Self-supervised Learning: Architecture Analysis and Implementation

Project Report #3

Mariateresa Nardoni

Abstract

Self-Supervised Learning (SSL) is a technique that uses unlabeled data, thereby reducing dependence on costly labeled data. Using pretext tasks, models learn meaningful representations from unlabeled data, which are then used in supervised learning tasks. Pre-training in SSL aims to obtain robust and generic representations, often through the use of Siamese networks. This project analyzes well-known approaches such as SimCLR, BYOL, and Barlow Twins, offering both a theoretical and practical overview of self-supervised learning, showing the differences and commonalities among the techniques while also considering the case of k-Nearest Neighbors used as a classifier.

1. Introduction

Self-Supervised Learning (SSL) represents a technique in machine learning that exploits unlabeled data for learning, reducing the need for large amounts of labeled data, which are often expensive and difficult to obtain, especially in fields that require skilled human annotators. In addition, labeled data can introduce spurious correlations and generalization errors, making models vulnerable to adversarial attacks such as adding noise that can lead to misclassification of analyzed data. In contrast, unlabeled data are abundant and free, providing a significant opportunity for learning.

The central idea behind Self-Supervised Learning is to create pretext tasks (or surrogate tasks) that allow models to learn meaningful representations from unlabeled data. Subsequently these representations are used in supervised learning tasks also known as downstream tasks.

The process of self-supervised training can be described through a sequence of basic steps:

- ▷ a pretext task is used to train a backbone on a large unlabeled dataset, using a self-designed objective
- ▷ once the backbone has been trained, its weights are frozen
- ▷ a classifier is then trained over the frozen backbone using a labeled dataset for a downstream task

This approach results in obtaining pre-trained models that can be useful for doing transfer learning later.

Pre-training in Self-Supervised Learning aims at obtaining representative features of unlabelled data that are robust and generic. This is achieved by training the model on pretext tasks that require the model to learn representations invariant to transformations and augmentations of the data. A common architecture used for pre-training in SSL is the Siamese network. A Siamese network uses the same weights to process two different input vectors, producing comparable output vectors. This is crucial to avoid the representation collapse phenomenon, where without augmentations, the encoder could learn a trivial solution.

Overall, Self-Supervised Learning offers an effective methodology to exploit unlabelled data, reducing dependence on expensive labelled data and improving the ability of models to generalise to disturbing factors.

1.1. Motivation

The main objective of this project is to conduct an analysis of various self-supervised learning techniques with emphasis on approaches based on contrastive loss, distillation and redundancy reduction, the precursors of which are SimCLR, BYOL and Barlow Twins. The classifier that will be used is k-Nearest Neighbors.

In addition to the theoretical description of these methods, the practical implementation will be analyzed with Python code examples.

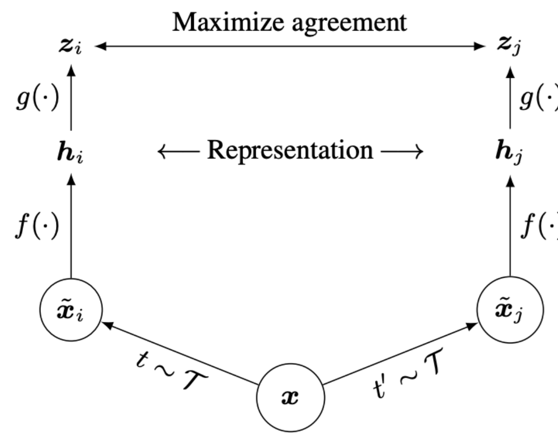
Through the combination of theoretical and practical analysis, the project aims to provide a general overview of the mentioned self-supervised learning methods by showing their various differences and commonalities.

2. Method

The project was written in Python language. All calculations and results obtained were processed on Google Colab. The following resources were used in the realisation of this project, an explanation follows:

2.1. SimCLR

SimCLR, which stands for "Simple Framework for Contrastive Learning of Visual Representations", is a self-supervised learning approach designed to extract meaningful representations of images using unlabelled data.



