SVEUČILIŠTE U ZAGREBU
**FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**

DIPLOMSKI RAD br. 2754

# Samonadzirane metode za vizualno raspoznavanje

Vedran Bogdanović

Zagreb, rujan 2022.

*Umjesto ove stranice umetnite izvornik Vašeg rada.*
*Da bi ste uklonili ovu stranicu obrišite naredbu* `\izvornik`.

# CONTENTS

# 1. Introduction

The field of deep learning has been rapidly growing in the last decade. Aside from fulfilling researchers' curiosity, created programs have been widely used in real-life examples. With the improvement of computational power and increase in data, methods have increasingly better results. Supervised methods have been dominating the scene because of the great results they have been producing. Besides the increase in accuracy due to the development of better approaches, these methods also have the exciting property of increasing accuracy by increasing the amount of data on which they are trained. Sadly this increase has its limitations. Mainly the data must be annotated, which can be a long and expensive process. Because of this, supervised methods are not limited by available data, which is growing rapidly, but rather by annotated data.

Self-supervised methods bring a solution to this problem. For these methods, we design pretext tasks on which we can learn useful representation. Pretext tasks are designed to use labels that can be programmatically created from the data. Unlike supervised methods, they require only a small amount of labeled data and are a lot more scalable because of this. Even though supervised methods still outperform self-supervised ones, the gap between them is closing. In computer vision, self-supervised methods have had great results in image recognition but, more importantly, in image segmentation, where labeling data is even more expensive. Since these methods also improve with more data and better resources, performance will only increase over time.

Recently proposed solutions with promising results perform instance prediction in pretext tasks and are forms of Siamese network [9]. Those methods are contrastive and non-contrastive. Contrastive methods use positive samples, augmentations of the same image, and negative samples, augmentations of different images, while non-contrastive methods use only positive samples.

This thesis analyse contrastive and non-contrastive methods on small datasets and compares their performance. It also tries to explain how non-contrastive methods avoid representation collapse, even though they only use positive samples.

# 2. Self-supervised learning

The main difference between supervised and unsupervised methods is the availability of annotated data. Supervised methods usually try to learn useful representations from data $D = \{\mathbf{x}_i, y_i\}_{i=1}^{N}$ by estimating label $\hat{y}$ for every data point and then comparing those estimates to actual data and minimizing the loss function.
Because unsupervised methods lack information about the correct label, those methods are based on extracting information from a distribution. Some examples are density estimation, clustering, learning to draw samples from a distribution, learning to denoise data from some distribution, and so on [8].
Like supervised methods, self-supervised methods also estimate $\hat{y}$ and compare it with the actual label, but because data isn't annotated, pseudo-labels must be generated. There are multiple methods for generating pseudo-labels. The most common methods will be explained in sections 2.2 and 2.3. After generating pseudo-labels, our data, $D = \{t(\mathbf{x}_i), y_{gi}\}_{i=1}^{N}$, where $t$ is a transformation used and $y_{gi}$ is generated pseudo-label, can be used to solve a described supervised task. Because of this, we can think of self-supervised learning as a combination of unsupervised and supervised learning.

## 2.1. Problem definition

Self-supervised methods require labeled and unlabeled data sources. We can denote the labeled one as $D_l = \{\mathbf{x}_i, y_i\}_{i=1}^{N}$ and unlabeled one as $D_u = \{\mathbf{x}_i\}_{i=1}^{M}$, where $M >> N$. To learn useful representations, we design a pretext task which we train on unlabeled data by generating labels based on knowledge about the data.
The general steps of self-supervised methods are very similar to supervised methods after we generate pseudo-labels. These general steps are:

1. Firstly, we have to generate pseudo-labels for the pretext task by using random transformation from transformation family $T$.

2. The pretext task defines the model as

$$H = \left\{ g(h(\mathbf{x}; \theta)) \right\}_{\theta}$$

   Here $g(\cdot)$ represents the prediction head, and $h(\cdot)$ represents the feature extractor. In this step, we try to minimize the empirical error by finding the best hypothesis, $h^*$.

3. After we have trained our model and obtained useful representation on a pretext task, we usually discard the prediction head and use fine-tuning or linear readout to prepare the model for a downstream task. This training is done on labeled data, but the amount is way smaller than what is needed to train standard supervised methods.

## 2.2. Pretext tasks

Pretext tasks are used to create supervised-like learning on unlabeled data. They are designed to learn useful representation by introducing some bias about data. Pretext tasks can further be divided into four subcategories: transformation prediction, instance discrimination, masked prediction, and clustering. Transformation prediction and instance discrimination are methods that perform very well on representation learning tasks. On the other hand, masked prediction methods have had more success in NLP.

Some examples of transformation prediction tasks that gave competitive results when they came out are rotation prediction [10], image colorization [11], and solving the jigsaw puzzle [12]. Each task tries to learn useful representation by learning a function $f : X \to Y$, where $X$ are augmented data points and $Y$ discrete labels representing one of the predefined augmentations used. In listed tasks, augmentations are rotation, image grayscaling, and shuffling of the cropped tiles. Labels for these tasks are: $n$ (usually 4) classes denoting the angle of rotation performed on an image, the color of the center pixel, index of permutation used chosen from $n$ predefined permutations. These tasks learn useful representation because it takes an excellent semantic understanding of an image to perform the task successfully. The validity of these methods can be empirically shown by training the networks on a large dataset, like ImageNet, and then evaluating them on some downstream task after fine-tuning.

One downside of these methods is that they introduce specific properties in representation. Because of this, there is no single task that outperforms others. Somewhat the success is heavily influenced by the downstream task that they are trying to solve. E.g. suppose we want representation to vary with orientation. In that case, we might use image rotation prediction, but if we are dealing with a dataset for which all orientations should produce the same result, then it might not be the best choice [13]. In Instance

**Figure 2.1:** Example of the transformation prediction task. The first image is the original data point, the second image is permutated data point used as an input for the pretext task, and the third image is the solution to the pretext task. The image was taken from [12].

discrimination methods, each data point can be observed as a different class. To create a pretext task, we must choose $n$ data points acting as surrogate classes. Next, we sample $m$ data points from each surrogate class using data augmentation $t \in T_i$, and train it as an N-classes classification problem. Popular method that uses this approach is Exemplar CNN [14]. Even though this method achieved good results when it came out, it has several drawbacks. The main drawback is that the performance stagnates after a certain number of surrogate classes. This stagnation happens because some classes will tend to be too similar to discriminate. In the original paper, the optimum number of surrogate classes was 8000 [14]. This limits one of the most significant advantages of unsupervised methods: learning on large amounts of unlabeled data.
This method's problem lies in the loss function. It uses standard cross-entropy loss, with loss function $l$:

$$L(X_u) = \sum_{\mathbf{x}_i \in X_u} \sum_{t \in T_i} l(i, t(\mathbf{x}_i))$$
$$l(i, t(\mathbf{x}_i)) = -\sum_k [[k = i]] \, log(f_k(t(\mathbf{x}_i)))$$

where $X_u$ are $n$ data points chosen as surrogate classes, $f_k(\cdot)$ denotes softmax output, and $t$ is the transformation used.

We can improve on this by using contrastive loss instead of cross-entropy loss and by using a variation of the Siamese network [9]. Using this loss, we reduce the number of classes to two. Instead of augmenting each data point once, we do it twice and then predict if two augmented images were generated from the same data point or not. The class is considered positive if augmented images are generated from the same data point and negative otherwise.

Similar to transformation prediction methods, augmentations used in instance discrimination methods introduce some invariances and influence performance on downstream tasks. E.g., if downstream tasks differentiate examples based on color information, using the Exemplar method will produce poor results [14].

## 2.3. Contrastive learning

In the contrastive method, we sample each point once, then try to maximize the agreement of positive pairs and minimize it for negative pairs. The basic framework of these methods can be seen in figure 2.2. Firstly, we use data augmentation $t \in T$ to generate two differently augmented data points. These augmented images are fed to the backbone, $f(\cdot)$, which calculates representations. The backbone is a ResNet [24] model whose depth depends on input data and network complexity. The backbone contains the most useful information and is used for downstream tasks. Representations are fed to the non-linear projection head, $h(\cdot)$, which calculates predictions. The backbone and projection head together form the encoder. Loss is calculated on predictions and tries to maximize the agreement between positive pairs and minimize it between negative ones in order to pull positive pairs closer together and push the negative pairs away.

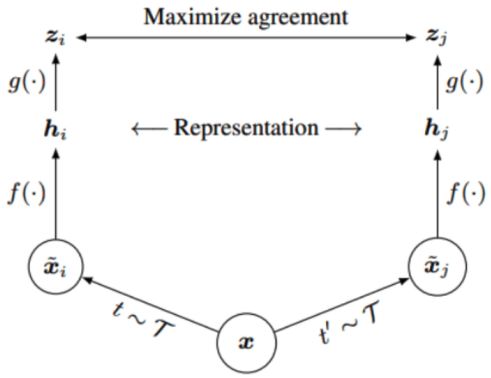


**Figure 2.2:** Framework for contrastive learning of visual representation. The image was taken from [15].

### 2.3.1. Loss function

The loss function used in contrastive learning is a form of InfoNCE loss [20]. The loss is used to infer which data point is positive in a batch of one positive data point and $N-1$ negative data points. This turns the N-class classification problem into a binary problem, where loss is a categorical cross-entropy of correctly classifying the positive data points. The loss function is:

$$L_N = -\mathbb{E}_X \left[ log \frac{s(\tilde{\mathbf{x}}_j, \mathbf{c})}{\sum_{\tilde{\mathbf{x}}_k \in X} s(\tilde{\mathbf{x}}_k, \mathbf{c})} \right] \tag{2.1}$$

$X$ is a set of data points. Context, $\mathbf{c}$, is a data point obtained from the same original point, $\mathbf{x}$, as $\tilde{\mathbf{x}}_j$ using different augmentation. Scoring function, $s(\cdot, \cdot)$, is a ratio between the probability of a data point being drawn from the distribution of positive samples and a data point being drawn from a distribution of negative samples.
The scoring function used in contrastive SSL methods is:

$$s(\tilde{\mathbf{x}}_j, \mathbf{c}) = exp\left( \frac{sim(\mathbf{z}_i, \mathbf{z}_j)}{\tau} \right) \tag{2.2}$$

where context $\mathbf{c}$ is equal to the other augmented image fed to siamese sub-network $\tilde{\mathbf{x}}_i$, $(i, j)$ is a positive pair of samples, $sim(\cdot, \cdot)$ is any function that calculates the similarity between two vectors, and $\tau$ is temperature parameter regulating how much similarity affects scoring function. Simple and popular choice for similarity function is the cosine similarity.
It can be shown that mutual information $I(\tilde{\mathbf{x}}_j, \mathbf{c})$ is an upper bound of negative loss [20]. Because of this, minimizing loss function maximizes the mutual information between context and positive sample.

$$I(\tilde{\mathbf{x}}_j, \mathbf{c}) \geq log(N) - L_N \tag{2.3}$$

Increasing batch size in contrastive methods has two benefits. Firstly, it reduces the noise during loss calculation and improves gradient approximation; secondly, it increases mutual information between positive pairs. Even though using large batch sizes is very beneficial for contrastive methods, it significantly increases training time or memory footprint. Figure 2.3 shows how contrastive methods learn the best representations on large batch sizes. The difference in the quality of learned representations is decreased with more training epochs because more negative samples are provided.

