AI604 Assignment 3

20214196 Sultan Rizky

June 6, 2021

Self-supervised Learning is a training paradigm that aims to train the model without manual labels provided by humans (i.e., a picture of cat accompanied by the label "cat"). In this assignment, we are going to learn the general representations of images without labels by implementing four different state-of-the-art self-supervised learning methods: SimCLR [1], MoCo [4], BYOL [3], and SimSiam [2].

1 Implementation: Self-Supervised Models

In this part, we will describe the details of implementation that we incorporate within our source code. To provide the simplified view of the aforementioned self-supervised models, please refer to Figure 1 below.

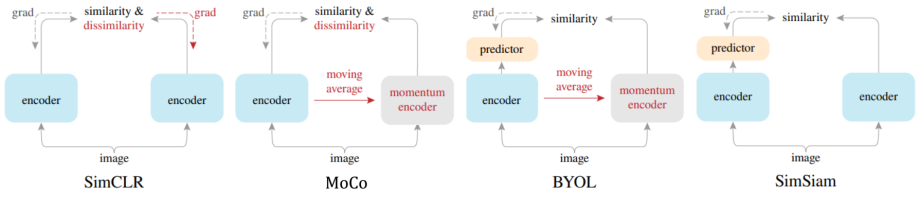

Figure 1: Simplified view of various self-supervised learning models.

1.1 Forward-Propagation

In the following section, we are going to complete the implementation of the overall model's forward propagation. Following the guideline that has been fixed in the homework document, Figure 2 below denotes the implementation of forwarding mechanism within the class `Model` in PyTorch within the file `model_factory.py`.

1.2 Contrastive Loss

In this section, we are going to complete the function `contrastive_loss` defined in `criterion.py`. By strictly following the provided guidelines and comments, Figure 3 below denotes the implementation of the loss, and with our implementation passing the test upon executing the `criterion.py` file as denoted in Figure 4.

1.3 Margin Loss

Now, we are going to complete the function `align_loss` in `criterion.py`. Upon strictly following the guidelines and comments provided, Figure 5 below denotes the

```python
def forward(self, im_aug1, im_aug2):
    z1, z2, p1, p2 = None, None, None, None
    ################################################################################
    # Question 1.1 Implementing the Model (10 points)                              #
    # Implement the model to forward the two agumented images im_aug1 and im_aug2  #
    # 1) if using the momentum_encoder, z2 is the output of the momentum_encoder   #
    # 2) If not using the predictor layer, just set p1 and p2 to None              #
    ################################################################################
    ### YOUR CODE HERE (~ 8 lines)
    # # f: backbone + projection mlp
    # f = self.encoder
    # # m: prediction mlp
    # h = self.predictor
    # Projections
    z1, z2 = self.encoder(im_aug1), self.encoder(im_aug2)
    # Momentum encoder
    if self.use_momentum_encoder:
        z2 = self.momentum_encoder(im_aug2)
    # Predictons
    if self.use_predictor:
        p1, p2 = self.predictor(z1), self.predictor(z2)
    ### END OF YOUR CODE

    return {'z1': z1, 'z2': z2, 'p1': p1, 'p2': p2}
```

Figure 2: Implementation of the model's forward-propagation.

```python
def contrastive_loss(self, z1, z2):
    B, D = z1.shape
    loss = None
    ################################################################################
    # Question 1.2 Implementing the Contrastive Loss (20 points)                   #
    # Implement the contrastive loss of representation z1 and z2                   #
    # B: batch_size                                                                #
    # D: representation_dimension                                                  #
    # 1) representation of each instance should be normalized                      #
    # 2) logits should be divided by temperature to yield a smoother distribution  #
    ################################################################################
    ### YOUR CODE HERE (~ 9 lines)
    z_hat = torch.cat([torch.nn.functional.normalize(z1, dim=1), torch.nn.functional.normalize(z2, dim=1)], dim=0)
    sim_matrix = torch.nn.functional.cosine_similarity(z_hat.unsqueeze(1), z_hat.unsqueeze(0), dim=-1)
    # Create 'positive' labels
    pos = torch.cat([torch.diag(sim_matrix, B), torch.diag(sim_matrix, B)]).view(2*B, 1)
    # Create 'negative' labels
    diag = torch.eye(2*B, dtype=torch.bool, device=z1.device)
    diag[B:, :B] = diag[:B, B:] = diag[:B, :B]
    neg = sim_matrix[~diag].view(2*B, -1)
    # calculate logits and divide logits by temperature
    logits = torch.cat([pos, neg], dim=1)/self.temperature
    # calculate loss
    loss = torch.nn.functional.cross_entropy(logits, torch.zeros(2*B, dtype=torch.int64, device=z1.device), reduction='sum')/(2*B)
    ### END OF YOUR CODE
    return loss
```