Taking a starting image x, two augmentations are performed, obtaining two images denoted as $\widetilde{x}_i$ and $\widetilde{x}_j$. SimCLR uses a common encoder, $f(\cdot)$, to extract representations of features h_i and h_j from the two augmented versions of the image. These feature representations are then projected into an embedding space with reduced dimensions using a $g(\cdot)$ projector, which is typically a small Multi-Layer Perceptron (MLP). Once the projected z_i and z_j representations have been obtained, SimCLR calculates the Contrastive Loss using the InfoNCE Loss.

$$\mathcal{L}_{\text{infoNCE}} = -\sum_{(i,j)\in\mathbb{P}} \log\left(\frac{e^{\text{CoSim}(\boldsymbol{z}_i,\boldsymbol{z}_j)/\tau}}{\sum_{k=1}^{N} e^{\text{CoSim}(\boldsymbol{z}_i,\boldsymbol{z}_k)/\tau}}\right)$$

The InfoNCE Loss (Normalized Cross Entropy) exploits cosine similarity between feature representations instead of the scalar product, uses positive and negative samples and sets the batch construction following the N-pair loss paradigm. Furthermore it applies an explicit normalisation factor, known as temperature τ, for the cosine similarity and does not apply any regularisation on the feature vector. The InfoNCE Loss is computed as the cross-entropy between the similarity predictions, which are computed using the cosine distance between the pairs of representations z_i and z_j and the similarity labels, represented by a one-hot vector indicating whether the pairs of examples are similar or not.

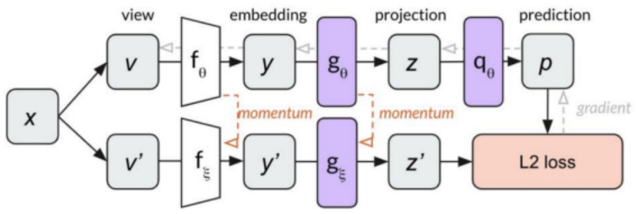
The main objective of the InfoNCE Loss is to maximise the similarity between pairs of positive examples and minimise the similarity between pairs of negative examples. This process allows the model to learn to correctly distinguish between similar and different examples, improving the quality of the learned representations.

A distinctive aspect of SimCLR is the importance of data augmentation. By using two different versions of the same image, SimCLR exploits data augmentation to improve the generalisation of the model and obtain more robust representations. Furthermore, SimCLR introduces a projector within the Siamese network to avoid representation collapse, a phenomenon in which the model learns trivial or uninformative representations of unlabelled data.

After pre-training with SimCLR, the projector is discarded and the trained Siamese network can be used for supervised learning tasks (downstream tasks), such as classification or image segmentation.

2.2. BYOL

BYOL (Bootstrap Your Own Latent) is a method of self-supervised learning that is based on the structure of MoCo (Momentum Contrast) but introduces significant modifications. BYOL doesn't use Contrastive Loss, but the L2 loss. This results in the absence of the need for negative samples, making the method less dependent on batch size. BYOL uses a student/teacher structure where the student network learns from the image and the target network is used as a reference point for learning.



A distinctive aspect of BYOL is the use of the Exponential Moving Average (EMA) update for updating the target network. In this process, the weights of the target network are updated using an exponential moving average of the weights of the student network. This type of asymmetric update allows the target network to gradually follow the student network in the learning process. The EMA step is:

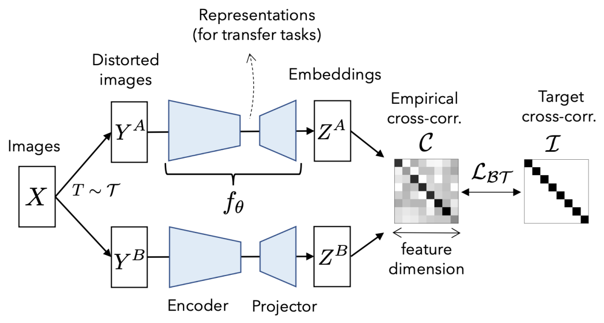
$$\theta_{\text{t}} \leftarrow \xi\theta_{\text{t}} + (1-\xi)\theta_{\text{s}}$$

starting from the initial condition in which the teacher net is initialized as the student $\eta = \theta_s$

The asymmetric update is controlled by a parameter ξ, which determines the extent to which the target network preserves its history with respect to the student network. Typical values for ξ are around 0.999. This mechanism contributes to greater stability and generalization of the model over time, as the target network maintains a "stabilized" version of the information learned from the student network. Asymmetric updating of the target network via EMA is a key element that contributes to the stability and quality of the learned representations.