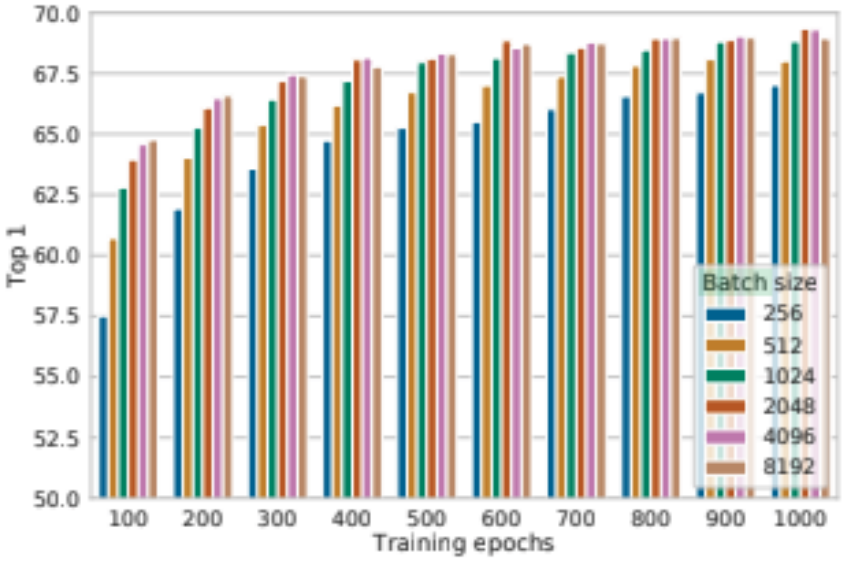


**Figure 2.3:** Linear evaluation models trained with different batch sizes and epochs. The image was taken from [15].

Issue of large batch sizes can be solved by using a dictionary as a queue of data samples. This dictionary contains representations from multiple batches. It is built as a queue by removing the oldest batch and adding the calculated representations from the latest batch. By doing this, we decouple batch size from negative examples and reduce memory footprint. Because we keep a lot of negative examples in the queue, it would be very inefficient to update the key encoder using back-prop. To avoid this query encoder is updated with back-prop and the key encoder uses momentum update. Finally, we can use large momentum $m$ to reduce the difference among encoders used to calculate representations for different batches. The formula for updating query encoder parameters is:

$$\theta_k \leftarrow m\theta_k + (1 - m)\theta_q \tag{2.4}$$

This method of building a moving dictionary of negative sample representations is used in MoCo method [16]. Figure 2.4 shows how this method calculates representations. On the left side, we calculate representation of first augmented sample $x^q$ using encoder, and on the right side we calculate representation of second augmented sample $x^k$ using momentum encoder. Representations from the right side are then added to the dictionary.

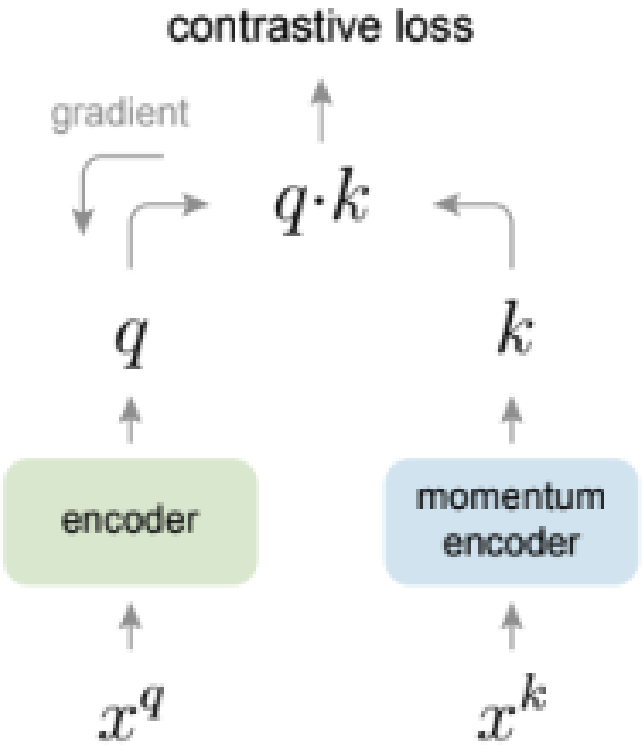


**Figure 2.4:** MoCo representation calculation. The image was taken from [16].

### 2.3.2. Backbone

Models for image classification have to be very deep to be expressive enough and learn useful representation. Although deeper networks can learn better representations they suffer from vanishing/exploding gradients.[1] Using normalization layers can solve this problem. However, deeper networks can still hurt performance during training, which should only be increased in theory. ResNet architecture solves this problem by implementing residual blocks. Residual blocks consist of layers with skip connections which can be seen in figure 2.5. These skip connections ensure that adding more layers can only increase the performance because it is trivial for the network to learn identity mapping and indirectly reduce the depth.

There are several variations of this architecture. They all have the same building blocks called residual blocks, and differ in the number of layers in each block. Variations with: 18, 34, 50, 101 and 152 layers were presented in the original paper [24]. The simplest architecture, ResNet-18, can be seen in figure 2.6.

[1]This problem occurs during back-propagation, where the gradient either constantly decreases or increases by multiplying the gradient components of each layer. When this happens, model performance either stagnates or deteriorates.

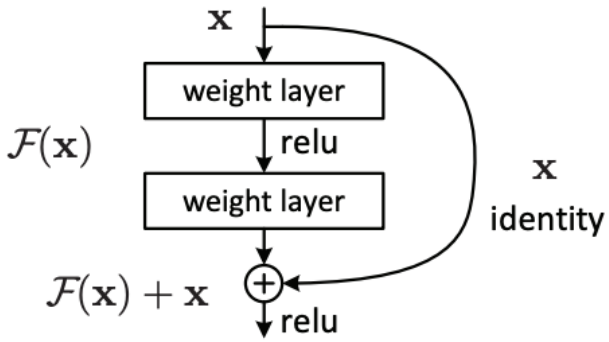


**Figure 2.5:** Skip connection. The image was taken from [24].

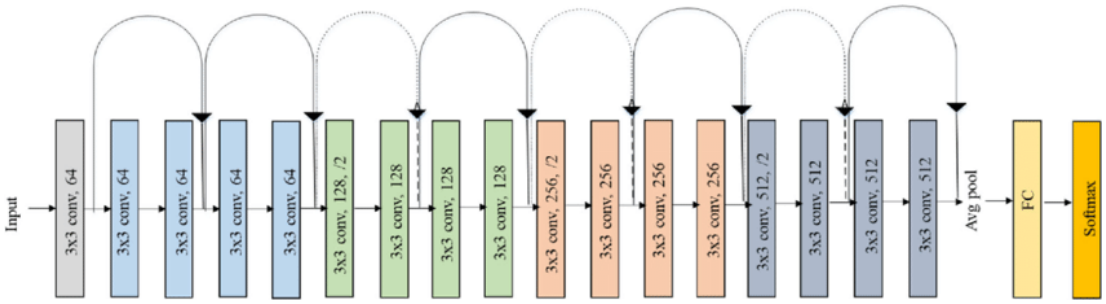


**Figure 2.6:** ResNet-18 architecture.

### 2.3.3. Data augmentation

In supervised classification problems, data augmentations are used to improve generalization. Applying different transformations to the input data increases the invariance to those transformations during prediction. Some of the transformations used are translation, rotation, color jittering, and gaussian noise.

Contrastive self-supervised methods also improve performance by applying augmentations on input data, but it is only partly due to improvement caused by generalization. Since contrastive methods aim to maximize the agreement between positive pairs, the method might compare pixel intensity histograms or some other non-generalizable feature and easily solve the contrastive task without learning useful features. To prevent this, we use very strong augmentations in contrastive learning to increase tasks's difficulty. These methods usually benefit from much stronger augmentations than supervised methods. SimCLR [15] finds that image cropping and color jittering transformations work exceptionally well when applied together. The base pipeline used in all the methods described in this work is composed of these transformations: cropping, resizing, horizontal flipping, color jittering, converting to grayscale, gaussian blurring

and solarization. The first two transformations are always applied, and other transformations are applied with some probability. Pipeline is shown in figure 2.7.

