Deep learning project Report

Prediction on MNIST data with only 100 labels

Supervised Learning and Unsupervised Learning with MoCo on MINST data

Students:
Do Thanh Dat LE
Assmaa AL SAMADI
Piseth KHENG

Teacher:
Prof. Blaise Hanczar
Institution:
University of Paris-Saclay, Évry

January 11, 2024

Contents

1 Introduction **2**

2 Methodology **2**

3 Baseline Neural Networks on supervised leaning with only 100 labeled images **3**

3.1 Data preparation 3
3.2 Data augmentation 3
3.3 Multilayer Perceptron - MLP 4
3.4 Convolutional Neural Network - CNN 5
3.5 Baseline Neural Network Results 5
3.5.1 Training and validation 5
3.5.2 Evaluation on the test set 7

4 Unsupervised Visual Representation Learning with Momentum Contrast - MoCo **8**

4.1 MoCo's architecture 8
4.2 Model setup and training 9
4.3 Training loss of MoCo 10

5 Linear Classification Protocol **10**

5.1 Supervised Learning 10
5.2 Training of Linear Classification Protocol 10
5.3 Results 11

6 Conclusion **12**

1 Introduction

Image classification is a crucial task in the field of artificial intelligence and image analysis. Deep learning with neural networks, in particular networks of convolutional neurons (CNN), have shown great performance in this area. Indeed, Deep Learning is a powerful and fashionable tool that allows you to solve tasks complex as we will see in this very project. In this project, we will study the application of several neural network models, such as multilayer perceptrons (MLP), Convolutional Neural Network (CNNs) and the Momentum Contrast (MoCo) for Unsupervised Visual Representation Learning [KHYR20] to solve the the image classification task on the handwritten digits data MNIST data. We will also present the implementation and evaluation of these models.

The objective of our project is to develop deep learning models for handwritten digits image classification and evaluate their performance using the database MNIST data. However, we assumed that we only have 100 labeled images from the MNIST dataset. Other images are unlabeled. This assumption could occur in reality, because the manual annotation of data could be tedious and expensive.

In this project, we constructed several baseline neural networks including MLP, CNNs for supervised learning on 100 labeled images of MNIST data. Moreover, we also implemented a unsupervised method called Momentum Contrast (MoCo) for learning the representations of the unlabeled images. Thus, this approach makes it possible to deal with the lack of a large number of labeled data in reality.

2 Methodology

For the supervised learning, we took randomly 100 labeled images including 10 classes (ten digits 0-9) from the MNIST train data to train the baseline neural network. These 100 labeled images were transformed into tensors and normalized by its mean and standard deviation. Those 100 images were split into a train set (75 images) and a validation set (25 images) with stratifying the proportions of 10 classes. We transformed image data into tensor and normalized by its mean and standard deviation. Since the train set has only 75 images, we apply data augmentation on the train set thanks to image rotations, scale, crop, translation, and random affine transformations to increase the number of train images. We constructed 3 neural networks including 2 Multilayer Perceptrons (MLP) and a Convolutional Neural Networks (CNNs) for training on the train set and evaluating on the validation set before testing on the MNIST test data.

For the unsupervised learning with MoCo, we use 59900 unlabeled image data to pre-train the MoCo model for transferring to downstream tasks by fine-tuning with 100 labeled image data. We transformed the unlabeled image data into tensor and augmented it by using the augmentation method which contains the Random Resize Crop, Grayscale, Collor Jitter, Random Horizontal Flip, and normalized by its mean and standard deviation. Moreover, we created two random crops of one image which returns a pair of query and key for the training process. In this project, we use Resnet50 as the base encoder for both query and key encoder.

Regarding the results, we compared the supevised learning with baseline neural networks and the unsupervised learning with MoCo to evaluate their performance in handwritten digit recognition of MNIST data.

3 Baseline Neural Networks on supervised leaning with only 100 labeled images

3.1 Data preparation

The MNIST dataset is a collection of 70,000 grayscale images of handwritten digits, divided into 60000 training images and 10000 test images. Each image has size 28x28 pixels. The examples of images in the MNIST data are shown in the **Figure 1**.

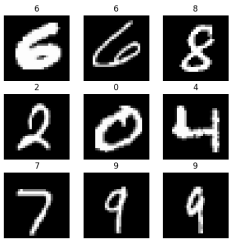


Figure 1: Images examples in MNIST data with labels

Then we took randomly 100 labeled images from the train dataset of MNIST data. These 100 labeled images were split into a train set (75 images) and a validation set (25 images) with stratifying the proportions of 10 classes. We will train our baseline neural networks with the train set and evaluate their performance on the validation set. We also transformed the images to tensors and normalized the data by the mean and standard deviation for the channel of the images. The code for the data preparation is represented in the **Appendix**.

3.2 Data augmentation

The neural networks, particularly the CNNs could work better if there is enough data. Therefore, the data augmentation method could be considered to make the network robust. We applied several transformations to increase the number of labeled data thanks to random rotation, random resize crop, translations, scale and shear. The augmented images for digit 5 are illustrated in **Figure 2**.

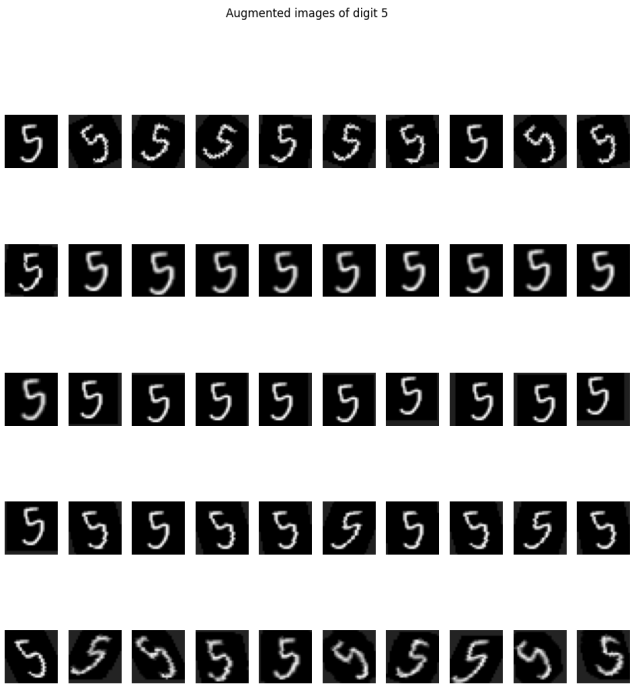
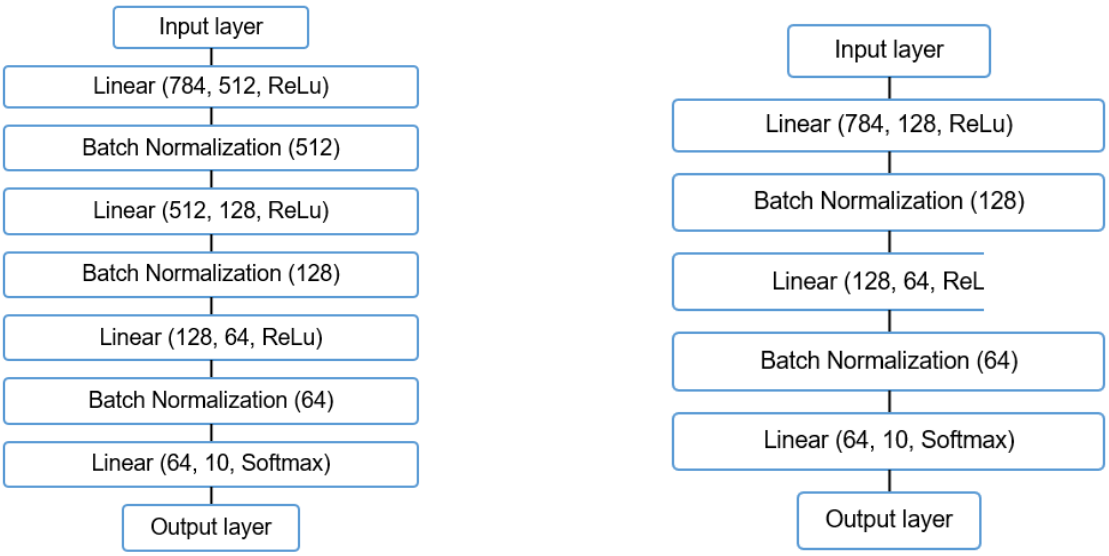


Figure 2: Augmented images for digit 5

The code for the data augmentation is represented in the **Appendix**.

