Self-Supervised Learning

Papers followed: MoCo by He et al.
Datasets used: CIFAR10 (https://www.cs.toronto.edu/~kriz/cifar.html)
Framework Used: PyTorch Lightning (1.6.4), PyTorch (1.11.0), TorchVision (0.12.0)
GitHub Repo: https://github.com/jainraj/ADRL-Course-Work
Code Inspiration: https://github.com/facebookresearch/moco

1) Code Details

a) Feature Extractor – Resnet based feature extractor. Depending on where it is being used, it is either set to trainable or not

b) OutputLayer – Simple linear output layer which gives logits for different classes. Number of classes is given as an input to make the code reusable

c) Classifier – class for part 1a models. For the base model, the feature extractor is trained. For other models, the base model feature extractor is given as an input and thus kept untrained, only the output layer is trained. For the base model, labels from the dataset are used. For the 2-class and 5-class models, labels are transformed using the two_class_mapper and five_class_mapper respectively.

d) MoCo Transforms – As mentioned in the assignment statement, we are using resizing and cropping, colour jittering, gaussian blurring and rotation augmentations. The test transforms is just normalisation for Resnet. Two augmentations are returned – one is used as the query and one is used as the key.

e) MoCo:

- Momentum and SoftMax temperature are kept same as the value in the paper
- Dictionary size is taken as input and two values are tried: 50X and 200X of the batch size
- 2 queues are used – one for training and validation
- Loss and accuracy were used to check model performance. Here, accuracy would just be based on class 0 prediction as we always keep the positive class as the first entry.
- SGD optimiser and Cosine Annealing LR scheduler were used to keep the same as the paper.

f) LinearClassifier – Similar to Classifier class but always expects a trained feature extractor which is the query encoder of the trained MoCo model. Additionally, the number of classes is fixed to 10 as we need to compare with the base model. train_fraction is taken as input which dictates how much of the training data should be used to train the output layer.

g) train_and_test: General function which allows running different configurations. It takes parameters such as max_epochs, which GPUs to use etc. Even takes the model_class and the model_kwargs to instantiate the model inside the function. It uses the Trainer class of PyTorch Lightning to train, validate and test the model on the given dataset and the model. It logs everything into a tensorboard (created inside the function), which can then be used to view loss curves

h) tsne_plot: Plots the t-SNE on 2D for the features generated per image coloured by their true class

i) All randomness is controlled by using seeds which ensures reproducibility.

j) All code is present in the mentioned GitHub repository organised as modules for ease of navigation and maintenance. It uses additional stuff like logging to WAndB and running updates of the code progress to a webhook. These are omitted in the ipynb file as they are just for maintenance and not central to running the code.

2) Architectural Choices & Details:

a) Class mappers are made such that there is no class imbalance while maintaining some semantic semblance.

b) Macro accuracy (balanced) is used instead of raw accuracy so as to handle any differences in preferential class output of the models.

c) When training classifiers, some basic transformations i.e., random horizontal flip is used to handle overfitting to the data. Preliminary experiments showed improvement in the validation accuracy.

d) Multiple learning rate schedules were tried. ReduceLROnPlateau seemed to perform best.

e) Normalisation was done based on Resnet's suggested values.

f) To implement the training fraction, we use a Subset Sampler. To maintain reproducibility and fairness, seed based random number generator is used.

g) For MoCo, the batch size was chosen as 128.

3) Experiments, Results & Inference:

Table 1 summarises the accuracies obtained in models of Part 1a and Fig. 1 shows the loss curves for the various models. The Resnet model clearly overfits for the base condition i.e., predicting the actual 10 classes of CIFAR10. For the remaining two conditions, the loss curves do not change much after a few iterations which is again not surprising as we are training only a few weights i.e., last layer. The validation accuracy increases as the number of classes go down – this could be because the number of parameters that need to be learned are less and hence we don't overfit.

#Classes	Train Acc	Val Acc
10	99.93%	82.60%
05	97.50%	87.70%
02	98.50%	94.73%

Table 1

Fig. 1 Loss Curves for Models of Part 1a

For 5-class classification, following pairs were made: airplane+ship, automobile+truck, bird+horse, cat+dog, deer+frog. For 2-class classification, airplane, automobile, truck, ship and horse were merged into one class and the rest in the other. This was done so that there is no class imbalance. Next set of figures show the t-SNE plots.