2.3. Barlow Twins

Barlow Twins is a self-supervised learning method that relies on using a cross-correlation matrix to learn augmentation invariant representations and reduce redundancy in the data. Barlow Twins aims to learn invariant and nonredundant representations through optimization of the cross-correlation matrix, thus contributing to greater stability and generalization of the learned representations.



Barlow Twins doesn't use Contrastive Loss or L2 loss, but focuses on the structure of the cross-correlation matrix. The cross-correlation matrix is constructed by associating each batch image with all the others, where the columns represent different augmented versions of the same image and the values on the main diagonal correspond to the augmented versions of the same image. The goal is to maximize the variance between an image and its augmented versions, while trying to minimize the covariance or correlation with all other images.

$$\mathcal{L}_{\mathcal{BT}} \triangleq \underbrace{\sum_i (1 - \mathcal{C}_{ii})^2}_{\text{invariance term}} + \lambda \underbrace{\sum_i \sum_{j \neq i} {\mathcal{C}_{ij}}^2}_{\text{redundancy reduction term}}$$

where C represents the cross-correlation matrix computed between the outputs of two identical networks along the batch dimension

To achieve these goals, the Barlow Twins loss consists of two terms: the first term maximizes the similarity between augmented versions of an image, while the second term minimizes the correlation between representations of different images. This seeks to maximize invariance to augmentations and minimize redundancy between representations, thus avoiding representation collapse.

2.4. k-Nearest Neighbors

The evaluation process using KNN in self-supervised learning methods involves several key steps. To begin with, features of the target dataset are extracted using a pretrained model, keeping the model weights frozen during this step.

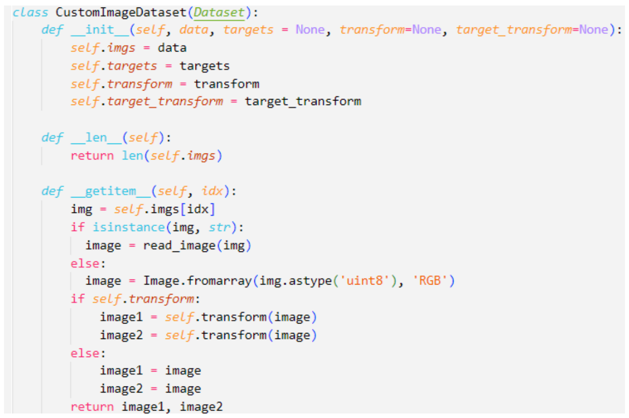
Next, the dataset is divided into two parts: a training set and a test set. To evaluate the performance of the model, the distances between the features of each image in the test set and those of all the images in the training set are calculated, commonly using metrics such as Euclidean distance.

Once the distances are obtained, the k closest neighbors are selected for each image in the test set, based on the previous calculations. Finally, to rank each image in the test set, a sort of ”voting” is performed among the classes of nearest neighbors. This process can be performed simply by choosing the most frequent class among the neighbors or by weighting the classes according to their frequency, especially if the dataset is unbalanced. The choice of parameter k is crucial as it affects the accuracy and robustness of the classifier.

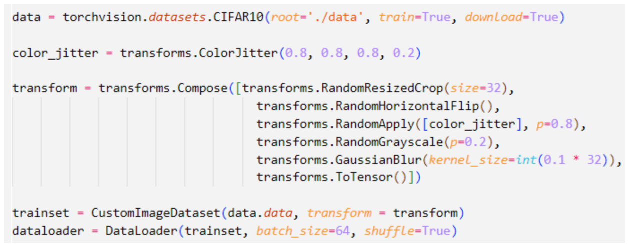
Using KNN as a classifier to evaluate representations learned through Self-supervised learning provides a measure of effectiveness in describing data and in the ability to generalize to new examples.

