INSTITUT FÜR INFORMATIK
Machine Learning
Universitätsstr. 1 D–40225 Düsseldorf

Heinrich Heine Universität Düsseldorf

Classification of Electrocardiograms using Contrastive Predictive Coding

Julian Winter

Masterarbeit

Date of issue: 18. August 2021
Date of submission: 18. Februar 2022
Reviewers: Prof. Dr. Stefan Harmeling
Prof. Dr. Michael Leuschel

Erklärung

Hiermit versichere ich, dass ich diese Masterarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Düsseldorf, den 18. Februar 2022

Julian Winter

Abstract

The following work deals with classifying multilabel electro-cardiographic data using a “self-supervised” learning method called Contrastive Predictive Coding (short CPC) [1], that learns lower dimensional data representations without requiring class labels. These data representations can then be used in classification tasks. Learning from data without labels is potentially very beneficial in fields where correctly labeled data is scarce and only obtainable slowly by human experts.

The goal of this work is to re-implement CPC, apply and test several changes to the architecture and evaluate its capability of classifying ECG-data. We try to answer the following central questions to shine a light on CPC’s strengths and weaknesses:

How does the CPC architecture compare against our baseline models in a fully supervised setting? How capable is CPC’s pretraining? How good are the learned representations on their own? Under what circumstances is CPC especially efficient to use? Can the original architecture be improved? Does CPC learn visually verifiable useful latent representations?

We will answer these questions by conducting various experiments that aim to emulate real-life circumstances such as low label availability and limited training times. CPC is compared to fully supervised models, where we make use of architectures like the TCN [2] and a special time-series classification focused residual net [3], but also introduce a wide variety of own baseline models.

For evaluation we look at predictions both quantitatively, through different metrics, and qualitatively, by either inspecting embeddings in lower dimensions of the learned representations or by visualizing the prediction probabilities in a wide variety of plots. Additionally we utilize the gradient of selected trained networks to show spots in the input data which possibly contain ECG-classes.

Our contributions to the field are a more precise explanation of the original publication [1], alterations to the one dimensional architecture suited for timeseries classification, namely non overlapping data windows, normalized latent representations, different encoder-, autoregressive context- and downstream-task networks. We also introduce a "no-context" architecture which predict latents without making use of the standard context recurrent network. In total we trained and tested over 1700[1] networks and will also release their model properties and average scores as table for all those interested.

Last but not least our generated prediction plots and "point of interest"-visualization mainly based on grad-cam [4], show the different models’ strengths beyond classification scores and could also assist cardiologists in the future.

[1] We count individually trained/tested networks, not necessarily different architectures

Contents

1 Introduction

1.1 Electrocardiographic Data

Electrocardiographic data is obtained by placing electrodes on different parts of the body, mainly around the heart, which measure the electrical changes in voltage over time, produced by the heart muscle [5]. Multiple electrodes are placed to get a better overview over the heart state, since other muscles will add noise under non ideal conditions. A high recording frequency of 257-10000hz and a recording length of 10 seconds up to 30 minutes[2] or even days (long time ECG used in hospitals) are desirable to not miss important information that could reveal an infarction and other muscular heart diseases.

We obtain sequential data that can be classified like other time-series, however since each individual electrode produces its own signal there is more than one data channel to analyze concurrently, which increases the difficulty to find a generalizing computer system. For example the conventional "12-lead ECG" uses 10 differently placed electrodes resulting in 12 different channels together with the two additional ones added by averaging [6]. Accounting for the high frequency and length this would sum up to $\mathbf{datapoints} = \mathbf{seconds} \cdot \mathbf{frequency} \cdot \mathbf{channels} = 10\mathbf{s} \cdot 1000\mathbf{hz} \cdot 12 = 120000$ data points for short recordings or $120\mathbf{s} \cdot 1000\mathbf{hz} \cdot 12 = 1440000$ data points for longer recordings, per patient (assuming a frequency of 1000hz). The varying lengths and much information make correct prediction challenging. For comparison, the widely used ImageNet [7] counts on average 469×387 pixels[8], which is equal to $469 \cdot 387 \cdot 3 = 544509$ data points. Additionally the recordings can contain more than one “true” label which makes correct classification even more difficult. Like in many medical fields, one big hurdle in training a fully automated system is the inaccessible correctly labeled data. The biggest reason for the data scarcity is that only trained professionals/doctors are able to successfully diagnose patients, while also requiring long periods of time to do so.

Nevertheless we found a few dataset sources that are openly available online:

PTB A dataset provided by the Physikalisch-Technische Bundesanstalt (PTB) containing 549 records with 9 diagnostic class labels. [9]

PTB-XL A dataset provided by the Physikalisch-Technische Bundesanstalt (PTB) containining 21837 records with 52 diagnostic class labels (5 superclasses, split into 52 unique scp codes).[10]

Georgia “Georgia” dataset which is hosted on `Kaggle.com` [11]

China A dataset provided during a chinese classification challenge and made openly available.

Nature A 12-lead electrocardiogram database for arrhythmia research covering more than 10,000 patients. [12]

Most of the above datasets and more are also used in a recent ECG classification challenge from 2020 [13] hosted by physionet.org [14]. Since the original datasets have incompatible

[2] attributes of ECG data reported from 1.1

labels (e.g. the same classes are mapped to different terms) we use the available challenge data because the labels across all datasets were already mapped to unique SNOMED CT codes [15]. The "Nature" data labels, which are not part of the challenge data, had to be dropped since they were incompatible to the other datasets, thus this data will be only used in pretraining for models that require it.

The data has 12 channels and one or more "true" labels per sample. However there is no hint on to where the label is found in the sequential data. See Figure 1 for an example ECG recording.

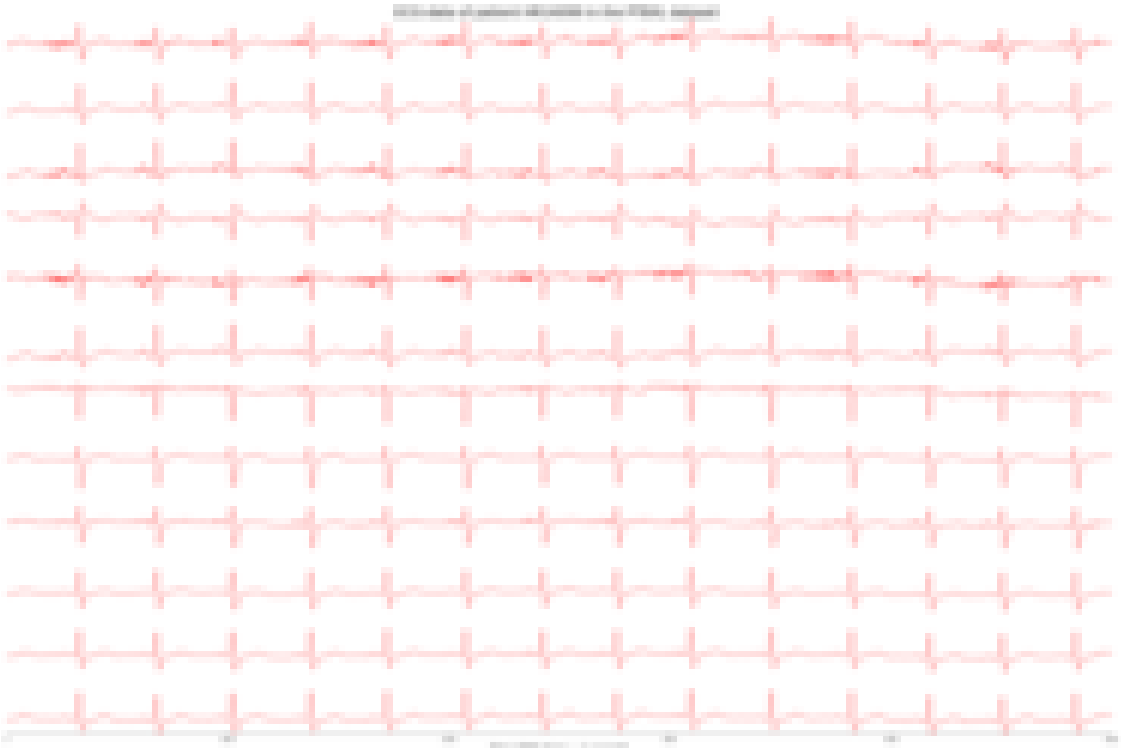


Figure 1: A single patients ECG with 500hz recording frequency. The classes *EKG: T wave abnormal* and *sinus rhythm* can both be found somewhere in the data.

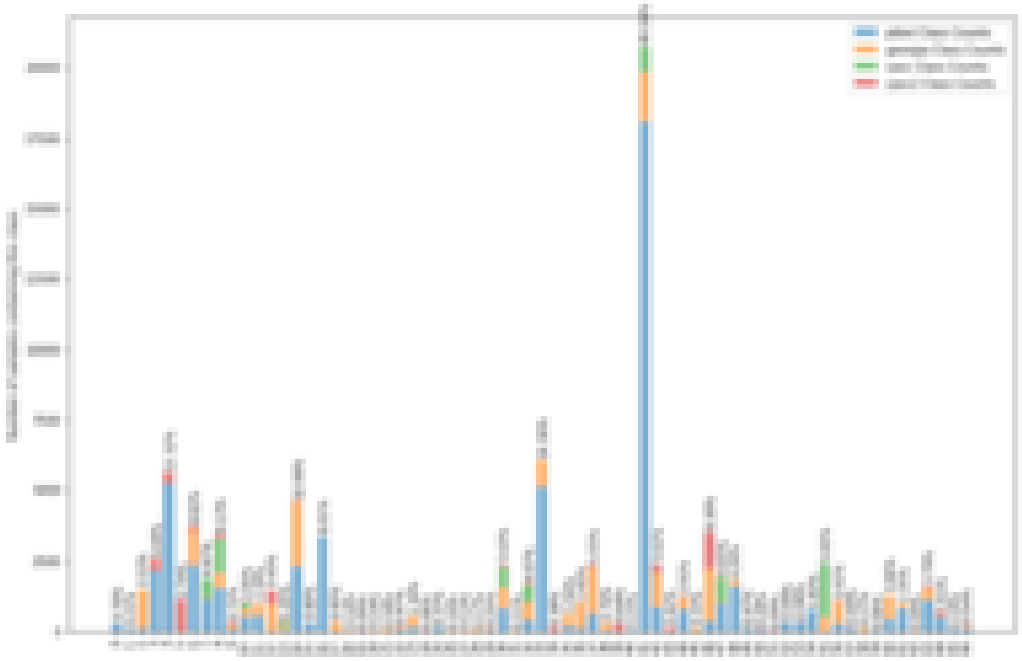
To better understand the data we count class occurrences over all datasets, with their respective SNOMED Code and the index number used in our implementation Figure 2. Readily identifiable is that not every class is present in all datasets and that some classes are very rare in comparison to others. Additionally some classes are closely related or even superclasses of others but are not consistently stated as true even though the subclass is true (see Bundle Branch Block vs. left/right Bundle Branch Block or Ventricular Hypertrophy vs. left/right Ventricular Hypertrophy). Since some class distinctions might be intentional, we did not apply any changes to the labels, but want to point out the possible classification difficulty increase due to the label inconsistencies.

Figure 2: All SNOMED Codes with their respective name and count in the datasets.