3.3 Multilayer Perceptron - MLP

We constructed 2 MLP whose architectures are illustrated in the **Figure 3a** and **Figure 3b**. We used ReLU as the activation function between layers. We trained the 2 MLPs on the train set with 75 labeled images and validate evaluated their performance on the validation set with 25 labeled images. The images were flatten into the size 784 = 28x28 before going through the MLP.



(a) MLP Net1 architecture

(b) MLP Net2 architecture

Figure 3: MLPs Architectures

The training and evaluation of those MLP models are conducted with 20 epochs, batch size = 20, the optimizer is Adam, and the learning rate is 0.001. We used Cross Entropy Loss as a criterion.

3.4 Convolutional Neural Network - CNN

We constructed a CNN whose architectures is illustrated in the **Figure 4**. We used ReLU as the activation function between layers. We trained this CNN on the train set with 75 labeled images and evaluated their performance on the validation set with 25 labeled images.

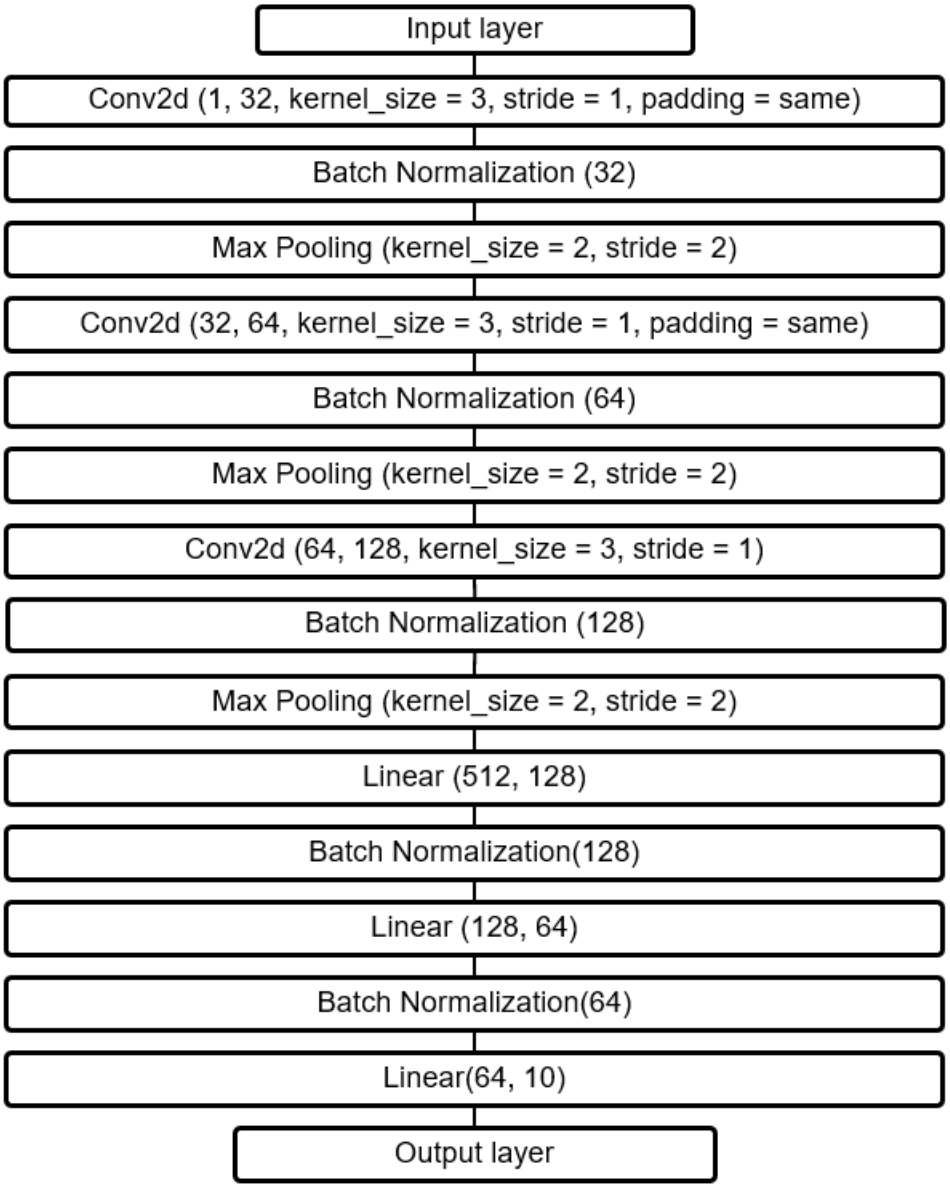


Figure 4: CNN Architecture

The training and evaluation of the CNN are conducted with 20 epochs, batch size = 20, the optimizer is Adam, and the learning rate is 0.001. We used Cross Entropy Loss as a criterion.

3.5 Baseline Neural Network Results

3.5.1 Training and validation

After we trained the MLP and CNN models on the train set and validate their performance on the validation set, the results of their loss and accuracy for each epoch are illustrated in the **Figure 5**, **Figure 6**, **Figure 7**.

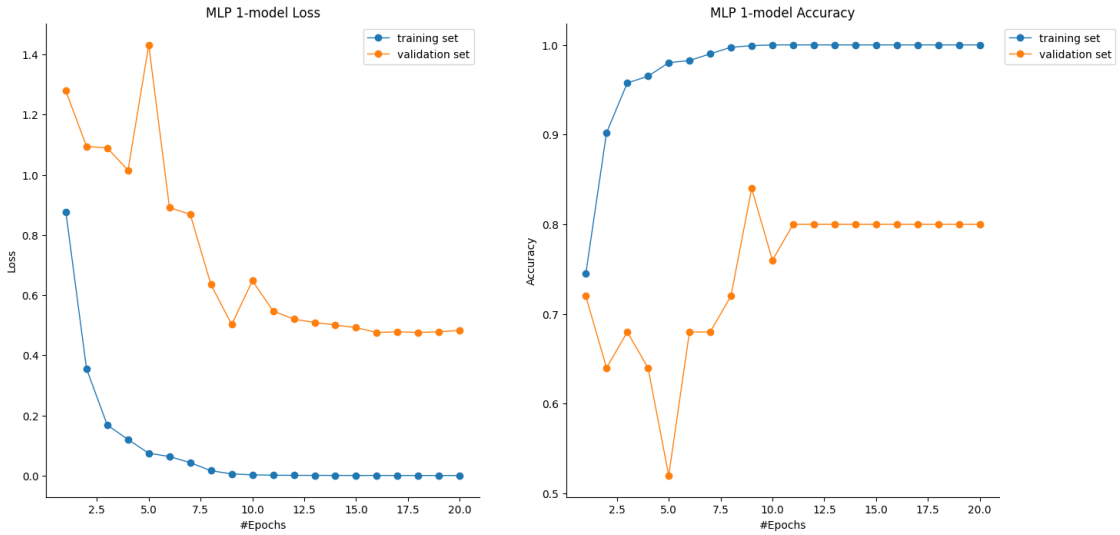


Figure 5: MLP 1 model loss and accuracy

Regarding the learning curves of the MLP 1, we could see that the validation loss is unstable and shows signs of diverging after epoch 18. The best validation loss achieved is about 0.5. The validation accuracy does not seem to have increased after epoch 11. The best validation accuracy is 0.85. Therefore, the MLP 1 is not robust.