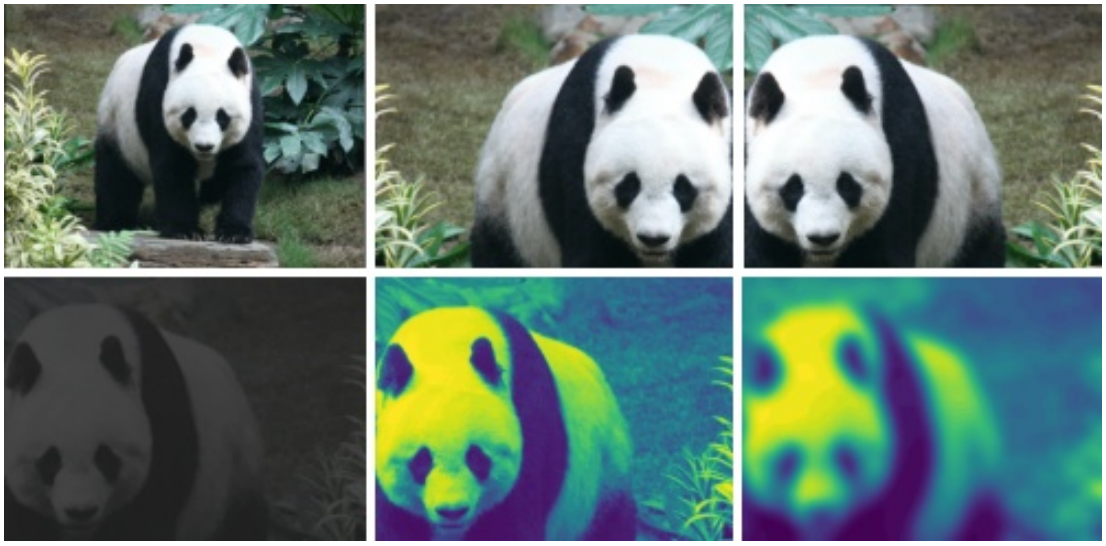

**Figure 2.7:** Example of an image passed through the data augmentation pipeline.

## 2.4. Non-contrastive learning

As pointed out in the previous section, although contrastive methods are a significant improvement over other instance discrimination methods, they still have drawbacks. Mainly the large amount of negative samples required to train them. This imposes one of two problems. Methods either require large batch sizes or increase the implementation complexity and memory overhead.

Non-contrastive methods try to solve this problem by training only on positive samples. Removing negative samples from the loss function, makes the method more robust to batch size and augmentation choice. Because the method does not use negative samples, batch size and the ability to avoid representational collapse are uncorrelated.

Batch size in non-contrastive methods has a similar effect on the network as in supervised methods. Because of this, methods are still slightly improved by using a larger batch size.

### 2.4.1. Architecture

Since negative samples are essential for avoiding representational collapse in contrastive methods, a natural question arises regarding how non-contrastive methods avoid representation collapse.

Collapse is avoided by creating asymmetry in the network and introducing a stop-gradient operation[2] to one of the Siamese [9] sub-networks. Asymmetry is achieved by adding prediction MLP to one sub-network (online network), and then the stop-gradient operation is added to the other sub-network (target network). Both of these modifications are needed to avoid representation collapse. Described architecture can be seen in figure 2.8.

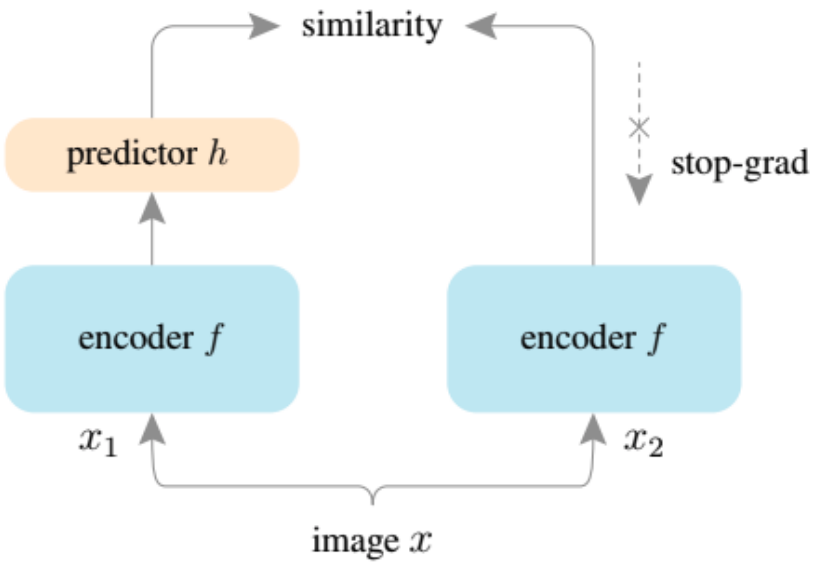


**Figure 2.8:** Non-contrastive method architecture. Image was taken from [21].

The encoder in figure 2.8 has the same architecture as the encoder in contrastive methods seen in figure 2.2. It consists of augmentation $t \in T$, backbone $f(\cdot)$, and projection MLP $g(\cdot)$. Like in contrastive learning, the backbone contains the most useful information and is used in downstream tasks. The difference between contrastive and non-contrastive models lies in the extra prediction head and stop-gradient opera-

[2]First non-contrastive method proposed was BYOL [18] which uses a momentum encoder in the target network and implicitly adds stop-gradient operation to that sub-network. SimSiam [21] later showed that the momentum encoder improves performance by stabilizing the representation calculation step but is not necessary for avoiding representation collapse. Since SimSiam [21] only removes momentum encoder from BYOL [18], it can be seen as an ablation study of that method.

tion seen in figure 2.8. Due to this asymmetry, the loss is not calculated on projection heads' output $z_1$ and $z_2$ but instead on projection head's output $z_1$ and one prediction head's output $p_2$. Stop-gradient operation in the target network causes gradients to be calculated only for the target network, while the online network is updated using EMA. To describe a general case, we can view the target network as a momentum encoder.[3] BYOL method uses a high momentum value (between 0.9 and 0.999) [18]. Because of this encoder in the target network is calculated as an exponential moving average of the encoder in the online network. SimSiam, on the other hand, uses a 0 momentum value, which means that the encoder in the target network has the same weights as the encoder in the online network.

It is essential to distinguish the duty of the momentum encoder in contrastive methods and non-contrastive methods. In contrastive methods, it is used to decouple batch size from the number of negative samples used. In non-contrastive methods, these two are naturally decoupled. Instead, it is used to introduce stop-gradient operation and to stabilize the representation calculation step.

### 2.4.2. Loss function

Loss function used in non-contrastive learning method SimSiam [21] is a cosine similarity between $l_2$-normalized predictions from online network and $l_2$-normalized projections from target network.[4] $l_2$-normalized cosine similarity is shown in the following equation:

$$D(\mathbf{p}, \mathbf{z}) = \frac{\mathbf{p}}{\|\mathbf{p}\|_2} \cdot \frac{\mathbf{z}}{\|\mathbf{z}\|_2}$$

Online predictions, $\mathbf{p}_1$ and $\mathbf{p}_2$, and target projections, $\mathbf{z}_1$ and $\mathbf{z}_2$, are calculated from augmented data points, $t_1(\mathbf{x})$ and $t_2(\mathbf{x})$, respectively. This makes the loss function

$$L = \frac{1}{2} D(\mathbf{p}_1, sg(\mathbf{z}_2)) + \frac{1}{2} D(\mathbf{p}_2, sg(\mathbf{z}_1))$$

symmetric and improves performance because it makes two times more predictions for each image. Not using symmetric loss can be compensated by calculating the loss on two pairs of samples at each step [21]. Non-contrastive loss function introduces one more subtle improvement. Since we do not use negative samples during loss calculation, these methods are more robust to augmentation choice.

[3] Since the momentum encoder naturally implements stop-gradient operation to the target network, we can obtain both methods by changing the momentum update value.

[4] Because targets are $l_2$-normalized using cosine similarity is the same as using MSE [18].

## 2.5. Barlow twins

Barlow twins is another SSL method that is very similar to contrastive methods but can not be categorized as such because of its unique loss function. The model uses a very similar architecture for encoder with ResNet backbone and MLP projection head. However, the projection head is slightly different with three layers, and the output dimension is significantly larger compared to contrastive methods like [15; 16].

### 2.5.1. Loss function

In this section loss function of Barlow twins [22] will be analyzed because all differences from contrastive methods stem from its loss function.
Loss function consists of two terms:

- **Invariance**: minimizes the difference between embeddings of the same data point augmented with augmentation pipeline from 2.3.3. By doing this, the model learns useful representation invariant to the distortions. This term has the same effect as the numerator in equation 2.1.
- **Covariance**: minimizes correlation between embeddings of different data points. By doing this, it stops the model from learning trivial solutions. This term has the same effect as the denominator in equation 2.1.

Described loss function is:

$$L_{BT} = \sum_i (1 - C_{ii})^2 + \lambda \sum_i \sum_{j \neq i} C_{ij}^2 \tag{2.5}$$

The cross-correlation matrix $C$ is computed between embeddings (along the batch dimension) calculated from the same data points with different augmentations. Regularization hyperparameter $\lambda$ balances invariance and covariance term. The loss function is minimal when the correlation matrix is an identity matrix.
This function connects with the Information Bottleneck principle, which states that representation should be informative about the sample and non-informative about the augmentations used [22]. This is precisely what we are trying to accomplish to obtain useful representations. The principle is given with the following equation:

$$IB_\theta = I(Z_\theta, Y) - \beta I(Z_\theta, X) \tag{2.6}$$

$$IB_\theta = H(Z_\theta|X) + \frac{1-\beta}{\beta} H(Z_\theta) \tag{2.7}$$

$I(\cdot, \cdot)$ represents mutual information and $\beta$ is a positive number and has a similar task as $\lambda$ in equation 2.5 but is inversely proportional to it. The equation 2.6 should be

minimized because the first term is the mutual information between embeddings and augmentations, and the second term is the mutual information between embeddings and represented sample.
Unfortunately, we can not use this equation directly because measuring the entropy of a large dimensional signal would also require large amounts of data. This would lead to the same need for large batch sizes as in contrastive loss. To avoid this, we could assume that the representation $Z_\theta$ is distributed as Gaussian. With this assumption, we would have to calculate the logarithm of the determinant of its covariance function. Because we do not have to estimate the entropy of $Z_\theta$ there is no need for large batch sizes [22]. This is the central assumption that allows this loss function to be minimized and the model to learn useful representations without large batches.
A few more assumptions must be made to fully connect the loss used in the paper and the Information Bottleneck principle. However, they will not be stated here because they are not responsible for minimizing batch size.