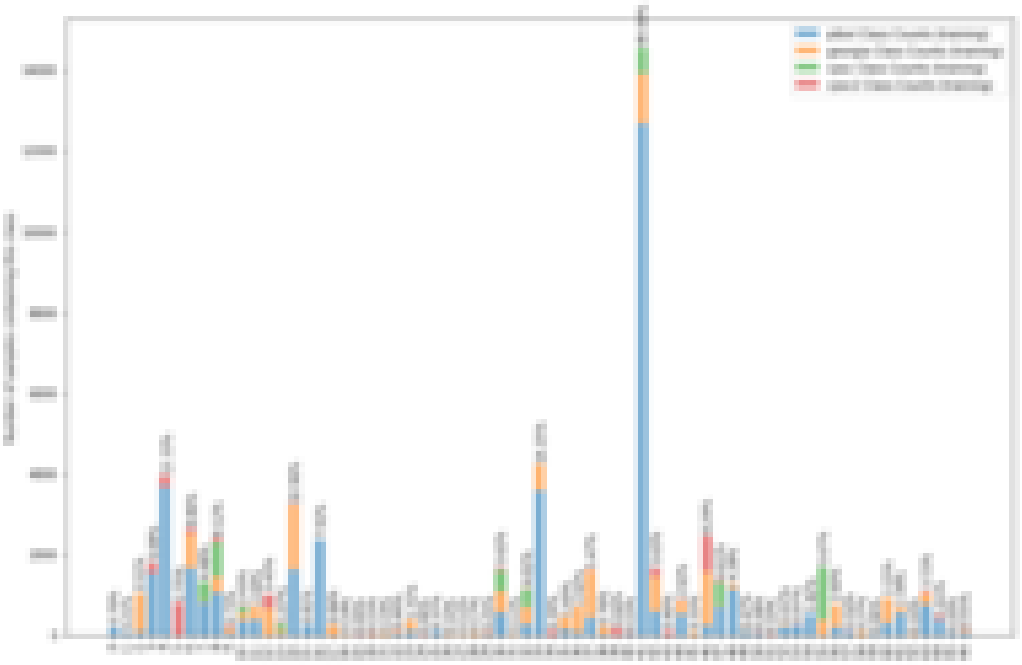
Index	Class Snomed Code	Class Term	ptbxl	georgia	cpsc	cpsc2
0	10370003	Rhythm from artificial pacing (298)	295	0	0	3
1	11157007	Ventricular bigeminy (89)	82	2	0	5
2	111975006	Prolonged QT interval (1493)	118	1372	0	3
3	164861001	EKG myocardial ischemia (2556)	2175	0	0	381
4	164865005	EKG: myocardial infarction (5640)	5261	7	0	372
5	164867002	EKG: old myocardial infarction (1160)	0	0	0	1160
6	164873001	EKG:left ventricle hypertrophy (3735)	2359	1226	0	150
7	164884008	ECG: ventricular ectopics (1894)	1154	41	699	0
8	164889003	ECG: atrial fibrillation (3441)	1514	561	1219	147
9	164890007	EKG: atrial flutter (303)	73	185	0	45
10	164909002	EKG: left bundle branch block (1038)	536	229	235	38
11	164917005	EKG: Q wave abnormal (1010)	548	461	0	1
12	164930006	ECG: ST interval abnormal (1453)	0	979	0	474
13	164931005	ST elevation (444)	28	133	220	63
14	164934002	EKG: T wave abnormal (4651)	2345	2284	0	22
15	164947007	Prolonged PR interval (340)	340	0	0	0
16	164951009	EKG: QRS complex abnormal (3389)	3389	0	0	0
17	17338001	Ventricular premature beats (359)	0	351	0	8
18	195042002	Second degree atrioventricular block (58)	14	23	0	21
19	195080001	Atrial fibrillation and flutter (39)	0	2	0	37
20	195126007	Atrial hypertrophy (61)	0	59	0	2
21	233917008	Atrioventricular block (77)	0	74	0	3
22	251120003	Incomplete left bundle branch block (201)	77	83	0	41
23	251146004	Low QRS voltages (552)	182	370	0	0
24	251180001	Ventricular trigeminy (24)	20	1	0	3
25	251200008	Indeterminate cardiac axis (156)	156	0	0	0
26	251266004	Ventricular pacing pattern (43)	0	43	0	0
27	251268003	Atrial pacing pattern (50)	0	50	0	0
28	253352002	Left atrial abnormality (70)	0	70	0	0
29	266249003	Ventricular hypertrophy (105)	30	70	0	5
30	270492004	First degree atrioventricular block (2385)	797	764	721	103
31	27885002	Complete atrioventricular block (46)	16	8	0	22
32	284470004	Premature atrial contraction (1722)	398	635	616	73
33	39732003	Left axis deviation (6080)	5146	934	0	0
34	413844008	Chronic myocardial ischemia (160)	0	0	0	160
35	425419005	EKG: inferior ischemia (660)	219	441	0	0
36	425623009	EKG: lateral ischemia (1039)	142	897	0	0
37	426177001	ECG: sinus bradycardia (2351)	637	1670	0	44
38	426434006	EKG: anterior ischemia (323)	44	279	0	0
39	426627000	ECG: bradycardia (273)	0	6	0	267
40	426761007	EKG: supraventricular tachycardia (62)	27	32	0	3
41	426783006	ECG: sinus rhythm (20760)	18090	1751	916	3
42	427084000	ECG: sinus tachycardia (2374)	826	1247	0	301
43	427172004	ECG: premature ventricular contractions (178)	0	0	0	178
44	427393009	ECG: sinus arrhythmia (1234)	772	452	0	10
45	428417006	Early repolarization (138)	0	138	0	0
46	428750005	Nonspecific ST-T abnormality on electrocardiogram (3513)	381	1864	0	1268
47	429622005	ST Depression (1967)	1009	36	867	55
48	445118002	Left anterior fascicular block on electrocardiogram (1805)	1626	179	0	0
49	445211001	Left posterior fascicular block on electrocardiogram (199)	177	22	0	0
50	446358003	Right atrial hypertrophy (117)	99	0	0	18
51	446813000	Left atrial hypertrophy (40)	0	0	0	40
52	47665007	Right axis deviation (421)	343	77	0	1
53	54329005	Acute myocardial infarction of anterior wall (414)	354	0	0	60
54	55930002	EKG ST segment changes (776)	770	6	0	0
55	59118001	Right bundle branch block (2390)	0	534	1855	1
56	59931005	Inverted T wave (1105)	294	806	0	5
57	63593006	Supraventricular premature beats (208)	157	1	0	50
58	6374002	Bundle branch block (115)	0	115	0	0
59	67198005	Paroxysmal supraventricular tachycardia (24)	24	0	0	0
60	67741000119109	Left atrial enlargement (1295)	427	868	0	0
61	698252002	Non-specific intraventricular conduction delay (994)	789	201	0	4
62	713422000	EKG: atrial tachycardia (40)	0	27	0	13
63	713426002	EKG: Incomplete right bundle branch block (1606)	1118	404	0	84
64	713427006	EKG: complete right bundle branch block (681)	542	27	0	112
65	74390002	Wolff-Parkinson-White pattern (82)	80	2	0	0
66	89792004	Right ventricular hypertrophy (229)	126	84	0	19

Figure 3 shows the class imbalance for the whole dataset and our train split.

Figure 3: Class counts for labels in the datasets as bar diagram. Percentages mean "Class is included in $x\%$ of files"



(a) Class counts for labels in all datasets.



(b) Class counts for labels in the training datasets.

Since we have multiple possible class labels per patient it might be interesting to see if

any of the classes have a correlation to each other. For that we created Figure 4 which shows the Pearson correlation coefficient.

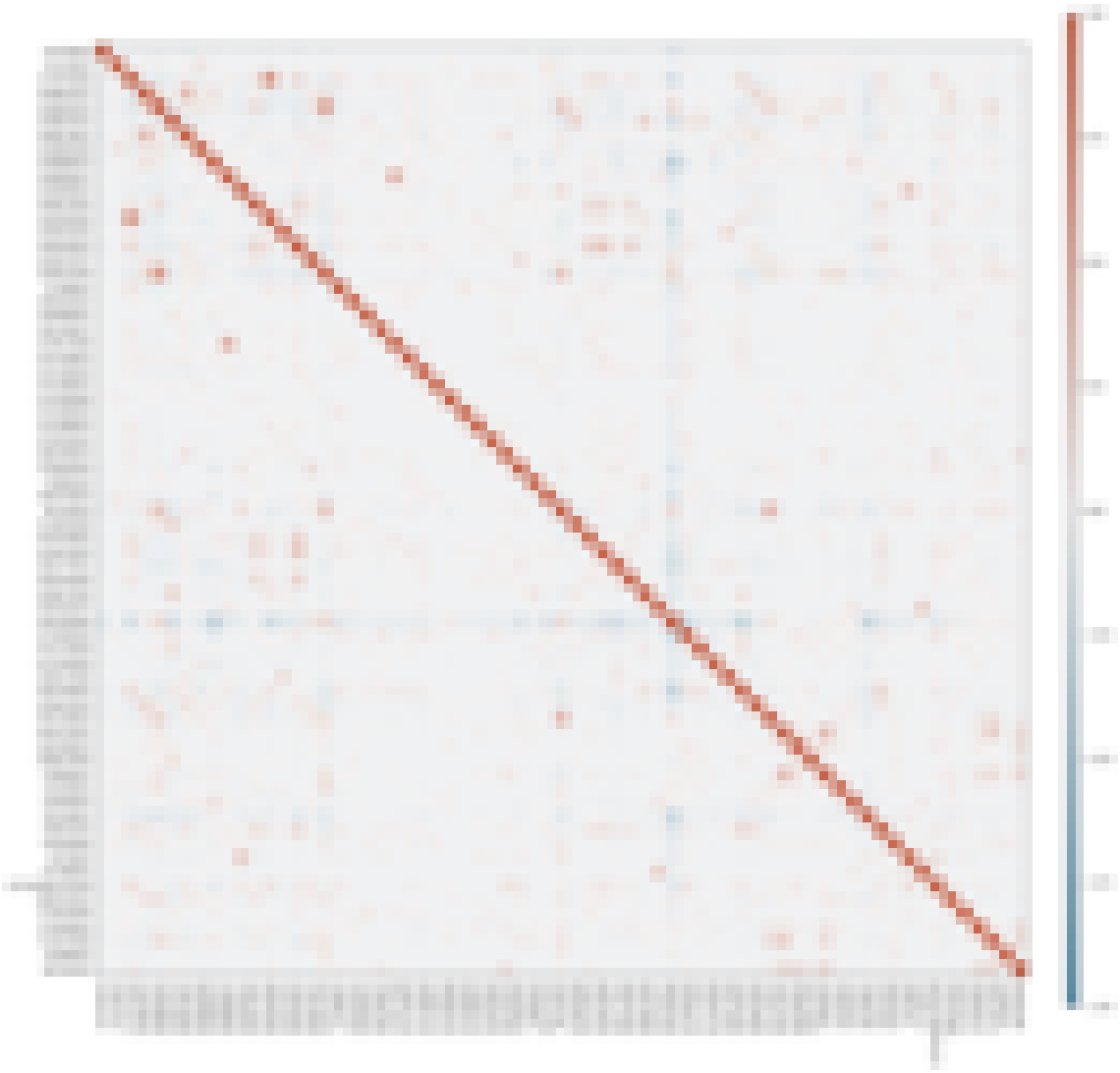


Figure 4: Pearson Class Correlation Coefficient $\rho_{X,Y}$ heatmap: Given class x, what other classes y are likely to appear at the same patient?. $\rho_{X,Y} > 0 =$ positive correlation, $\rho_{X,Y} < 0 =$ negative correlation, $\rho_{X,Y} \approx 0$ no correlation

Understanding and classifying ECG data (in real time) using an automated computer system could greatly help the likelihood of surviving an (incoming) infarction, since analysis by eye is time consuming and demands a trained medical professional. Monitoring patients at all times in the hospital could also be a great benefit. While doctors will probably still be mandatory to make the final diagnosis and prescribe medication, an automated classification system that also shows potential highly diagnostic locations in the signal could reduce analysis time by a big factor.

To address the problem of few data being publicly available, solutions different to fully supervised trained models could have a big impact on their success.

One of those more unsupervised learning domains is representation learning.

1.2 Representation Learning/Background

In representation/feature learning the algorithm (neural net), tries to learn representations, often in a lower dimension, by not using the real labels directly and instead uses secondary labels extracted from the data itself. After training on these artificial labels, most layers will be used again with their weights to train on the downstream task - the task to predict the desired primary labels. As such representation learning can be split into a pretraining and training phase, which are both needed to produce a working prediction model. The strength of representation learning comes from the fact that unlabeled data is often easier to come by, but which cannot be used by fully supervised methods.

Following Yann LeCun, this is also referred to as self-supervised learning[16], which is partly unsupervised learning since the models do not need the original labels, but there is also an supervisional aspect, where one needs to define secondary labels, related to the downstream task.

There are many different approaches to representation learning, some train the biggest fraction of the layers using labels that can be extracted with high certainty from the data directly (Jigsaw from image [17], Image Rotation [18], [19]). Others augment the data and the neural net has to calculate the same lower dimensional representation for both augmented images [20].

Another way to learn representation is to predict missing or contextual information, eg. by blacking out known parts of the data and training an algorithm to fill these spots out again. The algorithm will have to find meaningful representations and thus "understand" the data in order predict the missing spots correctly.

It is hypothesized that solving secondary tasks or predicting contextual information "are fruitful partly because the context from which we predict related values are often conditionally dependent on the same shared high-level latent information" [1]. That is to say by solving a secondary task and learning representations useful for that specific task, the model will be easier to train on the real problem because the necessary representations are similar to the ones already learned.

Unfortunately the methods mentioned above focus on image recognition and use knowledge not applicable to general sequential data. Also predicting high dimensional data is a very difficult task and simpler loss functions like Mean Squared Error do not work very well. Furthermore architectures like Recurrent Neural Nets that predict only one "step" into the future will likely resort to exploiting "local smoothness" - learning features that only describe the data locally but fail to infer a more global data description.

That's where methods like CPC, short for Contrastive Predictive Coding from the paper "Representation Learning using Contrastive Predictive Coding"[1] come into play.

1.3 Contrastive Predictive Coding

1.3.1 Overview

The goal of CPC is to learn data representations in a compact (latent) embedding space, which are universally useful in downstream tasks. CPC learns these representations by predicting the future not in the sequential input data, but rather its latent lower dimensional space. Due to the lower dimension, correct predictions of future latents are easier to model. Since there is no ground truth in the unknown embedding space, the architecture is trained with the help of a loss based on Noise-Contrastive Estimation, where the model has to differentiate between latents that stem from the current sample or other samples in the batch.