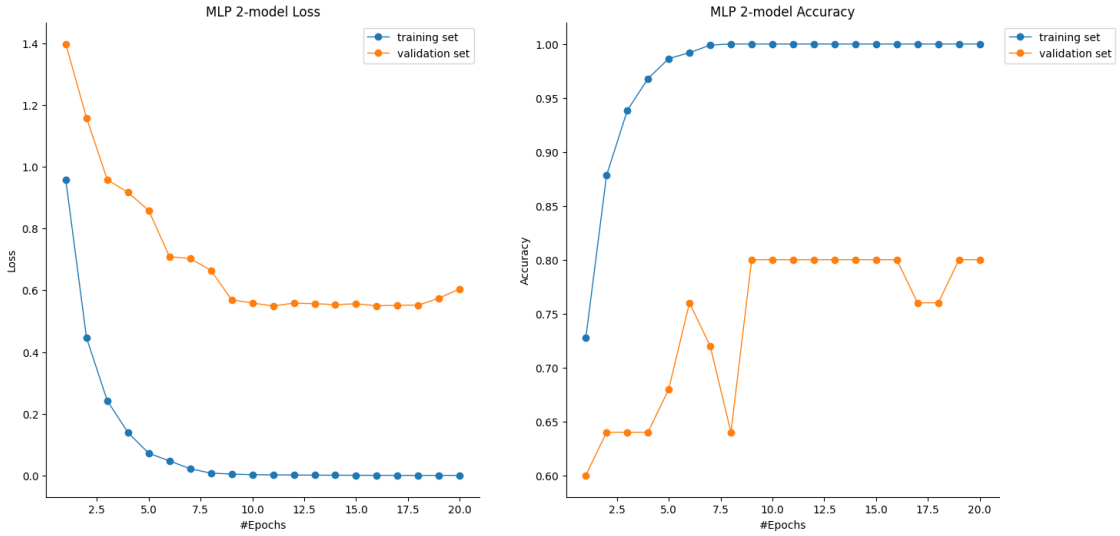


Figure 6: MLP 2 model loss and accuracy

Regarding the learning curves of the MLP 2, we can see that the validation loss gradually decreases throughout the training. However, it shows signs of diverging after epoch 18. The best validation loss achieved is above 0.5. The validation accuracy is unstable and does not improve after epoch 9. The best validation accuracy is about 0.8. Therefore, the MLP 2 is also not robust.

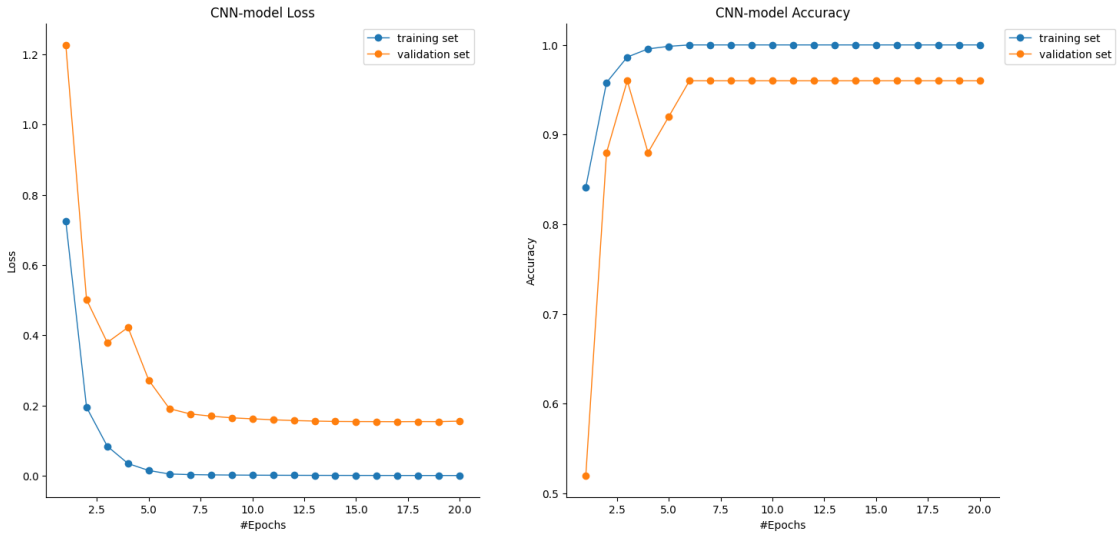


Figure 7: CNN model loss and accuracy

Regarding the learning curves of the CNN, we can see that the validation loss gradually decreases and shows signs of convergence during the training although the speed decreases slowly after epoch 6. The best validation loss achieved is below 0.2. The validation accuracy increased rapidly and reached 0.96 in epoch 7. It did not decrease but did not continue to improve in subsequent epochs. The best validation accuracy acheived is 0.96. Therefore, the CNN is far better than the 2 MLPs. It is the most robust model with the lowest validation loss and highest validation accuracy.

3.5.2 Evaluation on the test set

After training and validation, we continued to evaluate the CNN on the test data of MNIST data and we obtained the accuracy of 0.91. The confusion matrix of the CNN on the test data are shown in the **Figure** 8