The loss function can also be reduced by decreasing the value of embeddings. This results in a very sparse embedding matrix, and because of this method benefits from the high dimensional output space of the projection head. E.g., Barlow twins paper [22] reports the best performance on ImageNet with 16384 output dimension of projection head.

## 2.6. Downstream task

The downstream task is the actual task we are trying to solve. This task is solved using some variation of gradient descent optimization. Since the starting point of gradient descent optimization affects convergence, an excellent way to initialize weights is with an already pre-trained model. This method is called transfer learning, which was shown to speed up the training process of the target task. One drawback of transfer learning is that learned representations on source labels generalize less well on target labels [17]. Self-supervised learning improves on this because it does not use source labels. Rather than optimizing representation for source labels, it learns representation based on bias exploited by pretext tasks. Because of this, it is less likely to overfit on source tasks. Even though it is harder to overfit using SSL, we still need to pick the correct pretext task based on the downstream task to learn useful representations.
Data augmentations used in pretext tasks also greatly influence the quality of representations [15]. Some augmentations can also hurt the accuracy of the downstream model,

as described in section 2.2.

The backbone of the encoder is kept as the feature extractor for the downstream task, and the projection head is dropped and changed with the more suitable one for the downstream task. At this step, we train the network using labeled data. It is important to emphasize that the amount of labeled data needed is much smaller than we would typically use in standard supervised learning. Therefore, we can use linear readout or fine-tuning to adapt the network for the downstream task. Linear readout freezes the feature extractor and updates only the weights of a new projection head while fine-tuning updates all the weights. The choice is based on the similarity between the source and target tasks and the amount of available labeled data.

# 3. How do non-contrastive methods avoid collapse?

This chapter will analyze the stop-gradient's and predictor's contribution to avoiding representation collapse more thoroughly. This model for obtaining a better understanding of non-contrastive methods was proposed in [23]. All mathematical proofs can be found in the cited paper.

## 3.1. Simple linear model

To gain a better insight into how non-contrastive methods avoid representation collapse, we can observe a simple linear BYOL model. Architecture of this model can be seen in figure 3.1. In the experiments chapter, it will be shown that conclusions drawn from this simple model generalize to real-world examples.

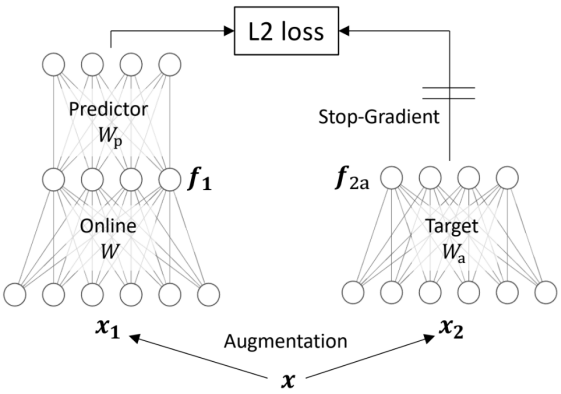


**Figure 3.1:** The simple linear architecture of the BYOL model. We can compare it to figure 2.8. In the online network, the first FC layer with weights $W$ is the encoder, and the second FC layer with weights $W_p$ is the predictor. In the target network, FC layer with weights $W_a$ is the momentum encoder. The image was taken from [23].

Loss used is the same as in BYOL [18]:

$$J(W, W_P) = \frac{1}{2}\mathbb{E}_{\mathbf{x}_1,\mathbf{x}_2}\left[||W_p\mathbf{f_1} - sg(\mathbf{f_{2a}})||_2^2\right] \tag{3.1}$$

where $\mathbf{f_1}$ and $\mathbf{f_{2a}}$ are encoder outputs from online network and target network, respectively. Due to the simple architecture, we can analytically calculate gradient updates for weights and analyze how stop-gradient and predictor affect learning. By removing the stop-gradient operation from the target network, gradient update for matrix W becomes:

$$\begin{aligned}\frac{d}{dt}vec(W) &= -\frac{\partial J}{\partial\, vec(W)} \\ &= -H(t)\, vec(W)\end{aligned}$$

where $t$ represents a single training step and derivation over the training step is an update of a matrix after each training step.[1] Matrix H is constucted as follows:

$$\begin{aligned}H(t) &= X' \otimes (W_p^T W_p + I_{n_2}) + X \otimes \tilde{W}_p^T \tilde{W}_p + \eta I_{n_1 n_2} \\ \tilde{W}_p &= W_p - I_{n_2}\end{aligned}$$

where $X$ is a covariance matrix of input data, $X'$ is a covariance matrix of augmentations used on input data, $I_x$ is an identity matrix, and $n_1$ and $n_2$ are dimensions of input data and encoder output, respectively.

Since all elements of matrix H are PSD matrices and both summing and applying the Kronecker product on two PSD matrices result in a PSD matrix, matrix $H$ is a PSD matrix. Because of this, we can prove that weights will collapse to 0 if eigenvalues of $H$ have a lower bound greater than 0. We do this by constructing the Lyapunov function

$$V(vec(W)) = \frac{1}{2}||vec(W)||_2^2$$

and derivating it. We get the update $-vec(W)^T H(t) vec(W)$ which is by definition always equal or greater than 0. Because of this $l_2$-norm will deteriorate to 0 and so will the weights.

This proves that removing the stop-gradient operation from the target network, under this assumption, causes representations to collapse. Removing predictor[2] and making the network symmetric will cause stop-grad to only reduce the update of weights by the number of two. Update, in this case, would be the same as just using half the loss from

[1] This notation will help us prove that weights will collapse.

[2] Adding the extra predictor to the target network would yield the same effect. We would just have a deeper projector.

the equation 2.4.2 without stop gradient. We analytically proved that non-contrastive methods could not work without these two additions.
To further simplify the model and show why stop-gradient operation helps avoid collapse, we will introduce three assumptions [23]:

1. Firstly, we assume that the weights of target network $W_a$, which are updated by EMA, are always linearly proportional to the online network with relation $W_a(t) = \tau(t)W(t)$. E.g. $\tau$ is always 1 in SimSiam [21].

2. Second assumption, which helps simplify the model, is that the data distribution $p(x)$ has 0 mean and identity covariance matrix and that augmentation distribution $p_{aug}(\cdot|x)$ has $\mathbf{x}$ mean and $\sigma^2$I covariance matrix.

3. The last assumption is that predictor is symmetric. This is a valid assumption since the predictor becomes more symmetric during training progress.

By incorporating these assumptions to gradient updates of weights, we can calculate gradient update of:

$$[F, W_p] = FW_p - W_pF \tag{3.2}$$

where F is a covariance matrix of online representation and $W_p$ are weights of the prediction head. This relation has the dynamics of negative semi-definite system. Using the same proof, as for the weights $W$ in the absence of stop-gradient operation, it can be shown that the relation will collapse to 0, which is equivalent to eigenvector alignment.
When eigenvectors align, we can prove that diagonal decomposition vector $U$, where $F(t) = U(t)diag(s_j)U(t)^T$, will not change over time. Because of this, we can replace weight matrices update for eigenvalues update. This turns original equations for updating weights to scalar updates of eigenvalues $p_j$, of matrix $W_p$, and eigenvalues $s_j$, of matrix $F$. Eigenvalue updates are given with following equations:

$$\begin{aligned}
\dot{p}_j &= \alpha_p s_j[\tau - (1+\sigma^2)p_j] - \eta p_j \\
\dot{s}_j &= 2p_j s_j[\tau - (1+\sigma^2)p_j] - 2\eta s_j \\
s_j\dot{\tau} &= \beta(1-\tau)s_j - \frac{\tau \dot{s}_j}{2}
\end{aligned}$$

where $\alpha_p$ is the predictor's learning rate, $\eta$ is weight decay, and $\sigma^2$ is the variance of augmentation distribution. From these equations we can obtain the dynamics of eigenvalues:

$$s_j(t) = \alpha_p^{-1} p_j^2(t) + e^{-2\eta t} c_j \tag{3.3}$$

which largely depends on weight decay. There are three interesting cases that we will observe.

1. With no weight decay, initial conditions are never forgotten, and representations will collapse if the eigenvalues of matrix F become 0. In order to avoid the collapse initial conditions should be $s_j(0) > p_j(0)^2/\alpha_p$. This condition explains findings in [18], that updating the predictor more frequently increases performance. It also supports findings in [21] stating that updating predictor too frequently hurts it. This is due to the eigenvalues of matrix F being minimally affected by the growth of the predictor's eigenvalues. This also proves that having a prediction head and stop-gradient operation in the same sub-network wouldn't work because $\alpha_p$ would be 0 and we would always fall into trivial stable point. Case with no weight decay is shown in figure 3.2a. There are two stable points, saddle which leads to trivial solutions and $p^*_{j+} = \frac{\tau}{1+\sigma^2}$ which doesn't.

2. With weight decay initial conditions are forgotten with time and equation 3.3 becomes parabola. When this happens, we can calculate three fixed points where $\dot{p}_j$ is 0. Two of those points are stable, and one is not. Increasing $\tau$ and decreasing $\eta$ decreases unstable points, shown in figure 3.2b with green color, and makes it harder for representation to collapse. However, it also increases the non-collapsed stable point $p^*_{j+}$ and slows the training.

3. By setting weight decay to be too high, only one stable point remains, and representations are bound to collapse. This case is shown in figure 3.2c.

