

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND
COMPUTING

MASTER THESIS no. 2470

Deep Learning Model of Nanopore Sequencing Pore

Rafael Josip Penić

Zagreb, June 2021.

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

Zahvala

CONTENTS

1. Introduction	1
1.1. Bioinformatics	1
1.2. Nanopore sequencing	1
1.3. Self-supervised learning	1
2. Data	3
3. Pore model	5
3.1. Model implementation	6
4. Contrastive Predictive Coding	7
4.1. Implementation details	8
4.2. Results	9
5. Representation learning algorithms	11
5.1. Signal augmentations	12
5.1.1. Unsuccessful attempts	12
5.1.2. Noise	14
5.1.3. Drop-point augmentation	14
5.2. SimCLR	15
5.2.1. Implementation	16
5.2.2. Results	17
5.3. MoCo	17
5.3.1. Implementation	19
5.3.2. Results	19
5.4. SimSiam	22
5.4.1. Implementation	23
5.4.2. Results	24
5.5. BYOL	26

5.5.1. Implementation	26
5.5.2. Results	27
5.6. DINO	29
5.6.1. Implementation	30
5.6.2. Results	31
5.7. Barlow Twins	32
5.7.1. Implementation	34
5.7.2. Results	35
6. Conclusion	37
Bibliography	38

1. Introduction

1.1. Bioinformatics

Bioinformatics is a field of science where computer resources are used for analysis of genome, proteins and other biological compounds.

1.2. Nanopore sequencing

A lot of problems that are dealt with in bioinformatics involve analysis of DNA or RNA sequences. Question arises: how exactly do we "read" those sequences from the nature? This is where the nanopore sequencing comes in.

Sequences are passed through the nanopore which measures ionic current that changes depending on the nucleotides that are in the pore. There are usually 5 or 6 nucleotides in the pore at the same time due to the pores' size. Information about the ionic current signal is then usually used for basecalling (converting a raw current signal to a specific sequence of nucleotides).

1.3. Self-supervised learning

Today we have an enormous amount of unlabeled data, but unfortunately most of the machine learning algorithms are restricted to using only labeled data, which is often very sparse and expensive to label. One might ask is there any way we could use unlabeled data to improve performance and accuracy of our models. That is exactly what self-supervised learning does.

Self-supervised learning, often referred to as the dark matter of intelligence [17], is a method of pretraining models with unlabeled data. For example, models are often tasked to predict future from the present or vice versa. Pretrained parameters of the model are then used as the "starting point" for the downstream task. This approach

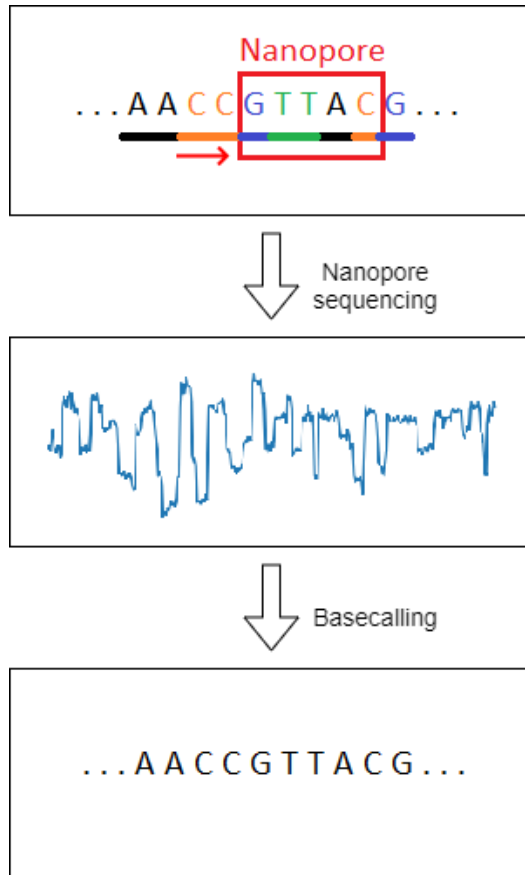


Figure 1.1: Example of basecalling "workflow".

achieved fantastic results in natural language processing and it is becoming a very important part of computer vision as well.

2. Data

For model training and validation we used the same data as the popular basecaller Bonito. Data has been divided into two parts. One part of data was used for model self-supervised pretraining and other for the actual training on the downstream task (basecalling). Pretraining data is about five times larger than the downstream task data.

Information about signals is stored in "hdf5" files. This file format is a hierarchical data format and it contains all relevant informations about nanopore signal. It is important to note that signals are stored in a discrete form and if we want to get actual values of ionic current (in picoamperes), we have to transform them into continuous signals:

$$\begin{aligned} scale &= range/digitisation \\ continuous &= scale * (discrete + offset) \end{aligned} \quad (2.1)$$

Signals are normalised with "med-mad" normalisation:

$$signal = (signal - median(signal))/mean_absolute_deviation(signal) \quad (2.2)$$

After normalisation, signals are chunked into smaller segments to prevent memory issues.

Nanopore signals are approximately rectangular.

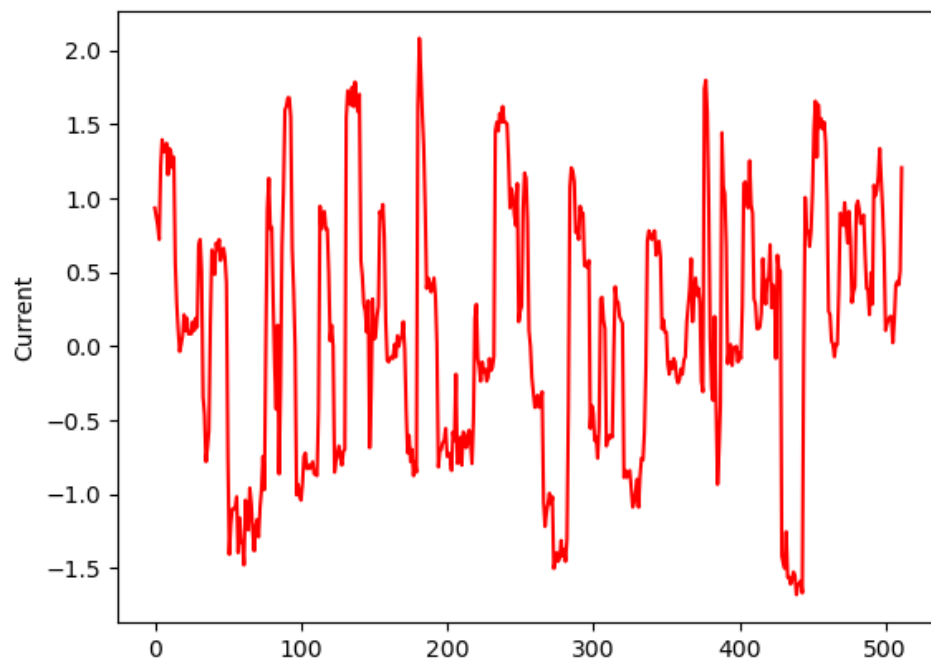


Figure 2.1: Example of a single normalised nanopore signal chunk.

3. Pore model

~~Before we explain which methods are used and how, we have to define what our model consists of.~~ Pore model is composed of two main components: feature encoder and transformer. This model structure is inspired by a model used in Wav2vec 2.0 [2].

Feature encoder is built of multiple convolutional blocks. Each block contains convolutional layer, batch normalisation and Gaussian Error Linear Unit (GELU) activation function.

Outputs of feature encoder are passed to transformer [16]. Transformer inputs are encoded with positional embedding just like in popular natural language processing model BERT [9].

Pore model is usually followed by another smaller model depending on a task it is solving. For example, during basecalling few linear projection layers could be "attached" to it.

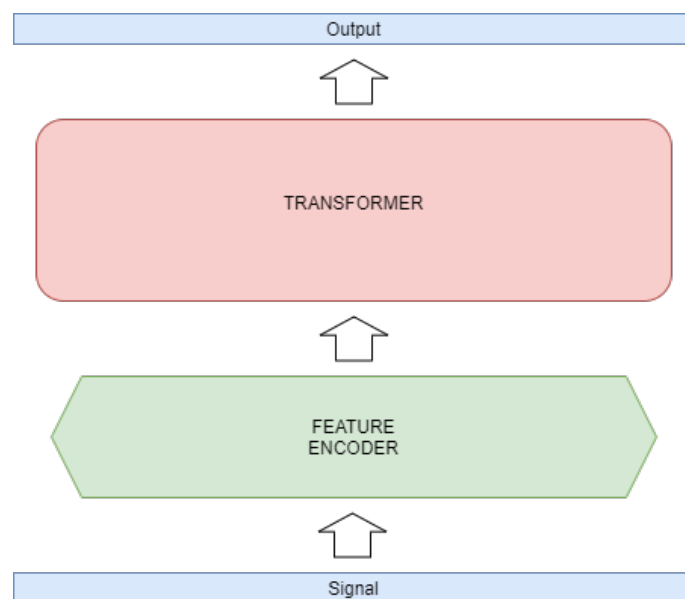


Figure 3.1: Pore Model.

3.1. Model implementation

Feature encoder consists of 4 blocks. First block outputs 64, second 128, third 256 and fourth 512 channels. Each block uses kernel size of 3 and all blocks use stride 2 besides the last one which uses stride 1.

Transformer uses 8 attention heads and it consists of 10 sub-layers. Dimension of the feed-forward network inside of transformer is set to 2048.

4. Contrastive Predictive Coding

Main idea of contrastive predictive coding (CPC) [15] is simple: predict "future" parts of the signal from the "current" context. In other words, we use output of transformer (context) to predict outputs of feature encoder (latent representations) in the future time steps. We don't just use any context vector, but only the latest one since that one is closest to the "future" we are predicting.

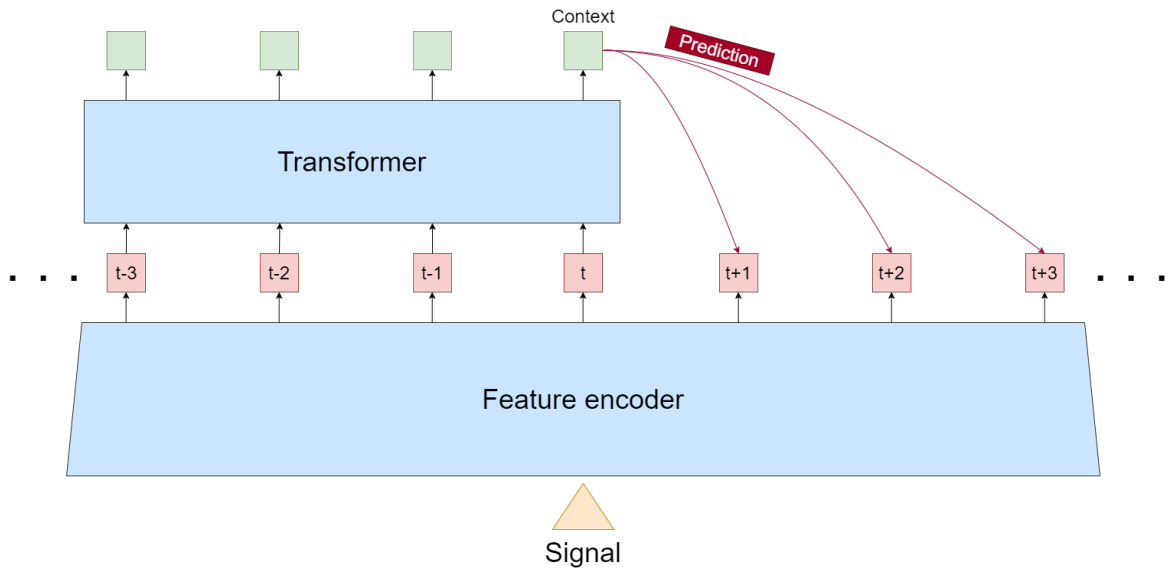


Figure 4.1: Contrastive predictive coding. "Latest" context vector is used to predict "future" latent representations.

Under the hood, CPC actually tries to maximise mutual information between context (c) and future latent representation (z):

$$I(z, c) = \sum_{z, c} p(z, c) \log\left(\frac{p(z|c)}{p(z)}\right) \quad (4.1)$$

Instead of directly modeling the mutual information, CPC models density ratio $f_k(z_{t+k}, c_t)$ which is proportional to $p(z_{t+k}|c_t)/p(x_{t+k})$. Density ratio is defined as:

$$f_k(z_{t+k}, c_t) = \exp(z_{t+k}^T W_k c_t) \quad (4.2)$$

where z_{t+k} is latent representation of the signal in k -th future time step and c_t represents "present" ($k = 0$) context. It is noteworthy to mention that prediction of each future time step uses separate weights (W_k) for linear projection of context vector. Finally, CPC uses density ratio in InfoNCE loss. Let us assume that size of a batch is N :

$$L_k = - \sum_{i=1}^{i=N} \log \frac{f_k(z_{t+k}^i, c_t^i)}{\sum_{j=1}^N f_k(z_{t+k}^j, c_t^j)} \quad (4.3)$$

$$L = \sum_k L_k \quad (4.4)$$

It is easy to see that InfoNCE loss is in fact categorical cross-entropy loss where we want to choose correct (positive) latent representation from a certain number of options (negatives). Negatives are sampled from other signals in the batch. For example, if we are predicting latent representation in k -th time step, negatives will be acquired by taking k -th output of feature encoder from all other batch samples.