Figure 3: Implementation of contrastive loss in criterion.py.

```
====Test Self-Supervised Loss Function====
[Test 1.1: Contrastive Loss w/o predictor]
[-----------------passed-----------------]
[Test 1.2: Contrastive Loss w/_ predictor]
[-----------------passed-----------------]
```

Figure 4: Test results upon executing criterion.py for contrastive loss.

implementation of the loss, and with our implementation passing the test upon executing the `criterion.py` file as denoted in Figure 6.

```python
def align_loss(self, z1, z2):
    B, D = z1.shape
    loss = None
    ##########################################################################
    # Question 1.3 Implementing the Align Loss (10 points)                   #
    # Implement the align loss of representation v1 and v2                   #
    # B: batch_size                                                          #
    # D: representation_dimension                                            #
    # 1) representation of each instance should be normalized                #
    # 2) what is the relationship between l2-distance and cosine similarity  #
    ##########################################################################
    ### YOUR CODE HERE (~ 1 line)
    loss = -torch.nn.functional.cosine_similarity(torch.nn.functional.normalize(z1, dim=1), torch.nn.functional.normalize(z2, dim=1), dim=-1).mean()
    ### END OF YOUR CODE
    return loss
```

Figure 5: Implementation of align loss in criterion.py.

```
[Test 2.1: Align Loss w/o predictor]
[--------------passed--------------]
[Test 2.2: Align Loss w/_ predictor]
[--------------passed--------------]
```

Figure 6: Test results upon executing criterion.py for align loss.

1.4 Updating the Momentum Encoder

In this section, we are going to implement parts of `main.py` that is responsible for updating the momentum encoder. Figure 7 denotes the code snippet for doing the momentum update.

```python
##############################################################################################################
# Question 1.4 Updating the Momentum-Encoder (10 points)                                                     #
# Update the momentum encoder by moving average the encoder's parameters to the momentum-encoder's           #
##############################################################################################################
if args.use_momentum_encoder:
    ### YOUR CODE HERE (~ 2 lines)
    for online, target in zip(model.encoder.parameters(), model.momentum_encoder.parameters()):
        target.data = args.momentum*target.data + (1-args.momentum)*online.data
    ### END OF YOUR CODE
```

Figure 7: Updating the momentum encoder in main.py.

1.5 K-Nearest Neighbor Classification

In this section, we are going to complete the function `KNN` in `validation.py`, which is responsible for employing K-nearest neighbor classifier in evaluating the learning process. Figure 8 denotes the implementation of KNN, with our implementation passing the test upon executing the `validation.py` file as denoted in Figure 9.