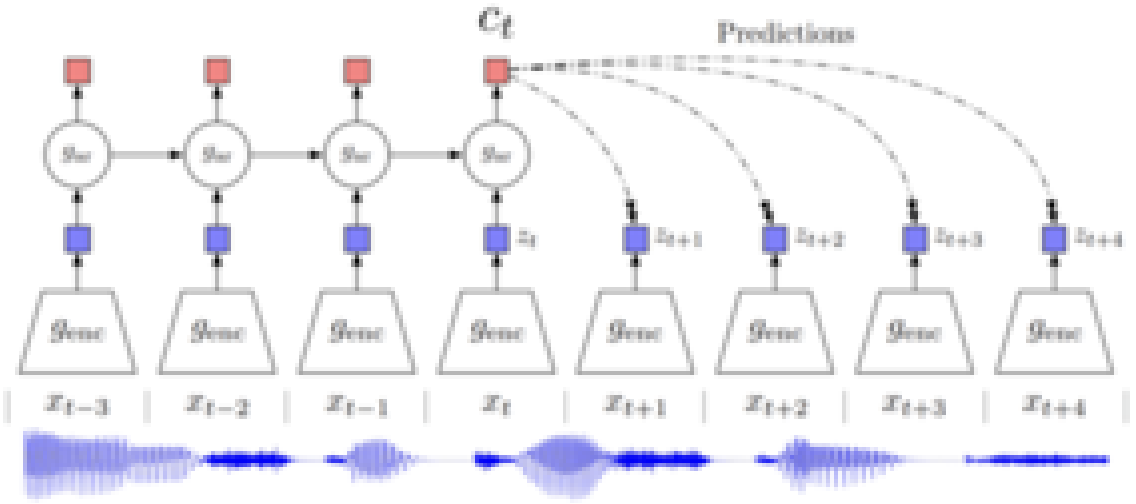


Figure 5: CPC audio architecture how it is visualized in [1]

Both Figure 5 and Figure 6 are CPC architectures, only differing in how the encoder encodes the data into latent representations. They can be used as a close reference for the following description:

First, CPC transforms the sequential data signal x into multiple latent variable vectors $(z_{t-m} \dots z_t, z_{t+1}, \dots, z_{t+k}, \dots, z_{t+n}) = g_{enc}(x)$ in an embedding space (with potentially lower dimension), using any encoder network. In Figure 5 g_{enc} encodes parts of the data separately, in Figure 6 the Encoder calculates all latents "in one go". t denotes the "current" timestep, m and n are steps into the past and future respectively and may vary depending on chosen architecture/hyperparameters and input data. The "past" latents $z_{t-m} \dots z_t$ are fed into any recurrent network/autoregressive model (e.g. g_{ar} in Figure 5), which produces a context matrix $g_{ar}(z_{t-m}, \dots, z_t) = (c_{t-m}, \dots, c_t)$, where c_t is the context vector for all seen latents $z_{t-m}, \dots, z_t$. Since an autoregressive model is used, any number of latent vectors can be summarized into a context, which can also be sized differently from the latent vectors. The last context c_t is then multiplied individually with n weight matrices $W_1, \dots, W_n$ to predict n future latent vectors $\hat{z}_{t+1}, \dots, \hat{z}_{t+n}$. Multiple latents are predicted at the same time to counteract the exploitation of local smoothness and force the network to encode globally useful information into the context. In the best

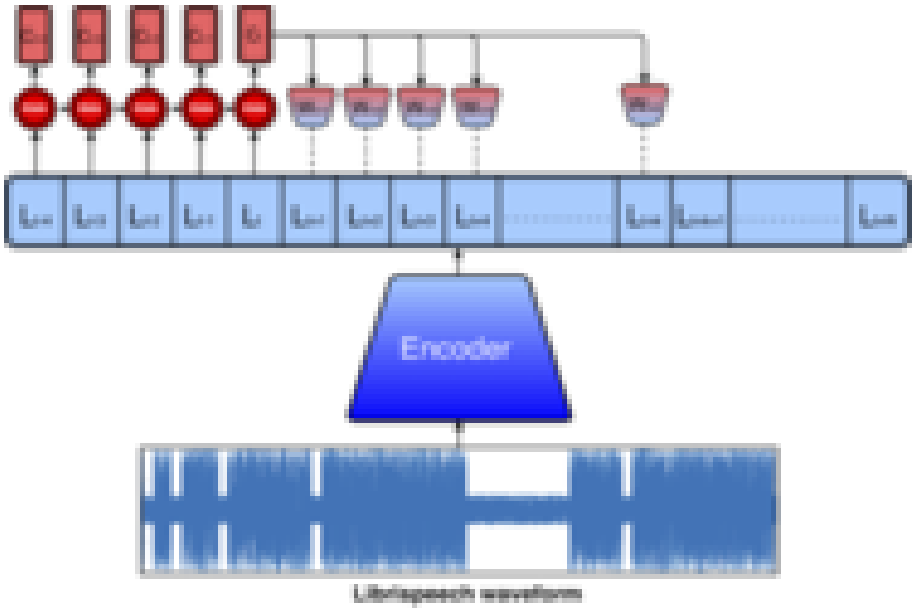


Figure 6: CPC audio architecture how it is visualized in [21]

case scenario the context represents a meaningful description of all past latents and will be a good feature matrix for following downstream tasks. The disparity between the predicted latents $\hat{z}$ and the latents z, produced by the encoder network earlier, can then be used to form the loss and thus train both the encoder and autoregressive model jointly.

To avoid a trivial encoding of e.g. an all-zero-vector for all latents, a loss based on noise contrastive estimation [22] is used, where the network has to differentiate between positive and negative samples. Positive samples are latents that stem from $p(x_{t+k}|c_t)$ — the same datasource that the context was calculated from. Negative samples are latents that were calculated from $p(x_{t+k})$ — the unconditioned data distribution, independent from the current context. Negative samples can e.g. be taken from the data batch or in concrete terms another patient.

1.3.2 Motivation

As in Subsection 1.2 already said it is assumed that unsupervised representation learning approaches "are fruitful partly because the context from which we predict related values are often conditionally dependent on the same shared high-level latent information" [1]. With this motivation we want to learn features (latent representations) that maximize the mutual information between the original input and the encoded representations. Mutual information describes how much two probability density functions have in common and how much knowing one leads to knowledge about the other. "More specifically, it quantifies the "amount of information" [...] obtained about one random variable through observing the other random variable". [23] The equation for mutual information for two probability mass functions is given by Equation 1, where we arrive exactly at the papers definition in the second to last line. Here $x \in X$ are samples from the original signal and

C the contexts derived by $x \in X$:

$$\begin{aligned} I(X;C) &= \sum_{c \in C} \sum_{x \in X} p_{X,C}(x,c) \log \left(\frac{p_{X,C}(x,c)}{p_X(x) p_C(c)} \right) \\ &= \sum_{c \in C} \sum_{x \in X} p_{X,C}(x,c) \log \left(\frac{p_{X,C}(x|c) p_C(c)}{p_X(x) p_C(c)} \right) \\ &= \sum_{c,x} p(x,c) \log \left(\frac{p(x|c)}{p(x)} \right) \\ &= \mathbb{E} \log \left(\frac{p(x|c)}{p(x)} \right) \end{aligned} \tag{1}$$

Mutual Information $I(X;C)$ could thus be described as the mean logarithmic difference between the conditional and the unconditional joint probability distributions of X and C. If the conditional probability mass function (PMF) is similar to the unconditional PMF ($\implies X \perp C$), the difference will be close to 0. If the logarithmic difference is bigger, we know that that X and C are conditionally dependent, which means they have a higher Mutual Information.

As our goal, we want to find weights for our model, that **maximize the mutual information between the input signal X and the calculated context C.**

1.3.3 Math in detail

To reach our goal, we define the probability that given a context c_t and a set of samples X, a specific sample $x_i \in X$ was drawn from the conditional distribution $p(x_{t+k}|c_t)$, rather than from the unconditional distribution $p(x_{t+k})$. This probability is given by Equation 2 with $[d = i]$ being the indicator that sample x_i is the "positive" sample [1, p. 4]:

$$\begin{aligned} p(d = i|X, c_t) &= \frac{p(x_i|c_t) \prod_{l \neq i} p(x_l)}{\sum_{j=1}^{N} p(x_j|c_t) \prod_{l \neq j} p(x_l)} \\ &= \frac{p(x_i|c_t) \left(\frac{1}{p(x_i)} \prod_{l=1}^{N} p(x_l) \right)}{\sum_{j=1}^{N} p(x_j|c_t) \left(\frac{1}{p(x_j)} \prod_{l=1}^{N} p(x_l) \right)} \\ &= \frac{\frac{p(x_i|c_t)}{p(x_i)}}{\sum_{j=1}^{N} \frac{p(x_j|c_t)}{p(x_j)}} \end{aligned} \tag{2}$$

In Equation 2 the numerator $p(x_i|c_t) \prod_{l \neq i} p(x_l)$ is the probability that x_i was drawn from $p(x_{t+k}|c_t)$, while all other $x_j \in X, j \neq i$ were drawn from $p(x_{t+k})$. This gets normalized by $\sum_{j=1}^{N} p(x_j|c_t) \prod_{l \neq j} p(x_l)$, the sum of all probabilities where x_j was drawn from $p(x_{t+k}|c_t)$ instead of x_i. Note that $p(x_i|c_t) \prod_{l \neq i} p(x_l)$ is also included in the sum, thus $p(d = i|X, c_t) \in [0, 1]$.

The goal of the loss function will be to maximize this probability over all possible combinations between each sample and the corresponding context. Which means the model

will need to distinguish between positive and negative samples, given a context. Knowing that the mutual information means: given C, how much do we know about X, it becomes clear that a model that maximized the mutual information between the original signal and its context, will have an easier "job" to distinguish between real and wrong sample.

The categorical cross entropy loss is a commonly used loss function for training on class probability scores for N classes: $\sum_{i=1}^{N} y_i log(\hat{y}_i)$ — where y_i is the ground truth - and $\hat{y}_i$ is the model prediction.

Combining both the cross entropy loss with Equation 2, we arrive at the following loss function:

$$\mathcal{L}_{\mathcal{N}} = - \mathop{\mathbb{E}}_{x_i \in X} \log \left[\frac{\frac{p(x_i|c_t)}{p(x_i)}}{\sum\limits_{x_j \in X} \frac{p(x_j|c_t)}{p(x_j)}} \right] \tag{3}$$

However, since we cannot evaluate $\frac{p(x_{t+k}|c_t)}{p(x_{t+k})}$ directly, we need a function f s.t. $f_k(x_{t+k}, c_t) \propto \frac{p(x_{t+k}|c_t)}{p(x_{t+k})}$, ($f$ needs to only be proportional to the optimal value and is not restricted to lay in the interval $[0, 1]$). In the paper $f_k(x_{t+k}, c_t) := \exp\left(z_{t+k}^T W_k c_t\right)$ is used to form a simple log-bilinear model. $W_k c_t$ is the models prediction for latent z_{t+k}, the dot product between the prediction and the ground truth expresses their similarity, because the dot product between two vectors implies whether they point in the same/opposing/orthogonal direction. This log-bilinear model is also used in the well-known word2vec model in natural language processing [24].

We thus finally arrive at the InfoNCE loss function from the paper (which can also be described as the logarithmic Softmax over vector similarities):

$$\begin{aligned} \mathcal{L}_{\mathcal{N}} &= -\mathop{\mathbb{E}}_{X} \log \left[\frac{f_k(x_{t+k}, c_t)}{\sum\limits_{x_j \in X} f_k(x_j, c_t)} \right] \\ &:= -\mathop{\mathbb{E}}_{X} \log \left[\frac{\exp\left(z_{t+k}^T W_k c_t\right)}{\sum\limits_{x_j \in X} \exp\left(z_j^T W_k c_t\right)} \right] \end{aligned} \tag{4}$$

The prove from the original paper, that this loss is indeed maximizing the mutual information between c_t and x_{t+k}, is given here with additional steps for easier following (Equation 5). For $f_k(x_j, c_t)$ it uses the optimal value $\frac{p(x_{t+k}|c_t)}{p(x_{t+k})}$ instead of $\exp\left(z_{t+k}^T W_k c_t\right)$:

(5)

Given a set X split into one positive sample x_{t+k} and $N-1$ negative samples X_{neg}.

$$\mathcal{L}_N^{opt} = -\mathop{\mathbb{E}}_X \log \left[\frac{\frac{p(x_{t+k}|c_t)}{p(x_{t+k})}}{\sum_{x_j \in X} \frac{p(x_j|c_t)}{p(x_j)}} \right] \tag{6}$$

$$= -\mathop{\mathbb{E}}_X \log \left[\frac{\frac{p(x_{t+k}|c_t)}{p(x_{t+k})}}{\frac{p(x_{t+k}|c_t)}{p(x_{t+k})} + \sum_{x_j \in X_{neg}} \frac{p(x_j|c_t)}{p(x_j)}} \right] \tag{7}$$

$$= \mathop{\mathbb{E}}_X \log \left[\frac{\frac{p(x_{t+k}|c_t)}{p(x_{t+k})} + \sum_{x_j \in X_{neg}} \frac{p(x_j|c_t)}{p(x_j)}}{\frac{p(x_{t+k}|c_t)}{p(x_{t+k})}} \right] \tag{8}$$

$$= \mathop{\mathbb{E}}_X \log \left[\frac{p(x_{t+k})}{p(x_{t+k}|c_t)} \cdot \left(\frac{p(x_{t+k}|c_t)}{p(x_{t+k})} + \sum_{x_j \in X_{neg}} \frac{p(x_j|c_t)}{p(x_j)} \right) \right] \tag{9}$$

$$= \mathop{\mathbb{E}}_X \log \left[1 + \frac{p(x_{t+k})}{p(x_{t+k}|c_t)} \sum_{x_j \in X_{neg}} \frac{p(x_j|c_t)}{p(x_j)} \right] \tag{10}$$

$$= \mathop{\mathbb{E}}_X \log \left[1 + \frac{p(x_{t+k})}{p(x_{t+k}|c_t)} (N-1) \sum_{x_j \in X_{neg}} \frac{1}{N-1} \frac{p(x_j|c_t)}{p(x_j)} \right] \tag{11}$$

Assuming all $N-1$ negative samples have approximately the same probability:

$$= \mathop{\mathbb{E}}_X \log \left[1 + \frac{p(x_{t+k})}{p(x_{t+k}|c_t)} (N-1) \mathop{\mathbb{E}}_{x_j \in X_{neg}} \frac{p(x_j|c_t)}{p(x_j)} \right] \tag{12}$$

$$\approx \mathop{\mathbb{E}}_X \log \left[1 + \frac{p(x_{t+k})}{p(x_{t+k}|c_t)} (N-1) \right], \text{Assuming } c_t, x_j \text{ are (nearly) independent} \tag{13}$$

$$= \mathop{\mathbb{E}}_X \log \left[\frac{p(x_{t+k})}{p(x_{t+k}|c_t)} N + 1 - \frac{p(x_{t+k})}{p(x_{t+k}|c_t)} \right] \tag{14}$$

$$\geq \mathop{\mathbb{E}}_X \log \left[\frac{p(x_{t+k})}{p(x_{t+k}|c_t)} N \right], \text{Assuming } p(x_{t+k}) \leq p(x_{t+k}|c_t) \tag{15}$$

$$= -\mathop{\mathbb{E}}_X \log \left[\frac{p(x_{t+k}|c_t)}{p(x_{t+k})} \right] + \log(N) \tag{16}$$

$$= -I(x_{t+k}, c_t) + \log(N) \tag{17}$$

Therefore $-I(x_{t+k}, c_t) \geq \log(N) - \mathcal{L}_N^{opt}$, which is also true for loss functions greater than $\mathcal{L}_N^{opt}$ (with non optimal $f \propto \frac{p(x_i|c_t)}{p(x_i)}$). "Equation (15) quickly becomes more accurate as N increases. At the same time $\log(N) - \mathcal{L}_N$ also increases, so it's useful to use large values of N" [1].