Convenient thing about CPC is that we can easily adjust difficulty of the self-supervised task by setting how many future time steps model is tasked to predict. The greater this number is, the more difficult task is.

4.1. Implementation details

Implementation of CPC algorithm has been very straightforward. Only somewhat problematic thing was a fact that we are using transformer as the context encoder instead of one-directional RNN (recurrent neural network) which is usually used with CPC. This is problematic because transformer is bidirectional and if nothing is done, context could "see" the future. It is very important to prevent this from happening otherwise CPC pretraining task would be too easy and useless. We averted this by masking transformer inputs of "future" time steps.

Algorithm 1 CPC algorithm [12]

Input: feature encoder fe , transformer $trnf$, batch size N , prediction window size P , weights W_k ($k \in \{1, \dots, P\}$).

for minibatch $\{s_1, \dots, s_N\}$ **do**

 Randomly choose "current" (present) timestep t

for $i \in \{1, \dots, N\}$ **do**

$z_i = fe(s_i)$

$c_i = trnf(z_i)$ {transformer inputs that come after timestep t are masked}

end for

$\{z, c \Rightarrow T \times N \times F\}$

$L = 0.0$

for $j \in 1, \dots, P$ **do**

$density_ratio = \exp(z[t+j]W_k c[t]^T)$ { $N \times N$ }

 {id(D) \Rightarrow identity matrix with dimensions $D \times D$ }

 {sum(M) \Rightarrow sums all elements of matrix M }

$L = L - sum(id(N) \log(softmax(\log(density_ratio))))$

end for

 Update transformer, feature encoder and weights W_k to minimize loss L

end for

4.2. Results

When we first tried out CPC on Pore model, it achieved decently high accuracy when predicting future time steps, even for time steps that are relatively far away from the "present".

We quickly realised that feature encoder had a very wide reception field, which might have made the future prediction task too easy so we decided to try out a bit "thinner" feature encoder. We decided to replace feature encoder with a single-layered convolutional network which had much narrower field of view.

This time self-supervised task accuracy was much worse. Model could only decently predict immediate future step.

In both cases pretraining did not cause any boosts in the downstream task (basecalling) performance. We did not manage to find the reason why this happens but it is possible that CPC has trouble predicting future latent representations because of signals "jumpy" (rectangular) nature.

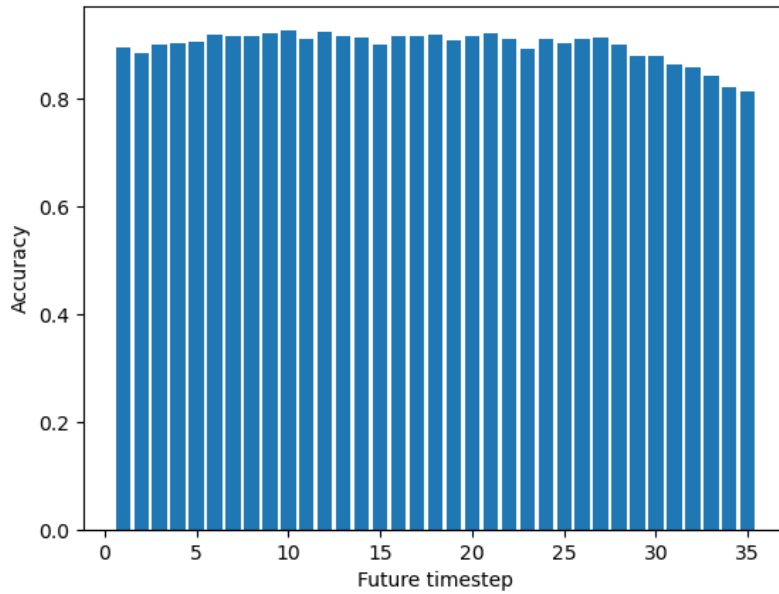


Figure 4.2: CPC prediction accuracy for each future time step with a feature encoder that had a very wide reception field. Model was tasked to predict 35 future time steps. Baseline accuracy is 0.03125.

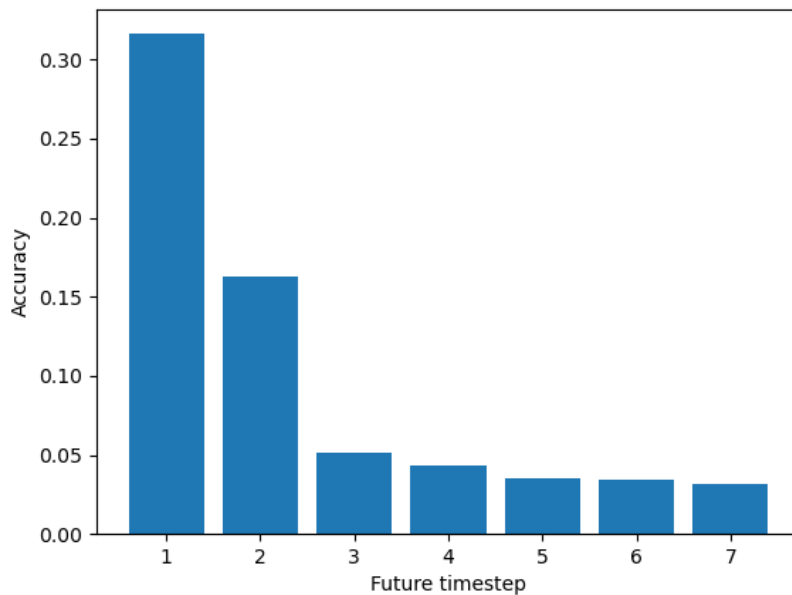


Figure 4.3: CPC prediction accuracy for each future time step. Note that in this case we used "thin" feature encoder that consisted of only one-layered convolutional neural network. Model was tasked to predict 7 future time steps. Baseline accuracy is 0.03125.

5. Representation learning algorithms

Representation learning algorithms are family of self-supervised methods which take two or more augmentations of a single instance (in our case signal chunks) and then try to learn a good representation of the original (non-augmented) signal. This task usually transfers well into downstream tasks and boosts model's accuracy and performance.

For these methods, we could not use Pore model as it is so we had to attach a smaller model to it. This attached model is dropped once the pretraining is done and Pore model is "imported" into basecaller (or whatever else the downstream task is).

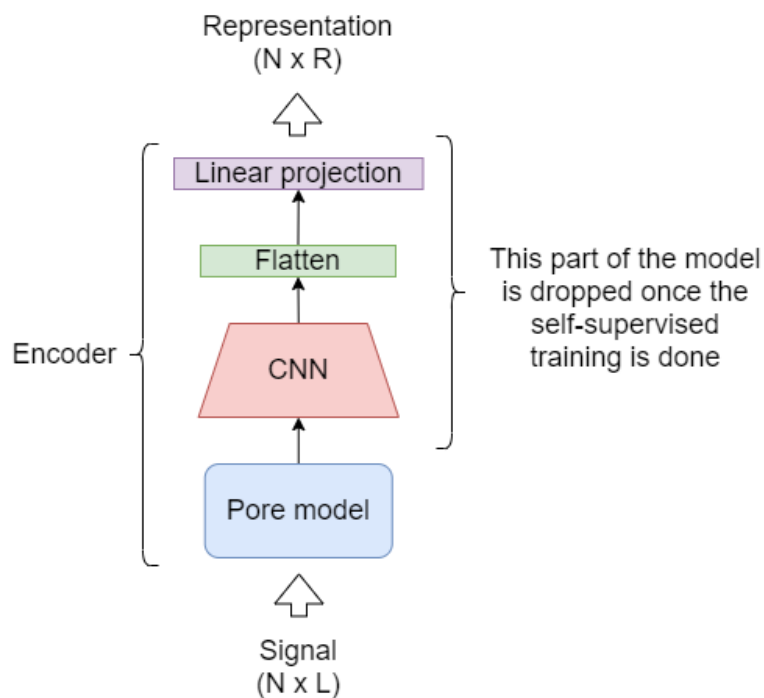


Figure 5.1: Model used with representation learning algorithms. N is size of a batch, L is length of the signal chunk and R is dimensionality of the final signal representation.

Attached model consists of two main parts: convolutional neural network (CNN) and a final linear projection. CNN is built of blocks each of which contains convolutional layer, batch normalisation, rectified linear unit activation function and a max pooling layer. Output of the CNN is flattened and passed to final linear projection block

which is composed of 2 linear projection layers. First linear projection is followed by batch normalisation and rectified linear unit activation function.

In the end, we decided that CNN will consist of only one block where convolutional network uses kernel size 7 and stride 1. Max pooling uses kernel size 4 and stride 4. Linear projection block differed from algorithm to algorithm since each method has its preferences when it comes to representation dimensionality.

5.1. Signal augmentations

Most important part of representation learning methods are augmentations. This is also their biggest flaw because it is not always easy to come up with good augmentations that will result in high-quality representations.

5.1.1. Unsuccessful attempts

Before we came up with good combination of augmentations, we tried a lot of augmentations that proved to give a pretty bad representation of signal which did not cause any improvements on downstream task. At first, we quite underestimated the importance of this segment but we quickly realised that we can't just use any augmentation.

One of these (ineffective) augmentations was shuffle augmentation. Main idea is to split signal into certain number of segments and then shuffle their order.

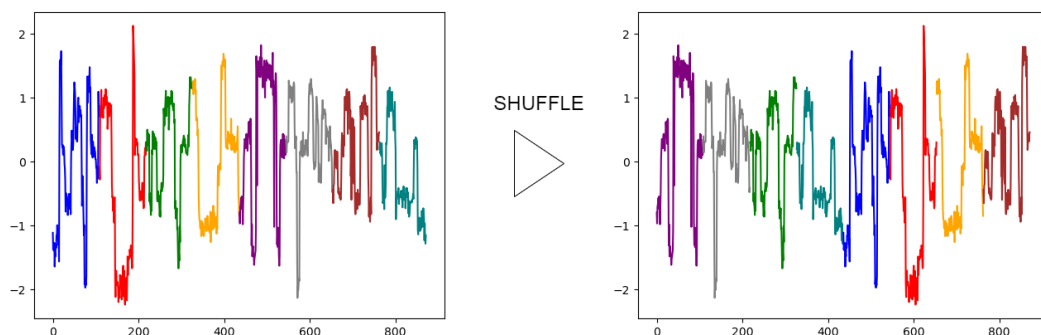


Figure 5.2: Example of shuffle signal augmentation. In this case signal is divided in 8 segments and then those segments are shuffled in random order.

We also tried out adding moving average into the mix to emphasize the "rectangularity" of the signal. We tried using both small and big moving average windows but in both cases, model was unable to learn good signal representations.

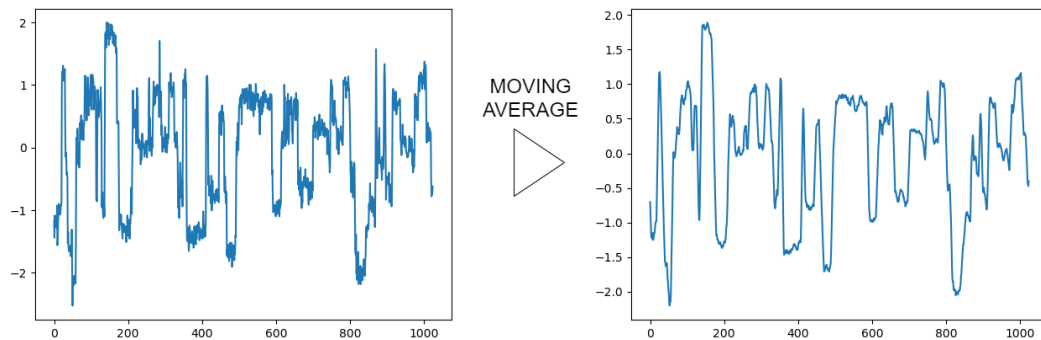


Figure 5.3: Example of moving average augmentation. Width of the moving window is 7.

Random signal crop augmentation is inspired by similar augmentations used in computer vision [4]. Certain percentage of the signal is taken and then widened with linear interpolation to the length of the original chunk.

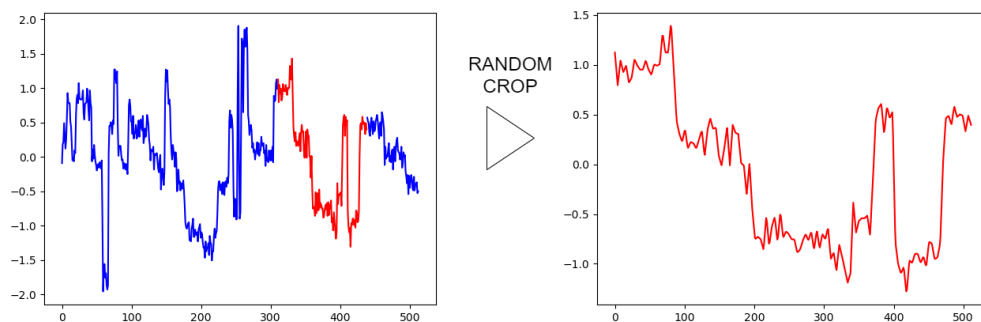


Figure 5.4: Random crop augmentation. Random part of the signal is "cut out" (red color) and then stretched out to the length of the chunk it was taken from.