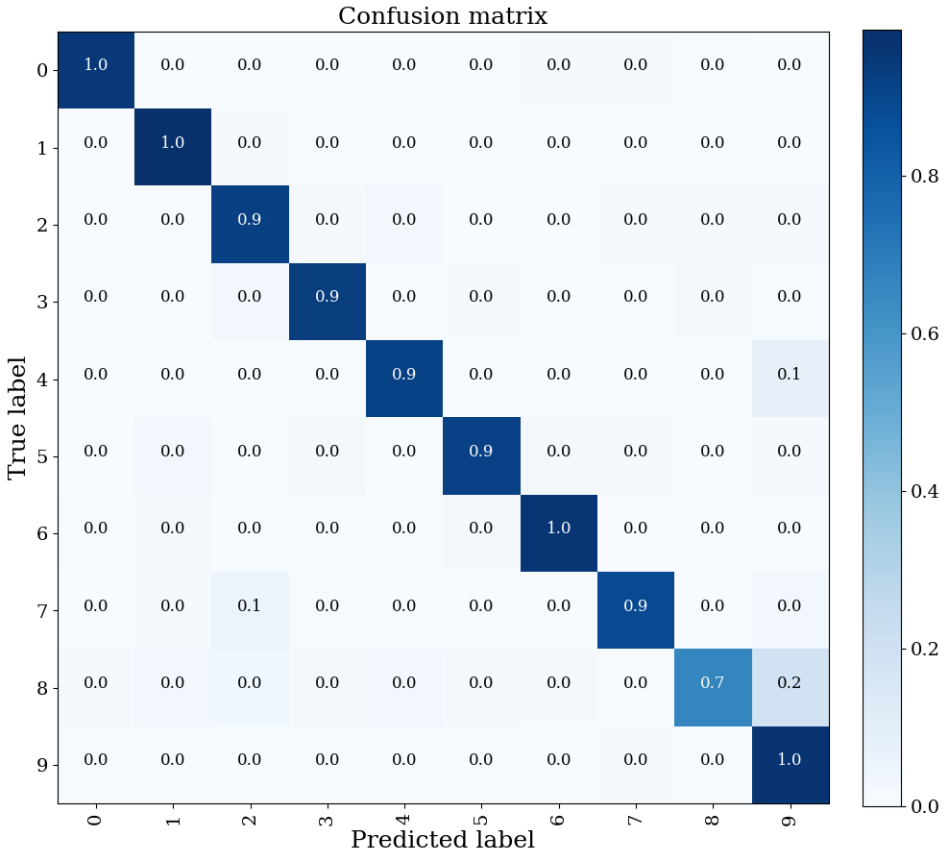


Figure 8: CNN Confusion matrix on the test data

As could be seen in the **Figure 8**, the confusion matrix is generally presented in the form of a table with the actual classes represented on the rows and the predicted classes represented on the columns. The diagonal elements of the table represent the rate of correct predictions for each class, while the non-diagonal elements represent the errors of classification. Therefore, we could see that the CNN model achieve very good performances for most digits except the digit 8, the model seems to sometimes misclassify images of the digit 8 as the digit 9. Moreover, images of the digit 7 are sometimes misclassified as the digit 2, and images of the digit 4 is sometimes misclassified as the digit 9. The evaluation of the CNN model on the test data is good according to the confusion matrix and the accuracy.

4 Unsupervised Visual Representation Learning with Momentum Contrast - MoCo

4.1 MoCo's architecture

The architecture of MoCo consists of several main components, such as inputs (query and keys), encoder query, momentum encoder (encoder keys), dictionary keys (queue), one-hot target, and contrastive loss. This model takes two inputs: a query and a key. These inputs are transformed from the image representation using two random augmentations, one for the query and one for the key. For the base encoder, any CNN architecture can be used. However, we used the Resnet50 architecture for our base encoder.

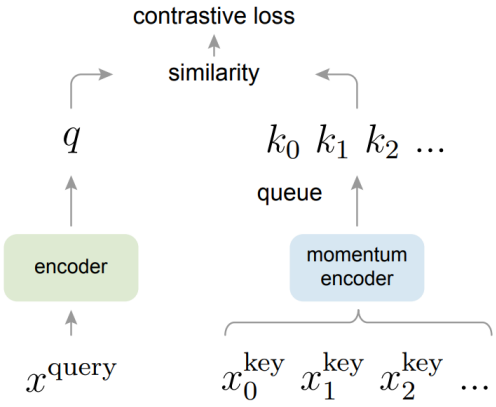


Figure 9: MoCo's architecture from the paper

During the training process, only the encoder query was updated by backpropagation, although the query and the key encoder have the same architecture. When using backpropagation, updating the key encoder can be expensive because the gradients will flow down to all of the dictionary representations. Copying the parameters of the key encoder from the query encoder may seem like a simple solution, but it can lead to bad experimental results. This is because the parameters may be rapidly changing, which can reduce the consistency of the dictionary representations. The solution proposed by the MoCo's paper is to use the momentum update on the encoder keys:

$$\theta_k \leftarrow m\theta_k + (1-m)\theta_q$$

Where $m \in [0, 1)$ is a momentum coefficient, θ_k is the parameters of the encoder key, and θ_q is the parameter of the encoder query.

Nevertheless, MoCo uses a dictionary as a queue to store key feature representations from preceding mini-batches. Known as the dictionary keys, this dictionary adds significant improvements to the results by providing more data samples for the model to learn from. The dictionary's flexible size allows for it to hold more data than a mini-batch, making it act as a queue. The more extensive the dictionary, the better the results.

In Figure 9, we can see the process of encoding the query q and the key k, followed by pushing the positive key k_+ (which corresponds to the pair image of query q) into the dictionary. Next, we use the contrastive loss function, called InfoNCE, to match the query q with dictionary keys that contain its positive key k_+. The loss function value will be low if the query q is similar to its positive key and dissimilar to the negative keys. This loss function is used in the MoCo paper and has the form of:

$$L_q = -\log \frac{\exp(q.k_+/\tau)}{\sum_{i=0}^{K} \exp(q.k_i/\tau)}$$

Where τ is the temperature hyper-parameter, K is the number of negative keys.

4.2 Model setup and training

Due to the limited computing resources for working on this model, we were able to train it on a single GPU on Colab. Firstly, we start by creating a class of MoCo model which takes the base encoder, encoder output dimension, queue size (K), momentum coefficient

(m), loss temperature (T), batch size, and the variable that can enable the model to add the multilayer perceptrons to the encoder query and encoder key.

In the training process, we first start by loading the MNIST data and transforming them by using two crops random transformations (224×224-pixel crop) as we mentioned earlier. Then we initialize the MoCo model by using Resnet50 architecture as the base encoder with the output dimension of 128 (dim=128), the size of the queue is 4096 (K=4096), batch size of data loader is 128, the momentum coefficient is 0.999 (as mentioned in [KHYR20], they suggest to use the large value of m), and the temperature of the contrastive loss function is 0.07. Moreover, we use Stochastic Gradient Descent as the optimizer of our model parameters with a learning rate of 0.03. Last but not least, we start to train the model following the process that we have mentioned in the MoCo architecture under 15 epochs.

4.3 Training loss of MoCo

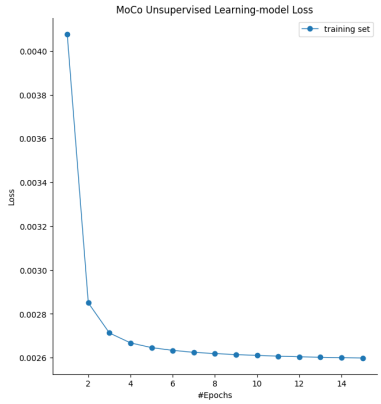


Figure 10: MoCo training loss score

As a result, we can see that the loss score during the training process dropped very quickly from 0.0043 to 0.0026, and this result is quite good where we can get the low score of loss function. Moreover, it took 2 hours to train this model with only 15 epochs on a single GPU.

5 Linear Classification Protocol

5.1 Supervised Learning

We follow the unsupervised learning by a linear classification protocol as in [KHYR20]. We freeze all layers of MocoModel except the last fully connected layer. Referring [KHYR20] to All keys from the dictionnaries are deleted except the one of the fully connected layers of the query encoder. Code is shown in the annexe . We did a supervised approach to classify the 100 samples used in the basline. After freezing layers, we train a fully connected layers with input dimension 2048 and output dimension equal to 10 (number of classes in the Minst Data). The 100 samples were prepared with using the data transformation method used in Moco to fit the training.

5.2 Training of Linear Classification Protocol

At first we train the model with 100 epochs, and batch size = 20, Adam optimizer with learning rate 1e-3 and CrossEntropyLoss criterion. The accuracy on test set was 20% Then we added an early stopping function to tune the number of epochs with epochs 200.The patience parameter of early stoping is set to 4 to avoid too early stop and delta

to 0.05 The training stop on the 73rd epochs with 34% as the top 1 accuracy on the test set. In addition, Top 2 accuracy were calculated as well to see the model performance.

