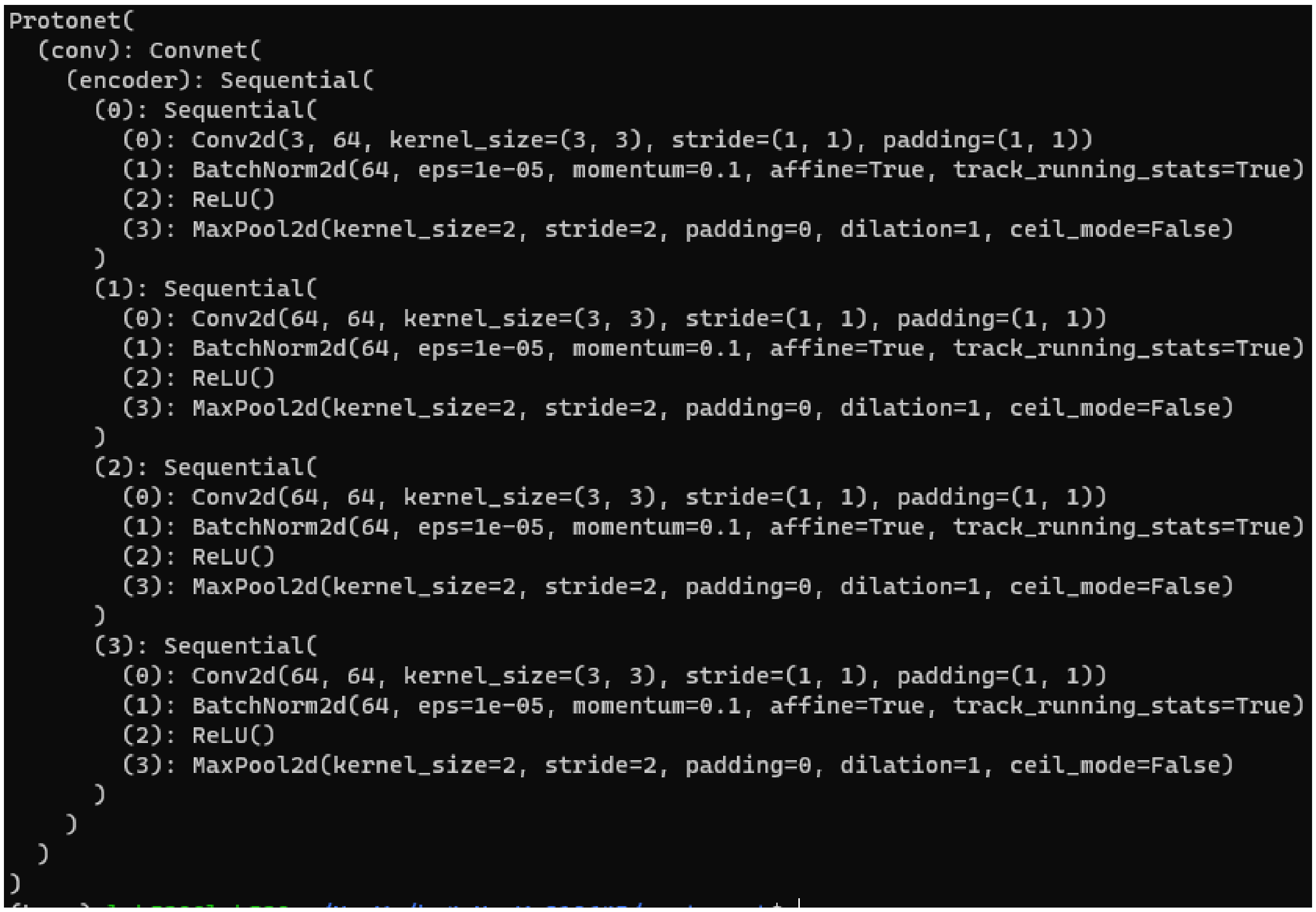
Deep Learning for Computer Vision (2021 Fall) HW4

R10942152 Ken Yu(游家權)