Signal flip is a rather straightforward augmentation. We used two variants: horizontal and vertical flip. Horizontal version reverses the other order of signal points and vertical one just flips the signal around the x-axis.

As mentioned before, none of the previously described augmentations provided good enough representations. In the end, we decided to only use noise and drop-point augmentation since this combination gave the best signal representation which boosted downstream tasks accuracy.

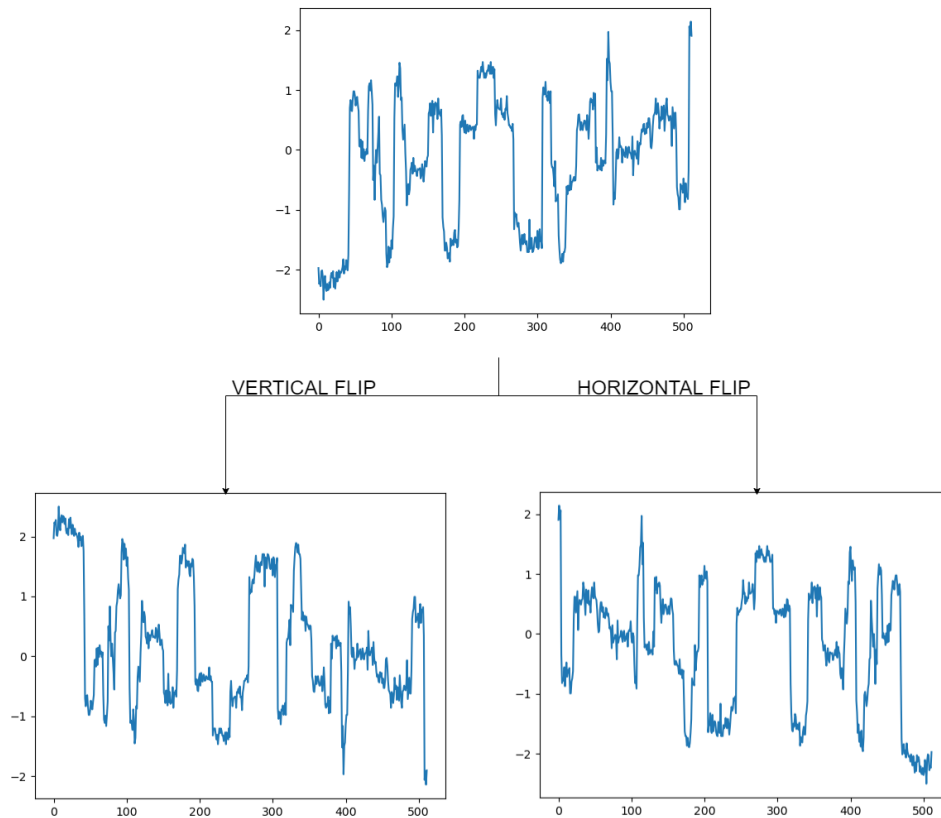


Figure 5.5: Example of horizontal and vertical signal flip.

5.1.2. Noise

Noise augmentation is probably the simplest one we tried. Signal is augmented by adding simple gaussian noise to each signal point:

$$\text{noised_signal} = \text{signal} + \alpha * N(0, 1) \quad (5.1)$$

$N(0, 1)$ generates a vector filled with random numbers from normal distribution (with mean 0 and standard deviation equal to 1). Generated vector has the same dimensionality as the signal that is augmented. We also used noise factor (α) with which one can easily adjust how much the signal will be distorted. During pretraining, we set the noise factor to 0.25.

5.1.3. Drop-point augmentation

Drop-point augmentation is in our case the most important augmentation. Without it, we were not able to achieve notable improvements on the downstream task. Despite its importance, it is not overly complex. ~~All it does is drop~~ each signal point with

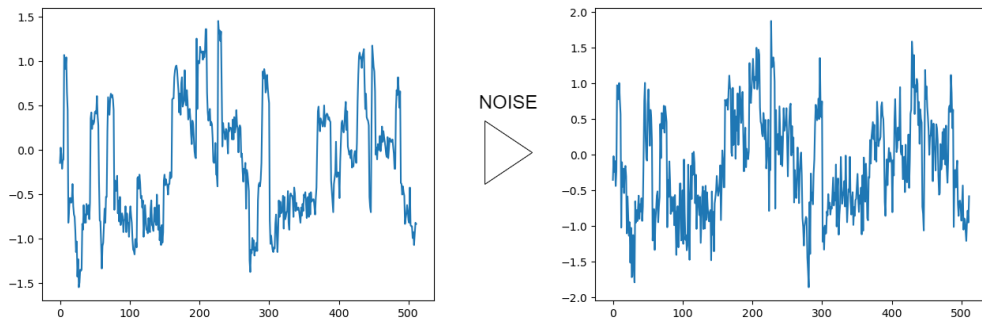


Figure 5.6: Noise augmentation. Noise factor is 0.25.

a certain probability. Implementation-wise this might cause incompatibility with the model and other augmentations since we shorten the signal. We prevent such problems by adding padding to the shortened signal and generating a mask which indicates which part of the signal is padding and which is not. Drop chance is obtained with uniform distribution from a given interval. We used values between 0.25 and 0.38.

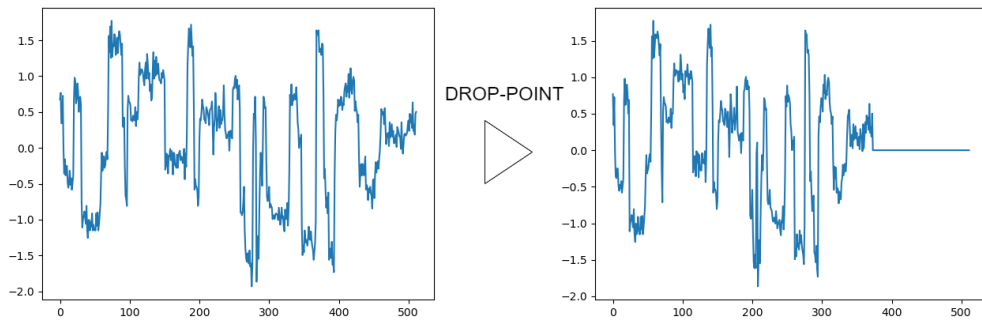


Figure 5.7: Drop-point augmentation.

5.2. SimCLR

SimCLR [6] is the first self-supervised contrastive learning algorithm and most modern self-supervised methods (especially in the computer vision) are in one way or another inspired by this one. Main idea is to create multiple views of the signal and then make the representations of those views as similar as possible. But there is one problem: how do we prevent model from finding the trivial solution where all signals have the same representation? SimCLR prevents this by maximizing similarity with positives (augmentations of the same signal) and minimizing similarity with negatives

(augmentations of other signals).

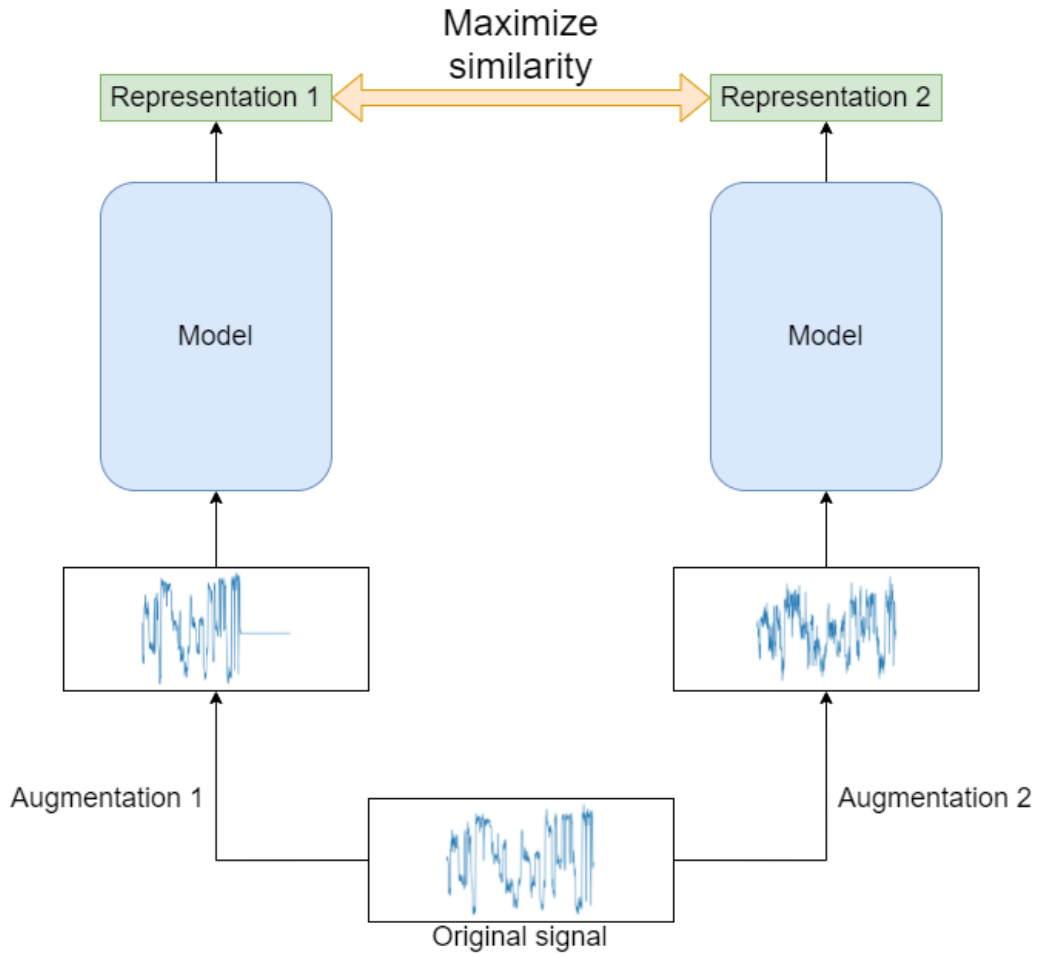


Figure 5.8: SimCLR method.



Loss for a positive pair of signal representations r_i and r_j is defined as follows:

$$l(r_i, r_j) = -\log \frac{\exp(\text{sim}(r_i, r_j)/\tau)}{\sum_{k=1}^{2N} \exp(\text{sim}(r_i, r_k)/\tau)} \quad (5.2)$$

$$L = \sum_{(r_i, r_j) \in \text{positive_pairs}} (l(r_i, r_j) + l(r_j, r_i)) \quad (5.3)$$

N is size of a single batch and two augmentations are generated for each signal chunk in the batch. Parameter τ represents cross entropy temperature.

5.2.1. Implementation

In our SimCLR implementation final representation dimensionality is 256. For training, we used Adam [13] optimizer with learning rate 3e-5. Although SimCLR "prefers" larger batch sizes we were forced to use a relatively small size (32) due to hardware

limitations. Cross entropy temperature τ was set to 0.1. As similarity function we used cosine similarity since that type of similarity was also used in the original paper [6].

Algorithm 2 SimCLR train algorithm

Input: model m , temperature τ , batch size N , set of signal augmentations A .

for minibatch $\{s_1, \dots, s_N\}$ **do**

for s_i in minibatch ($i \in [1, N]$) **do**

 randomly choose two augmentations a_1 and a_2 from set A

$x_i, x_{i+N} = a_1(s_i), a_2(s_i)$

$r_i, r_{i+N} = m(x_i), m(x_{i+N})$

end for

$$L = -\frac{1}{2N} \sum_{j=1}^N \left(\log \frac{\exp(\text{sim}(r_j, r_{j+N})/\tau)}{\sum_{k=1}^{2N} \exp(\text{sim}(r_j, r_k)/\tau)} + \log \frac{\exp(\text{sim}(r_{j+N}, r_j)/\tau)}{\sum_{k=1}^{2N} \exp(\text{sim}(r_{j+N}, r_k)/\tau)} \right)$$

 update model m parameters to minimize loss L

end for

5.2.2. Results

5.3. MoCo

Biggest flaw of the SimCLR method is that it is very dependant on big batch sizes which is not always convenient especially if we don't have sufficient computational power. MoCo [11] negates this limitation by not using negatives from the same batch but from the special representation queue which is updated in each training step.

First part of the algorithm is basically the same as in the SimCLR. Two views of the original signal are generated with augmentations. After that one of the views (query) is passed through the model (encoder) and the other view (key) is passed through the model with same architecture but with different parameters (momentum encoder). Once we get query (q) and key (k_+) representations we use them in contrastive loss:

$$L = -\log \frac{\exp(\text{sim}(q, k_+)/\tau)}{\sum_{k \in \text{queue}} \exp(\text{sim}(q, k)/\tau)} \quad (5.4)$$

Once the loss is calculated, oldest representation in the queue is replaced with the new key representation.

At first glance, the fact that query and key are not passed through the same model (like in SimCLR) might seem strange but there is a very good reason why this is necessary. If we used the same model to get representations of both query and key, model

would have quite an easy job detecting the representations in queue that are the oldest since the model as whole changes relatively drastically during training. Because of that we need a key encoder whose parameters ($\theta_{key_encoder}$) are updated a bit "slower" and we can do this by updating them like this:

$$\theta_{key_encoder} = m\theta_{key_encoder} + (1 - m)\theta_{query_encoder} \quad (5.5)$$

Hyper-parameter m is momentum coefficient which is usually somewhere between 0.9 and 0.999. Query encoder parameters ($\theta_{query_encoder}$) are updated as usual.