To hypertune parameters, optuna framework was used. Parameters used in optuna are show in table 1. Optuna shows that best optimizer is Adadelta, with 0.00776 0.007767271146682699 as learning rate and batch size of 10.

Optimizers	Adam	Adadelta	Adagrad
Learning rate range	1e-4	1e-3	
batch size	10	20	

Table 1: Tunned Parameters

5.3 Results

In this section, we present loss and accuracies of the linear classification Protocol. Fig 11 shows the fluctuation of loss over epochs. The loss varies between 1.5 and 2.4.

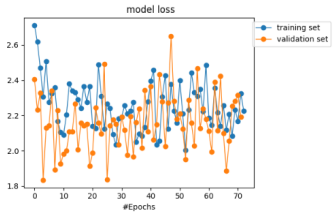
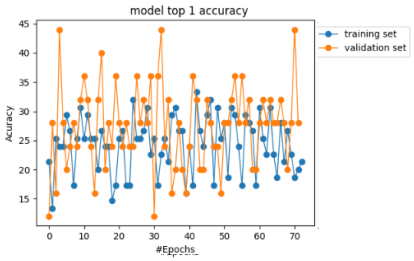
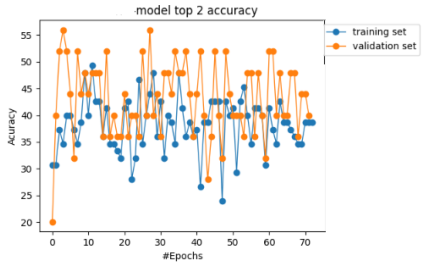


Figure 11: Loss of Linear Classifier

Moreover, Fig 12 shows that the accuracy 1 and 2 fluctate as well over epochs. Results reflect the need of adjustment of learning rate and other parameters of the training .



(a) Top 1 Accuracy over epochs of Linear Classifier



(b) Top 2 Accuracy over epochs of Linear Classifier

Figure 12: Accuracies of The Linear Classifier

Furthermore the top 2 accuracy reflects that the true class has the second high probability of prediction from the Training phase. Top 5 Accuracies of the testing phase are shown in table 2. This table demonstrates that the model deceive between 2 classes for each image since Top 2 accuracy and top 3 accuracy are equals. True class hasn't the Third high probability.

Top 1 Accuracy	**Top 2 Accuracy**	**Top 3 Accuracy**	**Top 4 Accuracy**	**Top 5 Accuracy**
34%	61%	61%	70%	77%

Table 2: Top 5 Accuracies

The Linear protocol was performed on classical google collab. Due to the Limitation of time and the low speed of traning in classical collab, we didn't get the results with Best Parameters. The results above show that the linear model should be tuned since the loss was fluctuated. and we get low accuracy. Comparing with [KHYR20] our Model have Top 1 accuracy 34% on Test set of size 100 images, and the unsupervised was trained on 59900 images of MINST Data, While Original Moco was trained on 1M ImageNet which has more features that the MINST (grayscale of 28*28) Original Moco is 77% which is near to our Top 2 Accuracy The Linear Classification should be tuned to figure out the performance of Linear Moco on MINST Data.

6 Conclusion

The baseline training consist of modifying the architecture of the classifier where MoCo focus on extracting feature from images before the classification. Our work shows the decreases in the accuracy comparing to the baseline. This could be caused by nature of data affects Linear Protocol after Moco since MINST data differs from ImageNet. Moreover, MoCo unsupervised learning and Linear Protocol for supervised learning need tunning

References

[KHYR20] He. Kaiming, Fan. Ha, Xie Yu, Wu. ans Sa, and Gi. Ross. *Momentum Contrast for Unsupervised Visual Representation Learning. Arxiv*, 2020.

Annexes

Baseline Neural Network

Data preparation

Compute mean and standard deviation, load train, test data, transform to tensors and normalize data

```
# Stick all the images together
x = np.concatenate([np.asarray(train_data[i][0]) for i in range(len(
    train_data))])
# calculate the mean and std along the (0, 1) axes
mean = np.mean(x, axis=(0, 1))/255
std = np.std(x, axis=(0, 1))/255

mean=mean.tolist()
std=std.tolist()

# Transform images to tensor and normalize by mean and standard deviation
transformer = transforms.Compose([transforms.ToTensor(), transforms.
    Normalize(mean,std)])

# Load train data and test data from MNIST
train_data = datasets.MNIST(root = './data', train = True, download = True,
     transform = transformer) # Train set
test_data = datasets.MNIST(root = './data', train = False, download = True,
     transform = transformer) # Test set
```

Sample 100 labeled images randomly from MNIST train data

```
# Function for sample 100 labeled images randomly from the train data of
    MNIST
def sample_100(data, random_bool = True):
  if random_bool:
    torch.manual_seed(42)
```

```
    subset_indices = torch.randperm(len(data))[:100]
    subset_data = torch.utils.data.Subset(data, subset_indices)
  else:
    subset_data = []
    for i in range(10):
      label_indices = torch.where(data.targets == i)[0]
      random.seed(42)
      subset_indices = random.sample(label_indices.tolist(), 10)
      subset_data.append(Subset(data, subset_indices))
    subset_data = ConcatDataset(subset_data)
  return subset_data

train_100 = sample_100(train_data) # data with only 100 images take
    randomly
len(train_100)
```

Train, validation set split stratify with labels

```
from sklearn.model_selection import train_test_split
random_state = 42 # for reproductible results
# Get labels to stratify the split
labels = []
for i in range(len(train_100)) :
  _, label = train_100[i]
  labels.append(label)

# Split indices if the dataset is composed of images
train_indices, val_indices = train_test_split(list(range(len(labels))),
                                              test_size = 0.25,
                                              stratify = labels,
                                              random_state = random_state,
                                              shuffle=True)

trainset = torch.utils.data.Subset(train_100, train_indices)
valset = torch.utils.data.Subset(train_100, val_indices)
```

Data Augmentation

Function to apply transformation on images

```
def data_transform(data, transformer_list = [], num_per_img = 10):
  """
  data augmentation applying each transforms for each original image,
  num_per_img is the number of images augmented from each original image
    for each transform
  """

  torch.manual_seed(42)
  augmented_data = []
  augmented_labels = []
  for i in range(len(data)):
    # Data augmentation
    img, label = data[i]
    augmented_data.append(img)
    augmented_labels.append(label)
    for t in transformer_list:

      # Generate new images from the original image
      for _ in range(num_per_img) :
        random.seed(42)
        new_img = t(img)
```

```
        augmented_data.append(new_img)

      # Update new labels
      augmented_labels.extend([label] * (num_per_img))

  augmented_data = torch.stack(augmented_data) # Concatenate a sequence of tensors
  augmented_labels = torch.tensor(augmented_labels)

  return TensorDataset(augmented_data, augmented_labels)
```

Apply data augmentation

```
rotation45 = transforms.Compose([transforms.RandomRotation(45),
                                 transforms.Normalize(mean, std)])

crop_resize = transforms.Compose([transforms.RandomResizedCrop(28, scale=(0.8, 1), ratio = (1,1)),
                                  transforms.Normalize(mean, std)])

translation = transforms.Compose([transforms.RandomAffine(degrees = 0, translate=(0.1, 0.1)),
                                  transforms.Normalize(mean, std)])

shear30 = transforms.Compose([transforms.RandomAffine(0,shear=30),
                              transforms.Normalize(mean, std)])

transform_pipeline1 = transforms.Compose([transforms.RandomRotation(45),
                                          transforms.RandomResizedCrop(28, scale=(0.8, 1), ratio = (1,1)),
                                          transforms.RandomAffine(degrees = 0, translate=(0.1, 0.1)),
                                          transforms.RandomAffine(degrees = 0, shear = 30),
                                          transforms.Normalize(mean, std)])
# create a transformer list
transformer_list1 = [rotation45, crop_resize, translation, shear30, transform_pipeline1]

# apply the transformer list to augment the data
data_aug = data_transform(trainset, transformer_list1, num_per_img = 10)
```