1.3.4 Algorithm

The original CPC Loss formulation can be seen in Figure 7a.

Figure 7: "Given a set $X = x_1, \ldots, x_N$ of N random samples containing one positive sample from $p(x_{t+k}|c_t)$ and $N-1$ negative samples from [...] $p(x_{t+k})$, we optimize":

$$L_N = -\mathop{\mathbb{E}}_{X} \log \left[\frac{f_k(x_{t+k}, c_t)}{\sum_{x_j \in X} f_k(x_j, c_t)} \right]$$

$$:= -\mathop{\mathbb{E}}_{X} \log \left[\frac{\exp\left(z_{t+k}^T W_k c_t\right)}{\sum_{x_j \in X} \exp\left(z_j^T W_k c_t\right)} \right]$$

(a) InfoNCE Loss from [1]

However we think that this formulation can be misunderstood once it has to be implemented and hence give a mathematical formulation more along the lines of the second paper [25]. Note that this version uses every sample $x_i \in X$ once as a positive sample: algorithm 1

Algorithm 1: InfoNCE Loss calculation

Data: A set of N *full* data-samples $X = (x_1, \ldots, x_N)$.

/* X can e.g. be a set/batch of patients' ECG records with size $N \times channels \times datalength$ */

Input: An index $t \in \mathbb{N}$.

/* t marks the "present" timestep and splits data into "past" and "future". */

Input: The encoder network $g_{enc}(x_i|\theta) \mapsto z_i = (z_{i,1}, \ldots, z_{i,t}, z_{i,t+1}, \ldots, z_{i,t+k})$

/* z_i is a set of $t+k$ latent vectors, representing consecutive timesteps of the input data in a lower dimension. How many get produced is dependent on the encoder network and data samples $x \in X$ */

Input: The context network $g_{ar}((z_{i,1}, z_{i,2}, \ldots, z_{i,t})|\psi) \mapsto c_{i,t}$

/* $c_{i,t}$ is the context vector for a set of t latent vectors, for data-sample x_i */

Input: A set of n weight matrices $\{W_1 \ldots W_n\}$

/* The weight matrices are used to calculate a set of n latent vectors $\hat{z}_i = (\hat{z}_{i,t+1}, \ldots, \hat{z}_{i,t+n}) := (W_1 c_{i,t}, W_2 c_{i,t}, \ldots, W_n c_{i,t})$ */

Result:

$$Loss_i := -\frac{1}{k}\sum_{k=1}^{n} \log\left[\frac{\exp(\hat{z}_{i,t+k}^T z_{i,t+k})}{\exp(\hat{z}_{i,t+k}^T z_{i,t+k}) + \sum\limits_{(j,l)\in\{(j,l)|j\neq i \wedge l\leq|z_i|\}} \exp(\hat{z}_{i,t+k}^T z_{j,l})}\right]$$

/* $Loss_i$ is the scalar loss value for a single positive data-sample x_i. This is close to the papers definition. */

$$Loss := \frac{1}{N}\sum_{i=1}^{N} Loss_i =$$

$$= -\frac{1}{kN}\sum_{i=1}^{N}\sum_{k=1}^{k} \log\left[\frac{\exp(\hat{z}_{i,t+k}^T z_{i,t+k})}{\exp(\hat{z}_{i,t+k}^T z_{i,t+k}) + \sum_{(j,l)\in\{(j,l)|j\neq i \wedge l\leq|z_i|\}} \exp(\hat{z}_{i,t+k}^T z_{j,l})}\right]$$

/* $Loss$ is the scalar loss value where each data-samples $x_i \in X$ is the positive sample once. */

Note that the set of negative latent samples $A := \{(j,l)|j \neq i \wedge l \leq |z_i|\}$ can be replaced by e.g. $A_1 \subset \{(j,l)|j \neq i \wedge l \leq |z_i|\} \leftrightarrow A_1$ *is a random subset of* A, if there are too many negative samples for the hardware to handle, or by e.g. $A_2 := \{(j,l)|j \neq i \wedge l = t+k\} \leftrightarrow A_2$ *is only using negative examples that share the same timestep as* $z_{i,t+k}$, to provide samples that might be more similar to the positive one.

Furthermore we provide an algorithm in Python pseudo code from our understanding, to further explain the loss function and clear up potential confusion. Listing 1 shows this pseudo code for a naive implementation of the formula we provided in algorithm 1 (with negative samples that stem from other batches but the same timestep).

```
def cpc_loss_naive(X:Tensor, W:List[Tensor], enc:nn.Module, rnn:nn.Module):
   batch, channels, data_length = X.shape #eg. (64 x 12 x 4500)
   #Encode data into latents:
   latent_matrix = enc(X)
   batch, timesteps, latent_size = latent_matrix.shape #eg. (64 x 27 x 128)
   #Calculate context from latents:
   context_matrix = rnn(latent_matrix)
   batch, timesteps, context_size = context_matrix.shape #eg. (64 x 27 x 256)
   #Select current timestep (here last possible):
   n=len(W)
   current_timestep = timesteps-n-1
   last_context_vector = context_matrix[:, current_timestep, :]
   #output shape: (64 x 256)
   loss = 0.
   for i in range(batch):
      loss_i = 0.
      for k in range(n):
         #Predict latent for specific timestep k from current context:
         latent_vector_pred_i = last_context_vector[i] @ W[k]
         #Select latent vector for batch i at correct timestep:
         latent_vector_i = latent_matrix[i, current_timestep+k+1, :]
         #calculate similiarity with dot product:
         sim = latent_vector_pred_i @ latent_vector_i #scalar
         #calculate exp(sim) for softmax
         p_xi_ct = exp(sim)
         nominator = p_xi_ct
         denominator = 0.
         for j in range(batch): #replace j with j, l for different sampling
            latent_vector_j = latent_matrix[j, current_timestep+k+1, :]
            #Calculate dot product to measure similarity:
            sim = latent_vector_pred_i @ latent_vector_j #scalar
            #Calculate exp(sim) for softmax
            p_xj_ct = exp(sim)
            denominator += p_xj_ct
         #take log for log softmax
         loss_i += log(nominator/denominator)
      loss += -loss_i/n
   return loss / batch
```

Listing 1: Naive CPC Loss Pseudo Code

```
def cpc_loss_fast(X:Tensor, W:List[nn.Module], enc:nn.Module, rnn:nn.Module):

   batch, channels, data_length = X.shape #eg. (64 x 12 x 4500)

   latent_matrix = enc(X)
   batch, timesteps, latent_size = latent_matrix.shape #eg. (64 x 27 x 128)

   context_matrix = rnn(latent_matrix)
   batch, timesteps, context_size = context_matrix.shape #eg. (64 x 27 x 256)

   timesteps_out = len(W) #eg. 12
   current_timestep = timesteps-timesteps_out
   last_context_vector = context_matrix[:, current_timestep-1, :] #shape: (64 x
     256)
   loss = 0.
   accuracy = 0.
   for k in range(timesteps_out):
      #predict latents for timestep k and all items in batch:
      pred_latent = W[k](last_context_vector) #shape: (64 x 128)
      #calculate similarity between all encoded latents against pred_latent:
      sim = latent_matrix @ pred_latent.T #shape eg.: (64 x 27 x 64)
      #reshape result for easier/efficient indexing:
      sim_r = sim.reshape(timesteps*batch, batch).T #shape eg.: (64, 1728)
      #Select the indices where pred_{i,k}==latent_{i,k}
      labels = torch.arange(batch) + batch*(current_timestep + k)
      #pytorch: crossentropyloss = LogSoftmax + NLL_Loss
      loss += crossentropyloss(sim_r, labels)
      #Count how often the max softmax value is at the correct idx
      accuracy += (sim_r.max(dim=-1).indices == labels).sum()

   loss = loss / (timesteps_out*batch)
   accuracy = accuracy / (timesteps_out*batch)  # * pred_latent.shape[0]
   return accuracy, loss, hidden

```

Listing 2: Optimized CPC Loss Pseudo Code. Uses CrossEntropyLoss from Pytorch

2 Implementation

In this section we describe our different models and their implementational details. All code is available at our github repository[3].

2.1 CPC

Since CPC is a “self-supervised” architecture, there are different parts of the network that are used according to the current task. The pretraining architecture without labels is described in Subsubsection 2.1.1. The downstream training architecture is described in Subsubsection 2.1.2.

2.1.1 Unsupervised

CPC was implemented following the original paper [1] with focus on the section "3.1 Audio", which uses CPC as a speaker and phone classification method on the librispeech dataset. Since audio data is also sequential and thus comparable to ECG data, the architectural details were a good starting point. We adopted the encoder network with only small changes, which means we use 5 convolutional layers with kernel-sizes $[10, 8, 4, 4, 4]$, strides $[5, 4, 2, 2, 2]$ and ReLU activations in between, directly on the input data. The amount of channels is set to 128 after the initial 12. This is different from the original paper [1] where 512 "hidden units" are used instead. This means that each latent vector has the length $l_{size} = 128$. The latents are calculated “in one go”, which means for this architecture and a data input with dimensions $batch \times 12 \times 4500$, a latent matrix sized $batch \times l_{size} \times t_{total} = batch \times 128 \times 26$ is calculated. The total number of latent timesteps $t_{total} = 26$ is directly given by the architecture and input data length. The transposed encoder output/latent matrix is fed into a GRU model with a hidden state dimension of 256 to form the models context for each of the columns in the latent-matrix, resulting in the context-matrix of shape $batch \times 26 \times 256$. In theory, since a recurrent network is used for its calculation, each context vector in the context-matrix can represent a summary or state of the prior latent vectors. From a selected column t in the context-matrix, $t_{out} = 12$ individual linear/fully-connected layers are used to predict $t_{out} = 12$ latent vectors into the future. The value of t should not be selected to low, as a meaningful context, from which can be predicted correctly, requires multiple latents. In our case, we select the latest possible context vector $t = t_{total} - t_{out} = 26 - 12 = 14$th timestep, which is also the most interesting context entry, since it encodes all seen latents. Note that you can also make the $t_{out} = 12$ predictions from every possible column $\{t | t_{in} \leq t \leq t_{total} - t_{out}\}$ in the context matrix in one forward pass, to make more use of the input data, however this comes at the cost of training time. t_{in} eg.$= 12$ is newly introduced additional hyperparameter which defines how many latent vector are at least used to encode the context. A randomly selected timestep from the set $\{t | t_{in} \leq t \leq t_{total} - t_{out}\}$ is another possibility which we used.

Now with the encoded and predicted latent values only, the CPC-loss can be drawn.

[3] https://github.com/Lullatsch/ecg-cpc

Like in the introduction already mentioned the CPC loss uses a modified noise contrastive estimation loss called *InfoNCE*. Here the negative samples are other ECG files drawn from the batch. We differentiate between the latent sets $A = \{z_{j,l} | j \neq i \wedge l \in [1, |z_j|]\}$ (all latents including past and future from other patients) and $A_2 = \{z_{j,l} | j \neq i \wedge l = t + k\}$ (only latents from other patients at the same timesteps) as negative examples. The details on the loss are explained in algorithm algorithm 1, Listing 1, Listing 2 or in Subsubsection 1.3.1.

2.1.2 Supervised

For the downstream task, the InfoNCE-loss function is not used anymore. Nevertheless the whole network that calculates latent-, context- or future latent-vectors will still be used. In the original paper [1], a single linear layer is used on the last context vector in conjunction with a multi-class linear logistic regression classifier to predict the data labels. We applied multiple different architectures to the models output; for a full list see section Subsubsection 2.2.5

2.2 CPC with modifications

Apart from the model described above, additional architectures similar to the CPC audio architecture were examined with changes to different parts of the network.

2.2.1 Loss calculation changes

2.2.1.1 Latent sampling As already mentioned (see right sum in denominator of algorithm 1) there are multiple possibilities as to what negative samples (other latent vectors) are compared to the "current" latent prediction: $N = batch\ size, t = total\ timesteps, t_{out} = number\ of\ predicted\ timesteps$

same uses all latent vectors from the same timestep as the predicted latent vector in the batch. The model has thus to recognize one value under N values for each predicted timestep, for all items in the batch $= N^2 \cdot t_{out}$ total guesses.

future uses all latent vectors from future timesteps in the loss calculation. The model has to recognize the correct latent value under $N \cdot t_{out}$ values for each predicted timestep, for all items in the batch $= N^2 \cdot t_{out}^2$ total guesses.

all uses all latent vectors from all available timesteps in the loss calculation. The model has to recognize the correct latent value under $N \cdot t$ values for each predicted timestep, for all items in the batch $= N^2 \cdot t \cdot t_{out}$ total guesses. In the results this method is the same as crossentropy which just uses a slightly different implementation and is always used in conjunction with "Fewer Latent Vectors" (Paragraph 2.2.1.2).

2.2.1.2 Fewer latent vectors One handicap we came across during training is that with many latent vectors, which might get generated for long input sequences or an encoder architecture with a lower downscaling factor, the loss calculation takes a long time because the model has to calculate big dot product matrices and the softmax afterwards, which is a costly operation. Furthermore recurrent architectures tend to consume large amounts of memory for long input sequences which even made training impossible altogether in some cases.

As a solution we limited the latents which are to be used in the loss calculation to a fixed number, instead of trying to use all latent vectors. During Downstream training, all data is used again, which allows for a more representative context matrix and is important for correct classification. The network is called `cpc_intersect_manylatents` (because it "deals with" many latents), as opposed to the standard architecture `cpc_intersect`.