2 Experiment Results and Discussion

Upon finalizing the required implementations that has been specified in Section 1, we proceed to run various experiments to examine how different model components impact the performance of the representations. Initially, we run the self-supervised learning methods for SimCLR [1], MoCo [4], BYOL [3], and SimSiam [2] with varying batch size, learning rate, network architecture, and momentum coefficient by following the guidelines specified in the . Once we finish training the model in a self-supervised

```python
def KNN(features, train_features, labels, train_labels, K=1):
    B, _ = features.shape
    num_correct = 0
    ################################################################################################
    # Question 2 Implementing the K-NN Evaluation (10 points)                                      #
    # K: number of neighbors to vote                                                               #
    # Implement the K-NN evaluation by                                                             #
    # 1) find the K nearest train features for each feature                                        #
    # 2) predict the labels as the most frequent train-labels from the nearest neighbors           #
    #    (If the number of neighbors are identical, return the smallest index)                     #
    #    (e.g. [0(label_idx):32(number_of_neighbors), 1:6, 2:32] --> class:0)                      #
    # 3) return the number of correct labels (labels == predicted_labels)                          #
    ################################################################################################
    ### YOUR CODE HERE (~ 6 lines)
    sim_matrix = torch.mm(features, train_features)
    # find k-nearest train features for each feature
    _, sim_indices = sim_matrix.topk(k=K, dim=-1)
    sim_labels = torch.gather(train_labels.expand(B, -1), dim=-1, index=sim_indices)
    # counts for each class
    one_hot_label = torch.zeros(B*K, labels.unique().size(0), device=sim_labels.device)
    one_hot_label = one_hot_label.scatter(dim=-1, index=sim_labels.view(-1, 1), value=1.0)
    pred_scores = torch.sum(one_hot_label.view(B, -1, labels.unique().size(0)), dim=1)
    pred_labels = pred_scores.argsort(dim=-1, descending=True)
    num_correct += (pred_labels[:, 0] == labels).sum().item()
    ### END OF YOUR CODE
    return num_correct
```

Figure 8: Implementation of K-Nearest Neighbor classifier in validation.py.

```
=====Test K-Nearest Neighbor======
[------passed------]
```

Figure 9: Test results upon executing validation.py for K-Nearest Neighbor.

manner, we proceed to train the linear model to further evaluate the classification performance.

2.1 Effects of Batch Size & Learning Rate in Classification Performance

Table 1 indicates the classification performance of KNN-validation after we finished training the model in self-supervised manner and linear-evaluation with varying batch size, default hyperparameters, and default training setups specified in `main.py` and `main_linclis.py`. We incorporate the best accuracy that each model achieves throughout the training as an indicator of the model's classification performance. To indicate the best and second-best performance within the table, we'll put a **bold** and underline in the respective models.

Here, we may notice that in case of SimCLR [1] and SimSiam [2], decreasing the batch size consecutively from 256 to 128 (in case of SimCLR [1]) and from 512 to 256 to 128 (in case of SimSiam [2]) increase both the performance in KNN-validation and top-1 linear-evaluation. However, in case of BYOL [3], increasing the batch size from 128 to 256 decrease the KNN-validation accuracy and increase the top-1 linear-evaluation accuracy by 0.02%. Unfortunately, performing similar analysis in MoCo [4] is impossible due to limitation of computing power when the batch size is increased from 128 to 256. The same reason also caused some of the models being trained only with batch size of 256 during the self-supervised training instead of the default batch

Method	Batch Size	KNN (%)	Linear (Top-1) (%)
SimCLR [1]	256	88.05	91.62
SimCLR [1]	128	**88.07**	**92.15**
MoCo [4]	128	86.55	91.72
BYOL [3]	256	85.89	91.10
BYOL [3]	128	86.72	91.08
SimSiam [2]	512	86.59	91.22
SimSiam [2]	256	86.89	91.48
SimSiam [2]	128	87.39	91.94

Table 1: ResNet-18 Classification Performance for each models under varying batch-size

size of 512 specified in `main.py`. Here, we can reach the conclusion that for most models, training with smaller batch size can result in better KNN-validation and top-1 linear evaluation. This involves less amount of GPU consumption upon training, but at a cost of longer training time.

Moving on to the next set of experiments, we are going to examine the effect of varying the learning rate for each model during the self-supervised training. Using the batch size of 128 for training each models, default training setup for rest of the hyper-parameters, and varying learning rate, Table 2 indicates the classification performance of KNN-validation after we finished training the model in self-supervised manner and top-1 linear-evaluation. We incorporate the best accuracy that each model achieves throughout the training as an indicator of the model's classification performance. We follow the same notation for the best and second-best performance as in Table 1, but this time, we do performance comparison with respect to each state-of-the-art methods instead of performing simultaneous comparisons.