MLP 1 architecture

```
class Net1(nn.Module):
  def __init__(self):
    super().__init__()

    self.lin1 = nn.Linear(in_features = 784, out_features = 512)
    #self.dp1 = nn.Dropout(p = 0.2)
    self.batchnorm1 = nn.BatchNorm1d(512)
    self.lin2 = nn.Linear(in_features = 512, out_features = 128)
    #self.dp2 = nn.Dropout(p = 0.3)
    self.batchnorm2 = nn.BatchNorm1d(128)
    self.lin3 = nn.Linear(in_features = 128, out_features = 64)
    #self.dp3 = nn.Dropout(p = 0.2)
    self.batchnorm3 = nn.BatchNorm1d(64)
    self.lin4 = nn.Linear(in_features = 64, out_features = 10)

  def forward(self, x):
```

```
    x = torch.flatten(x, 1)
    x = F.relu(self.lin1(x))
    x = self.batchnorm1(x)
    #x = self.dp1(x)
    x = F.relu(self.lin2(x))
    x = self.batchnorm2(x)
    #x = self.dp2(x)
    x = F.relu(self.lin3(x))
    x = self.batchnorm3(x)
    #x = self.dp3(x)
    x = self.lin4(x)
    return x
```

MLP 2 architecture

```
class Net2(nn.Module):
  def __init__(self):
    super().__init__()

    self.lin1 = nn.Linear(in_features = 784, out_features = 128)
    self.batchnorm1 = nn.BatchNorm1d(128)
    self.lin2 = nn.Linear(in_features = 128, out_features = 64)
    self.batchnorm2 = nn.BatchNorm1d(64)
    #self.dp = nn.Dropout(p = 0.2)
    self.lin3 = nn.Linear(in_features = 64, out_features = 10)

  def forward(self, x):
    x = torch.flatten(x, 1)
    x = F.relu(self.lin1(x))
    x = self.batchnorm1(x)
    x = F.relu(self.lin2(x))
    x = self.batchnorm2(x)
    #x = self.dp(x)
    x = self.lin3(x)
    return x
```

CNN architecture

```
class CNN(nn.Module):
  def __init__(self):
    super().__init__() # always subclass
    self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding = "same") # conv
    layer 28x28x32
    self.batchnorm1 = nn.BatchNorm2d(32)
    self.pool1 = nn.MaxPool2d(2) # maxpooling 14x14x32
    #self.dp1 = nn.Dropout(p = 0.2)

    self.conv2 = nn.Conv2d(32, 64, kernel_size=3) # first conv layer 12
    x12x64
    self.batchnorm2 = nn.BatchNorm2d(64)
    self.pool2 = nn.MaxPool2d(2) # maxpooling 6x6x64
    #self.dp2 = nn.Dropout(p = 0.3)

    self.conv3 = nn.Conv2d(64, 128, kernel_size=3) # conv layer 4x4x128
    self.batchnorm3 = nn.BatchNorm2d(128)
    self.pool3 = nn.MaxPool2d(2) # maxpooling 2x2x128
    #self.dp3 = nn.Dropout(p = 0.2)

```

```
    self.fc1 = nn.Linear(128*2*2, 128) # we have 10 probability classes to predict so 10 output features
    self.batchnorm4 = nn.BatchNorm1d(128)
    self.fc2 = nn.Linear(128, 64)
    self.batchnorm5 = nn.BatchNorm1d(64)
    #self.dp4 = nn.Dropout(p = 0.3)
    self.fc3 = nn.Linear(64, 10)

  def forward(self, x):
    x = self.conv1(x)
    x = self.batchnorm1(x)
    x = F.relu(x)
    x = self.pool1(x)
    #x = self.dp1(x)

    x = self.conv2(x)
    x = self.batchnorm2(x)
    x = F.relu(x)
    x = self.pool2(x)
    #x = self.dp2(x)

    x = self.conv3(x)
    x = self.batchnorm3(x)
    x = F.relu(x)
    x = self.pool3(x)
    #x = self.dp3(x)

    x = torch.flatten(x, 1)
    x = self.fc1(x)
    x = self.batchnorm4(x)
    x = F.relu(x)
    x = self.fc2(x)
    x = self.batchnorm5(x)
    x = F.relu(x)
    #x = self.dp4(x)
    x = self.fc3(x)

    return x
```

Train function

Early stopping class

```
# class to implement Early Stopping
class EarlyStopping:
  """Early stops the training if validation loss doesn't improve after a given patience."""
  def __init__(self, patience=1, min_delta=0):
    self.patience = patience
    self.min_delta = min_delta
    self.counter = 0
    self.min_validation_loss = float('inf')

  def early_stop(self, validation_loss):
    if validation_loss < self.min_validation_loss:
      self.min_validation_loss = validation_loss
      self.counter = 0
    elif validation_loss > (self.min_validation_loss + self.min_delta):
      self.counter += 1
```

```
      print(f'EarlyStopping counter: {self.counter} out of {self.patience}')
    if self.counter >= self.patience:
      return True
  return False
```

Function to get accuracy

```
def get_accuracy(y_true, y_pred):
    return int(np.sum(np.equal(y_true,y_pred))) / y_true.shape[0]
```

Function to load data with batch size = 20

```
def load_data(train, val, test, train_batch, val_batch, test_batch):
  torch.manual_seed(42)
  trainloader = torch.utils.data.DataLoader(train, batch_size=train_batch,
                                            shuffle=True, num_workers=2)
  valloader = torch.utils.data.DataLoader(val, batch_size=val_batch,
                                          shuffle=False, num_workers=2)
  testloader = torch.utils.data.DataLoader(test, batch_size=test_batch,
                                          shuffle=False, num_workers=2)
  return trainloader,valloader,testloader

trainloader,valloader,testloader = load_data(trainset, valset, test_data,
    20, 20, 20)
```