2.2.1.3 Normalized latent vectors We test normalizing the latent vectors z and $\hat{z}$ to unit vectors (length is 1), before their multiplication in the loss to receive scalar values in range [-1, 1], instead of an open interval. As a result we can observe the following properties:

1. If $z \cdot \hat{z} = 1 \implies z$ and $\hat{z}$ are parallel and point in the same direction.
2. If $z \cdot \hat{z} = 0 \implies z$ and $\hat{z}$ are orthogonal.
3. If $z \cdot \hat{z} = -1 \implies z$ and $\hat{z}$ are parallel and point in the opposite direction.

We justify this change because higher valued latent vectors that are not calculated from the current context, but still point in the generally same direction, might be falsely classified as the positive sample. Note that during downstream tasks, we tested both to return the un-normalized calculated latents and the unit vectors, where returning the normalized vectors worked better by a margin. The loss function thus becomes:

$$Loss_i := -\frac{1}{k}\sum_{k=1}^{n}\log\left[\frac{\exp(\frac{\hat{z}_{i,t+k}}{\|\hat{z}_{i,t+k}\|}^T \frac{z_{i,t+k}}{\|z_{i,t+k}\|})}{\exp(\frac{\hat{z}_{i,t+k}}{\|\hat{z}_{i,t+k}\|}^T \frac{z_{i,t+k}}{\|z_{i,t+k}\|}) + \sum\limits_{(j,l)\in\{(j,l)|j\neq i\wedge l\leq|z_i|\}} \exp(\frac{\hat{z}_{i,t+k}}{\|\hat{z}_{i,t+k}\|}^T \frac{z_{j,l}}{\|z_{j,l}\|})}\right] \tag{18}$$

To circumvent an erroneous division by 0 and the loss becoming undefined, we clip all vector lengths to be in the interval $[\epsilon, \infty]$, where ϵ is a constant scalar (we chose $1e-6$):

$$\|z\|^{\cdot} = \begin{cases} \|z\|, & \text{if } \|z\| > \epsilon \\ \epsilon, & \text{else} \end{cases}$$

However in practice our model scores in the result section got slightly downgraded by normalizing the latent vectors. We suspect that although having access to the properties of comparing unit vectors, this change also restricted the model, hurting classification accuracy.

2.2.2 Encoder networks

The encoder architecture from the paper produces 26 latent vectors for data with length 4500. There one cannot control how many latent vectors get produced, besides changing the encoder architecture or varying data length. Since each latent vector is calculated from at least 465 datapoints, which can be calculated by transposing the convolutions in the architecture, and each vector represents a window shifted by $5{\cdot}4{\cdot}2{\cdot}2{\cdot}2 = 160$ (product of strides, or the downsampling factor given in the original paper), this is equal to an overlap of $\frac{465-160}{465} = \frac{305}{465} \approx \frac{2}{3}$ in the data for neighboring vectors, making the exploitation of local smoothness in the next step prediction likely. Furthermore we would like to point out that the statement from the paper: "The total downsampling factor of the network is 160 so that there is a feature vector for every 10ms of speech", is ambiguous, since it conveys the impression that one latent vector is being created from 10ms of audio. In reality one latent vector is being calculated from 465 data points $\frac{465}{16000hz} \cdot 1000 \approx 29ms$ rather than from $\frac{160}{16000hz} \cdot 1000 = 10ms$. It would be more correct to say that one latent vector is calculated from $\frac{465}{16000hz} \cdot 1000 \approx 29ms$ of audio, with an overlap of $\frac{305}{16000hz} \cdot 1000 \approx 19ms$, creating an additional non disjoint latent vector after every $\frac{160}{16000hz} \cdot 1000 = 10ms$.

2.2.2.1 Data sliding window We compare the original "intersecting" model to a different encoder network that encodes the data by breaking it into multiple non-overlapping windows and then encodes each separately. This ensures that the autoregressive model has to predict future latent vectors which are not calculated from partly identical data and thus have to learn non-local features that do not exploit "local smoothness". Additionally the differentiation between positive and negative samples should be a lot harder too, improving the latent representations. In detail that means an input array of size $batch \times channels \times length$ is split into a fixed number e.g. $M \times batch \times channels \times \frac{length}{M}$ of non-overlapping windows, which are then encoded into $M \times batch$ latent vector. The model is called "strided" in the results and `cpc_encoder_as_strided` in the code, where it acts as a module wrapper applicable to every encoder network.

One downside is that the window-size $\frac{length}{M}$ needs to be fixed or more specifically for an input with dimensions $batch \times channels \times \frac{length}{M}$ the architecture needs to produce exactly $batch \times latentsize \times 1$ values. This can be easily solved with mean pooling or similar methods but not relying on those techniques may yield more accurate results. Furthermore, using the same encoder as in the original architecture, due to no data overlap, we produce less latent vectors which becomes a problem for small data lengths. Using different encoders that can deal with smaller input sizes than 465 would solve this data restriction.

2.2.2.2 Baseline-like Network To evaluate whether potential downstream task performance differences were stemming from architecture distinctions rather than the special CPC pretraining, we used the good working baseline architecture `baseline_v8` (explained later) as an encoder network, called `cpc_encoder_likev8`. It comes with the property of reducing less values from the input data, making the output larger which can be useful if the data contains many information or if a larger output is needed for following tasks. One difficulty that arises due to the smaller downsampling factor in comparison to the standard cpc encoder, is that more latent vectors are created, which

in turn leads to heavily inflated training durations, mostly because of the latent sampling method. Additionally recurrent architectures sometimes fail to cope with very long input sequences. The issue was solved by combining this encoder network with a different architecture that uses a fixed-size subset of latent vectors, already explained in Paragraph 2.2.1.2, returning in fewer latents during the loss calculation.

2.2.3 Context prediction networks

In the original paper a GRU model is used to summarize all latent vectors prior to the current timestep into a context vector. We assume that, in the paper, the output for each latent vector is meant with "model output" and took the last output as the context. In contrast to that we implemented a network which uses the hidden state as context and predicts the future latent values from there. Since the hidden state is internally used to calculate the autoregressive model output, it should contain a lot of information about past latents, which makes it a good candidate for next-step latent prediction as well. It is called `cpc_autoregressive_hidden` in the code.

2.2.4 Latent prediction networks

Originally CPC uses a fixed number of simple weight matrices/ single linear layers to predict future latents from the current context. Next to this predictor which internally uses 12 linear layers to predict 12 latent vectors, called `cpc_predictor_v0`, we additionally implemented a predictor that completely eliminates the use of an autoregressive context network by directly predicting future latent vectors from past latents. For this we used a Temporal Convolutional Network [2], which is a convolutional many-to-many sequence architecture we also used as one of our baselines (see Baseline Subsubsection 2.3.2). As a result, no pretrained context matrix is available in downstream-training, but we relief the model from first summarizing latent values and then secondly applying a number of linear layers synchronously.

2.2.5 Downstream task networks

For each ECG sample, CPC calculates latent vectors with the encoder model, and a context vector with the autoregressive model. Although both can be used for classification later on, it is easier to use the context vector because only one is getting calculated no matter the ECG signal length. Nevertheless if we use the latent vectors during downstream training, to account for varying dimensions, latents have either been flattened, summarized with a new autoregressive architecture, or pooled to obtain a single dimension for prediction with linear layers. The output of the networks is then activated with a sigmoid function. We differentiate between multiple network-types:

- Single Linear Layer + Pooling In the original paper, to test the capabilities of CPC, a single linear layer was used on the latent vectors. However since our input size might change we used a pooling operation to condense multiple latents into one

feature vector. Afterwards we apply a linear layer to the output. Because the context vector was used to predict future latents, it should contain enough information about the data for classification. As a result we can use the context vector as a feature vector too, which we do by feeding it to a linear layer. If both latents and context are desired, we add both together to form the output, before a sigmoid activation is used. This model is called `cpc_downstream_only` in the results.

- Single Conv. Layer + RNN Following works that omit linear layers completely, we also test a single convolutional layer that predicts the output classes in the channel dimension. If latents are used in the prediction we summarize latents with an additional GRU with hidden size 256. If both latents and context are used in downstream training the values are concatenated, instead of added. (`cpc_downstream_cnn` in the results.)

- Linear Layer + Pooling In this network we make class prediction for all latents with a single linear layer. To obtain the final result we simply take the maximum value across the data dimension for each of the separate classes. If the context is used a single linear layer is used on the context vector. If both latents and context are used we simply add both outputs together before calculating the sigmoid activation. The model `cpc_downstream_latent_maximum` uses the maximum pooling operation. The idea behind this network is that each latent might include some class that is supposed to be propagated into the final result with the maximum pooling, no matter how often the class appears. The model `cpc_downstream_latent_average` applies an average pooling operation instead, which makes classes that appear in the data at multiple locations more likely to be considered true in the final probability output.

- 2 Linear Layers + RNN If the predicted classes are not linearly separable from context and latents, additional layers have to be used. We examined two stacked linear layers with ReLU activation in between. Again, when latents are used, they are summarized with a GRU network with hidden size 256. If both context and latents are used they are stacked together and then forwarded through the two linear layers. (`cpc_downstream_twolinear` in the code)

- 2 Linear Layers + Max Pool In addition to the 2 linear layers above, we also used an identical network with the difference that latents are maximum pooled instead, and latents and context are added instead of stacked, after they have been feed through the linear layers. (`cpc_downstream_twolinear_v2` in the code)

We decided against using deeper, more powerful models, as we want to focus on the latent representations learnt during pretraining, rather than finding the theoretically best performing model in general.

2.3 Baselines

We implemented different fully supervised models to compare the CPC-models' performance on the ECG data.

2.3.1 Custom baseline architectures

Multiple hand crafted fully supervised convolutional architectures were examined, each with slight modifications to test for the best performance.

The different baseline architectures were build to have different sets of the following parameters:

Number of Convolutions: Two up to six stacked layers of one dimensional convolutional layers have been tested

Use of BatchNorm: We tried if including a BatchNorm[26] layer after each Convolution helps training and found that it slightly sped up convergence but since we trained for a fixed set of epochs this benefit was negligible

Strides: We found that using strides > 1 in early network layers improved accuracy and lead to better performance in most networks.

Pooling: Different Pooling operations have been tried either between all layers, or at the end of the network. Namely Global Average Pooling (AdaptivePooling with output size 1), Maximum and Average Pooling with different kernelsizes.

Dilation: We used dilation in different layers. As argued in the TCN paper [3] dilated convolutions can build big receptive fields in a short number of layers (receptive field is exponential to number of layers) making it ideal for our ECG data.

Kernel size: Different kernel sizes were experimented with ranging from 3 to 12 (or 1 in special cases). We observe that using bigger kernelsizes in early layers helped accuracy.

Final output Layer: We applied a linear layer to the convolutional network output (filter output map), in order to calculate scores for all the diagnostic classes. However multiple approaches have been taken to reshape the network output of shape $(N, C, L) := (batch \times convchannels \times datalength)$

1. Flatten the tensor to receive a shape of $(N, C \cdot L)$
2. Take only the last value in the data dimension to receive a shape of $(N, C, 1)$
3. Use a global pooling (mean or max) operation to receive a shape of $(N, C, 1)$
4. Use an additional convolutional layer with L input channels, an output channel of 1 and a kernel size of 1, on the last dimension of the transposed data to receive a shape of $(N, C, 1)$
5. Use a recurrent architecture to summarize the values in the last dimension by outputting a vector at each step and taking the last as context, to receive a shape of $(N, C, 1)$

Although 1 seems like a good idea because no information is lost, the accuracy was low when L was large and the weight matrix for the linear layer gets substantially bigger, resulting in longer training times, higher memory requirements and a harder training process. Option 2 is the least taxing operation, however a lot of information is lost in the

process which also reflected in a lowered accuracy. Option 3 does not introduce new parameters and worked approximately as good as option 2. However the bigger the filter output map, the worse the network performed in general. Option 4 performed the best by a margin, but also introduced new parameters since a new weight matrix for the filter has to be learned. However we opt to still use this option because it is more expressive and can even learn both option 2 or 3, if desired. Furthermore convolutions with a 1x1 kernel (1 in our case because of data dimension) are widely used as a downsampling measure and are established in many models. Option 5 can be compared to the CPC architecture, however when trained supervised end-to-end this method did perform poorly. We suspect that the partly large values for L make it difficult to train recurrent architectures.

While we think that there are many possible approaches to summarizing the last layers output, the best solution might be to choose architectures that obtain small values for L.

Most models follow a general structure:

Multiple one-dimensional Convolution Layers with decreasing kernel sizes and either stride or dilation, get stacked with ReLU after each Layer. Batchnorm is added to some models. The first few layers of the network are supposed to downsample the data using big kernel sizes and strides, reducing memory footprint for the model drastically. After the downsampling layers, additional one-dimensional convolutional layers have been used, this time with a mostly fixed kernel size of 3, low or no strides and no or increasing dilation, which increase the networks receptive field, potentially extracting wide spanning features. The number of filters is increased from 12 to 128 to allow enough features to be learned by the model. The filter number has been kept constant at 128, which is unusual but since we had differing layer numbers we opted for a simple solution. Finally the non-channel dimension has been summarized by a one of the techniques described in Final output Layer. The resulting data of shape (Batch-size, 128, 1) is fed into a fully connected or convolutional layer with sigmoid as activation function, to predict class labels with probability scores $\in [0, 1]$ each. Note that Softmax is not used as the probabilities do not add up to 1 in multi-label classification.

Table 1 summarizes most of the differing model attributes described above. The model names are the same as in the results.

Because the summary table can not fully capture all details of the different models, we will briefly describe our baseline models here, that either do not fit the above general description or get used often:

`BL_v0` 2 one-dimensional convolutinal layers, with kernel-sizes $[7, 3]$, dilations $[1, 3]$ and ReLU activations. The output is summarized by a convolutional with kernel size 1 (option 4).