Problem 1: Few-Shot Learning - Prototypical Network(70%)

1. (20%) Describe the architecture & implementation details of your model. (Include but not limited to the number of training episodes, distance function, learning rate schedule, data augmentation, optimizer, and N-way K-shot setting for meta-train and meta-test phase)

Accuracy on validation set = **46.98%** +- 0.82



```
Protonet(
  (conv): Convnet(
    (encoder): Sequential(
      (0): Sequential(
        (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU()
        (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      )
      (1): Sequential(
        (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU()
        (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      )
      (2): Sequential(
        (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU()
        (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      )
      (3): Sequential(
        (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU()
        (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      )
    )
  )
)
```

<u>Network Design:</u>

I didn’t use any layers in MLP. I simply just reshape the features from Conv4 and use it as model output. I have tried using three linear layers for MLP and output a vector with 400 elements. But after a few trials and errors, I found out that not using any linear layers is actually the best. By training without MLP layers, I was able to achieve the baseline in 40 epochs.

My explanation is that the prototypical net uses features to find prototypes; thus, if the task isn’t super-hard, the raw features from the backbone would suffice. Any layer added to MLP might just mingled the output into a worse feature, or just reduce the dimension of the output. Since this prototypical network is mainly based on calculating distance, I think it’s better to give it a higher dimension feature to calculate with.

During the training process, I also found that the standard deviation of accuracy will be smaller if I train it without MLP, which means the performance of the model is steadier, and it’s because it has fewer parameters to train.

I’ve tried data augmentation method in meta-training, but it didn't help a lot. I used color-jitter, horizontal-flip, random-rotation, but all ended up with a mediocre result. I think the reason is that augmented images are very similar to the original image; thus, it won’t help the prototypical network to get a better prototype. I guess the prototype is about the same after using data augmentation. I think the other reason is I didn’t use MLP for prediction. This design is making it very hard to learn the similarities between original image and augmented image, since there’s no MLP to train with.

2. (20%) When meta-train and meta-test under the same 5-way 1-shot setting, please report and discuss the accuracy of the prototypical network using 3 different distance function (i.e., Euclidean distance, cosine similarity and parametric function). You should also describe how you design your parametric function.

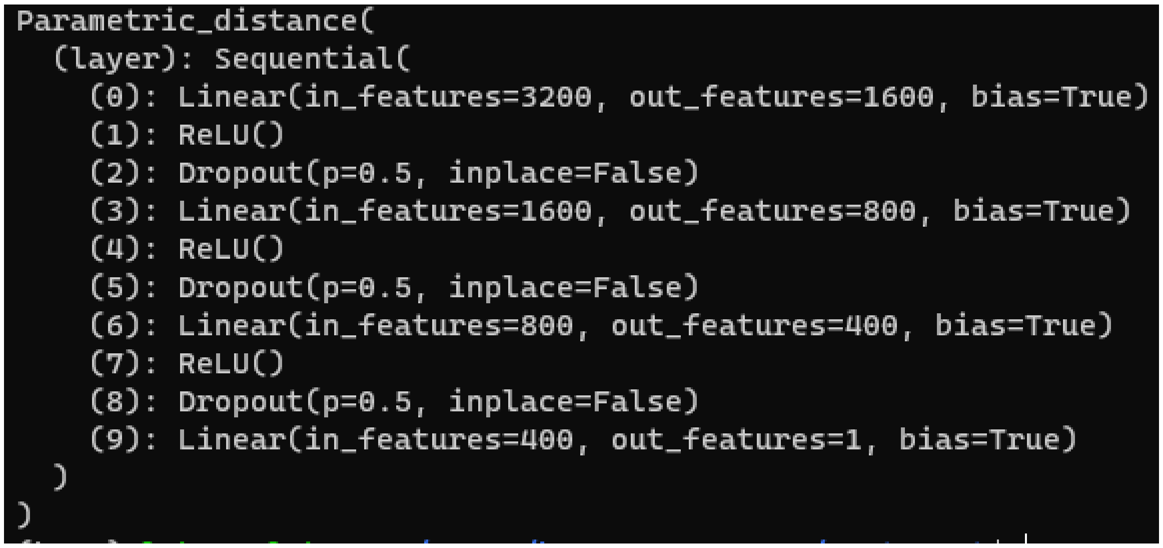
Euclidean distance	**cosine similarity**	**parametric function**
40.71% +- 0.77	**38.77% +- 0.75**	**45.15% +- 0.81**