Train function

```
# Train function
def train_model(model, epochs, train_loader, val_loader, optimizer_name = '
    Adam', patience = 1, early_stopping = True, learning_rate = 0.001,
    device=None):
  torch.manual_seed(42)
  # Init
  output_fn = torch.nn.Softmax(dim=1) # we instantiate the softmax
    activation function for the output probabilities
  criterion = nn.CrossEntropyLoss() # we instantiate the loss function
  optimizer = getattr(optim, optimizer_name)(model.parameters(), lr=
    learning_rate) # we instantiate Adam optimizer that takes as inputs the
     model parameters and learning rate

  loss_valid,acc_valid =[],[]
  loss_train,acc_train =[],[]

  # initialize the early_stopping object
  if early_stopping:
    early_stopper = EarlyStopping(patience=patience)

  for epoch in tqdm(range(epochs)):
    torch.manual_seed(42)
    # Training loop
    model.train() # always specify that the model is in training mode
    running_loss = 0.0 # init loss
    running_acc = 0.

    # Loop over batches returned by the data loader
    for idx, batch in enumerate(train_loader):

      # get the inputs; batch is a tuple of (inputs, labels)
      inputs, labels = batch
      inputs = inputs.to(device) # put the data on the same device as the
    model
      labels = labels.to(device)
```

```
30
31      # put to zero the parameters gradients at each iteration to avoid
    accumulations
32      optimizer.zero_grad()
33
34      # forward pass + backward pass + update the model parameters
35      out = model(x=inputs) # get predictions
36      loss = criterion(out, labels) # compute loss
37      loss.backward() # compute gradients
38      optimizer.step() # update model parameters according to these
    gradients and our optimizer strategy
39
40      # Iteration train metrics
41      running_loss += loss.view(1).item()
42      t_out = output_fn(out.detach()).cpu().numpy() # compute softmax (
    previously instantiated) and detach predictions from the model graph
43      t_out=t_out.argmax(axis=1)  # the class with the highest energy is
    what we choose as prediction
44      ground_truth = labels.cpu().numpy() # detach the labels from GPU
    device
45      running_acc += get_accuracy(ground_truth, t_out)
46
47    ### Epochs train metrics ###
48    acc_train.append(running_acc/len(train_loader))
49    loss_train.append(running_loss/len(train_loader))
50
51    # compute loss and accuracy after an epoch on the train and valid set
52    model.eval() # put the model in evaluation mode (this prevents the use
    of dropout layers for instance)
53
54    ### VALIDATION DATA ###
55    with torch.no_grad(): # since we're not training, we don't need to
    calculate the gradients for our outputs
56      idx = 0
57      for batch in val_loader:
58        inputs,labels=batch
59        inputs=inputs.to(device)
60        labels=labels.to(device)
61        if idx==0:
62          t_out = model(x=inputs)
63          t_loss = criterion(t_out, labels).view(1).item()
64          t_out = output_fn(t_out).detach().cpu().numpy() # compute softmax
     (previously instantiated) and detach predictions from the model graph
65          t_out=t_out.argmax(axis=1)  # the class with the highest energy
    is what we choose as prediction
66          ground_truth = labels.cpu().numpy() # detach the labels from GPU
    device
67        else:
68          out = model(x=inputs)
69          t_loss = np.hstack((t_loss,criterion(out, labels).item()))
70          t_out = np.hstack((t_out,output_fn(out).argmax(axis=1).detach().
    cpu().numpy()))
71          ground_truth = np.hstack((ground_truth,labels.detach().cpu().
    numpy()))
72        idx+=1
73
74      acc_valid.append(get_accuracy(ground_truth,t_out))
75      loss_valid.append(np.mean(t_loss))
76
77    print('| Epoch: {}/{} | Train: Loss {:.4f} Accuracy : {:.4f} '\
```

```
        '| Val: Loss {:.4f} Accuracy : {:.4f}\n'.format(epoch+1,epochs,
    loss_train[epoch],acc_train[epoch],loss_valid[epoch],acc_valid[epoch]))

    # early_stopping check if the validation loss has decresed, if yes, it
    will make a checkpoint of the current model
    if early_stopping:
      stop_bool = early_stopper.early_stop(loss_valid[epoch])
      if stop_bool:
        print("Early stopping")
        break

  # load the last checkpoint with the best model
  #if early_stopping:
    #model.load_state_dict(torch.load('checkpoint.pt'))

  return model, loss_train, acc_train, loss_valid, acc_valid
```

Test model function

```
    def test_model(model, test_loader):
  output_fn = torch.nn.Softmax(dim=1) # we instantiate the softmax
    activation function for the output probabilities
  model.eval()
  torch.manual_seed(42)
  with torch.no_grad():
    idx = 0
    for batch in test_loader:
      inputs,labels=batch
      inputs=inputs.to(device)
      labels=labels.to(device)
      if idx==0:
        t_out = model(x=inputs)
        t_out = output_fn(t_out).detach().cpu().numpy()
        t_out=t_out.argmax(axis=1)
        ground_truth = labels.detach().cpu().numpy()
      else:
        out = model(x=inputs)
        t_out = np.hstack((t_out,output_fn(out).argmax(axis=1).detach().cpu
    ().numpy()))
        ground_truth = np.hstack((ground_truth,labels.detach().cpu().numpy
    ()))
      idx+=1

  print("Test accuracy {}".format(get_accuracy(ground_truth,t_out)))
  return t_out, ground_truth
```

MoCo model

Create MoCo class

```
class MoCoModel(nn.Module):

  def __init__(self, base_encoder, dim=128, K=4096, m=0.999, T=0.07,
    batch_size=128, mlp=False):

    super(MoCoModel, self).__init__()

    self.K = K
    self.m = m
```

```
    self.T = T
    self.batch_size = batch_size

    # create the encoders
    self.encoder_q = base_encoder(num_classes=dim)
    self.encoder_k = base_encoder(num_classes=dim)

    if mlp:
      dim_mlp = self.encoder_q.fc.weight.shape[1]
      self.encoder_q.fc = nn.Sequential(
          nn.Linear(dim_mlp, dim_mlp), nn.ReLU(), self.encoder_q.fc
      )
      self.encoder_k.fc = nn.Sequential(
          nn.Linear(dim_mlp, dim_mlp), nn.ReLU(), self.encoder_k.fc
      )


    for param_q, param_k in zip(self.encoder_q.parameters(), self.encoder_k
    .parameters()):

      param_k.data.copy_(param_q.data)
      param_k.requires_grad = False # no gradient update

    # create the queue
    with torch.no_grad():
      self.queue = torch.randn(batch_size,moco_K).cuda()
      self.queue = nn.functional.normalize(self.queue, dim=0)

  @torch.no_grad()
  def _momentum_update_key_encoder(self):

    for param_q, param_k in zip(self.encoder_q.parameters(), self.encoder_k
    .parameters()):
      param_k.data = param_k.data * self.m + param_q.data * (1.0 - self.m)

  @torch.no_grad()
  def _dequeue(self):
      return self.queue[:,:(self.K - self.batch_size)]

  @torch.no_grad()
  def _enqueue(self,k):
      return torch.cat([k, self.queue],dim=1)

  def forward(self, im_q, im_k):
    """
    Input:
        im_q: a batch of query images
        im_k: a batch of key images
    Output:
        logits, targets
    """

    # compute query features
    q = self.encoder_q(im_q)  # queries: NxC
    q = nn.functional.normalize(q, dim=1)

    # compute key features
    with torch.no_grad():
      self._momentum_update_key_encoder()

```

```
      k = self.encoder_k(im_k)
      k = nn.functional.normalize(k, dim=1)

    # compute logits
    # positive logits: Nx1
    l_pos = torch.einsum("nc,nc->n", [q, k]).unsqueeze(-1)
    # negative logits: NxK
    l_neg = torch.einsum("nc,ck->nk", [q, self.queue.clone().detach()])

    # logits: Nx(1+K)
    logits = torch.cat([l_pos, l_neg], dim=1)

    # apply temperature
    logits /= self.T

    # labels: positive key indicators
    labels = torch.zeros(logits.shape[0], dtype=torch.long).cuda()

    # dequeue and enqueue
    self.queue = self._dequeue()
    self.queue = self._enqueue(k)

    return logits, labels
```

Data augmentation

```
class TwoCropsTransform:
  """Take two random crops of one image as the query and key."""

  def __init__(self, base_transform):
    self.base_transform = base_transform

  def __call__(self, x):
    q = self.base_transform(x)
    k = self.base_transform(x)
    return [q, k]

# Create augmentation parameters for transforming the data
augmentation = [
            transforms.RandomResizedCrop(224, scale=(0.2, 1.0)),
            transforms.Grayscale(num_output_channels=3),
            transforms.ColorJitter(0.4, 0.4, 0.4, 0.4),
            transforms.RandomHorizontalFlip(),
            transforms.ToTensor(),
            transforms.Normalize(0.1306, 0.3081),
        ]

transformer = TwoCropsTransform(transforms.Compose(augmentation))
```