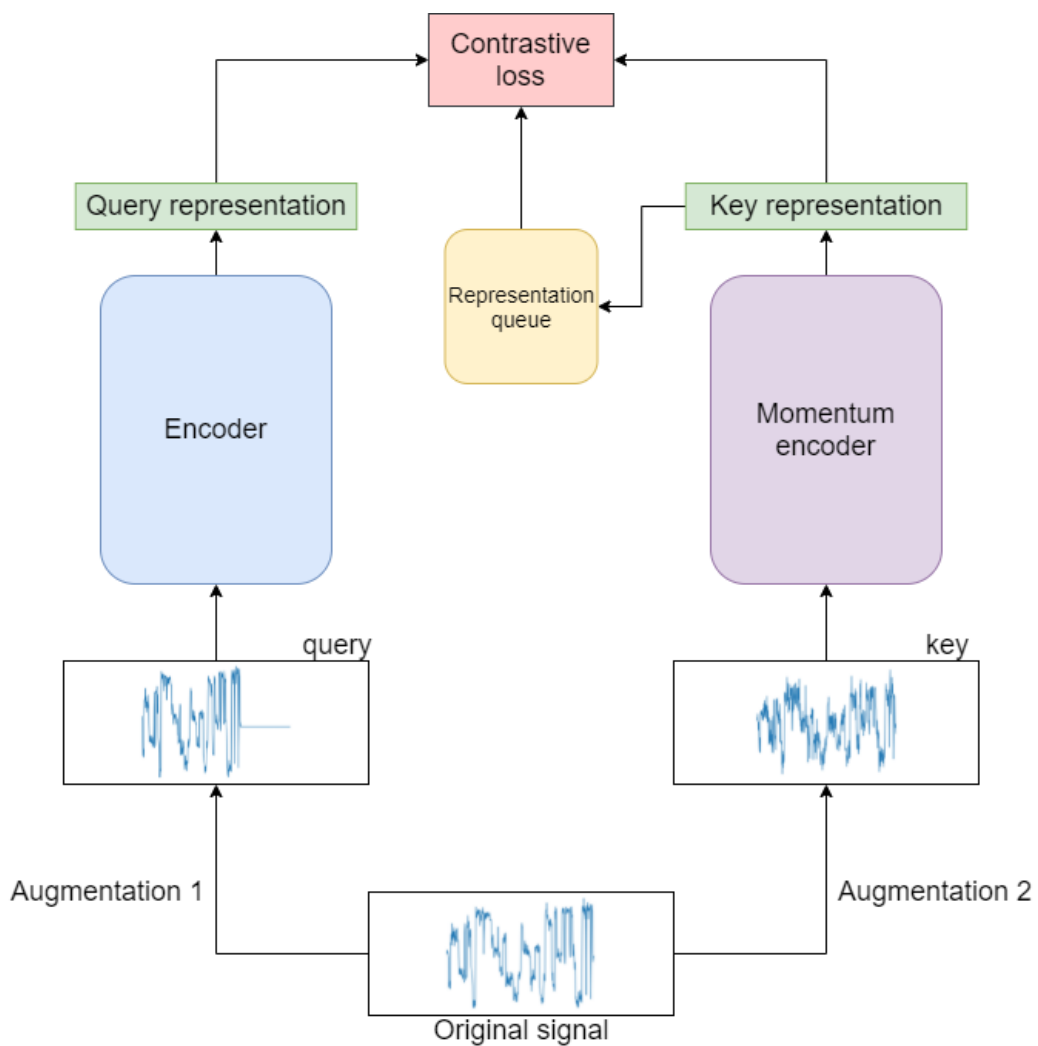


Figure 5.9: MoCo algorithm. Algorithm requires two separate models with the same architecture. Encoder (query encoder) is updated via stochastic gradient descent and momentum encoder (key encoder) is updated via momentum.

5.3.1. Implementation

One thing we have to clear up is what happens at the very beginning of training when the representation queue is empty. To be completely accurate, queue will never be empty. Once the queue is initialized, it is filled with randomly initialized vectors that simulate signal representations. This doesn't cause problems during training because these "dummy" representations will quickly be replaced with real ones.

For MoCo self-supervised training we used same hyper-parameters as in SimCLR. We set queue size to 8192 and update momentum to 0.999.

Algorithm 3 MoCo train algorithm

Input: model architecture $arch$, temperature τ , batch size N , set of signal augmentations A , momentum m , representation queue size K .

query_encoder, key_encoder = init_model($arch$), init_model($arch$)

key_encoder.parameters = query_encoder.parameters

queue = init_queue(K)

for minibatch $\{s_1, \dots, s_N\}$ **do**

for s_i in minibatch ($i \in [1, N]$) **do**

 randomly choose two augmentations a_1 and a_2 from set A

$q_i, k_{i+} = \text{query_encoder}(a_1(s_i)), \text{key_encoder}(a_2(s_i))$

end for

$$L = -\frac{1}{N} \sum_{i=1}^N \log \frac{\exp(\text{sim}(q_i, k_{i+})/\tau)}{\sum_{k \in \text{queue}} \exp(\text{sim}(q_i, k)/\tau)}$$

 update query_model parameters to minimize loss L

 key_model.param = $m * \text{key_model.param} + (1 - m) * \text{query_model.param}$

 Replace oldest queue entries with k_{i+} , $i \in \{1, \dots, N\}$

end for

5.3.2. Results

During self-supervised pretraining model quickly achieved nearly perfect accuracy on the training set (accuracy is defined as the accuracy of choosing positive among negatives). On the validation set, on the other hand, it took few epochs before such thing happened. Loss for some reason started rising after sixth epoch so we soon stopped pretraining.

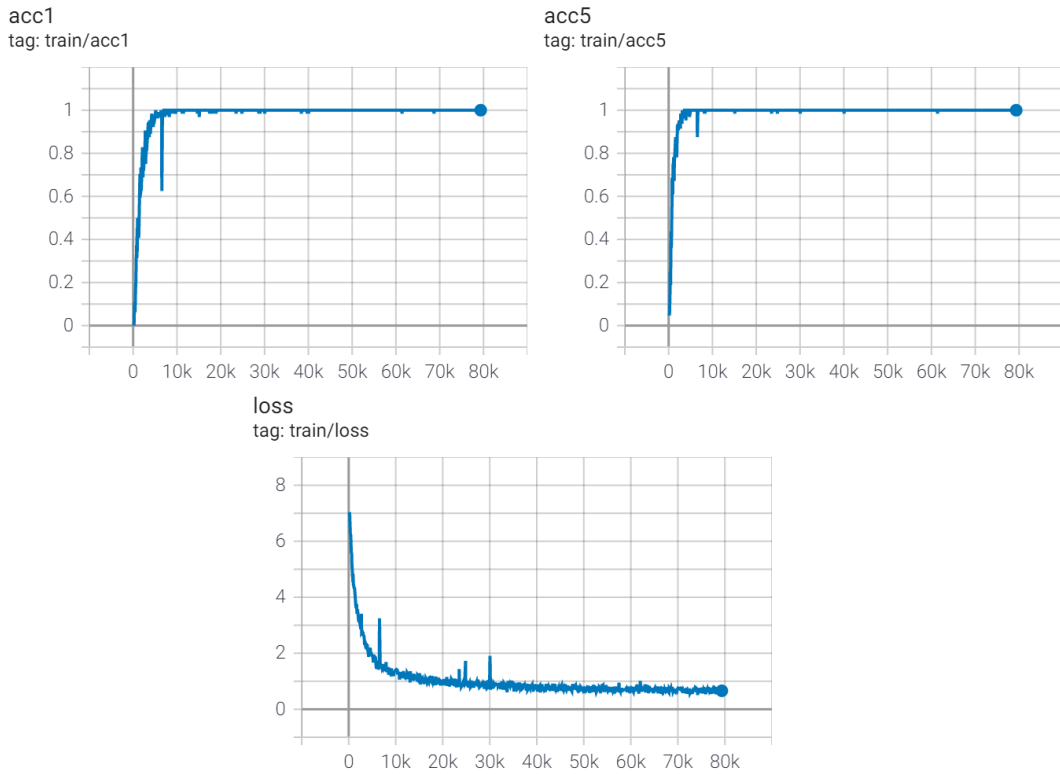


Figure 5.10: Training top-1 accuracy (top left), top-5 accuracy (top right) and loss (bottom middle) during MoCo self-supervised pretraining.

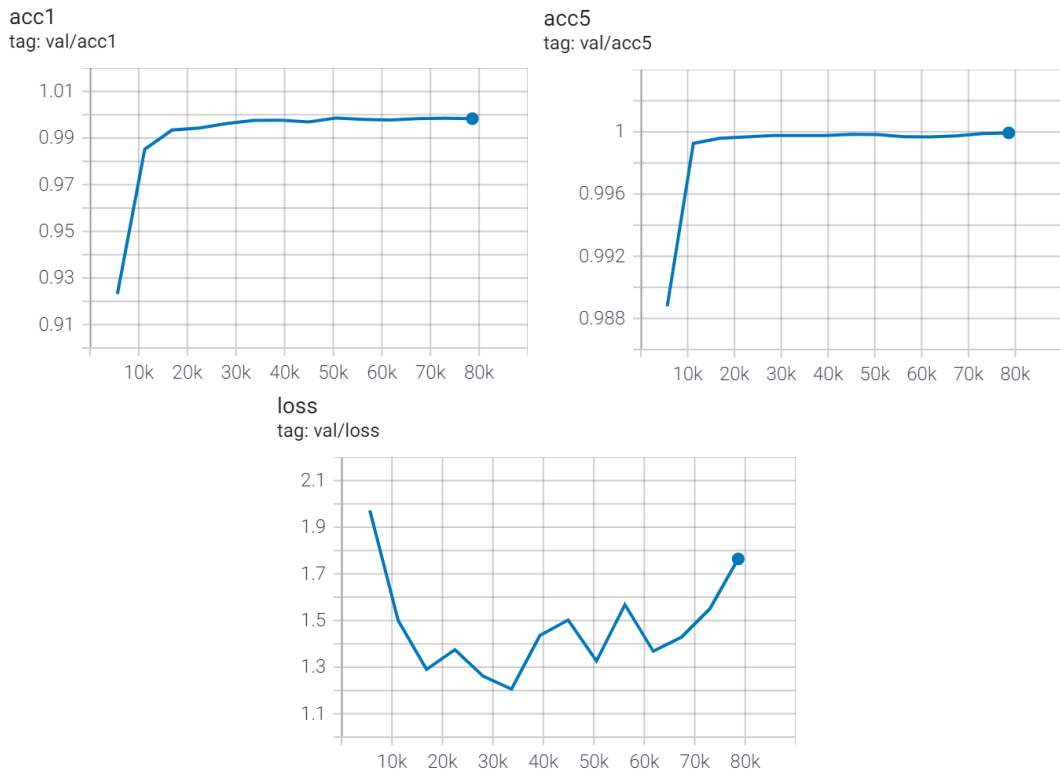


Figure 5.11: Validation top-1 accuracy (top left), top-5 accuracy (top right) and loss (bottom middle) during MoCo self-supervised pretraining.

Because model got to the perfect accuracy very fast we thought that self-supervised task was a bit too easy and we doubted that model would cause any changes on the downstream task training. However, pretraining managed to boost the accuracy on the downstream task for about 0.1%.