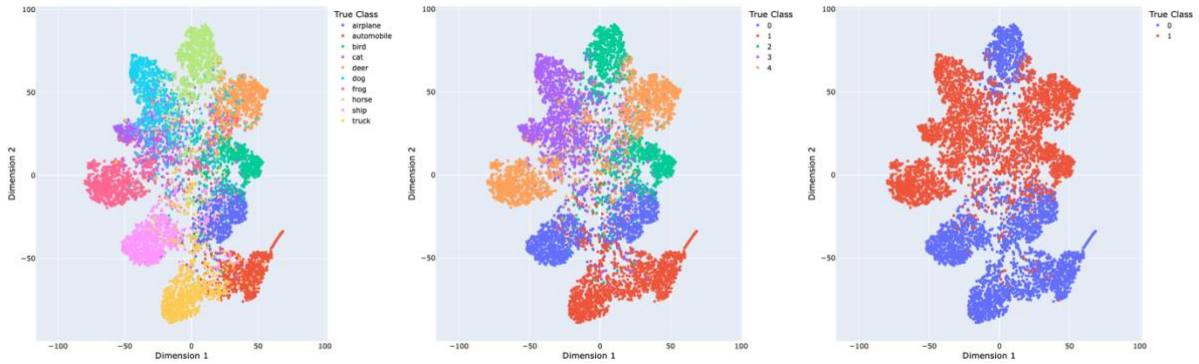

Fig. 2 t-SNE plots for various classification problems

From the first plot, very clearly all the non-living things capture a different space compared to the living things. Even in that, automobile and truck are closer to each other – which says the model seems to learn from the fact that both of these are similar in that they have tires and other similar visual features. Ship and airplane are the not that close, though they share features like being generally in the open sky, and are very separate from land vehicles, indicating that the model is very well able to learn how to distinguish a ship from airplane. In living things, frogs and horses seem to be the most distinct. Bird is nearer to airplane which is very interesting to see. Dogs and cats are also nearby and seem to overlap the most. For the remaining plots, since the feature extractor is unchanged, the plots are visually similar just the class labels change. It may be worthwhile to investigate all the points in the centre of the plot which seem to have the most confused points. Additionally, there were a bunch of points from the automobile class which were significant outliers (not shown here) and could be worthwhile to investigate them too.

As part of MoCo, we tried dictionary sizes of 6400 and 25600 i.e., 50X and 200X of batch size (=128).

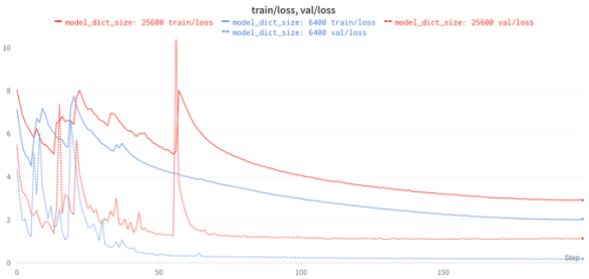

Fig. 3 MoCo loss curves

Loss curves for the MoCo model are shown in fig 3. We see that the validation loss is generally lower than the training loss because the validation dataset is a simpler dataset. 2 linear classifiers were trained for each of these dictionary sizes with training fraction of 10% and 50%. The summarised accuracies are mentioned in Table 2.

MoCo Dictionary Size	10% Data		50% Data	
	Train Acc	Val Acc	Train Acc	Val Acc
6400	77.60%	53.01%	65.54%	56.31%
25600	77.90%	52.73%	65.89%	55.82%

Table 2

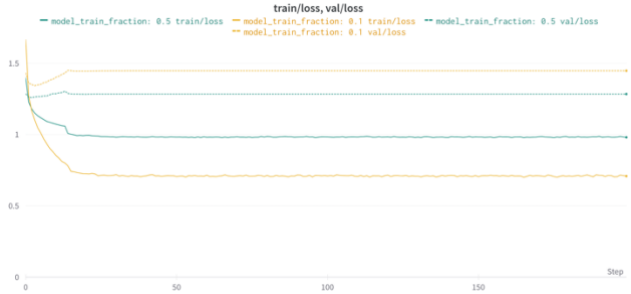

Fig. 4 Loss Curves for linear classifiers trained on 25600 dictionary size MoCo Model

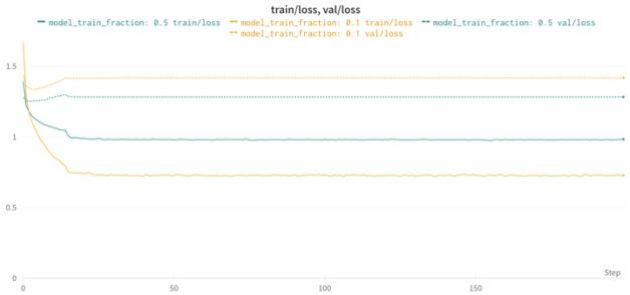

Fig. 5 Loss Curves for linear classifiers trained on 6400 dictionary size MoCo Model

We see an advantage of training with higher amount of data which was expected. What we don't see here is the advantage of using a higher dictionary size (4X higher). Maybe we need to explore much higher dictionary sizes to see the effect we are expecting. And overall the validation accuracies are much lower (~30%) than the base model. A careful exploration of all the hyperparameters involved may give better results. Fig. 4 and 5 show the loss curves of the linear classifiers trained on the MoCo models. As expected the loss curves are almost flat after a while as very few parameters need to be trained.

4) Problems faced, solutions used:

a) We initially tried with custom CNN but had accuracies ~50%. Shifting to Resnet gave much better accuracies.

b) In training MoCo, many different combinations of the augmentations were tried to get the best accuracies, still we could not beat the base model.

5) Conclusion:

a) Personal win was to see the many features of PyTorch and PyTorch Lightning in implementing this algorithm

b) Future work is to try out many different hyperparameters