`BL_v0_2` consists of 5 Conv1D Layers with kernel sizes of $[7, 5, 5, 3, 3, 3]$, dilations of $[1, 1, 1, 1, 2, 4]$, strides $[4, 3, 3, 1, 1, 1]$ and ReLU inbetween. The channel number is dropped down to 32 in the first layer with stride 1, and then increased back up to 64. The idea behind this network is to downsample the input in the first few layers and then extract spanning features with dilations without striding.

`BL_v2` `BL_v2`, consists of 5 Conv1D Layers with kernel sizes of 3 and dilations of $[1, 2, 4, 8, 16]$, the channel size is 128 after the initial 12. BatchNorm and ReLU is

applied after each layer. At the end of each besides the last layer a maximum pooling layer with filter size 3 is used. The output is summarized with a global average pool like option 3 describes.

`BL_v8` Similar to `BL_v2` but smaller, `Baseline_v8` consists of 4 Conv1D Layers with kernel sizes of 3 and dilations of $[1, 2, 4, 8]$, the channel size is 128 after the initial 12. BatchNorm and ReLU is applied after each layer. At the end of each besides the last layer a maximum pooling layer with filter size 3 is used. The output is downsampled in the data dimension like option 4 describes.

2.3.2 Literature architectures

In addition to the custom-designed architectures above, we tested already existing or slightly modified architectures that were reported to work on timeseries classification:

MLP The Multi-Layer-Perceptron is a three linear layer network with dropout and ReLU activations from [27] (`BL_MLP` in results)

FCN The Fully Convolutional Network consists of three convolutional layers with BatchNorm, ReLU and a global pooling operation from [27]. (`BL_FCN` in results)

Multi Scale 1D ResNet A Residual Network which uses one instead of two dimensional convolutions, tuned for timeseries classification [28]. Multiple subarchitectures are used in parallel to allow for modeling close and long distance coherence in sequential data. Each subarchitecture uses different filter sizes and striding to produce local or global features at the same time. (`BL_v14` in results.)

TCN Temporal Convolutional Networks [2] aim to provide an almost fully convolutional sequence-to-sequence alternative to RNNs, which brings perks like low memory requirements, more stable gradients and parallelism (predicting many timesteps at once). Exponentially increasing dilation sizes help in building a big receptive field, theoretically ideal for modeling long distance coherence in sequential data. (`BL_TCN` in results) Since this model's output has the same length as the models input, we had yet again to find a solution to reduce the output length and tested multiple variants: `TCN_down` uses a convolutional layer with filter-size 1 to reduce output. `TCN_last` uses the last value in the output of the data dimension, following one of the examples in the authors github repository [4]

Alexnet The Alexnet is an architecture known for image classification. Here we implemented it with 1D convolutions and two different kernel sizes otherwise following the implementation from pytorch [29]. (`BL_alex_v2` in the code)

Again, Table 1 summarizes most model attributes described above.

[4] https://github.com/locuslab/TCN/tree/master/TCN/mnist_pixel

Model Name	Convolutional Layer Number	Sum of Strides	Sum of Dilation	Sum of Paddings	Sum of Filters	uses BatchNorm	uses Max Pool	uses Adaptive Average Pooling	uses Linear	uses LSTM	Final Layer
BL_alex_v2	5	25	8	7	174	no	yes	yes	yes	no	1
BL_FCN	3	3	3	0	16	yes	no	no	no	no	3
BL_MLP	0	0	0	0	0	no	no	no	yes	no	2
BL_rnn_simplest_lstm	0	0	0	0	0	no	no	no	yes	yes	5
BL_TCN_block	3	4	4	0	7	no	no	no	yes	no	1
BL_TCN_down	15	15	31	56	39	no	no	no	yes	no	4
BL_TCN_flatten	15	15	31	56	39	no	no	no	yes	no	1
BL_TCN_last	14	14	30	56	38	no	no	no	yes	no	2
BL_v0	3	3	5	0	11	no	no	no	yes	no	4
BL_v0_1	3	4	5	0	11	no	no	no	yes	no	4
BL_v0_2	7	14	11	0	27	no	no	no	yes	no	4
BL_v0_3	7	14	11	0	27	no	no	no	yes	no	4
BL_v1	6	17	6	0	36	yes	no	yes	yes	no	3
BL_v2	5	20	36	0	30	yes	yes	yes	yes	no	3
BL_v3	5	5	16	0	13	yes	no	no	yes	no	4
BL_v4	5	5	16	0	13	no	no	no	yes	no	4
BL_v5	5	5	16	0	13	no	no	no	yes	no	4
BL_v6	7	7	64	0	19	no	no	no	yes	no	4
BL_v7	6	7	32	0	20	no	no	no	yes	no	4
BL_v8	5	14	19	0	22	yes	no	no	yes	no	4
BL_v9	5	14	8	0	22	yes	yes	no	yes	no	4
BL_v14	29	71	30	22	143	yes	no	no	yes	no	4
BL_v15	5	14	96	0	26	yes	yes	no	yes	no	4

Table 1: Model attributes summarized. Final Layer number refers to the enumeration Final Output Layer 2.3.1.

3 Training

3.1 Data preprocessing

All data has been resampled to 500hz, if it were not already. The networks are trained on cropped data with length fixed to 4500 (mostly down from 5000), all 12 channels have been used. In rare cases, where the available data was shorter than 4500, we repeatedly padded the rightmost value for each channel. The data was split into training-, validation- and test-set randomly with a fraction of $\frac{7}{10}$, $\frac{2}{10}$, and $\frac{1}{10}$ respectively, while still ensuring that every class follows those fractions approximately. The data distribution is displayed in the introduction in Figure 3.

We count label occurrences and exclude all labels that occur less than 20 times over all datasets, which results in 67 different ECG classes (down from 94). The class/label counts obtained will also be used later as loss function weights in training, where we will differentiate between calculating the loss with and without these weights.

Three different data normalization methods have been tested independently:

min-max The minimum and maximum for each channel was drawn over all datapoints in each channel to rescale the data into the interval $[0, 1]$. This means for data X of shape 12×4500, 12 different minima and maxima are drawn. The calculation for each datapoint independently is given here, where $x_{i,j} \in X$, the first index i is the channel and the second index j is the entry in the data dimension:

$$\forall i, j : x'_{i,j} = \frac{x_{i,j} - \min\limits_{d}(x_{i,d})}{\max\limits_{d}(x_{i,d}) - \min\limits_{d}(x_{i,d})}$$

This method is equal to the "Min-Max Normalization" found elsewhere.

min-max (augmented) The minimum and maximum at each datapoint is drawn over all channels to rescale each datapoint into the interval $[0, 1]$. For example this means for data $X$of shape 12×4500, 4500 different minima and maxima are drawn over all channels:

$$\forall i, j : x'_{i,j} = \frac{x_{i,j} - \min\limits_{c}(x_{c,j})}{\max\limits_{c}(x_{c,j}) - \min\limits_{c}(x_{c,j})}$$

This has the effect that each datapoint $x_{i,j}$ captures the relationship between all channels. We are aware that this changes the input data in a drastical manner and originally we applied this normalization method by mistake. Nevertheless the models were mostly able to train successfully and since all models had to train with the handicap, the results are still interesting to look at.

z-Score Z-Score Normalization shifts the data by substracting the mean from each point and rescales it afterwards by dividing by the standard deviation.

$$\forall i, j : x'_{i,j} = \frac{x_{i,j} - \text{mean}(x_i)}{\text{std}(x_i)}$$

As a result the normalized data has a mean of 0 and a standard deviation of 1 , which is said to improve model training stability. This method resulted in the overall best model performances, especially for the baseline architectures. We used it almost exclusively for all models, besides in our experiment Subsubsection 5.1.5.

Principle Component Analysis has been tested but did not yield good results, feature (heart beat) extraction as in [30] has been applied but random samples suggested big variations in the extracted beat's quality. This is to say that multiple beats got extracted as one, longer beats got cut short etc. Furthermore beat extraction works best on beats that follow the normal pattern, however the most interesting ones for classification might differentiate enough to not be detected anymore and thus introduces a potentially harming bias, where interesting beats get discarded with a higher probability. Also keep in mind that data labels only exist for the whole file where multiple heart beats are present, meaning each beat on its own cannot be correctly labeled with high confidence.

In addition to the beat extraction method from [30] we briefly tested outlier detection based on [31] to find possibly interesting candidates for classification of single beats, but the results were varying widely between different channels and datasets. Furthermore we raise the same concern as above about missing out on beats that are not getting classified as outliers.

3.2 Train settings

Adam was used as optimizer with the parameters set to $lr = 3e-4, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 1e-8, weight\_decay = 0$, all values were not changed for the most part. For models trained with z-score normalization, we used a learning rate scheduler with an initial learning rate of $3e-3$ for pretraining and $3e-4$ for downstream training, which multiplies the learning rate by a factor of 0.1 after 5 epochs with no validation loss decrease.

We used a batch-size of 128, the loss function is Binary Cross Entropy with or without the class weights multiplied to the loss (obtained in Subsection 3.1).

To allow a fair comparison between CPC and the baselines, we give access to the data for 120 epochs in total. Our baselines that utilize no pretraining can thus train for unrestricted 120 data iterations — while CPC needs sufficient time to pretrain in order for the loss to be minimized, we chose 100 epochs for the pretraining without labels and 20 epochs with labels for training on the downstream (classification) task. Since trivially 120 epochs with labels should be better or equal than 100 epochs without labels plus 20 epochs with labels, we know that a possible performance edge for CPC is achieved due to its properties and not because it has access to the data for longer. The split 100+20 was obtained by looking at CPC loss curves, where we focused on decreasing the pretraining loss sufficiently. However we want to mention that this specific allocation is not a given and performance might increase with more downstream- and less pretrain epochs. With more GPU time, a training until convergence for all models could possibly be a better alternative. In some experiments we had to downstream train our CPC models for up to 40 epochs which we will denote where applied.

3.2.1 CPC

As already mentioned The CPC architecture from the paper was trained for 100 epochs on the data without labels to find the optimal weights for the encoder network. After the 100 epochs, we trained the network with the added downstream layers for the downstream task, for 20 epochs with labels available.

We differentiate between two downstream training settings:

1. All weights are updated (Gradients for both encoder architecture and classification layers are calculated and updated).
2. Only weight gradients for the classification layer(s) are updated. (All weight gradients in the CPC network remain untouched aka. weights are “frozen”).

We make this differentiation because we want to better evaluate the CPC pretraining step, which is very important when weights are "frozen".

4 Evaluation

4.1 CPC Evaluation during Pretraining

As an addition to the CPC loss value, we evaluate how well CPC is able to differentiate between positive and negative latent vectors during training. For that we take the maximum value indices in the logsoftmax matrix and compare those to the index with the correct latent vectors. If the indices match we add one to the count of correctly classified latents (see line `28` in Listing 2). In the end we divide the sum by the number of total predicted latent vectors to receive values $\in [0, 1]$ (see line `31` in Listing 2).

4.2 Model Evaluation during Downstream Training

As an addition to the loss value we calculate metrics during downstream training.

To better evaluate performance on a sparse probability label vector, which means only one to five values of 67 classes (in case of all datasets combined) are set to something other than 0.0, the following accuracy functions have been used during training to evaluate progress. They are based on the widely used Mean Squared Error/Brier score [32], but are split into parts which measure different things:

Given the class label probabilities $Y = (y_1, \ldots, y_i, \ldots, y_n)$ the prediction probabilities $\hat{Y} = (\hat{y}_1, \ldots, \hat{y}_i, \ldots, \hat{y}_n)$ with $y_i \in [0, 1]$ and $\hat{y}_i \in [0, 1]$

$$\mathbf{zerofit}(Y, \hat{Y}) = 1 - \frac{\sum_{i,\mathbf{where}\ y_i=0} (y_i - \hat{y}_i)^2}{|\{\forall i, \text{where } y_i = 0\}|} \tag{19}$$

$$\mathbf{classfit}(Y, \hat{Y}) = 1 - \frac{\sum_{i,\mathbf{where}\ y_i\neq 0} (y_i - \hat{y}_i)^2}{|\{\forall i, \text{where } y_i \neq 0\}|} \tag{20}$$

$$\mathbf{meanfit}(Y, \hat{Y}) = \frac{\mathbf{zerofit}(Y, \hat{Y}) + \mathbf{classfit}(Y, \hat{Y})}{2} \tag{21}$$

Equation Equation 19 is supposed to capture how well the model reproduces the probability scores for classes labels with probability $= 0$. $\mathbf{zerofit}(Y, \hat{Y}) \in [0, 1]$, where $\mathbf{zerofit}(Y, \hat{Y}) = 0$ is the worst and $\mathbf{zerofit}(Y, \hat{Y}) = 1$ the best score.

Equation Equation 20 is supposed to capture how well the model reproduces the probability scores for classes labels with probability $\neq 0$. $\mathbf{classfit}(Y, \hat{Y}) \in [0, 1]$, where $\mathbf{classfit}(Y, \hat{Y}) = 0$ is the worst and $\mathbf{classfit}(Y, \hat{Y}) = 1$ the best score.

Equation Equation 21 is the mean between Equation 19 and Equation 20 and captures if both goals have been met. Again it holds that $\mathbf{meanfit}(Y, \hat{Y}) \in [0, 1]$.

Since the loss function is independent of the above metrics, the custom accuracy functions are solely a tool to better monitor network performance during training and do not change the weights in any way. They were used in conjunction with the loss values to determine a suitable fixed number of training epochs across all models — the aforementioned 120 epochs for baselines and 20 epochs for cpc networks.