Method	Learning Rate	KNN (%)	Linear (Top-1) (%)
SimCLR [1]	0.06	87.64	**92.38**
SimCLR [1]	0.03	88.07	92.15
SimCLR [1]	0.015	**88.18**	91.95
MoCo [4]	0.06	85.06	**92.33**
MoCo [4]	0.03	86.55	91.72
MoCo [4]	0.015	**86.73**	91.48
BYOL [3]	0.06	83.90	88.76
BYOL [3]	0.03	**86.72**	91.08
BYOL [3]	0.015	86.59	**91.33**
SimSiam [2]	0.06	86.29	91.60
SimSiam [2]	0.03	**87.39**	**91.94**
SimSiam [2]	0.015	86.13	90.89

Table 2: ResNet-18 Classification Performance for each models under varying learning rate

From Table 2, we can observe that in SimCLR [1] and MoCo [4], decreasing the learning rate by a multiple of 2 improves the KNN-validation classification performance, whereas increasing the learning rate by a multiple of 2 improves the top-1 linear-evaluation performance. In case of BYOL [4], the best KNN-validation classification

performance can be achieved when we fix the learning rate to be 0.03 or the default value, whereas the best top-1 linear evaluation performance can be achieved by fixing the learning rate to be 0.015, which is half of the default value. For SimSiam [2], both the best KNN-validation and top-1 linear-evaluation classification performance can be achieved when we fixed the learning rate during the self-supervised training to be 0.03, which is the default value. Thus, we can conclude that using small learning rate will result in better performance for training models that utilize contrastive loss (SimCLR [1] and MoCo [4]), whereas the default learning rate is already the best choice for training models that utilize margin loss (BYOL [3] and SimSiam [2]). On the other hand, linear evaluation result suggests that self-supervised models that use contrastive loss will benefit upon using pre-trained models with higher learning rate.

2.2 Effects of Changing Momentum Coefficient for the Momentum Encoder

In this part, we are going to analyze the effect of changing momentum coefficient m for self-supervised models that utilize the momentum encoder towards the classification performances. Table 3 indicates the classification performance of KNN-validation after we finished training models in self-supervised manner and linear-evaluation with batch size of 128, varying momentum coefficients, and default training setups specified in `main.py` and `main_linclis.py`. Note that we ONLY perform comparison with self-supervised models that employs momentum encoder, which is MoCo [4] and BYOL [3].

Method	m	KNN (%)	Linear (Top-1) (%)
`MoCo` [4]	0.999	86.69	**91.82**
`MoCo` [4]	0.99	86.55	91.72
`MoCo` [4]	0.9	**86.82**	91.46
`BYOL` [3]	0.999	**86.77**	90.97
`BYOL` [3]	0.99	86.72	**91.08**
`BYOL` [3]	0.9	86.66	91.00

Table 3: ResNet-18 Classification Performance for models employing momentum encoder under varying momentum coefficients

Referring to Table 3, it is possible to notice that for MoCo [4], the best KNN-validation result can be achieved when we reduce the momentum coefficient from the default value of 0.99 to 0.9. This result matches with the trend that can be observed in the MoCo's original paper [4]. However, upon performing top-1 linear evaluation on these MoCo [4] pre-trained models, the self-supervised pre-trained model of MoCo [4] which uses momentum coefficient of 0.999 achieves the best performance. Now, in case of BYOL [3], the best KNN-validation result can be obtained when we increase the momentum coefficient from the default value of 0.99 to 0.999. Also for BYOL [3], by choosing the default value of momentum coefficient ($m = 0.99$), we achieved the best linear-evaluation performance compared to other models.

2.3 Effects of Employing Predictor Layer and Stop Gradient

In this section, we are going to report the effect of employing predictor layer and stop gradient towards these self-supervised models. Table 4 indicates the classifica-

tion performance of KNN-validation after we finished training the selected models in `main.py` and `main_linclis.py`.