My experiment shows that using a parametric function to learn the distance really makes a difference. Accuracy went up very quickly in early epochs, and after 30 epochs, it outperformed the other two distance functions, quite impressive.

I think it's because we use 1-shot learning in this experiment, and the mean of the support set is dependent on a single image. This mean can be biased if we draw an "unlucky" sample. The parametric function here helps us to tackle this problem. The weight it learned can be viewed as "taking average" across episodes. The effect of a single outliner can be mitigated by the parametric distance function.

As for L2 distance and Cosine function, L2 distance actually gives a better result. I think it's because the Cosine function only ranges from -1 to 1, which makes the Cosine function have a smaller range to express "distance". On the other hand, the value of L2 distance can go very big due to its square; therefore, it has a better chance to get a distinctive distance.

My Parametric function:



```
Parametric_distance(
  (layer): Sequential(
    (0): Linear(in_features=3200, out_features=1600, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=1600, out_features=800, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=800, out_features=400, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.5, inplace=False)
    (9): Linear(in_features=400, out_features=1, bias=True)
  )
)
```

I used four linear layers to make a downsample-like design to make the network deep enough while maintaining the number of parameters in a reasonable range. I also used ReLU as an activation layer and used Dropout to prevent overfitting.

3. (10%) When meta-train and meta-test under the same 5-way K-shot setting, please report and compare the accuracy with different shots. (K=1, 5, 10)

K=1	**K=5**	**K=10**
40.71% +- 0.77	**60.32% +- 0.7**	**77.03% +- 0.47**

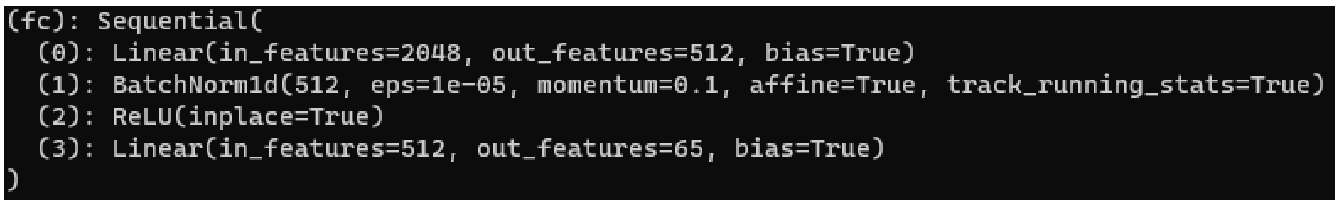
When we increase the size of the support set in the meta-training process, the accuracy goes up significantly. It received a 20% boost from 1-shot to 5-shot, and a 37% boost from 1-shot to 10-shot. I think it’s because the prototypical network uses support sets to calculate means, and uses these means to represent the classes. If we only allow 1-shot, the mean could become very biased by an “unlucky” sample. However, if we allow more shots in our support set, the effect of the outliers can be neutralized; therefore, get a more reliable and robust means to represent our support sets.

Also I think it’s worth noting that 10-shot 5-way learning has already reached 77% accuracy for image classification tasks. Although traditional supervised learning probably can reach almost 100% on image classification tasks, it actually needs to consume a huge amount of training data(i.e. 38400-shot 5-way in the mini dataset). So I think it’s very impressive that this “learn to compare” method shows its efficiency in these experiments.