3. Code implementation

We first define a dataset that given an image generates two augmented versions of it. The `CustomImageDataset` class is used to manage the dataset and apply data augmentation transformations, while `getitem()` returns two augmented versions of the same image. If transformations have been specified, they are applied twice to the original image to generate two augmented versions (*image1* and *image2*), otherwise two identical copies of the original image are returned.



```python
class CustomImageDataset(Dataset):
    def __init__(self, data, targets = None, transform=None, target_transform=None):
        self.imgs = data
        self.targets = targets
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.imgs)

    def __getitem__(self, idx):
        img = self.imgs[idx]
        if isinstance(img, str):
          image = read_image(img)
        else:
          image = Image.fromarray(img.astype('uint8'), 'RGB')
        if self.transform:
            image1 = self.transform(image)
            image2 = self.transform(image)
        else:
            image1 = image
            image2 = image
        return image1, image2
```

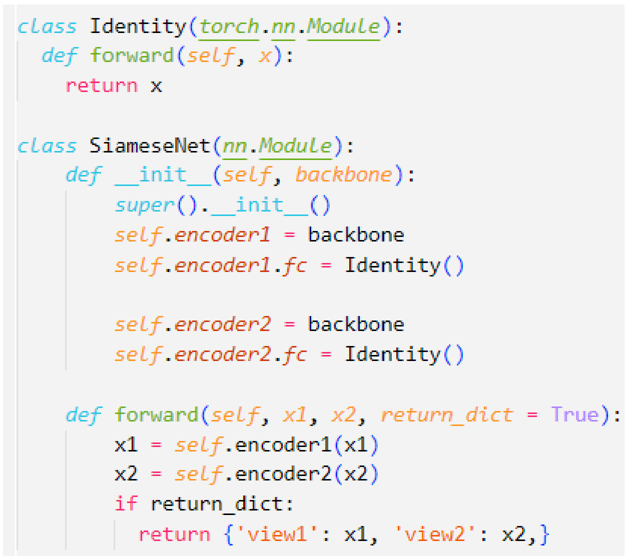


```python
data = torchvision.datasets.CIFAR10(root='./data', train=True, download=True)

color_jitter = transforms.ColorJitter(0.8, 0.8, 0.8, 0.2)

transform = transforms.Compose([transforms.RandomResizedCrop(size=32),
                                transforms.RandomHorizontalFlip(),
                                transforms.RandomApply([color_jitter], p=0.8),
                                transforms.RandomGrayscale(p=0.2),
                                transforms.GaussianBlur(kernel_size=int(0.1 * 32)),
                                transforms.ToTensor()])

trainset = CustomImageDataset(data.data, transform = transform)
dataloader = DataLoader(trainset, batch_size=64, shuffle=True)
```

Now let's see how to implement the above-mentioned models.

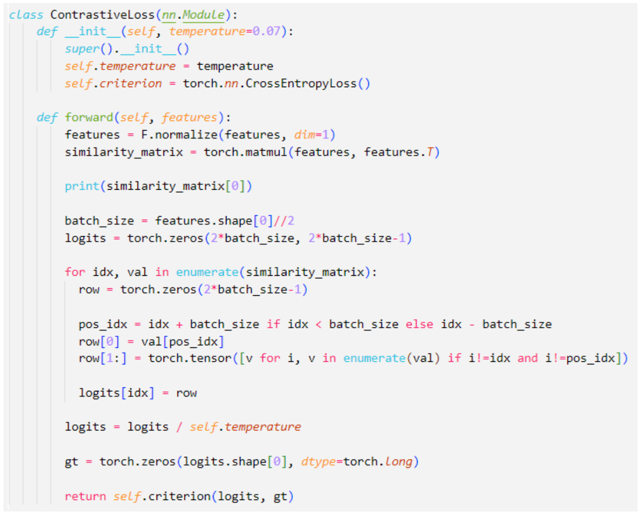
3.1. SimCLR

We implement a Siamese Network that takes as input two augmented versions of the same image. The Siamese Network consists of two identical encoders that simultaneously process the two augmented versions of the image.



```python
class Identity(torch.nn.Module):
  def forward(self, x):
    return x

class SiameseNet(nn.Module):
    def __init__(self, backbone):
        super().__init__()
        self.encoder1 = backbone
        self.encoder1.fc = Identity()

        self.encoder2 = backbone
        self.encoder2.fc = Identity()

    def forward(self, x1, x2, return_dict = True):
        x1 = self.encoder1(x1)
        x2 = self.encoder2(x2)
        if return_dict:
          return {'view1': x1, 'view2': x2,}
```