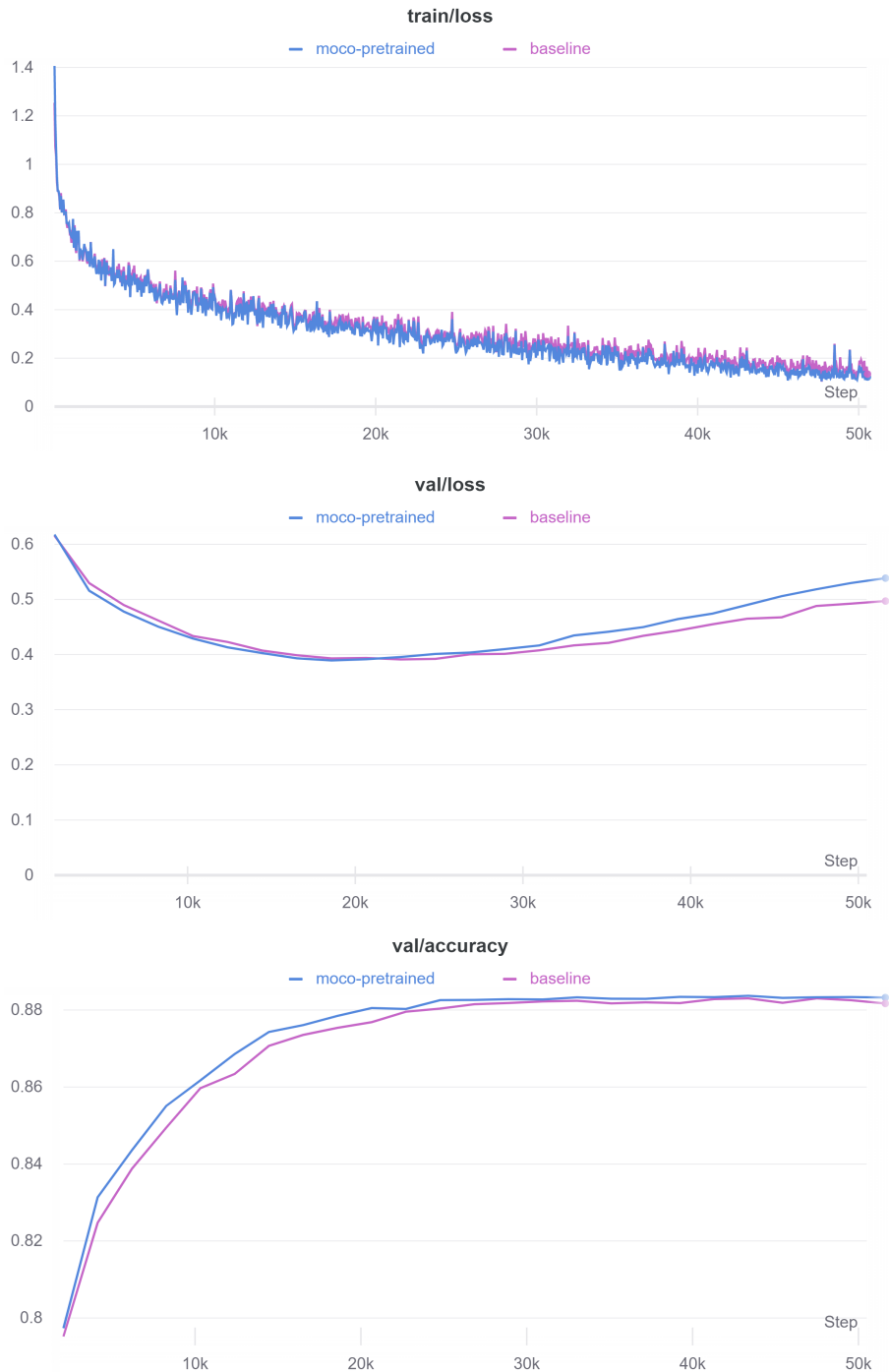



Figure 5.12: Comparison between randomly initialised basecaller ("baseline") and basecaller with pretrained Pore model ("moco-pretrained"). Figure shows training loss (top), validation loss (middle) and validation accuracy (bottom) on the basecalling task. 

5.4. SimSiam

SimSiam [6] is an interesting method for the fact that it does not require negatives during training like SimCLR and MoCo do. Instead, it uses stop-gradient functionality to prevent collapsing solution.

As always, two views of the original signal are created with augmentations. Both views are then passed through the model we are pretraining (encoder) to get representations. Finally, these representations are also passed through the special model attached to the encoder (predictor). This projection head usually consists of few linear layers and dimension of input and output vectors is the same. In our case, it is built of two linear projection layers, first of which is followed by batch normalization and ReLU activation function.

Let us assume we augmented some signal chunk and got two signal views: v_1 and v_2 . Main goal of the SimSiam algorithm is to make v_2 encoder output and v_1 predictor output as similar as possible. As mentioned before, it is very important to prevent trivial solution so we have to use stop-gradient in the computational "branch" of v_2 .

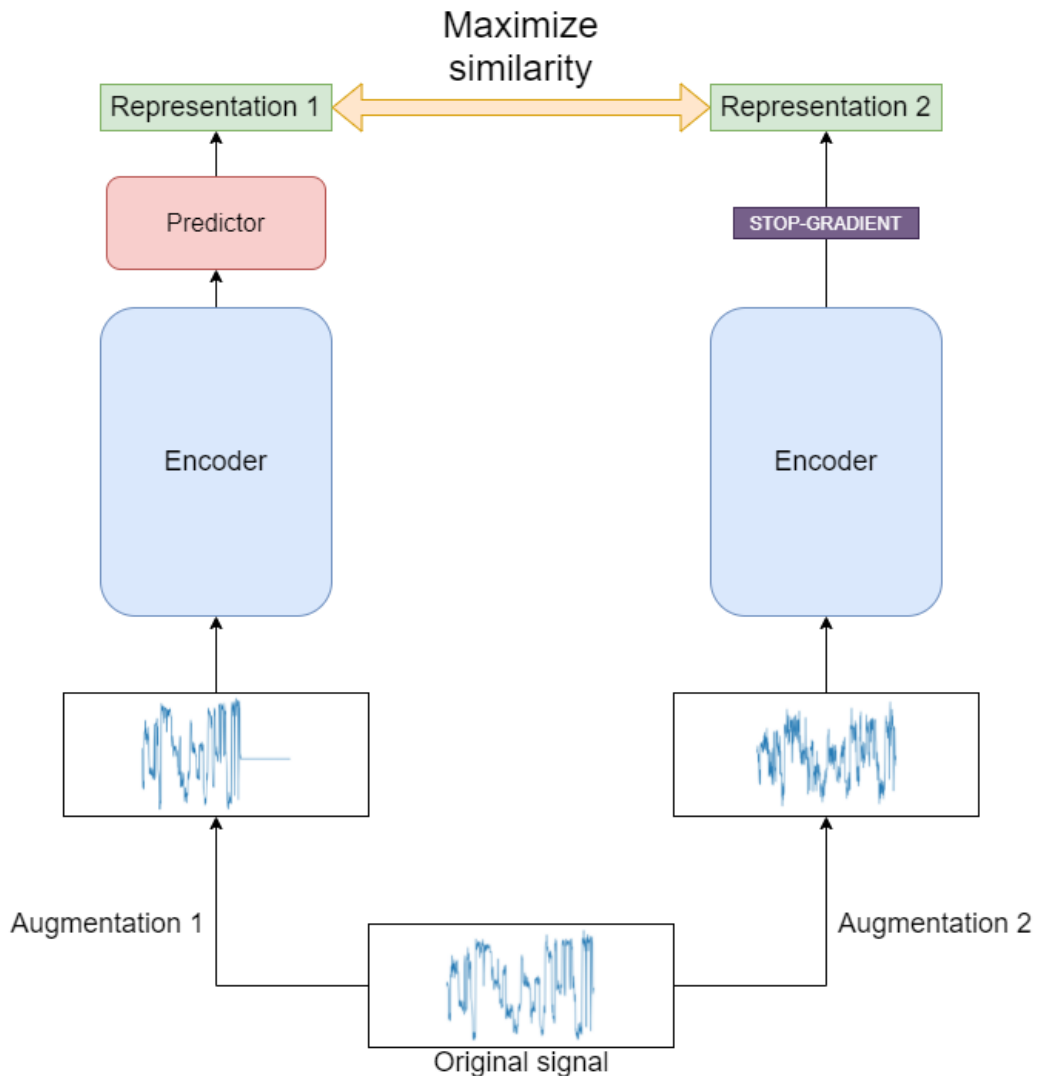


Figure 5.13: SimSiam algorithm. Both signal views are passed through the same encoder.

5.4.1. Implementation

Stop-gradient might seem like something complicated to implement at first but is rather simple. In most modern deep learning frameworks like Pytorch and Tensorflow you can easily detach tensors from the computational graph and essentially do what stop-grad operation does.

Predictor model block has the same architecture as the linear projection block in the encoder we described at the beginning of this chapter.

It is also worth noting that both signal views are passed through both encoder and predictor. In the end we want to maximize cosine similarity between output of the predictor of each signal view to the output of the encoder of the "paired" view.

Final representation dimensionality is 1024. For training, we once again used

Adam [13] optimizer with learning rate $3e-4$. Batch size was set to 64.

Algorithm 4 SimSiam train algorithm

Input: model m , predictor p , batch size N , set of signal augmentations A .

for minibatch $\{s_1, \dots, s_N\}$ **do**

for s_i in minibatch ($i \in [1, N]$) **do**

 randomly choose two augmentations a_1 and a_2 from set A

$x_{i1}, x_{i2} = m(a_1(s_i)), m(a_2(s_i))$

$h_{i1}, h_{i2} = p(x_{i1}), p(x_{i2})$

end for

$\{sg \Rightarrow \text{STOP-GRADIENT}\}$

$L = \frac{1}{N} \sum_{i=1}^N (2.0 - \text{cosine_sim}(sg(x_{i1}), h_{i2}) - \text{cosine_sim}(sg(x_{i2}), h_{i1}))$

 update model m and predictor p parameters to minimize loss L

end for

5.4.2. Results

During self-supervised pretraining loss was very unstable and it often spiked. Validation loss started stagnating after seventh epoch.

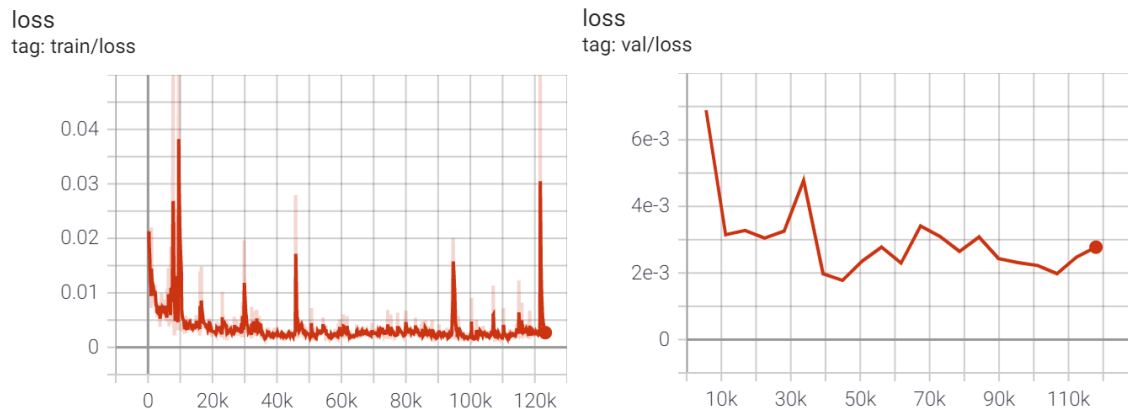


Figure 5.14: Training (left) and validation (right) loss during SimSiam self-supervised pretraining.

SimSiam had a similar performance on the downstream task (basecalling) to MoCo. It improved basecalling validation accuracy for 0.1%.

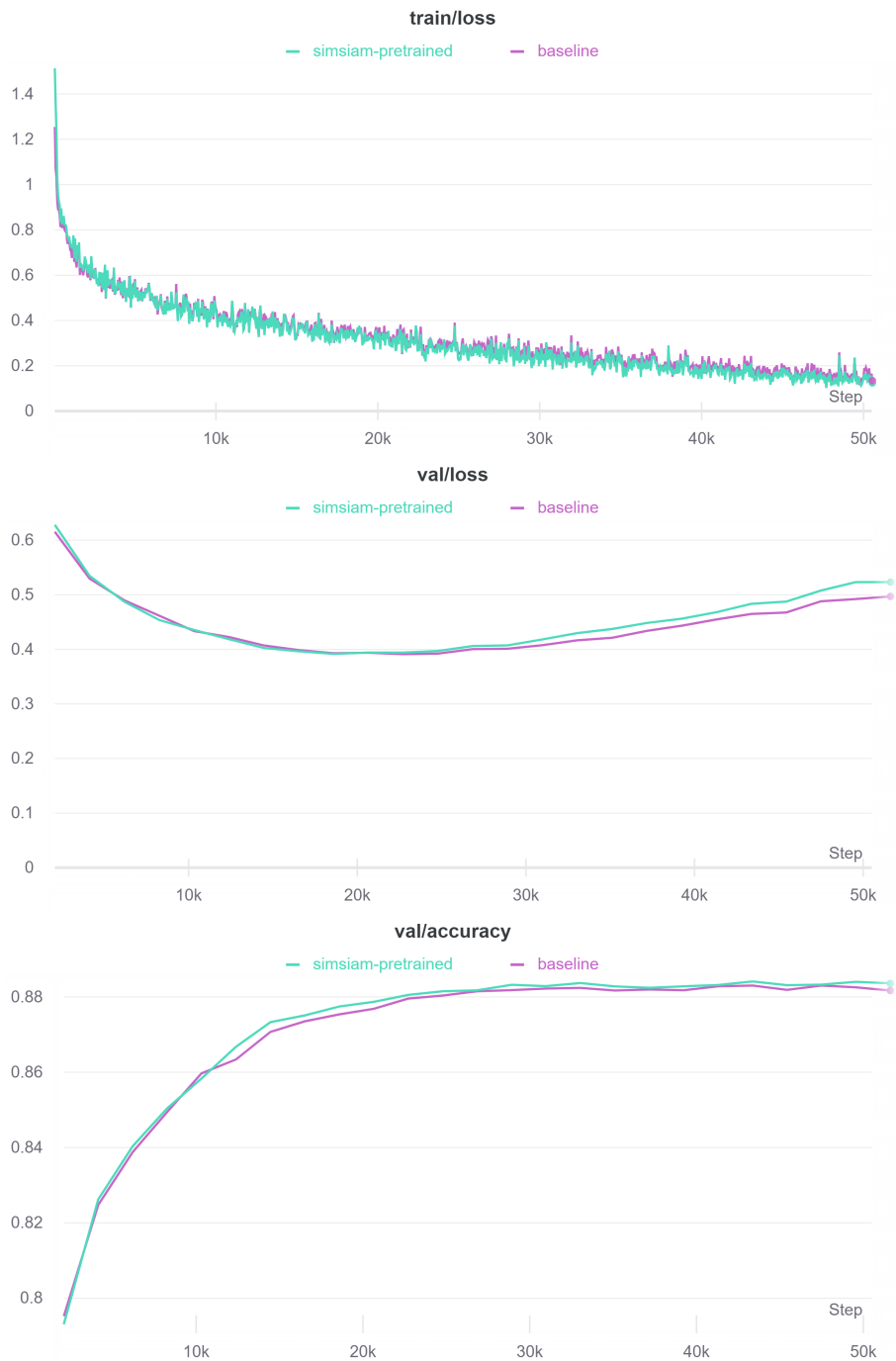


Figure 5.15: Comparison between randomly initialised basecaller ("baseline") and basecaller with Pore model that was pretrained with SimSiam algorithm ("simsiam-pretrained"). Figure shows training loss (top), validation loss (middle) and validation accuracy (bottom) on the basecalling task.