Computing accuracy for MoCo

```
def accuracy(output, target, topk=(1,)):
    """Computes the accuracy over the k top predictions for the specified
    values of k"""
    with torch.no_grad():
        maxk = max(topk)
        batch_size = target.size(0)

        _, pred = output.topk(maxk, 1, True, True)
        pred = pred.t()
```

```
        correct = pred.eq(target.view(1, -1).expand_as(pred))

        res = []
        for k in topk:
            correct_k = correct[:k].view(-1).float().sum(0, keepdim=True)
            res.append(correct_k.mul_(100.0 / batch_size))
        return res
```

Training function for MoCo

```
def training(model, epochs, train_loader, optimizer, criterion, device=None):
  train_losses = []
  train_acc = []

  for e in tqdm(range(epochs)):
      epoch_acc1, epoch_loss = 0.0, 0.0
      running_loss, running_acc1 = 0.0, 0.0

      #switch to training mode
      model.train()

      for i,(images, _) in enumerate(train_loader):

          images[0] = images[0].cuda(device)
          images[1] = images[1].cuda(device)

          # compute output
          output, target = model(im_q=images[0], im_k=images[1])

          loss = criterion(output, target)

          acc1 = accuracy(output, target,  topk=(1,))

          #updating query encoder
          optimizer.zero_grad()
          loss.backward()
          optimizer.step()

          epoch_acc1 += acc1[0].cpu().numpy()
          epoch_loss += loss.item()

          running_loss += loss.item()
          running_acc1 += acc1[0]

          if ((i+1)%100 == 0):
            print('Epoch :',e+1,'Batch :',(i+1),'Loss :',float(running_loss/100),'Top_acc1 :',float(running_acc1/100))
            running_loss, running_acc1 = 0.0, 0.0

      saveModel(e, model, optimizer, epoch_loss, "moco_model.pth")

      train_losses.append(epoch_loss/len(train_loader))
      train_acc.append(epoch_acc1/len(train_loader))
      print('Epoch :',e+1, 'Loss :',train_losses[e], 'Top acc1 :',train_acc[e])

  return model, train_losses, train_acc
```

Setup model parameters and training

```
# Create MoCo model
model = MoCoModel(models.__dict__["resnet50"],
                  dim=128, K=4096, m=0.999, T=0.07,
                  batch_size=128, mlp=False).cuda(device)
# define loss function (criterion) and optimizer
criterion = nn.CrossEntropyLoss().cuda(device)
optimizer = torch.optim.SGD(
    model.parameters(),
    lr=0.03,
    momentum=0.9,
    weight_decay=1e-4,
)
# training moco model
moco_train, moco_losses, moco_acc = training(model, epochs=15, train_loader
    , optimizer, criterion, device=device)
```

Linear Classifier

Train Model Function

```
def train_val(net, data_loader, train_optimizer):
    is_train = train_optimizer is not None
    net.train() if is_train else net.eval()

    total_loss, total_correct_1,total_correct_2,total_correct_3,
    total_correct_4, total_correct_5, total_num,  data_bar =
    0.0,0.0,0.0,0.0, 0.0, 0.0, 0, tqdm(data_loader)
    with (torch.enable_grad() if is_train else torch.no_grad()):
        for data, target in data_bar:
            data, target = data.to(device), target.to(device)
            out = net(data)
            loss = loss_criterion(out, target)

            if is_train:
                train_optimizer.zero_grad()
                loss.backward()
                train_optimizer.step()

            total_num += data.size(0)
            total_loss += loss.item() * data.size(0)
            prediction = torch.argsort(out, dim=-1, descending=True)
            total_correct_1 += torch.sum((prediction[:, 0:1] == target.
    unsqueeze(dim=-1)).any(dim=-1).float()).item()
            total_correct_2 += torch.sum((prediction[:, 0:2] == target.
    unsqueeze(dim=-1)).any(dim=-1).float()).item()
            total_correct_3 += torch.sum((prediction[:, 0:3] == target.
    unsqueeze(dim=-1)).any(dim=-1).float()).item()
            total_correct_4 += torch.sum((prediction[:, 0:4] == target.
    unsqueeze(dim=-1)).any(dim=-1).float()).item()
            total_correct_5 += torch.sum((prediction[:, 0:5] == target.
    unsqueeze(dim=-1)).any(dim=-1).float()).item()
            data_bar.set_description('{} Epoch: [{}/{}] Loss: {:.4f} ACC@1:
     {:.2f}% ACC@2: {:.2f}% ACC@3: {:.2f}% ACC@4: {:.2f}% ACC@5: {:.2f}%'
                                     .format('Train' if is_train else '
    Validation', epoch, epochs, total_loss / total_num,
                                             total_correct_1 / total_num *
    100, total_correct_2 / total_num * 100,  total_correct_1 / total_num *
```

```
    100, total_correct_3 / total_num * 100, total_correct_4 / total_num *
    100, total_correct_5 / total_num * 100))

    return total_loss / total_num, total_correct_1 / total_num * 100,
    total_correct_2 / total_num * 100,total_correct_3 / total_num * 100,
    total_correct_4 / total_num * 100,total_correct_5 / total_num * 100
```

Freezing Feature

```
checkpoint = torch.load('/content/drive/MyDrive/Deep Learning M2DS/Project/
    moco_model.pth', map_location=torch.device('cpu'))
state_dict = checkpoint["state_dict"]
for k in list(state_dict.keys()):
    # retain only encoder_q up to before the embedding layer
    if k.startswith("module.encoder_q") and not k.startswith(
                    "module.encoder_q.fc"):
    # remove prefix
      state_dict[k[len("module.encoder_q.") :]] = state_dict[k]
    # delete renamed or unused k
    del state_dict[k]
```

Training Model

```
optimizer = optim.Adam(model.fc.parameters(), lr=1e-3, weight_decay=1e-6)
results = {'train_loss': [], 'train_acc@1': [], 'train_acc@2': [],'
    train_acc@3': [], 'train_acc@4': [],'train_acc@5': [],
               'val_loss': [],'val_acc@1': [], 'val_acc@2': [],'val_acc@3':
     [], 'val_acc@4': [],'val_acc@5': []}

best_acc = 0.0
for epoch in range(1, epochs + 1):
        train_loss, train_acc_1,train_acc_2,train_acc_3,train_acc_4,
    train_acc_5 = train_val(model, trainloader, optimizer)
        results['train_loss'].append(train_loss)
        results['train_acc@1'].append(train_acc_1)
        results['train_acc@2'].append(train_acc_2)
        results['train_acc@3'].append(train_acc_3)
        results['train_acc@4'].append(train_acc_4)
        results['train_acc@5'].append(train_acc_5)
        with torch.no_grad():
          val_loss, val_acc_1,val_acc_2,val_acc_3,val_acc_4, val_acc_5 =
    train_val(model, valloader, None)
          if early_stopper.early_stop(np.mean(results['val_loss'])):
            print("Early stopping triggered!")
            break
          results['val_loss'].append(val_loss)
          results['val_acc@1'].append(val_acc_1)
          results['val_acc@2'].append(val_acc_2)
          results['val_acc@3'].append(val_acc_3)
          results['val_acc@4'].append(val_acc_4)
          results['val_acc@5'].append(val_acc_5)
        # save statistics
          data_frame = pd.DataFrame(data=results, index=range(1, epoch + 1)
    )
          if val_acc_1 > best_acc:
            best_acc = val_acc_1
```