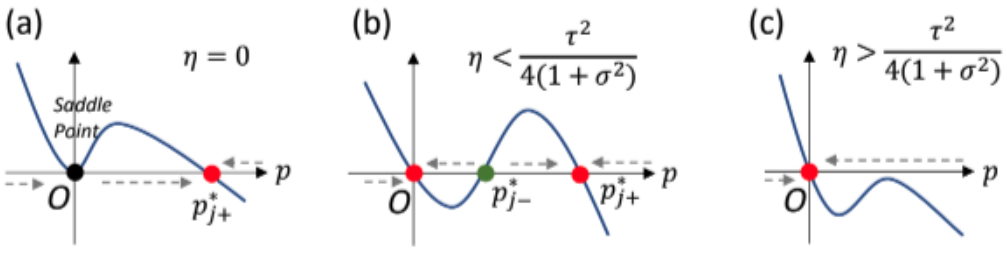


**Figure 3.2:** Fixed points for 3 different values of weight decay. Stable points are depicted with red and unstable with green color. The image was taken from [23].

This simple linear model shows how weights update collapses to 0 in the absence of prediction head or stop-gradient operation. It also shows how other hyperparameters

affect representation collapse and learning stability. Weight decay, predictor's learning rate, and EMA weight need to be carefully chosen to avoid representation collapse and allow the model to learn useful representation in a reasonable time.

## 3.2. DirectPred

DirectPred [23] is a non-contrastive model derived from observations presented in previous section. On a simple linear model, it was shown that the eigenvectors of the prediction matrix and covariance matrix of online representations gradually align.[3] Instead of updating predictor with back-prop, this method sets it. The predictor is calculated with the following equation:

$$p_j = \sqrt{s_j} + \epsilon \max_j s_j, \; W_p = \hat{U} diag(p_j) \hat{U}^T \tag{3.4}$$

Here $\hat{U}$ and $s_j$ are values obtained by computing eigendecomposition of $\hat{F}$. Estimated correlation matrix $\hat{F}$ is computed using a moving average. The correlation matrix is not zero-centered because this deteriorates the performance. Regularization hyperparameter $\epsilon$ helps by increasing small eigenvalues. Before calculating eigenvalues of predictor, eigenvalues $s_j$ are normalized so that the highest eigenvalue is set to one [23].

For this implementation to work, the predictor needs to be a one-layer MLP. Without this, we could not set its weights. Interestingly, even with single layer prediction head constraint, DirectPred [23] can perform equally well as BYOL [18] with a three-layer MLP prediction head.

[3] It will also be shown later in experiments that this happens during training of BYOL model.

# 4. Experiments

## 4.1. Google Colab

Google colab [1] is a Jupyter notebook environment that runs in the cloud. The main benefit of this service is that programs can be run on various GPUs. Some of the available GPUs are K80, T3 P100, and V100. These GPUs have between 12 and 16 GB of RAM, which allows complex networks to be trained on large datasets. Another benefit is a seamless connection to google drive. All the experiments done in this thesis were run on Google Colab.

## 4.2. Dataset

CIFAR-10 dataset [7] consists of 60,000 32x32 images in 10 classes. There are 50,000 images in the training set and 10,000 in the test set. The classes are completely mutually exclusive. Classes and examples can be seen in figure 4.1. This dataset was chosen because of its size. The training time of one epoch for the experiments was between one and three minutes.
Training images are augmented according to the pipeline described in subsection 2.3.3. Since CIFAR-10 is a low-resolution dataset, and described pipeline was designed for ImageNet, in these experiments, gaussian blur is omitted from the pipeline. Images used for validation are normalized.
The batch size is 512 in all experiments.

## 4.3. Encoder

All the models used in experiments contain an encoder network. The encoder consists of a backbone and projection head. Backbone contains the most useful representations and is later used for the downstream task. The ResNet [24] model was used for

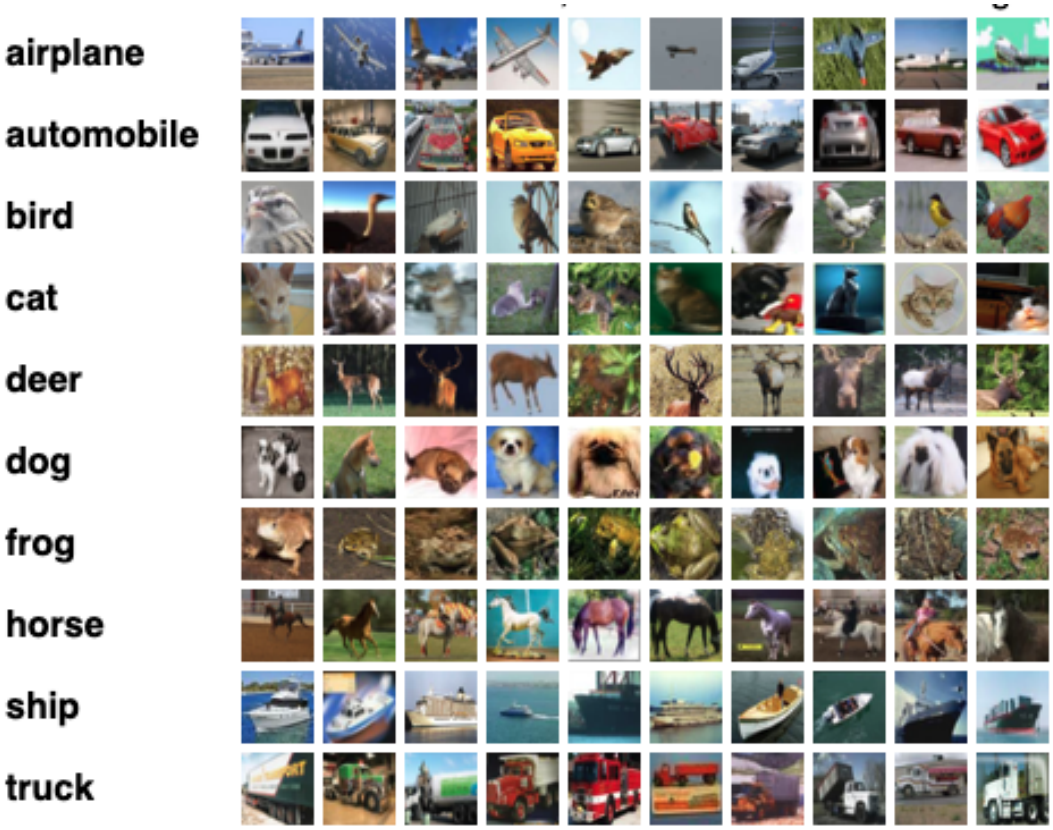


**Figure 4.1:** Random images from each class of CIFAR-10 dataset.

the backbone, and the number of layers is based on the dataset on which the model is trained. Because we are using a very small dataset in experiments ResNet-18 architecture is deep enough to learn useful representations. Even though ResNet-18 is sufficiently deep, we could boost the performance by using some deeper variations like Resnet-34 or ResNet-50. Learning time is the main reason for using 18-layer architecture over 34 or 50 layers. Resnet-34 has two times more parameters; hence the learning is way slower.

ResNet model was loaded using Lightly framework [2].[1] Projection head used in experiments is a 2 layer MLP consisting of:

1. Fully connected layer with ReLU activation.
2. Fully connected layer.

It is essential to use the outputs of the projection head and not the backbone during loss calculation. Representations learned this way are better, and the backbone will

[1]Complete implementation of ResNet models can be found on https://github.com/lightly-ai/lightly/blob/master/lightly/models/resnet.py

perform better on downstream tasks.

The problem with using outputs of backbone for loss calculation is that it becomes invariant to some transformations used, which are inevitable for making the SSL task challenging but hurt the model when classifying images. By using a nonlinear projection head, more information can be stored in the backbone, and the projection head can model some invariances.

The difference in representation quality between MoCo models with and without projection head can be seen in figure 4.2. In the case of the encoder without a projection head, it can be seen that the training loss keeps decreasing while the validation accuracy stagnates. This means that the backbone overfits to the training set. On the other hand, it can also be seen that the encoder with a projection head does not overfit to the training set.

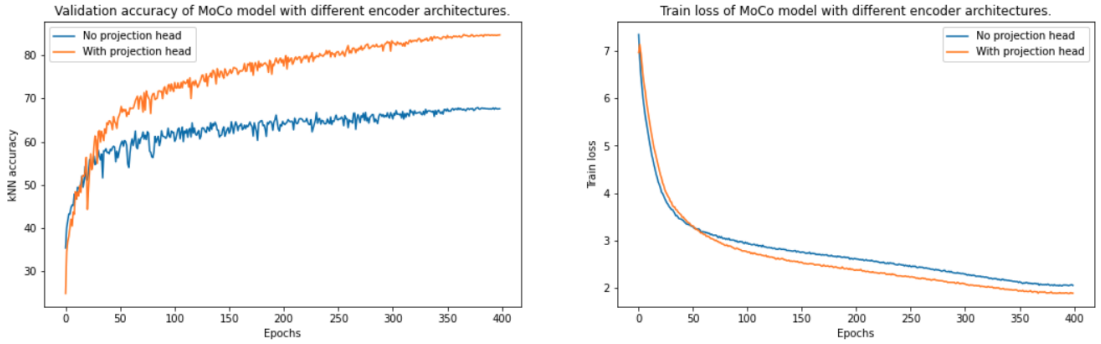


**Figure 4.2:** MoCo model performance with different encoder architectures.

## 4.4. Optimization algorithm

SGD with momentum and cosine decayed learning rate is used for the optimization algorithm.[2] In all experiments value for the momentum hyperparameter is 0.9, while the other hyperparameters (weight decay and base learning rate) differ.

The learning rate of the Barlow twins method used in experiments can be seen in figure 4.3

[2]Since contrastive methods require large batch sizes, they usually use LARS [25] optimization algorithm. MoCo method is an exception because it does not require a large batch size to give competitive results. Because of this SGD is also used in MoCo method.