5.5. BYOL

BYOL [10] is in a way a mixture of MoCo and SimSiam algorithms. It uses special momentum updated encoder but it does not require negative samples during training. All things considered, BYOL doesn't really contain any components we haven't explained in previous chapters. It pretty much works just like SimSiam method but instead of stop-gradient operation it uses momentum encoder that functions the same as the one in MoCo.

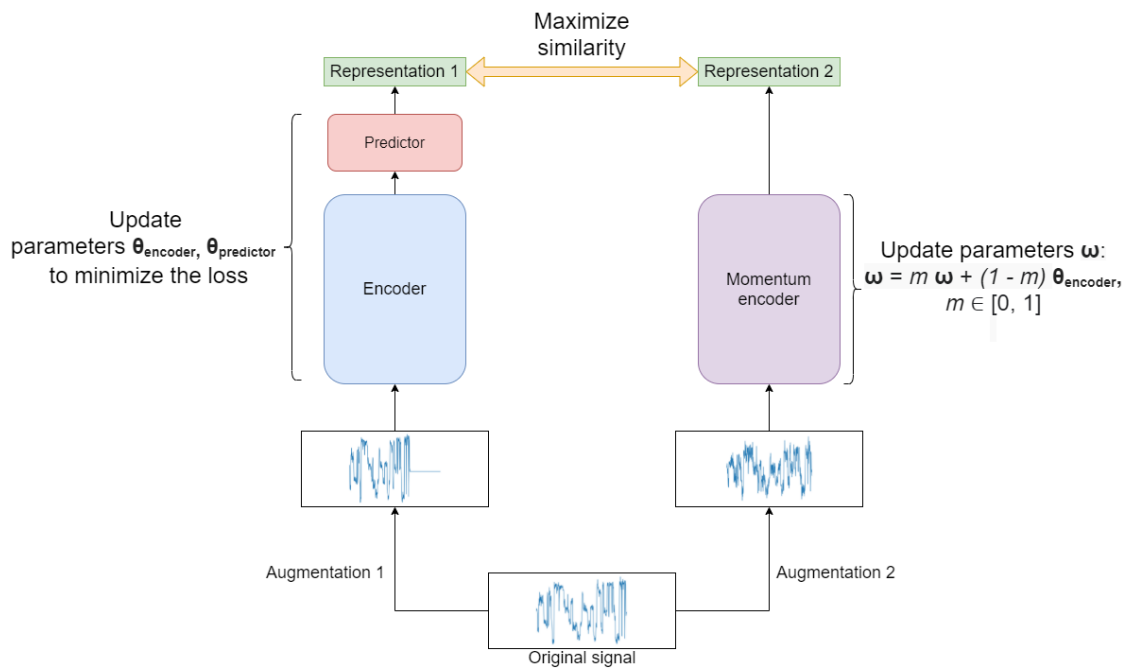


Figure 5.16: BYOL algorithm.

5.5.1. Implementation

We trained representation encoder with Adam optimizer and we used learning rate of $3e-4$. Momentum encoder was updated with exponential moving average with momentum 0.998. Representation dimensionality is set to 1024.

Algorithm 5 BYOL train algorithm

Input: model architecture $arch$, predictor p , batch size N , set of signal augmentations A , momentum m .

online, target = init_model($arch$), init_model($arch$)

target.parameters = online.parameters

for minibatch $\{s_1, \dots, s_N\}$ **do**

for s_i in minibatch ($i \in [1, N]$) **do**

 randomly choose two augmentations a_1 and a_2 from set A

$x_{i1}, x_{i2} = \text{online}(a_1(s_i)), \text{online}(a_2(s_i))$

$h_{i1}, h_{i2} = p(x_{i1}), p(x_{i2})$

$r_{i1}, r_{i2} = \text{target}(x_{i1}), \text{target}(x_{i2})$

end for

$L = \frac{1}{N} \sum_{i=1}^N (2.0 - \text{cosine_sim}(r_{i1}, h_{i2}) - \text{cosine_sim}(r_{i2}, h_{i1}))$

 update online encoder and predictor p parameters to minimize loss L

 target.param = $m * \text{target.param} + (1 - m) * \text{online.param}$

end for

5.5.2. Results

Self-supervised pretraining with BYOL was stable and loss kept steadily declining. At one point training loss significantly dropped in value and started stagnating.

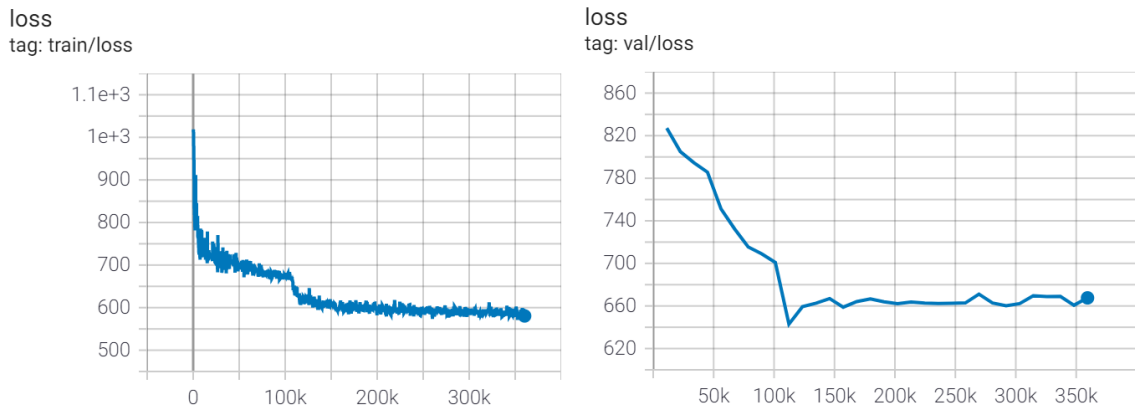


Figure 5.17: Training (left) and validation (right) loss during BYOL self-supervised pretraining.

Despite its relatively successful pretraining, BYOL did not increase performance of the basecaller. Moreover, it worsened the accuracy and it heightened the loss. We also tried to "import" parameters of the momentum encoder (instead of encoder) but

results were the same.

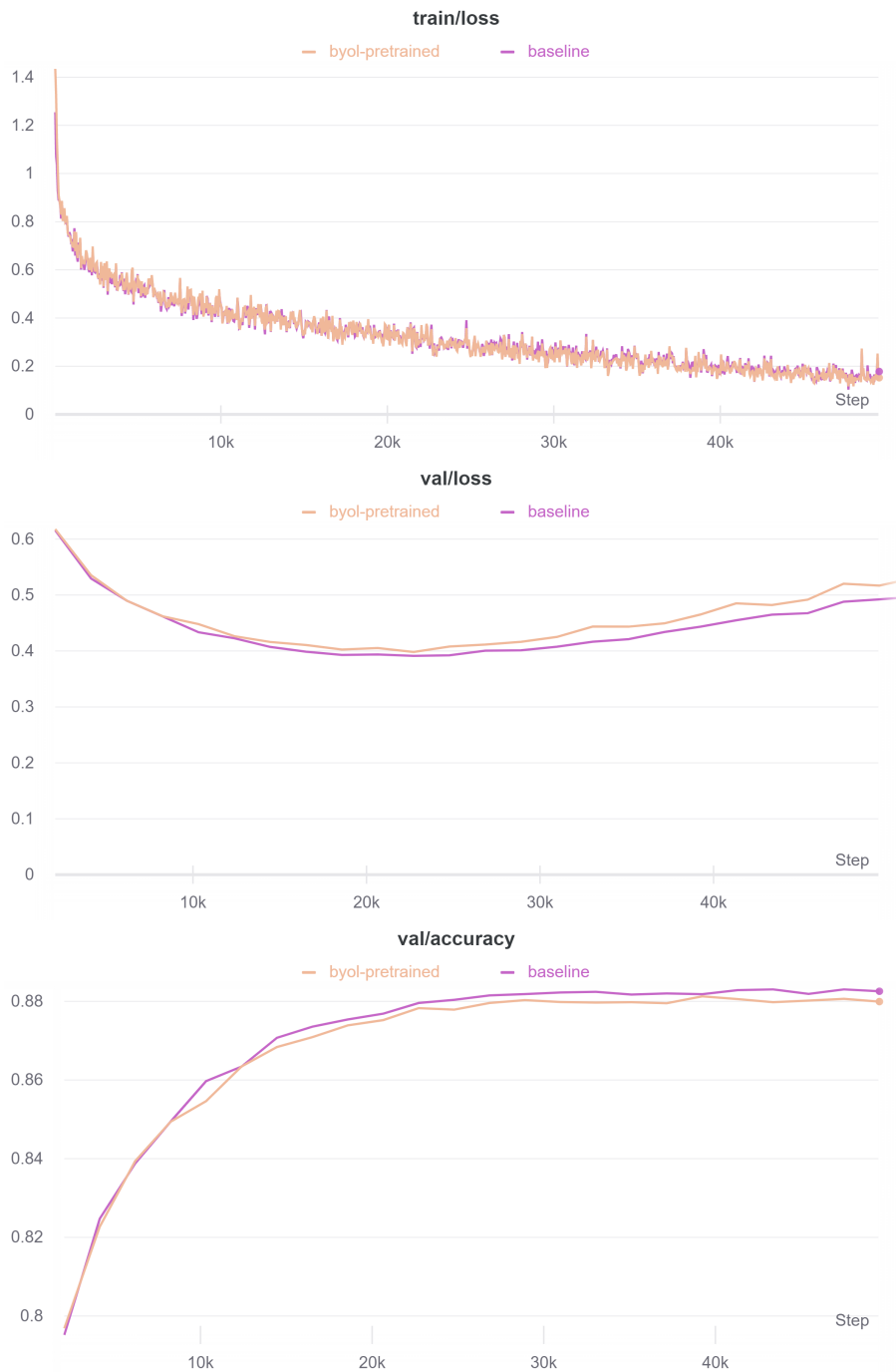


Figure 5.18: Comparison between randomly initialised basecaller ("baseline") and basecaller with Pore model that was pretrained with BYOL algorithm ("byol-pretrained"). Figure shows training loss (top), validation loss (middle) and validation accuracy (bottom) on the basecalling task.

5.6. DINO

DINO [5] draws a great deal of inspiration from knowledge distillation methods. Knowledge distillation is often used when we want to transfer knowledge from a larger model (teacher) to a smaller one (student). Main idea is to match output distributions of the teacher and the student. We can do this by minimizing cross entropy between those two distributions. But how can we use such approach for self-supervised learning? We do not have labels and we do not have some trained model we could use a teacher model.

Although we do not have specific labels, we know that two views of the same signal should have the same output distribution (representation). When it comes to teacher model, DINO uses the momentum encoder (same as MoCo) which is slowly updated in the direction of student's update. It is important to note that, unlike in classical knowledge distillation, student and teacher models have the same architecture. They only differ in the way they update their parameters.