The `Identity` class replaces the fully connected layer of the encoder with an identity operation, which simply returns the input without modifying it; this is useful because we want to use the features extracted from the encoder without further transformation. The `SiameseNet` class defines the architecture of the Siamese network. It takes as input a backbone model (such as ResNet18) and creates two copies of the encoder, replacing their fully connected layer with the identity. The method `forward()` takes as input two augmented views of the image (*x1* and *x2*) and returns feature vectors.



```python
class ContrastiveLoss(nn.Module):
    def __init__(self, temperature=0.07):
        super().__init__()
        self.temperature = temperature
        self.criterion = torch.nn.CrossEntropyLoss()

    def forward(self, features):
        features = F.normalize(features, dim=1)
        similarity_matrix = torch.matmul(features, features.T)

        print(similarity_matrix[0])

        batch_size = features.shape[0]//2
        logits = torch.zeros(2*batch_size, 2*batch_size-1)

        for idx, val in enumerate(similarity_matrix):
          row = torch.zeros(2*batch_size-1)

          pos_idx = idx + batch_size if idx < batch_size else idx - batch_size
          row[0] = val[pos_idx]
          row[1:] = torch.tensor([v for i, v in enumerate(val) if i!=idx and i!=pos_idx])

          logits[idx] = row

        logits = logits / self.temperature

        gt = torch.zeros(logits.shape[0], dtype=torch.long)

        return self.criterion(logits, gt)
```

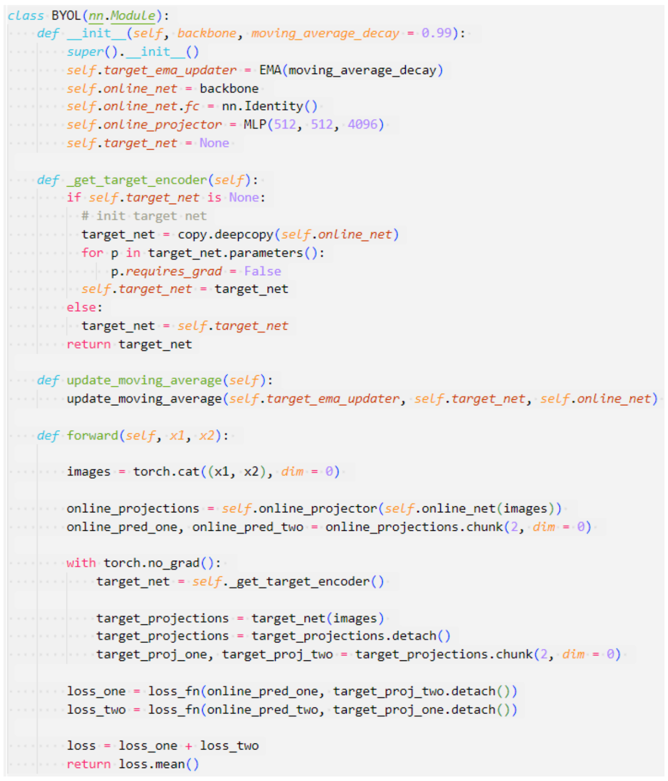
In the `ContrastiveLoss` class, features are normalized to have a unit norm, which is important for correctly computing cosine similarities between feature vectors. Next, a similarity matrix is calculated via the scalar product between the normalized features. For each image, logits are calculated, representing the similarity between one image and all others in the batch.

The indices of the logits are organized in such a way that the first element is always the similarity with the positive pair. The logits are scaled using a temperature to control the concentration of the probability distribution. Finally, Cross-Entropy Loss is applied on the logits with the ground truth having the positive pairs as the correct class (labeled 0).