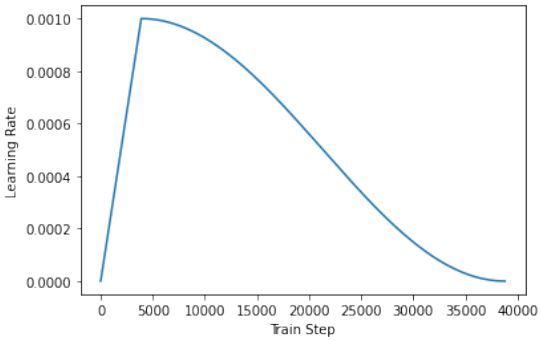


**Figure 4.3:** Learning rate in model trained for 400 epochs on CIFAR-10.

## 4.5. Evaluation protocol

For model evaluation on the validation set, a non-parametric weighted kNN classifier is used [19]. This metric is consistent and easy to use during validation, unlike the parametric linear classifier, which we have to train on a frozen backbone.

Implementation of weighted kNN classifier was taken from [3] and can be seen in figure 4.4. At the end of each epoch, we freeze weights and calculate features (from training data points without augmentation) used in the feature bank. During the validation step, features from the validation set are calculated and passed to the weighted kNN classifier along with the feature bank. The cosine similarity function is used to measure the similarity between features. Top $k$ nearest neighbors are used to predict the class with weighted voting. The weight of each neighbor is

$$\alpha_j = exp\left(\frac{\mathbf{f}_i \cdot \mathbf{f}}{\tau}\right)$$

where temperature $\tau$ regulates how much each weight will be affected by similarity change. Feature $\mathbf{f}$ is the one we want to classify, and feature $\mathbf{f}_i$ is one of the $k$ most similar features from the training set used for calculating weights.

Temperature $\tau$ and the number of nearest neighbors $k$ are hyperparameters in the algorithm, denoted with knn_t and knn_k. Their values are same for all experiments and are set to $\tau = 0.1$ and $k = 200$. Similar values are chosen in [19]. Models that achieved the best accuracy on the validation set are also evaluated with a linear classifier by freezing the backbone and training a linear classifier for ten epochs with SGD. The performance of the backbone trained with BYOL was tested with different learning rates. Since they all achieved comparable results, shown in table 4.1, for all other evaluations $lr = 0.3$ was chosen.

Linear classifier was also trained with up to fifty epochs. Difference in accuracy after

training for fifty and ten epochs was only 0.5%. Because of this all linear classifiers were trained for ten epochs.

**Table 4.1:** BYOL model accuracy with different learning rates during linear readout under linear evaluation.

| Learning rate | 0.05 | 0.1 | 0.15 | 0.2 | 0.25 | 0.3 | 0.35 |
|---|---|---|---|---|---|---|---|
| Top-1 accuracy | 85.3 | 85.48 | 85.48 | 85.51 | 85.52 | 85.55 | 85.7 |

```python
def knn_predict(feature, feature_bank, feature_labels, classes: int, knn_k: int, knn_t: float):
    """Helper method to run kNN predictions on features based on a feature bank
    Args:
        feature: Tensor of shape [N, D] consisting of N D-dimensional features
        feature_bank: Tensor of a database of features used for kNN
        feature_labels: Labels for the features in our feature_bank
        classes: Number of classes (e.g. 10 for CIFAR-10)
        knn_k: Number of k neighbors used for kNN
        knn_t: Temperature for contributing weight of neighbor
    """
    # compute cos similarity between each feature vector and feature bank ---> [B, N]
    sim_matrix = torch.mm(feature, feature_bank)
    # [B, K]
    sim_weight, sim_indices = sim_matrix.topk(k=knn_k, dim=-1)
    # [B, K]
    sim_labels = torch.gather(feature_labels.expand(feature.size(0), -1), dim=-1, index=sim_indices)
    # we do a reweighting of the similarities
    sim_weight = (sim_weight / knn_t).exp()
    # counts for each class
    one_hot_label = torch.zeros(feature.size(0) * knn_k, classes, device=sim_labels.device)
    # [B*K, C]
    one_hot_label = one_hot_label.scatter(dim=-1, index=sim_labels.view(-1, 1), value=1.0)
    # weighted score ---> [B, C]
    pred_scores = torch.sum(one_hot_label.view(feature.size(0), -1, classes) * sim_weight.unsqueeze(dim=-1), dim=1)
    pred_labels = pred_scores.argsort(dim=-1, descending=True)
    return pred_labels
```

**Figure 4.4:** Weighted kNN classifier implementation.

## 4.6. Model comparison

In this section, details of each implemented model will be explained. At the end of the section performance of implemented models will be compared. If not stated otherwise, implementation choices that are common for all models were stated in previous sections of this chapter.

### 4.6.1. MoCo

MoCo [16] is the only contrastive method that was implemented. Even though SimCLR [15] reports better results, with MoCO we can reduce memory footprint by using

a smaller batch size and maintaining a momentum encoder. Implementation is modeled after Barlow twins implementation [4]. The model was loaded using Lightly framework [2].[3] The loss function used in experiments is temperature normalized InfoNCE loss. Loss is given with equation 2.1 and scoring function with equation 2.2.
Encoder size is set to 4096, momentum to 0.99, and temperature to 0.1. Hyperparameter values are taken from the original paper [16].
Some implementation choices, like adding a projection head and augmentation pipeline, were motivated by SimCLR paper [15]. These modifications drastically improve the performance of the model, as can be seen in figure 4.2.

### 4.6.2. BYOL

BYOL [18] implementation is modeled after Barlow twins implementation [4]. The model was loaded using Lightly framework [2].[4] Loss function used in experiements is negative cosine similarity loss 2.4.2.
The momentum for the update of the encoder is 0.9 and the base learning rate is 0.06.

### 4.6.3. SimSiam

SimSiam [21] implementation is modeled after SimSiam implementation on CIFAR-10 [5]. Model was loaded using Lightly framework [2].[5] Loss function used in experiment is negative cosine similarity loss 2.4.2.
The base learning rate is 0.04.

### 4.6.4. Barlow twins

Barlow twins [22] implementation is taken from [4]. The model was loaded using Lightly framework [2].[6] Loss function used in experiment is given with equation 2.5. Regularization hyperparameter $\lambda$ is 0.005 as suggested in original paper [22].

[3]Complete implementation of MoCo model can be found on https://github.com/lightly-ai/lightly/blob/master/lightly/models/moco.py

[4]Complete implementation of BYOL model can be found on https://github.com/lightly-ai/lightly/blob/master/lightly/models/byol.py

[5]Complete implementation of SimSiam model can be found on https://github.com/lightly-ai/lightly/blob/master/lightly/models/simsiam.py

[6]Complete implementation of Barlow twins model can be found on https://github.com/lightly-ai/lightly/blob/master/lightly/models/barlowtwins.py

### 4.6.5. DirectPred

DirectPred [23] implementation is modeled after SimSiam implementation on CIFAR-10 [5]. Model was written using BYOL model from Lightly framework [2] and implementation [6] supplied with original paper [23] as references.
The projection head is the same as in the BYOL model. Due to restriction mentioned in section 3.2 prediction head is one layer MLP. The main difference between BYOL and DirectPred is how the prediction head is updated. Implementation of analytical update of prediction head, using eigendecomposition, can be seen in figure 4.5

```
def calculate_predictor(self, f0, f1):
    self.update_F(f0, f1)
    f_eigenvals, f_eig_vec = torch.linalg.eigh(self.F)
    f_eigenvals = torch.real(f_eigenvals)
    f_eig_vec = torch.real(f_eig_vec)

    max_eigenval = torch.max(f_eigenvals)
    f_eigenvals = torch.divide(f_eigenvals, max_eigenval)
    f_eigenvals = torch.clip(f_eigenvals, min=0, max=max_eigenval)

    p = torch.pow(f_eigenvals, 0.5) + self.eps
    p_diag = torch.diag(p)

    predictor = torch.matmul(f_eig_vec, p_diag)
    predictor = torch.matmul(predictor, torch.transpose(f_eig_vec, 0, 1))

    return predictor

def update_F(self, z0, z1):
    corr = self.calculate_corr_without_centering(z0, z1)
    if (self.F == None):
        self.F = corr
    else:
        self.F = self.m_f * self.F + (1 - self.m_f) * corr
```

**Figure 4.5:** Analytical predictor calculation implementation.

It is worth noticing that $\epsilon$ hyperparameter is not multiplied with the most significant eigenvalue because they are normalized before making it equal to 1.
The base learning rate is 0.06. The momentum $m$ for the update of the encoder is 0.9. The momentum for updating the correlation matrix $\hat{F}$ is 0.99. Regularization parameter $\epsilon$ is 0.1.

### 4.6.6. Performance on CIFAR-10 dataset

Implemented models mentioned in previous subsections were compared on CIFAR-10 dataset [7]. On the first graph 4.6 performance on validation set is shown.
We can see that non-contrastive methods perform similarly when trained for 400 epochs, with the highest accuracy of around 90%. SimSiam might be underperforming during the first half of the training because of the wrong choice for the learning rate hyper-parameter. Learning seems quite unstable, which could be caused by the high base learning rate. Another possibility is that the lack of stability in the early epochs is caused by setting the target projector to be equal to the online projector instead of using EMA for setting the target projector.
Non-contrastive method MoCo also performs well with the highest accuracy of around 85%, and the Barlow twins method has the worst highest accuracy of around 80%.