4.3 Evaluation after training

4.3.1 Quantitative

To evaluate our models we calculate `precision`, `recall`, `accuracy`, `ROC-AUC` and `F1`-scores [33] with the prediction probabilities on the test-dataset. Since the models output probability scores for each class and no binary values, which are however needed for most metrics, we need to set thresholds that define whether a class gets classified as True or False for a given datasample:

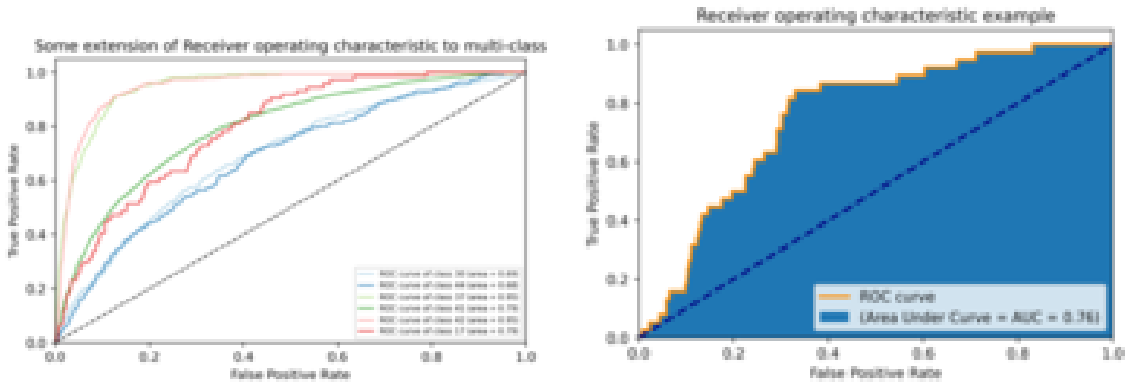
$$class_i := \begin{cases} False, & \text{if } p(class_i) < threshold_i \\ True, & \text{if } p(class_i) \geq threshold_i \end{cases} \tag{22}$$

With the decision boundary for each class set, we can count the "True Positives" = TP, "False Positives" = FP, "True Negatives" = TN and "False Negatives" = FN, which means that the data has been either classified correctly/falsely as positive/negative sample respectively (see table 4.3.1).

		Class is:	
		TRUE	FALSE
Classified as:	TRUE	True Positive (TP)	False Positive (FP)
	FALSE	False Negative (FN)	True Negative (TN)

We select the "best" thresholds based on the Receiver Operating Characteristic-curve for each individual prediction class. The ROC-curve (see Figure 8b) is the False Positive Rate (FPR) plotted against the True Positive Rate (TPR) for every possible threshold in Equation 22 for class i. The best threshold is then selected by evaluating $\underset{i}{argmax}(TPR_i - FPR_i)$. This takes place on the validation dataset and each model has their own optimal decision boundaries for each class.

Figure 8: ROC example



(a) Multiple ROC-curves with different amount of thresholds each.

(b) **A**rea **U**nder **C**urve for The Receiver operating characteristic of a single class.

The TP, FP, TN, FN counts then form the base for the following metrics (see equations 9):

$$
\begin{aligned}
precision_i &:= \frac{TP_i}{TP_i + FP_i} \\
recall_i &:= \frac{TP_i}{TP_i + FN_i} \\
accuracy_i &:= \frac{TP_i + TN_i}{TP_i + TN_i + FP_i + FN_i} \\
TPR_i &:= \frac{TP_i}{TP_i + FN_i} \\
FPR_i &:= \frac{FP_i}{FP_i + TN_i} \\
F_{1,i} &:= 2 \cdot \frac{precision_i \cdot recall_i}{precision_i + recall_i} \\
ROCAUC_i &:= \text{AUC of ROC-curve}
\end{aligned}
\tag{23}
$$

Figure 9: Different score metrics to measure performance. All score functions calculate the score for a given class i.

$$
\begin{aligned}
\mathbf{macroavg}(\mathbf{score}) &:= \frac{1}{N} \sum_i^N \mathbf{score}(TP_i, FP_i, TN_i, FN_i) \\
\mathbf{microavg}(\mathbf{score}) &:= \mathbf{score}(TP, FP, TN, FN) \\
&= \mathbf{score}(\sum TP_i, \sum FP_i, \sum TN_i, \sum FN_i) \\
&\text{For example:} \\
\mathbf{macroavg}(TPR) &:= \frac{1}{N} \sum_i^N \frac{TP_i}{TP_i + FN_i} \\
\mathbf{microavg}(TPR) &:= \frac{\sum_i^N TP_i}{\sum_i^N TP_i + \sum_i^N FN_i}
\end{aligned}
\tag{24}
$$

Figure 10: Micro- and Macro-average calculations for a given score function. Example for TPR at bottom

To obtain comparable results we also calculate *micro*- and *macro*-average scores (see equation 10). The *macro*-average sums the metrics in Figure 9 for every individual class and averages by dividing through the number of classes. That way every class weighs the same in the resulting average, no matter how many samples that class has. The *micro*-average can be calculated by summing the TPs, FPs, TNs and FNs for all classes respectively and then calculate the metrics in Figure 9, or by weighting the calculated class scores by their fraction of the total number of samples. The micro average thus weights the score metrics with the underlying class distribution. It is important to look at both micro and macro average scores to judge a models performance — high micro average scores but low macro average scores suggest that small classes are being overlooked by the model, which is especially disastrous considering fatal illnesses are very rare in most cases. The opposite; a high macro average- but low micro average score suggests that the models accuracy in larger parts of the data is small, which, in our case, would most likely

mean that healthy patients get diagnosed with illnesses on accident.

4.3.2 Qualitative

Apart from giving scores that allow a basic and quick evaluation of all of our models, we show various graphics for selected models to allow a more thorough overview of the models performance. They are explained in their respective section.

Table 2: Baseline Results

model	micro	macro
BL_alex_v2	0.903	0.774
BL_FCN	0.458	0.503
BL_MLP	0.500	0.500
BL_TCN_block	0.876	0.638
BL_TCN_down	0.924	0.806
BL_TCN_last	0.869	0.649
BL_v0	0.916	0.787
BL_v0_1	0.930	0.824
BL_v0_2	0.867	0.664
BL_v0_3	0.895	0.757
BL_v1	0.920	0.814
BL_v2	0.940	0.855
BL_v4	0.930	0.830
BL_v5	0.865	0.668
BL_v7	0.924	0.817
BL_v8	0.937	0.847
BL_v9	0.933	0.844
BL_v14	**0.951**	**0.885**
BL_v15	0.931	0.840

Micro and Macro AUC-ROC scores for all baseline models. Baseline Models are trained on the downstream task for 120 epochs.

5 Results

5.1 Quantitative

5.1.1 Baselines compared to CPC

How does the CPC architecture compare against our baseline models in a fully supervised setting?

Table 2 shows the results for our baseline/literature models. The literature architecture `BL_v14`, which is tuned for timeseries classification by making use of parallel architectures with differently sized receptive fields, performs the best for both micro and macro ROC-AUC scores. This model performed the best compared to all models in general.

The fully connected network `BL_FCN` and the multilayer perceptron `BL_MLP` both failed to learn the data beyond random guessing classes.

The best Temporal Convolutional Network is `BL_TCN_down` which uses a convolutional layer with filtersize 1 to summarize the values in the data dimension.

At this place we also want to mention the `BL_v8` model which performed well and is used as encoder network in selected CPC models later on.

We compare the baseline network results to different CPC models (with the standard encoder-, context- and predictor-network), that were trained fully supervised for 120

Table 3: CPC as Baseline Results

model	Freeze CPC	strided	Downstream Model	micro	macro
CPC	False	False	cpc_downstream_latent_average	0.943	**0.879**
CPC	False	False	cpc_downstream_latent_maximum	**0.944**	0.877
CPC	False	False	cpc_downstream_twolinear_v2	0.933	0.845
CPC	False	True	cpc_downstream_latent_average	0.941	0.872
CPC	False	True	cpc_downstream_latent_maximum	0.941	0.873
CPC	False	True	cpc_downstream_twolinear_v2	0.924	0.831

CPC Models trained without pretraining for 120 epochs (with labels). All models shown use the standard encoder-, context-network

epochs, to obtain their best possible scores within the 120 epoch limit. These results can be seen in Table 3: The models trained without pretraining have overall high average scores, only falling slightly behind `BL_v14` (Multi-Scale-Residual Network), which proves the CPC architecture itself is clearly capable of classifying the data sufficiently in a supervised setting.

The intersected models have a slim edge over strided models (Paragraph 2.2.2.1), most likely because they are less restrictive.

Surprisingly the models that utilized the more expressive two layer linear network fell slightly behind in performance, which we can only explain by a higher training difficulty, as the deeper model should be able to at least match the performance of smaller models. Similar occurrences have been already observed in the past, well explained in the Deep Residual Network paper [34].

5.1.2 CPC Downstream task with frozen weights

How capable is CPC's pretraining? How good are the learned representations on their own?

We want to measure how good the latent vector features are suited in our downstream task, by pretraining the model for 100 epochs without labels and downstream train for only 20 epochs with labels available. After pretraining, we are not updating any weights in the whole CPC network (CPC network is "frozen"). This is to say only the weights in the specific downstream task network (see Subsubsection 2.2.5) are altered during backpropagation. The results in Table 4 show good results, however it is clear that they fall behind in performance compared to the models who had access to the labeled data for 120 epochs. We observe that the two layered linear network is the best performing downstream model, suggesting that the learned latent vectors are not fully linearly separable for this downstream task. The added complexity from introducing a second layer helped classifying the learned latent representations, in contrast to the networks trained purely on the downstream task, where the added complexity lead to lower scores.

Additionally the strided models show lower macro scores.

In comparison we show the results for the same models with randomly initialized

Table 4: CPC ROC-AUC scores, with no CPC layers updated during downstream training

model	Freeze CPC	strided	CPC Sampling Mode	Downstream Model	micro	macro
CPC	True	False	all	cpc_downstream_latent_average	0.899	0.792
CPC	True	False	all	cpc_downstream_latent_maximum	0.910	0.797
CPC	True	False	all	cpc_downstream_twolinear_v2	0.889	0.816
CPC	True	False	crossentropy	cpc_downstream_latent_average	0.912	0.788
CPC	True	False	crossentropy	cpc_downstream_latent_maximum	0.908	0.780
CPC	True	False	crossentropy	cpc_downstream_twolinear_v2	0.872	0.809
CPC	True	False	same	cpc_downstream_latent_average	0.903	0.772
CPC	True	False	same	cpc_downstream_latent_maximum	0.901	0.802
CPC	True	False	same	cpc_downstream_twolinear_v2	0.882	**0.823**
CPC	True	True	all	cpc_downstream_latent_average	0.902	0.766
CPC	True	True	all	cpc_downstream_latent_maximum	0.904	0.779
CPC	True	True	all	cpc_downstream_twolinear_v2	0.900	0.806
CPC	True	True	crossentropy	cpc_downstream_latent_average	0.896	0.715
CPC	True	True	crossentropy	cpc_downstream_latent_maximum	0.896	0.708
CPC	True	True	crossentropy	cpc_downstream_twolinear_v2	0.872	0.793
CPC	True	True	same	cpc_downstream_latent_average	0.905	0.780
CPC	True	True	same	cpc_downstream_latent_maximum	0.903	0.794
CPC	True	True	same	cpc_downstream_twolinear_v2	**0.917**	0.808

ROC-AUC scores for standard CPC models trained with pretraining for 100 and training on the downstream task for 20 epochs (with the CPC weights frozen).

Table 5: CPC ROC-AUC scores, random weight initialization, no CPC layers updated during downstream training

model	Freeze CPC	strided	Downstream Model	micro	macro
CPC	True	False	cpc_downstream_latent_average	0.852	0.514
CPC	True	False	cpc_downstream_latent_maximum	0.861	0.580
CPC	True	False	cpc_downstream_twolinear_v2	**0.864**	**0.582**
CPC	True	True	cpc_downstream_latent_average	0.853	0.510
CPC	True	True	cpc_downstream_latent_maximum	0.852	0.575
CPC	True	True	cpc_downstream_twolinear_v2	0.861	0.577

Table 6: ROC-AUC scores for standard CPC models without pretraining (random weights); trained for 20 epochs on the downstream task (with the CPC weights frozen)

weights and no pretraining whatsoever, as to prove the pretraining is actually relevant and the data is not just classifiable with random weights and a single/two layer(s) within 20 epochs of training time: Table 6.

As expected the models fail to satisfactorily classify the ecg-data, which becomes clear when looking at the macro ROC-AUC scores. The high micro scores suggest that one layer is still enough to learn the big parts of the data somehow, where one has to keep in mind that e.g. the biggest class "Sinus Rythm" is included in $\approx 50\%$ of all ecg-files, which means correctly predicting this class has a big positive impact on micro average scores. (Reminder: see Figure 3 for all classes.) Both table Table 4 and Table 6 together suggest that the features learned by CPC in an unsupervised fashion are useful for the supervised downstream task, even if the network is restricted by making weight updates only in a previously untrained new layer, which fits the extracted features to the classes.

Table 7: CPC ROC-AUC scores, with all layers updated in downstream training

model	Freeze CPC	strided	CPC Sampling Mode	Downstream Model	micro	macro
CPC	False	False	all	cpc_downstream_latent_average	0.911	0.800
CPC	False	False	all	cpc_downstream_latent_maximum	0.914	0.810
CPC	False	False	all	cpc_downstream_twolinear_v2	0.900	0.818
CPC	False	False	crossentropy	cpc_downstream_latent_average	0.898	0.763
CPC	False	False	crossentropy	cpc_downstream_latent_maximum	0.902	0.785
CPC	False	False	crossentropy	cpc_downstream_twolinear_v2	0.887	0.803
CPC	False	False	same	cpc_downstream_latent_average	0.898	0.828
CPC	False	False	same	cpc_downstream_latent_maximum	0.901	0.832
CPC	False	False	same	cpc_downstream_twolinear_v2	0.896	**0.851**
CPC	False	True	all	cpc_downstream_latent_average	0.921	0.837
CPC	False	True	all	cpc_downstream_latent_maximum	0.919	0.821
CPC	False	True	all	cpc_downstream_twolinear_v2	**0.931**	0.845
CPC	False	True	crossentropy	cpc_downstream_latent_average	0.888	0.791
CPC	False	True	crossentropy	cpc_downstream_latent_maximum	0.892	0.792
CPC	False	True	crossentropy	cpc_downstream_twolinear_v2	0.881	0.824
CPC	False	True	same	cpc_downstream_latent_average	0.911	0.837
CPC	False	True	same	cpc_downstream_latent_maximum	0.913	0.828
CPC	False	True	same	cpc_downstream_twolinear_v2	0.905	0.847

ROC-AUC scores for CPC models trained with pretraining for 100 epochs and training on the downstream task for 20 epochs with the CPC weights also updated.