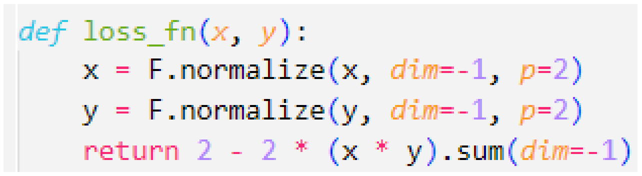
3.2. BYOL

BYOL is based on an asymmetric Siamese network structure, similar to MoCo, but with some key differences:

- **Online Network**: network that is updated by standard optimization method (SGD)
- **Target Network**: is the copy of the Online Network which is updated by an Exponential Moving Average (EMA) method

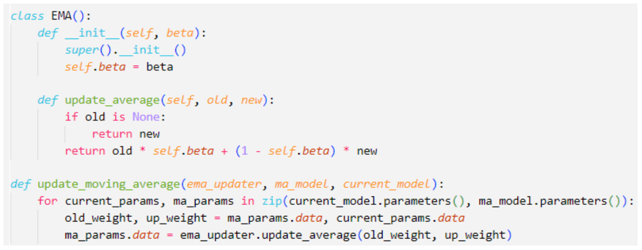


```python
class BYOL(nn.Module):
    def __init__(self, backbone, moving_average_decay = 0.99):
        super().__init__()
        self.target_ema_updater = EMA(moving_average_decay)
        self.online_net = backbone
        self.online_net.fc = nn.Identity()
        self.online_projector = MLP(512, 512, 4096)
        self.target_net = None

    def _get_target_encoder(self):
        if self.target_net is None:
            # init target net
            target_net = copy.deepcopy(self.online_net)
            for p in target_net.parameters():
                p.requires_grad = False
            self.target_net = target_net
        else:
            target_net = self.target_net
        return target_net

    def update_moving_average(self):
        update_moving_average(self.target_ema_updater, self.target_net, self.online_net)

    def forward(self, x1, x2):

        images = torch.cat((x1, x2), dim = 0)

        online_projections = self.online_projector(self.online_net(images))
        online_pred_one, online_pred_two = online_projections.chunk(2, dim = 0)

        with torch.no_grad():
            target_net = self._get_target_encoder()

            target_projections = target_net(images)
            target_projections = target_projections.detach()
            target_proj_one, target_proj_two = target_projections.chunk(2, dim = 0)

        loss_one = loss_fn(online_pred_one, target_proj_two.detach())
        loss_two = loss_fn(online_pred_two, target_proj_one.detach())

        loss = loss_one + loss_two
        return loss.mean()
```



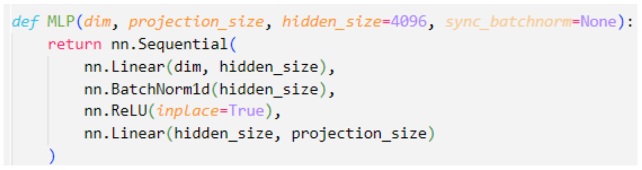
```python
def loss_fn(x, y):
    x = F.normalize(x, dim=-1, p=2)
    y = F.normalize(y, dim=-1, p=2)
    return 2 - 2 * (x * y).sum(dim=-1)
```

The loss used in BYOL is based on the Euclidean distance between the representations of the two augmented versions of the same image. This approach avoids the need for negative pairs, as required by Contrastive Loss.



```python
class EMA():
    def __init__(self, beta):
        super().__init__()
        self.beta = beta

    def update_average(self, old, new):
        if old is None:
            return new
        return old * self.beta + (1 - self.beta) * new

def update_moving_average(ema_updater, ma_model, current_model):
    for current_params, ma_params in zip(current_model.parameters(), ma_model.parameters()):
        old_weight, up_weight = ma_params.data, current_params.data
        ma_params.data = ema_updater.update_average(old_weight, up_weight)
```

BYOL uses an exponential moving average (EMA) technique to update the Target Network. This update is crucial to maintain learning stability.