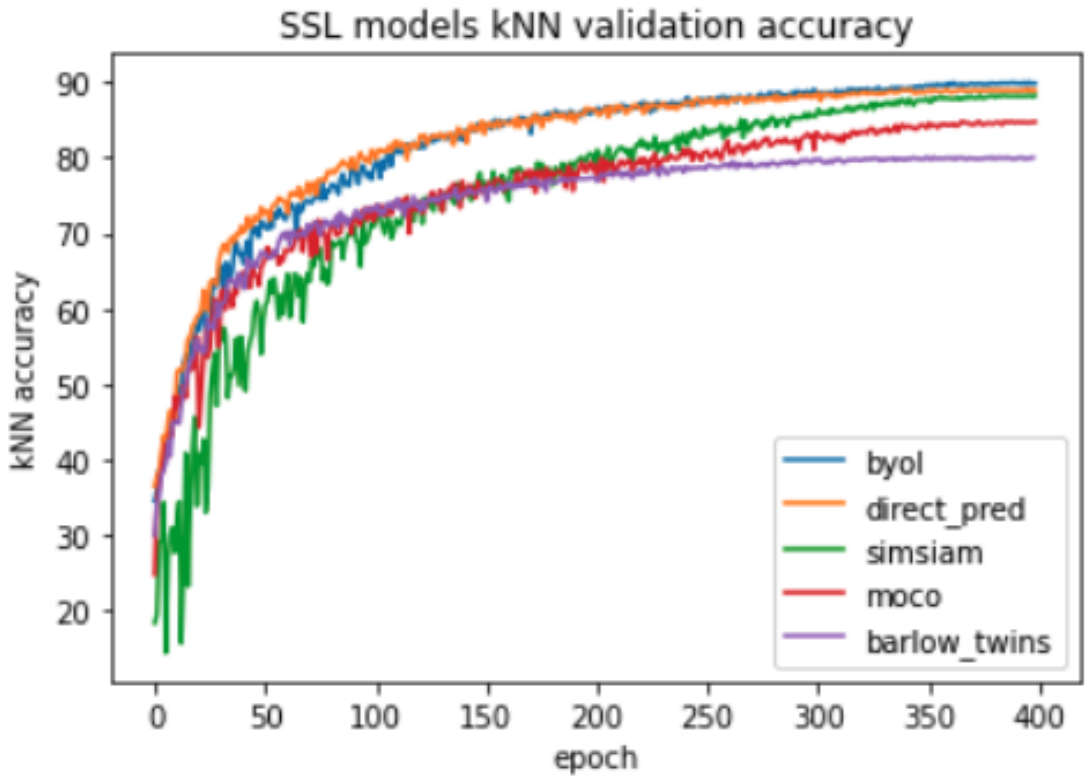


**Figure 4.6:** Comparison of implemented models on validation set.

We can see that the best performing models on validation set, also have the best results on test set shown in table 4.2. Even though the results are in same order, there is a significant drop in top-1 accuracy, of around 6%, for non-contrastive models [18;

21; 23]. This gap might be decreased with:

- Different optimizer during linear evaluation. SimSiam paper [21] reports a slight increase in accuracy when using LARS optimizer [25] and bigger batch size.
- Different augmentations during linear evaluation. In these experiments, only random cropping was used.
- Training for more epochs with smaller learning rate.

| Method | Top-1 |
|---|---|
| MoCo [16] | 82.37 |
| BYOL [18] | **85.55** |
| SimSiam [21] | 82.88 |
| Barlow twins [22] | 81.97 |
| DirectPred [23] | 84.43 |
| ResNet-18 [24] (Supervised) | 92.34 |
| ViT-H/14 (Supervised SOTA) [26] | **99.5** |

**Table 4.2:** Models top-1 accuracy under linear evaluation.

Models were evaluated using the linear evaluation protocol described in section 4.5. Results along with the results of the ResNet-18 model and the SOTA supervised method are shown in table 4.2. The ResNet-18 model when trained for 200 epochs performs better than SSL methods with the same backbone. This gap is not that wide, considering that the backbone in SSL methods is trained on completely unannotated data. However, it can be reduced if we use fine-tuning. Fine-tuning experiment was done on the best-performing method, BYOL, and was run for 70 epochs with $lr = 0.005$. For fine-tuning, we used both 10% and 100% of training data. Results are shown in table 4.3.

Even though the gap between implemented SSL methods and SOTA supervised method is pretty wide, it is worth noting that the goal of this thesis was not aimed at achieving high accuracy on the test set, so the hyperparameters were not tuned, and the backbone model was kept as small as possible. Better results could be achieved by using deeper backbone, e.g. ResNet-50, and by fine-tuning the model on part of the dataset. In addition, the listed supervised method has 60 times more parameters than the best-performing method, BYOL.

| Method | Top-1 | |
|---|---|---|
| | 10% | 100% |
| BYOL [18] | 82.8 | 91.66 |
| ResNet-18 [24] (Supervised) | 65.7 | 92.34 |

**Table 4.3:** Models top-1 accuracy after fine-tuning.

## 4.7. Non-contrastive methods analysis

In this section, we will try to see which observations made on a simple linear model 3.1 translate to real-life examples.

### 4.7.1. SimSiam without prediction head

It was shown in section 3.1 that a simple linear model without a prediction head would cause weights to collapse. Because of this collapse, representations should become constant, and loss should become minimal, making it impossible for the model to learn anything useful.

Results of SimSiam [21] model trained for 100 epochs without a prediction head are shown in figure 4.8. It can be seen that loss is immediately minimized (as expected), the standard deviation of representations[7] also drops to 0 (which means that they become constants) and accuracy on the validation set stagnates.

This experiment supports observations made from analyzing a simple model.

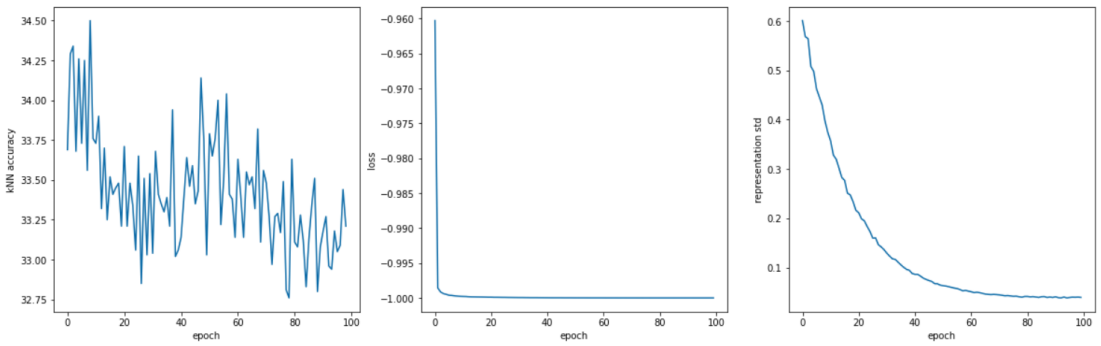


**Figure 4.7:** SimSiam model without prediction head trained for 100 epochs.

[7] Standard deviation is calculated on the last batch in an epoch.

### 4.7.2. BYOL eigenspace alignment

It was shown in section 3.1 that relation in equation 3.2 collapse to 0 during training. It can be shown that matrices that commute are simultaneously diagonalizable, which means that their eigenspaces align.

We could prove this collapse only by adding three assumptions to our model. One of which was that predictor becomes symmetric and PSD during training. This assumption will also be tested in this experiment.[8]

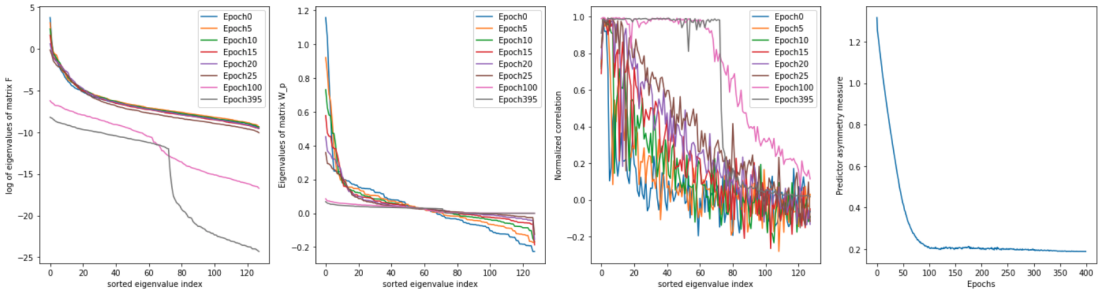


**Figure 4.8:** Properties of predictor during training of a BYOL [18] model. The first graph shows sorted eigenvalues of correlation matrix F. Second graph shows sorted eigenvalues of the predictor. The third graph shows normalized correlation between $W_p * U$ and $U$, where $U$ are eigenvectors of matrix F. 1 means that values are correlated and that the eigenspace is aligned, and 0 means that the values are not correlated and that the eigenspace is not aligned. The fourth graph represents the asymmetry of the predictor. The lower the value, the more symmetric the matrix is.

From the fourth graph, we can see that the predictor becomes more symmetric with time which validates the third assumption in 3.1. Asymmetry of a matrix was measured with the following equation:

$$|W - W^T|/|W| \tag{4.1}$$

The first two graphs show that the leading eigenvalues decrease but not nearly as much as the smaller eigenvalues. We can see that the first 70 eigenvalues carry all the information and that eigenvectors for those eigenvalues are aligned. Since 0 eigenvalues represent trivial solutions, it makes sense that the eigenvectors of those eigenvalues are entirely uncorrelated.

On the third graph, we can also observe that the eigenspace becomes more aligned during training, which was our hypothesis.

[8]This experiment was originally done in [23].