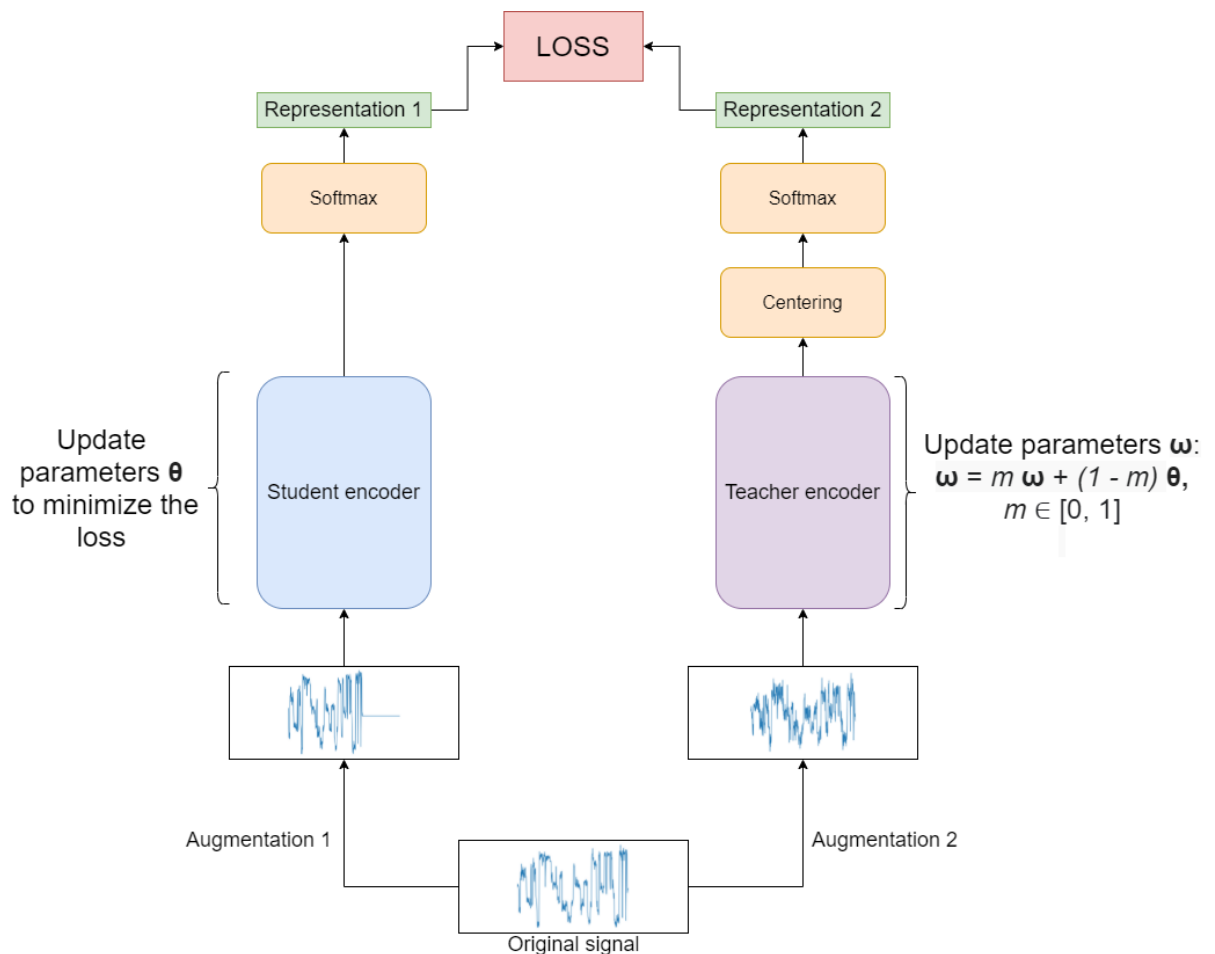


Figure 5.19: DINO algorithm.

To prevent solution collapse, DINO also introduced centering and sharpening of the teacher output (softmax input). Center is one of the parameters in the teacher model and it is subtracted from the output of the teacher model. It is updated similarly to other teacher’s parameters (with exponential moving average):

$$c = m_c c + (1 - m_c) \frac{1}{N} \sum_{j=1}^N teacher(s_j) \quad (5.6)$$

where m_c is center update momentum, N batch size and $t(s_i)$ is output of teacher model for the input signal s_i .

Main goal of centering is to prevent one of the output dimensions dominating, and thus collapsing the solution. On the other hand, DINO also uses distribution sharpening which is achieved with low softmax temperature values. This prevents collapse to uniform distribution.

DINO defines loss for a pair of positive signal views as follows:

$$l(s_1, s_2) = -softmax(student(s_1)/\tau_s) \log(softmax(teacher(s_2 - center)/\tau_t)) \quad (5.7)$$

$$L(s_1, s_2) = l(s_1, s_2) + l(s_2, s_1) \quad (5.8)$$

5.6.1. Implementation

When it comes to training, as with all other algorithms, we used Adam optimizer. Learning rate was set to 3e-5 and batch size to 32. Student softmax temperature is 0.1 and teacher softmax temperature has a value of 0.06. Center update momentum is 0.9 and teacher network update momentum is 0.999.

Algorithm 6 DINO train algorithm

Input: model architecture $arch$, batch size N , set of signal augmentations A , center momentum m_c , teacher momentum m_t , student temperature τ_s , teacher temperature τ_t .

student, teacher = `init_model(arch)`, `init_model(arch)`

teacher.parameters = student.parameters

for minibatch $\{s_1, \dots, s_N\}$ **do**

for s_i in minibatch ($i \in [1, N]$) **do**

 randomly choose two augmentations a_1 and a_2 from set A

$x_i = \text{student}(a_1(s_i))$

$y_i = \text{teacher}(a_2(s_i))$

$x_{i+N} = \text{student}(a_2(s_i))$

$y_{i+N} = \text{teacher}(a_1(s_i))$

end for $\{mean(v) = \frac{1}{d} \sum_{j=1}^d v_j\}$

$L = \frac{1}{2N} \sum_{i=1}^{2N} (mean(-softmax(x_i/\tau_s) * \log(softmax((y_i - center)/\tau_t))))$

 update student parameters to minimize loss L

 teacher.param = $m_t * \text{teacher.param} + (1 - m_t) * \text{student.param}$

 center = $m_c * \text{center} + (1 - m_c) \frac{1}{2N} \sum_{i=1}^N (y_{i1} + y_{i2})$

end for

5.6.2. Results

Training and validation kept declining through the whole self-supervised DINO pre-training. As the pretraining progressed training loss became a bit unstable but it always kept a relatively low value.

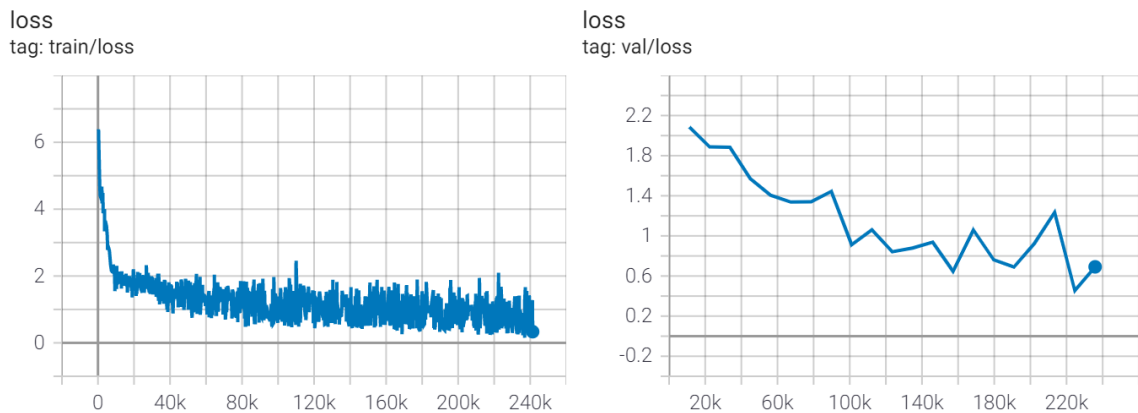


Figure 5.20: Training (left) and validation (right) loss during DINO self-supervised pretraining.

Pretrained Pore model considerably lowered the training loss on basecalling task but validation accuracy was improved for only 0.1% just like in most previously explained methods.

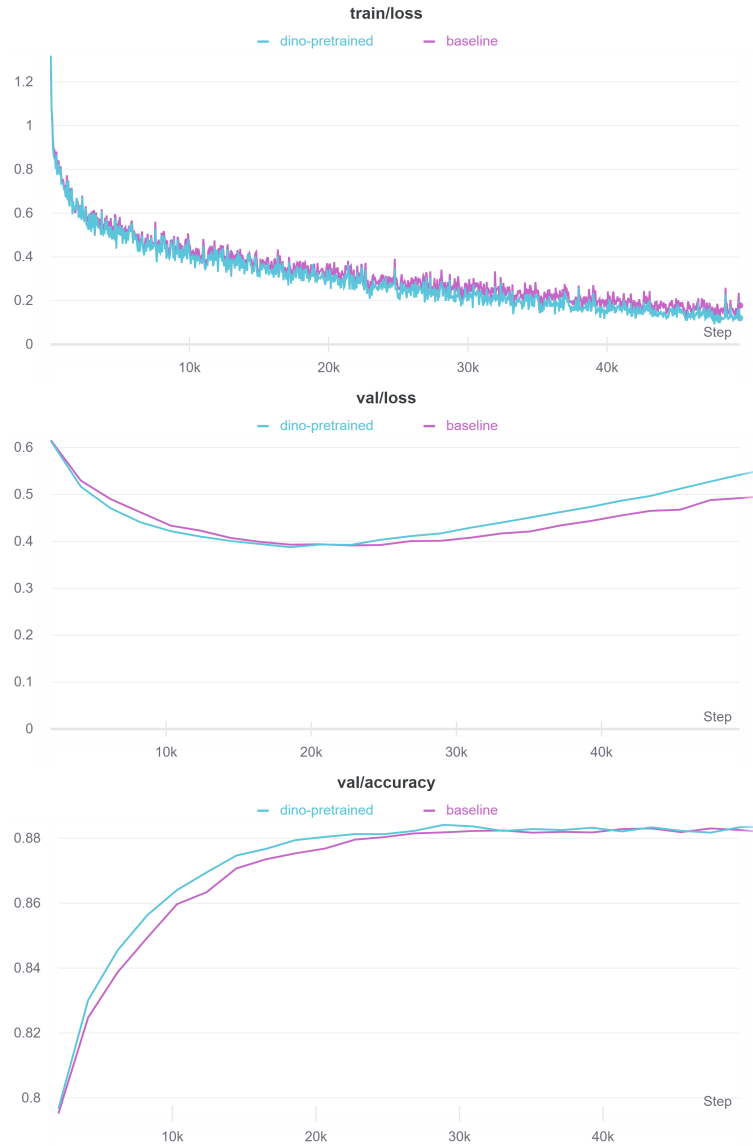


Figure 5.21: Comparison between randomly initialised basecaller ("baseline") and basecaller with Pore model that was pretrained with DINO algorithm ("dino-pretrained"). Figure shows training loss (top), validation loss (middle) and validation accuracy (bottom) on the basecalling task.

5.7. Barlow Twins

Barlow Twins [18] is a very unique algorithm because unlike most others self-supervised representation learning methods, it doesn't require some special methods of preventing

collapsed solutions. Its loss is defined in such way that such situation is impossible and this makes this algorithm very simple to implement.

Two views of the same signal are passed through the same model (encoder) to get their vector representations. Cross-correlation matrix is then calculated from those two representations and main goal of this algorithm is to make this matrix as close as possible to the identity matrix.

Barlow Twins' loss is defined as follows:

$$L = \sum_{i=1}^D (1 - C_{ii}) + \lambda \sum_{i=1}^D \sum_{j \neq i} C_{ij}^2 \quad (5.9)$$

where D is representation dimensionality and λ is a trade off hyper-parameter with which we can set how "important" is the second component of the loss in comparison to the first one. C is cross-correlation matrix and it is calculated like this:

$$C_{ij} = \frac{\sum_{b=1}^N r_{1,b,i} r_{2,b,j}}{\sqrt{\sum_{b=1}^N r_{1,b,i}^2} \sqrt{\sum_{b=1}^N r_{2,b,j}^2}} \quad (5.10)$$

where b is batch index (N is batch size), i and j represent a specific dimension in the representation. Representations are normalised along the batch dimension so the values in matrix C can be only in the interval between -1 and 1 (as it should be).

But why does this work? It doesn't feel very intuitive to force representation components to be as little as correlated as possible. Answer lies in the information bottleneck theory. Basically, what we do by forcing the decorrelation between representation components is that we prevent model from learning redundant information (e.g. information about augmentations, etc.) and we make sure it learns only the vital information, which in our case are the features that make nanopore signal look and behave the way it does.

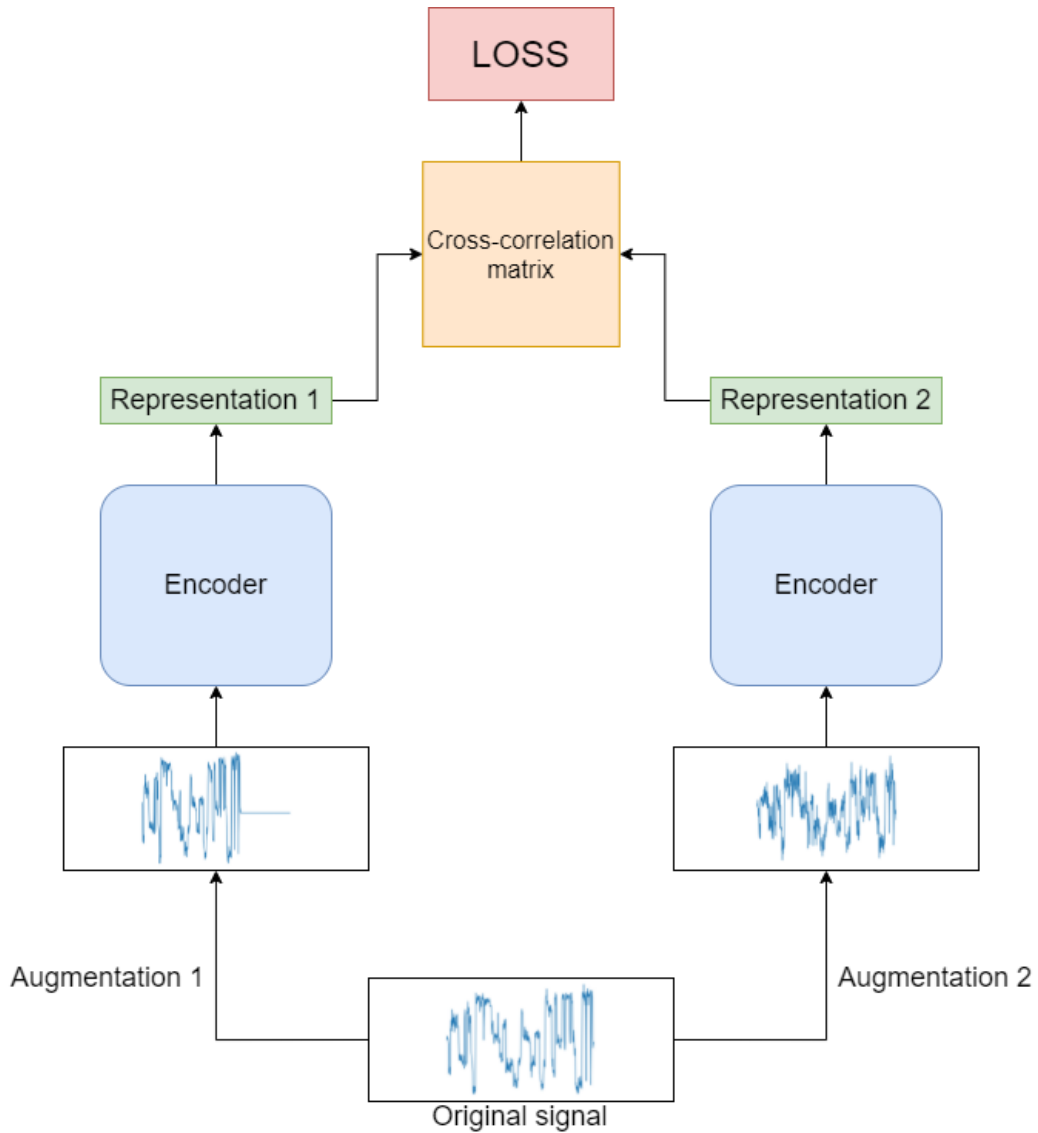


Figure 5.22: Barlow Twins algorithm.