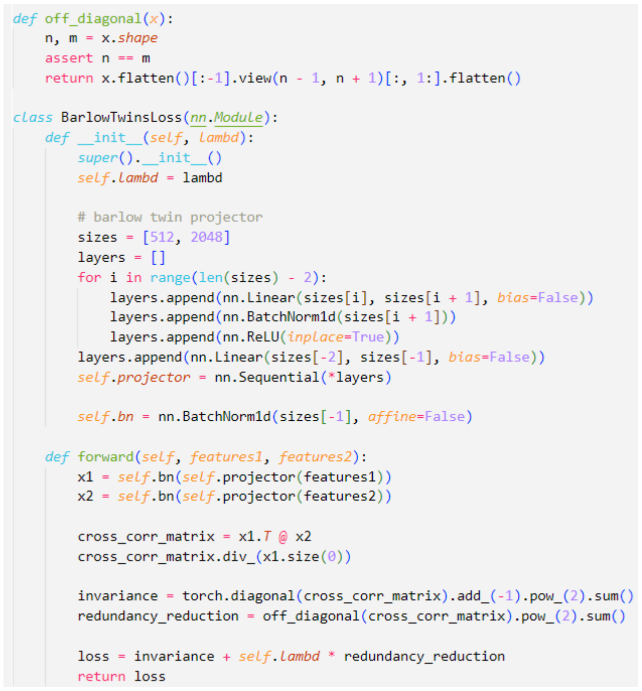
```python
def MLP(dim, projection_size, hidden_size=4096, sync_batchnorm=None):
    return nn.Sequential(
        nn.Linear(dim, hidden_size),
        nn.BatchNorm1d(hidden_size),
        nn.ReLU(inplace=True),
        nn.Linear(hidden_size, projection_size)
    )
```

The projector used is a Multi-Layer Perceptron (MLP), which is responsible for transforming the representations extracted from the Online Network into a hidden or "latent" representation space. This MLP is designed to process image features so that they can be better understood and used for learning representations.

3.3. Barlow Twins

We can start from the Siamese network implementation of SimCLR, because what Barlow Twins goes to do is to act on the similarity matrix. Therefore we are going to consider again the code of the Siamese network already defined above.

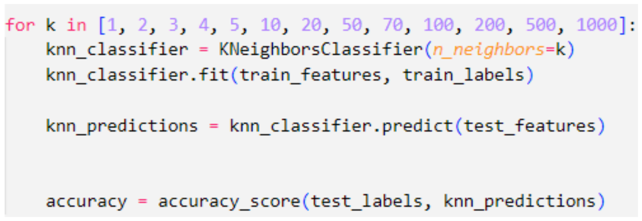
Barlow Twins Loss is calculated using two main components: invariance, calculated on the main diagonal of the cross-correlation matrix, and redundancy reduction, calculated on the off-diagonal elements. These components are then combined with the regularization parameter λ to obtain the final loss.



```python
def off_diagonal(x):
    n, m = x.shape
    assert n == m
    return x.flatten()[:-1].view(n - 1, n + 1)[:, 1:].flatten()

class BarlowTwinsLoss(nn.Module):
    def __init__(self, lambd):
        super().__init__()
        self.lambd = lambd

        # barlow twin projector
        sizes = [512, 2048]
        layers = []
        for i in range(len(sizes) - 2):
            layers.append(nn.Linear(sizes[i], sizes[i + 1], bias=False))
            layers.append(nn.BatchNorm1d(sizes[i + 1]))
            layers.append(nn.ReLU(inplace=True))
        layers.append(nn.Linear(sizes[-2], sizes[-1], bias=False))
        self.projector = nn.Sequential(*layers)

        self.bn = nn.BatchNorm1d(sizes[-1], affine=False)

    def forward(self, features1, features2):
        x1 = self.bn(self.projector(features1))
        x2 = self.bn(self.projector(features2))

        cross_corr_matrix = x1.T @ x2
        cross_corr_matrix.div_(x1.size(0))

        invariance = torch.diagonal(cross_corr_matrix).add_(-1).pow_(2).sum()
        redundancy_reduction = off_diagonal(cross_corr_matrix).pow_(2).sum()

        loss = invariance + self.lambd * redundancy_reduction
        return loss
```

3.4. k-Nearest Neighbors

Let us now look at the classifier we chose to evaluate the performance of the models.



```python
for k in [1, 2, 3, 4, 5, 10, 20, 50, 70, 100, 200, 500, 1000]:
    knn_classifier = KNeighborsClassifier(n_neighbors=k)
    knn_classifier.fit(train_features, train_labels)

    knn_predictions = knn_classifier.predict(test_features)


    accuracy = accuracy_score(test_labels, knn_predictions)
```

This code iterates over a series of k values, initialises a kNN classifier for each k value and then trains the classifier on the training data. Next, the trained classifier is used to make predictions on the test data and the accuracy of the predictions is calculated by comparing the actual labels of the test data with the predictions made by the classifier. The choice of parameter *k* is crucial, which is why several values have been proposed.