Method	Batch Size	Pred. Layer	Stop Grad.	KNN (%)	Linear (Top-1) (%)
`SimCLR` [1]	256	✗	✗	**88.05**	**91.62**
`SimCLR` [1]	256	✓	✗	87.84	**91.62**
`SimCLR` [1]	256	✓	✓	86.22	91.57
`MoCo` [4]	128	✗	✓	86.55	91.72
`MoCo` [4]	128	✓	✓	**87.85**	**92.12**
`MoCo` [4]	128	✗	✗	86.75	91.73
`BYOL` [3]	256	✓	✓	85.89	91.10
`BYOL` [3]	256	✗	✓	31.57	30.86
`BYOL` [3]	256	✓	✗	**86.13**	**91.26**
`SimSiam` [2]	128	✓	✓	**87.39**	**91.94**
`SimSiam` [2]	128	✗	✓	28.03	34.01
`SimSiam` [2]	128	✓	✗	30.52	32.43

Table 4: ResNet-18 Classification Performance for models with inclusion/exclusion of predictor layer and stop gradient

Referring to Table 4, one may notice that in case of SimCLR [1], adding predictor layer and stop gradient towards the original model doesn't improve both the KNN-validation and top-1 linear classification performance. In case of MoCo [4], adding predictor layer actually improve both the KNN-validation and top-1 linear classification performance. Similar result can also be achieved when stop gradient is NOT employed towards MoCo [4]. For BYOL [3], it can be noticed that prediction layer is very important during the self-supervised training phase. This is because once we remove the prediction layer, BYOL [3] suffers from collapse. Notice how the KNN-validation and top-1 linear classification accuracy drops to only 30%. However, removing the stop gradient mechanism in BYOL [3] actually improves both the KNN-validation and top-1 linear classification accuracy. Finally, in case of SimSiam [2], it is essential to include both predictor layer and stop gradient towards the model. This is because once they are removed, both will suffer in terms of KNN-validation and top-1 linear classification accuracy. In case we do not employ predictor layer, the loss will diverge at some point, meaning that optimum classification performance will not be reached within this network. On the other hand, not including stop gradient towards SimSiam [2] will immediately degenerate the loss value during self-supervised training. To examine this phenomena, we can refer to the loss plot of different SimSiam [2] architectures specified in Figure 10.

To conclude, the role of predictor layer in these self-supervised models is to prevent collapsing during training. Without predictor layer, we obtain suboptimal performance in KNN-validation and top-1 linear estimation for both BYOL [3] and SimSiam [2], which utilizes margin loss instead of contrastive loss. Now, even though stop gradient is often accompanied with the default implementation of self-supervised models that utilizes momentum encoder (as denoted in BYOL [3] and MoCo [4]), it turns out that they are not really essential for improving the accuracy in most self-supevised models except SimSiam [2]. In SimSiam [2], however, stop-gradient is essential for preventing the collapse during the self-supervised training.

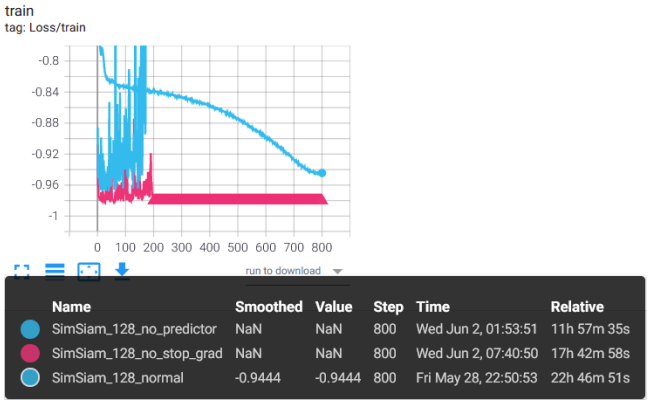

Figure 10: SimSiam architecture with inclusion/exclusion of predictor layer and stop gradient

References

[1] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pages 1597–1607. PMLR, 2020. 1, 3, 4, 5, 6, 7

[2] Xinlei Chen and Kaiming He. Exploring simple siamese representation learning. *arXiv preprint arXiv:2011.10566*, 2020. 1, 3, 4, 5, 6, 7

[3] Jean-Bastien Grill, Florian Strub, Florent Altché, Corentin Tallec, Pierre H Richemond, Elena Buchatskaya, Carl Doersch, Bernardo Avila Pires, Zhaohan Daniel Guo, Mohammad Gheshlaghi Azar, et al. Bootstrap your own latent: A new approach to self-supervised learning. *arXiv preprint arXiv:2006.07733*, 2020. 1, 3, 4, 5, 6, 7

[4] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9729–9738, 2020. 1, 3, 4, 5, 6, 7