5.7.1. Implementation

To optimize calculation of the loss, we had to fully utilize matrix computation methods. Once we calculated the cross-correlation matrix, we subtracted identity matrix from it. We then squared all values in the matrix and multiplied off-diagonal values with λ hyper-parameter. In the end, we just summed up all values in the final matrix to get the value of loss function.

For training we used Adam optimizer with learning rate of $1e-5$. Hyper-parameter λ was set to $5e-3$. We used pretty large representation dimensionality (2048) since Barlow Twins greatly benefits from such representations. Batch size was set to 64.

Algorithm 7 Barlow Twins train algorithm

Input: model m , , batch size N , representation dimensionality D , trade off hyperparameter λ , set of signal augmentations A .

for minibatch $\{s_1, \dots, s_N\}$ **do**

for s_i in minibatch ($i \in [1, N]$) **do**

 randomly choose two augmentations a_1 and a_2 from set A

$r_{1i}, r_{2i} = m(a_1(s_i)), m(a_2(s_i))$

end for

$\{r_1, r_2 \Rightarrow Nx D\}$

for $i \in \{1, \dots, N\}$ **do**

$r_{1i} = \frac{r_{1i} - \frac{1}{N} \sum_{k=1}^N r_{1k}}{std(r_1)}$

$r_{2i} = \frac{r_{2i} - \frac{1}{N} \sum_{k=1}^N r_{2k}}{std(r_2)}$

end for

$C = \frac{r_1^T r_2}{N}$

$\{id(D) \Rightarrow \text{creates identity matrix with dimensions } D \times D\}$

$L = \text{sum}((C - id(D))^2 \lambda (1 - id(D)))$

 update model m parameters to minimize loss L

end for

5.7.2. Results

Unlike most other self-supervised representation learning methods, Barlow Twins' self-supervised validation loss did not start stagnating after 5-6 epochs but it kept declining for much longer. Another interesting thing is that there were really no "spikes" in the loss that we often encountered in other self-supervised methods.

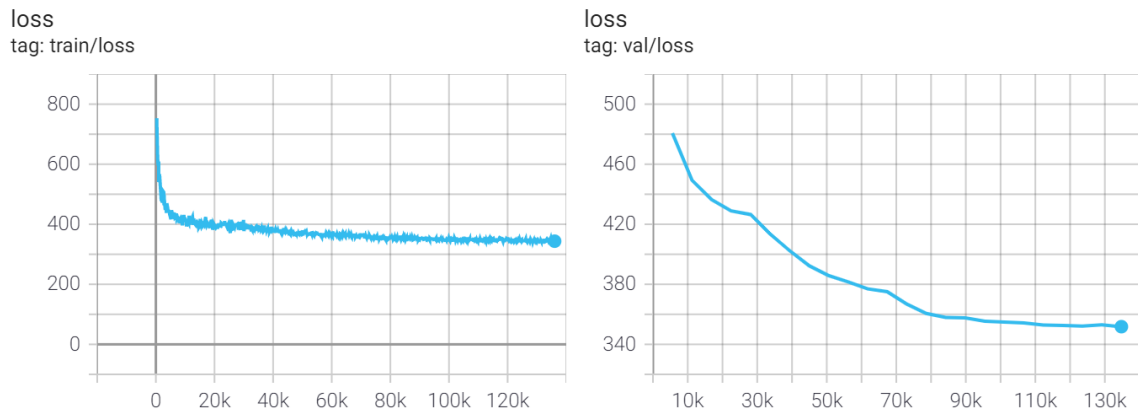


Figure 5.23: Training (left) and validation (right) loss during Barlow Twins self-supervised pretraining.

Barlow Twins improved basecalling accuracy for about 0.3%. At the start of the basecalling training, pretrained Pore model did not act much different than the randomly initialized one (training loss was more or less the same) but as the training went on, decrease in the training loss was starting to get more and more noticeable.

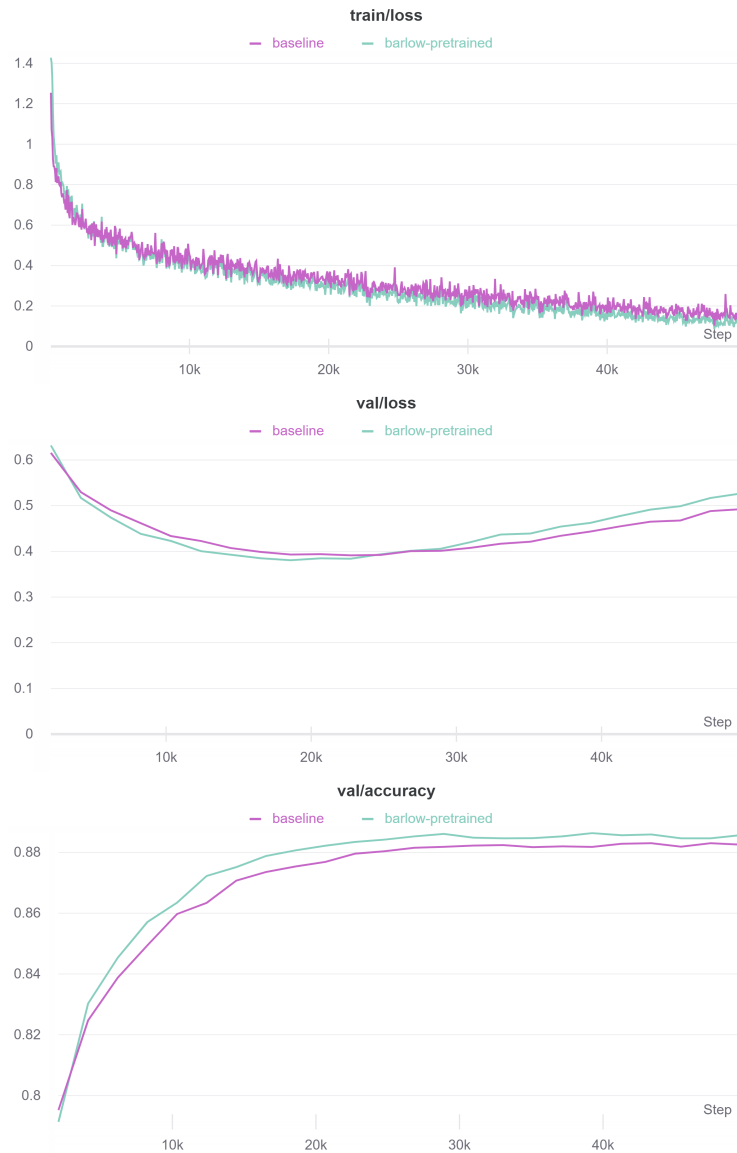


Figure 5.24: Comparison between randomly initialised basecaller ("baseline") and basecaller with Pore model that was pretrained with Barlow Twins algorithm ("barlow-pretrained"). Figure shows training loss (top), validation loss (middle) and validation accuracy (bottom) on the basecalling task.

6. Conclusion

In this thesis we explored various self-supervised algorithms and their effects on the nanopore sequencing deep learning model.

We tried out contrastive predictive coding method in which we tried to predict future latent representations. Sadly, this algorithm did not manage to improve performance of the model on the downstream task.

We also implemented a lot of self-supervised representation learning methods which improved model's basecalling accuracy. This improvement was especially noticeable when we used Barlow Twins algorithm which boosted basecalling accuracy for around 0.3%.

Although we showed that self-supervised methods are very usable in the domain of nanopore sequencing, there is still a lot of room for improvement. As the more obvious route of improvement one could take a look at additional signal augmentations in self-supervised representation learning methods. As we mentioned before, we only used two augmentations and introducing more good augmentations would probably result in higher-quality representations and therefore better performance on the downstream task.

Another idea worth investigating are semi-supervised methods like PAWS [1] and AdaMatch [3]. These approaches are very similar to classical self-supervised methods but unlike them, they also utilise a small amount of labeled data (usually to create some sort of pseudo-labels for unlabeled data).

All in all, we proved that self-supervised methods are a very powerful tool in machine learning and hope that our work will help in development of future sequencing tools and basecallers.

BIBLIOGRAPHY

- [1] Mahmoud Assran, Mathilde Caron, Ishan Misra, Piotr Bojanowski, Armand Joulin, Nicolas Ballas, i Michael Rabbat. Semi-supervised learning of visual features by non-parametrically predicting view assignments with support samples, 2021.
- [2] Alexei Baevski, Henry Zhou, Abdelrahman Mohamed, i Michael Auli. wav2vec 2.0: A framework for self-supervised learning of speech representations. 2020.
- [3] David Berthelot, Rebecca Roelofs, Kihyuk Sohn, Nicholas Carlini, i Alex Kurakin. Adamatch: A unified approach to semi-supervised learning and domain adaptation, 2021.
- [4] Mathilde Caron, Ishan Misra, Julien Mairal, Priya Goyal, Piotr Bojanowski, i Armand Joulin. Unsupervised learning of visual features by contrasting cluster assignments, 2021.
- [5] Mathilde Caron, Hugo Touvron, Ishan Misra, Hervé Jégou, Julien Mairal, Piotr Bojanowski, i Armand Joulin. Emerging properties in self-supervised vision transformers, 2021.
- [6] Ting Chen, Simon Kornblith, Mohammad Norouzi, i Geoffrey Hinton. A simple framework for contrastive learning of visual representations, 2020.
- [7] Xinlei Chen i Kaiming He. Exploring simple siamese representation learning, 2020.
- [8] Xinlei Chen, Haoqi Fan, Ross Girshick, i Kaiming He. Improved baselines with momentum contrastive learning, 2020.
- [9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, i Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.

- [10] Jean-Bastien Grill, Florian Strub, Florent Alché, Corentin Tallec, Pierre H. Richemond, Elena Buchatskaya, Carl Doersch, Bernardo Avila Pires, Zhaohan Daniel Guo, Mohammad Gheshlaghi Azar, Bilal Piot, Koray Kavukcuoglu, Rémi Munos, i Michal Valko. Bootstrap your own latent: A new approach to self-supervised learning, 2020.
- [11] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, i Ross Girshick. Momentum contrast for unsupervised visual representation learning, 2020.
- [12] John Hughes. How to boost emotion recognition performance in speech using contrastive predictive coding, 2020. URL <https://medium.com/speechmatics/boosting-emotion-recognition-performance-in-speech-using-cpc-ce6b>
- [13] Diederik P. Kingma i Jimmy Ba. Adam: A method for stochastic optimization, 2017.
- [14] Daisuke Niizumi, Daiki Takeuchi, Yasunori Ohishi, Noboru Harada, i Kunio Kashino. Byol for audio: Self-supervised learning for general-purpose audio representation, 2021.
- [15] Aaron van den Oord, Yazhe Li, i Oriol Vinyals. Representation learning with contrastive predictive coding, 2018.
- [16] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, i Illia Polosukhin. Attention is all you need, 2017.
- [17] Ishan Misra Yann LeCun. Self-supervised learning: The dark matter of intelligence, 2021. URL <https://ai.facebook.com/blog/self-supervised-learning-the-dark-matter-of-intelligence>.
- [18] Jure Zbontar, Li Jing, Ishan Misra, Yann LeCun, i Stéphane Deny. Barlow twins: Self-supervised learning via redundancy reduction. *arXiv preprint arXiv:2103.03230*, 2021.

Model dubokog učenja pore za sekvenciranje nanoporama

Sažetak

Sažetak na hrvatskom jeziku.

Ključne riječi: bioinformatika, samo-nadzirajuće učenje, sekvenciranje, nanopora, DNK

Deep Learning Model of Nanopore Sequencing Pore

Abstract

Abstract.

Keywords: bioinformatics, self-supervised learning, sequencing, nanopore, DNA