Problem 2: Self-Supervised Pre-training for Image Classification (50%)

1. (10%) Describe the implementation details of your SSL method for pre-training the ResNet50 backbone. (Include but not limited to the name of the SSL method you used, data augmentation for SSL, learning rate schedule, optimizer, and batch size setting for this pre-training phase)

I used BYOL(Bootstrap Your Own Latent) as my SSL method, and used two linear layers as my classifier in the Resnet50 network. The classifier network part shows below.



```
(fc): Sequential(
  (0): Linear(in_features=2048, out_features=512, bias=True)
  (1): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): ReLU(inplace=True)
  (3): Linear(in_features=512, out_features=65, bias=True)
)
```

This design is based on [2]. Since the input feature is only 2048, I think a two-layer network is sufficient to fulfill image classification tasks, which have only 65 classes.

The SSL pre-training process is surprisingly slow. It took almost 24 hour to finish 100 epochs on my Titan RTX, and since there's no label during this training process, all I can observe is its training loss. The loss goes down quickly at first, but after a few epochs, loss goes down at a very slow speed. I also found that performance between 40-epochs-trained model and 100-epochs-trained model are about the same when testing on downstream tasks. This means SSL already taught backbone how to extract features in general images in just 40 epochs.

I tried tuning the batch size during SSL training, to speed up the process. However, I found that it doesn't get faster if I use a bigger batch size. This might be because the SSL training method isn't necessarily able to do parallel computing. Nevertheless, I found the loss drop slightly faster if I used a bigger batch size. So I end up using 128 batch sizes.

I tried data augmentation in the pre-training part, but it's not very helpful. I used color-jitter, horizontal-flip, random-rotation, but all ended up with a worse result after training on downstream tasks. I think it's because the momentum encoder inside the BYOL is ruined by augmented data. The momentum encoder should receive a big momentum because every batch is very different. However, when I use augmented data in my batches, it will probably generate a smaller momentum to the encoder and corrupt its mechanism.

2. (10%) Following Problem 2-1, please conduct the Image classification on the Office-Home dataset as the downstream task for your SSL method. Also, please complete the following Table, which contains different image classification settings, and compare the results.

Setting	Pre-training (Mini-ImageNet)	Fine-tuning (Office-Home dataset)	Classification accuracy on valid set (Office-Home dataset)
A	-	Train full model	26.11%
B	w/ label	Train full model	34.00%
C	w/o label	Train full model	**36.94%**
D	w/ label	Fix the backbone. Train classifier only	25.61%
E	w/o label	Fix the backbone. Train classifier only	35.22%

3. (10%) Discuss or analyze the results in Problem 2-2

Among all settings, the SSL pretrained model(setting C) gives the best accuracy in the Office-Home dataset. This means the SSL method I used truly helps Resnet50 initialize a better weight before training, making it adapt better at downstream tasks. On the other hand, the B setting, which did a supervised pretrain on the Mini dataset, performs slightly worse than the SSL pretrained model. As for A setting, which didn’t pretrain at all, gives even worse performance on downstream tasks. This shows how important pretrained weight can be when doing image classification on a small dataset. The “adapt speed” can be majorly improved, if we do a SSL pretrain first.

Setting D and E show the effect of fixing backbone when training downstream tasks. Comparing B to D, we see a performance drop of 9%, if we fix the backbone. This means TA provided models mainly rely on the backbone when doing the finetune training. The backbone learns how to extract new features of the new dataset, and uses it to classify images. On the other hand, Comparing C to E, the performance only drops a little bit, which implies my SSL pretrained model doesn’t heavily rely on backbone. It has already learned how to extract features from images in general; thus, it doesn’t matter if we fix the backbone.The backbone has already been able to do a good job. Overall, by observing these experiments, we can say that SSL pretrained technique is very beneficial when the dataset is small. I think the performance gap will be even bigger, if we test the model on a smaller dataset.

Reference

[1] Implement protonet

https://github.com/orobix/Prototypical-Networks-for-Few-shot-Learning-PyTorch/blob/master/src/protonet.py

[2] Implement SSL’s MLP

https://github.com/sthalles/PyTorch-BYOL/blob/master/models/resnet_base_network.py

[3] byol

https://github.com/lucidrains/byol-pytorch