### 4.7.3. BYOL performance with different prediction head updates

It was stated that the larger learning rate of predictor helps avoid representation collapse and, by doing so, increases performance but that too high a learning rate might hurt it because eigenvalues of correlation matrix F will not increase with an increase of eigenvalues of prediction head. Results of this experiment can be seen on graph 4.9. This experiment does not confirm the hypothesis. It can be seen that models

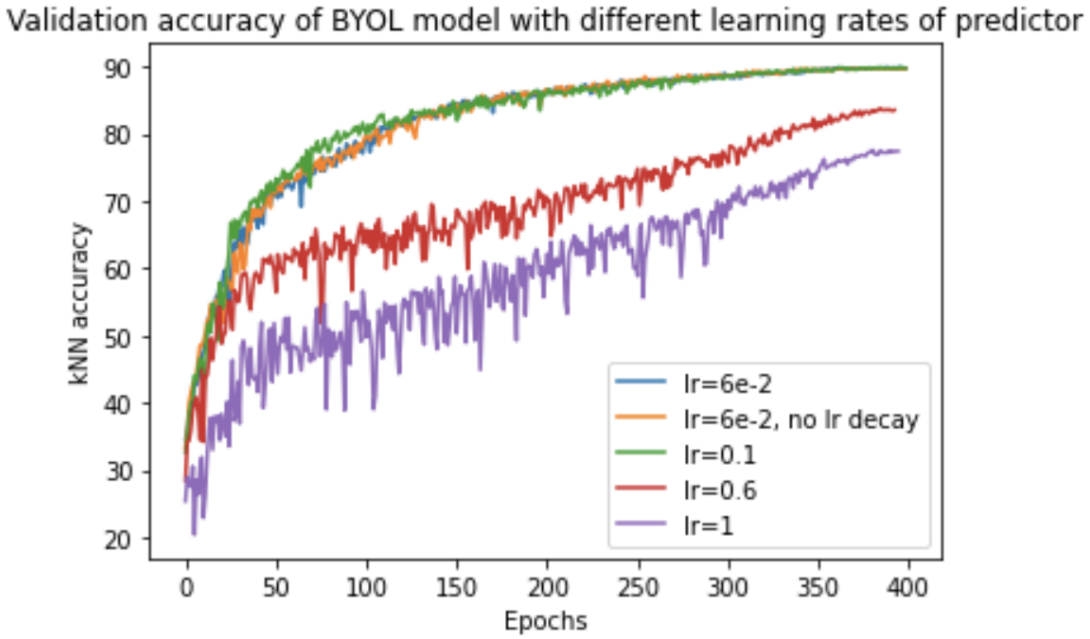


**Figure 4.9:** BYOL model trained with different learning rates of prediction head.

with a learning rate up to 0.1 have the same kNN accuracy graph. Models with larger learning also learn useful representations, but the maximum accuracy is way smaller. This seems to be due to unstable learning because graphs become way wavier with an increase in learning rate and not the problems mentioned in 3.1.

### 4.7.4. BYOL performance with large decay

It was stated that too large weight decay inevitably leads to representation collapse because there is only one stable point, which is trivial. In this experiment, we compare the BYOL model with suggested weight decay and with too large weight decay. Results can be seen in figure 4.10. Even though accuracy is decreased, it still learns useful representations. A decrease in accuracy can be caused by too large regularization, followed by larger training loss. Because of this experiment does not confirm the hypothesis.

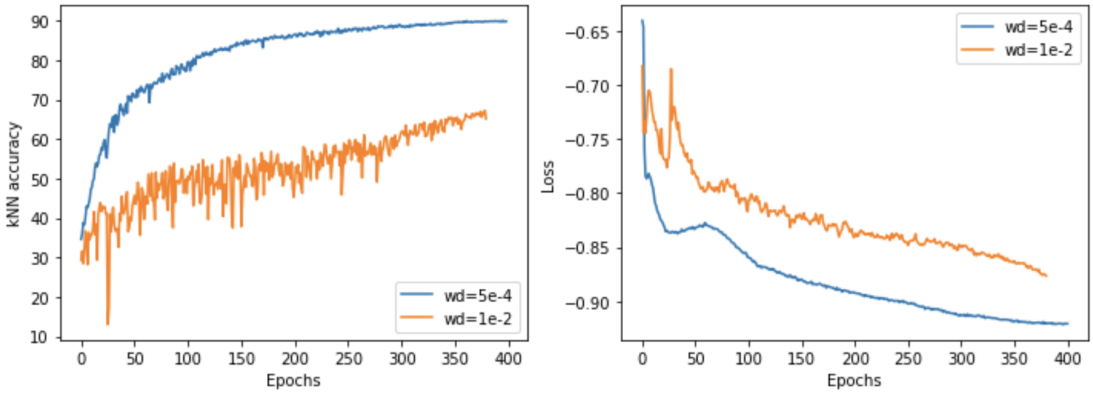


**Figure 4.10:** BYOL model trained suggested and large weight decay.

# 5. Conclusion

This thesis was focused on analyzing different self-supervised methods for visual recognition. In chapter 2, it was explained how different design choices of non-contrastive methods significantly improved upon SOTA contrastive methods. This was backed up in the experiments section showing how well different methods perform on CIFAR-10 dataset. Table 4.2 shows that the non-contrastive models are more memory efficient and robust to augmentations choices and that they also learn better representations. Furthermore, it was shown how non-contrastive methods avoid representation collapse and how different components affect performance on the validation set throughout the training process. Results of eigenspace alignment and predictor symmetrization during non-contrastive method training, shown in section 4.7.2 are significant for proving why non-contrastive methods avoid representation collapse and are the basis for novel non-contrastive method DirectPred [23]. It was shown that this method could learn better representation than other contrastive methods when trained for a small number of epochs, like 100. It was also shown that it has comparable performance to SOTA non-contrastive methods when trained for a longer time.
Even though this thesis showed that self-supervised methods, especially non-contrastive methods, can learn useful representations and achieve good results on the CIFAR-10 dataset, they still performed much worse than SOTA supervised methods. This gap was due to the time and resources constraint of this thesis. For future work, more competitive results can be achieved using a deeper backbone network, longer training time, and fine-tuning the model.

# BIBLIOGRAPHY


[1] Google Colab, https://colab.research.google.com/. 12.8.2022.

[2] Lightly, https://www.lightly.ai/ 13.8.2022.

[3] IgorSusmelj, barlowtwins, https://github.com/IgorSusmelj/barlowtwins/blob/main/utils.py

[4] IgorSusmelj, barlowtwins, https://github.com/IgorSusmelj/barlowtwins/blob/main/main.py

[5] IgorSusmelj, simsiam-cifar10, https://github.com/IgorSusmelj/simsiam-cifar10/blob/main/main.py

[6] facebookresearch, luckmatters, https://github.com/facebookresearch/luckmatters/tree/main/ssl

[7] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images, 2009.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep Learning. MIT Press, 2016. http://www.deeplearningbook.org.

[9] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Sackinger, and Roopak Shah. Signature verification using a "Siamese" time delay neural network. In NeurIPS, 1994.

[10] Spyros Gidaris, Praveer Singh, and Nikos Komodakis, Unsupervised Representation Learning by Predicting Image Rotations, ICLR, 2018.

[11] Gustav Larsson, Michael Maire, and Gregory Shakhnarovich. Colorization as a Proxy Task for Visual Understanding, 2017.

[12] Mehdi Noroozi and Paolo Favaro. Unsupervised learning of visual representations by solving jigsaw puzzles. In European Conference on Computer Vision, pp. 69–84. Springer, 2016.

[13] Linus Ericsson, Henry Gouk, Chen Change Loy, and Timothy M. Hospedales. Self-Supervised Representation Learning: Introduction, Advances and Challenges, 2021.

[14] Alexey Dosovitskiy, Philipp Fischer, Jost Tobias Springenberg, Martin Riedmiller, and Thomas Brox. Discriminative Unsupervised Feature Learning with Exemplar Convolutional Neural Networks, in NeurIPS, 2014.

[15] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. arXiv:2002.05709, 2020.

[16] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick.Momentum contrast for unsupervised visual representation learning. arXiv:1911.05722, 2019.

[17] Xingyi Yang, Xuehai He, Yuxiao Liang, and Yue Yang. Transfer Learning or Self-supervised Learning? A Tale of Two Pretraining Paradigms, 2020.

[18] Jean-Bastien Grill, Florian Strub, Florent Altche, Corentin Tallec, Pierre H. Richemond, Elena Buchatskaya, Carl Doersch, Bernardo Avila Pires, Zhaohan Daniel Guo, Mohammad Gheshlaghi Azar, Bilal Piot, Koray Kavukcuoglu, Remi Munos, and Michal Valko. Bootstrap your own latent: A new approach to self-supervised learning. arXiv:2006.07733v1, 2020.

[19] Zhirong Wu, Yuanjun Xiong, Stella Yu, and Dahua Lin. Unsupervised feature learning via non-parametric instance discrimination. In CVPR, 2018.

[20] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding. arXiv:1807.03748, 2018.

[21] Xinlei Chen, Kaiming He. Exploring simple siamese representation learning. arXiv:2011.10566, 2020.

[22] Jure Zbontar, Li Jing, Ishan Misra, Yann LeCun, and Stéphane Deny. Barlow twins: Self-supervised learning via redundancy reduction. arxiv:2103.03230, 2021.

[23] Yuandong Tian, Xinlei Chen, and Surya Ganguli. Understanding Self-Supervised Learning Dynamics without Contrastive Pairs, arxiv:2102.06810 2021.

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition, arXiv:1512.03385, 2015.

[25] Yang You, Igor Gitman, and Boris Ginsburh. Large Batch Trainign of Convolutional Networks, arXiv:1708.03888, 2017.

[26] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby, An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale, arXiv:2010.11929, 2021.

**Samonadzirane metode za vizualno raspoznavanje**

## Sažetak

U zadnje vrijeme, samonadzirane metode postižu rezultate koji su sumjerljivi s nadziranim metodama. Ove metode znaju davati i bolje rezultate od nadziranih u području vizualnog raspoznavanja jer se skaliraju s neoznačenim podacima, kojih je mnogo više. Ove metode su možda i bitnije u zadacima kod kojih je označavanje podataka jako skupo. U ovom radu su predstavljene neke od trenutno popularnih samonadziranih metoda. Jedna od tih metoda je implementirana i trenirana na dostupnim skupovima podataka. Dobiveni rezultati su objašenje i uspoređeni s drugim popularnim metodama.

**Ključne riječi:** Samonadzirano učenje, Kontrastivno učenje, Vizualno raspoznavanje, Duboko učenje

**Self-supervised methods for visual recognition**

## Abstract

Self-supervised methods have been producing results that are comparable to supervised methods. These methods even surpass many supervised methods in visual representation learning tasks because they scale with unlabeled data. These methods are also very important in tasks for which labeling data is very expensive, like dense prediction. In this thesis some more popular contrastive self-supervised methods are described. One of those methods was implemented and trained. The results obtained are explained and compared to other SOTA methods.

**Keywords:** Self-supervised learning, Contrastive learning, Non-contrative learning, Visual recognition, Deep learning