5.1.3 CPC Downstream task with all weights updated

Can the performance be improved by updating the weights in the whole architecture?

To answer this question, Table 7 shows yet another way of training the CPC architecture: pretrain the representations for 100 epochs and then, when training on the downstream task, do not freeze the weights in the CPC network layers. We assume, compared to the frozen model, this would lead to the better overall prediction accuracy, because the CPC pretraining is utilized in order to train the autoregressive architecture and then all weights are fine-tuned to optimize the classification loss.

As expected we observe that most models receive a slight performance boost compared to the "frozen" model variants and again the two layer downstream network has better scores overall. For the models however, that are trained with fewer latent vectors (`cpc_intersect_manylatents`) and use "crossentropy" non-strided, we observe that the complete opposite is the case: While the micro ROC-AUC scores are still in line with the other models, the macro average scores are worse than the models that did not have their weights updated. We assume the cpc objective lead to a specific latent representation that were in an opposite direction of the downstream loss function. With more training epochs, we assume that the model will converge towards the results of Table 3, assuming the model can get out of the suboptimal local minimum. We cannot completely rule out overfitting, but we think it is unlikely in this case as the micro average scores are also lower than before.

5.1.4 Low label availability

Under what circumstances is CPC especially efficient to use?

To further investigate the strengths of representation learning we reproduce an experiment from [25] that omits labeled data in the training process to simulate data scarcity. Since the representations learned by CPC only rely on unlabeled data, a clear advantage for all CPC models compared to the baselines, which cannot make use of unlabeled data, can be expected.

We start by utilizing the old train-, validation- and test-splits and make changes to the training split by taking a fixed fraction of each class. For this experiment, class label proportions of $\{0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1, 0.05, 0.01, 0.005, 0.001\}$ have been desired to achieve. Since the data can have multiple labels per sample, it is difficult or sometimes even impossible to exactly match the desired fractions for all classes, hence why the class distribution will be approximate. Our algorithm uses a greedy approach where all files are put into "class buckets" for each of the files corresponding class. Afterwards a random file is taken from each class bucket ordered by the buckets size, starting with the smallest one. If a file gets randomly drawn, it is removed from all other buckets. This gets repeated until all the desired fractions for each bucket are met. Code can be found in `experiment/create_fewer_labels_data`.

Again, like in the other experiments, we train 120 epochs per baseline model, while CPC trains only for 20 epochs on the labeled data plus 100 epochs on ALL of the original training dataset without the labels.

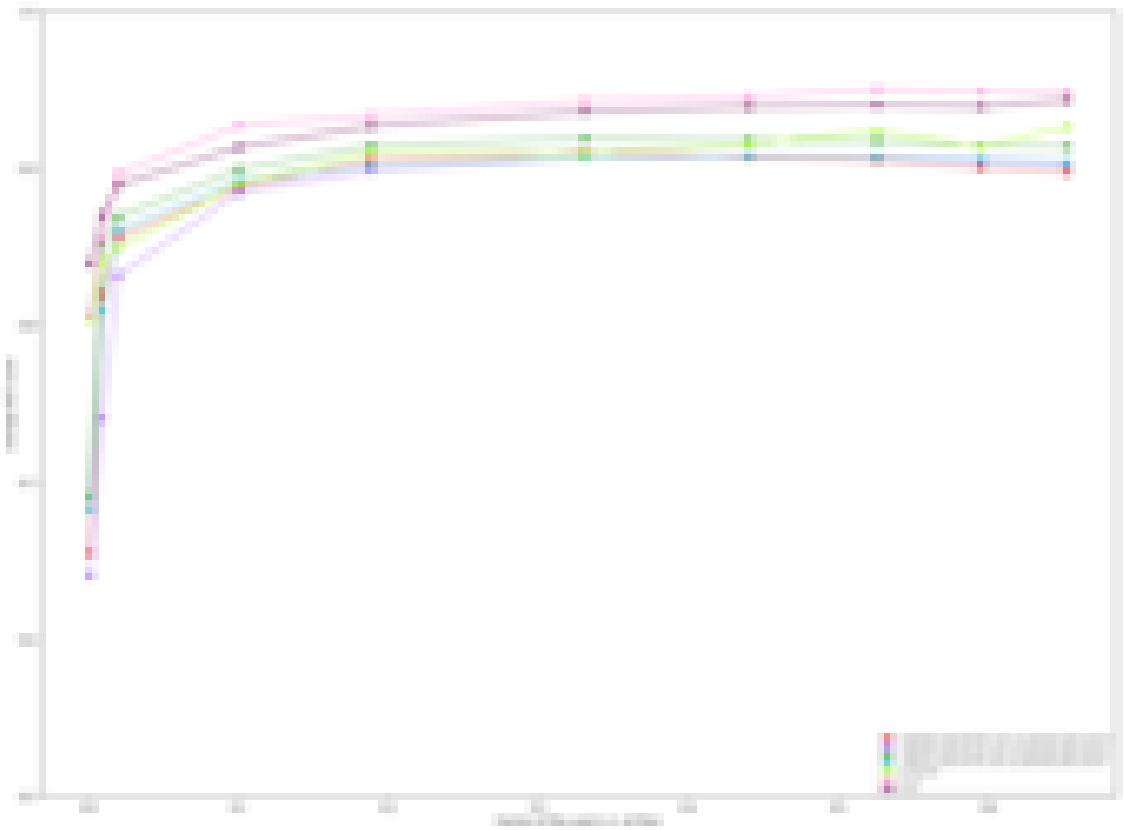
Finally we measure micro and macro ROC-AUC scores on the same test dataset and plot the results. Instead of the desired proportion we use the actual fraction of total data files used as x-axis, because again, multiple classes are possible per sample, and the actual fraction of data used might differ from the desired proportions. The results for 20 epochs on cpc models and 120 epochs on baseline models can be seen in Figure 11. As one can see more data $\equiv$ better score in general, however the CPC model scores are lower than in baseline models, where few labels are given. The lower CPC ROC-AUC score is contrary to what is to be expected since representation learning makes use of the complete dataset without labels first, hence it should perform considerably better in tasks that provide only few labeled data.

To investigate this counter-intuitive behavior we train selected CPC models for 50 epochs instead of 20. We visualize the scores again in Figure 12 and this time see results closer to what we expected, where CPC models have a slight advantage over baseline models in very low label regimes. Interestingly the performance advantage quickly fades as more labels are given, no matter if trained for 20 or 50 epochs. Our hypothesis is that each class has to be shown to the model a specific number of times at least to successfully fit the model to the classification problem. By using less and less labels, while also limiting the CPC network to training for only 20 epochs, this hypothetical threshold, where the loss is not decreased enough for each individual class, must have been reached. We conclude that CPC is able to slightly outperform non-representation learning networks, where only very few labels are available, but needs sufficient epochs to do so.

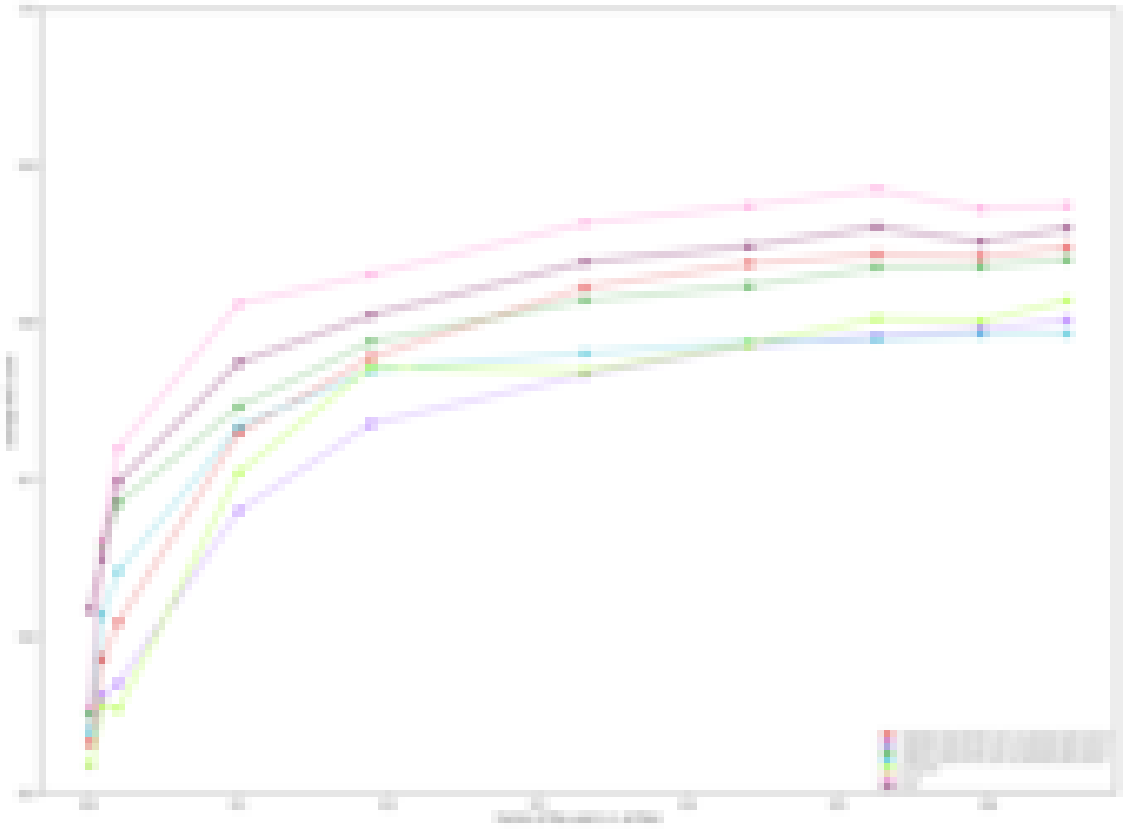
Nevertheless we are astounded how well the baselines performed in general and suspect that more tests on different data were few labels are available should be conducted in the future.

Furthermore this experiment shows one of representation learnings biggest strength:

Figure 11: Test scores for different models trained with a fraction of data labels

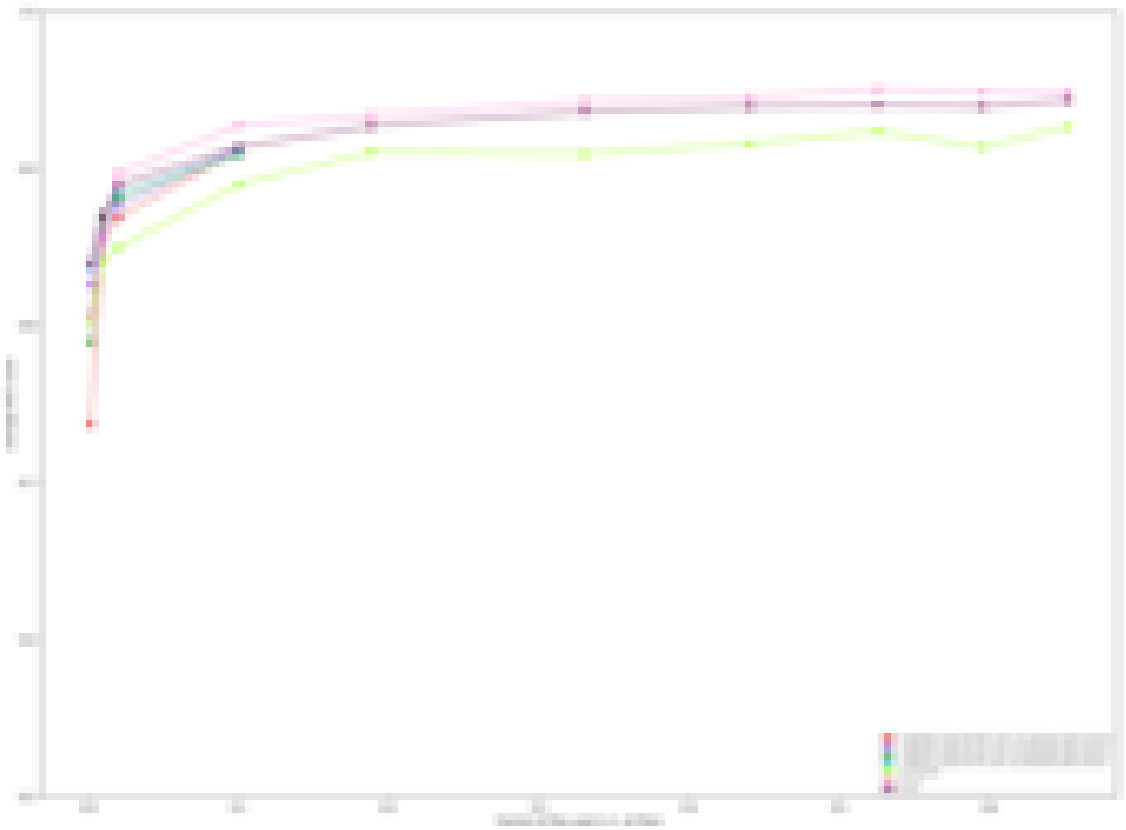


(a) Micro average ROC-AUC score. Dashed Line: CPC models, Full Line: Baselines. Baseline-epochs: 120, CPC-epochs: 20

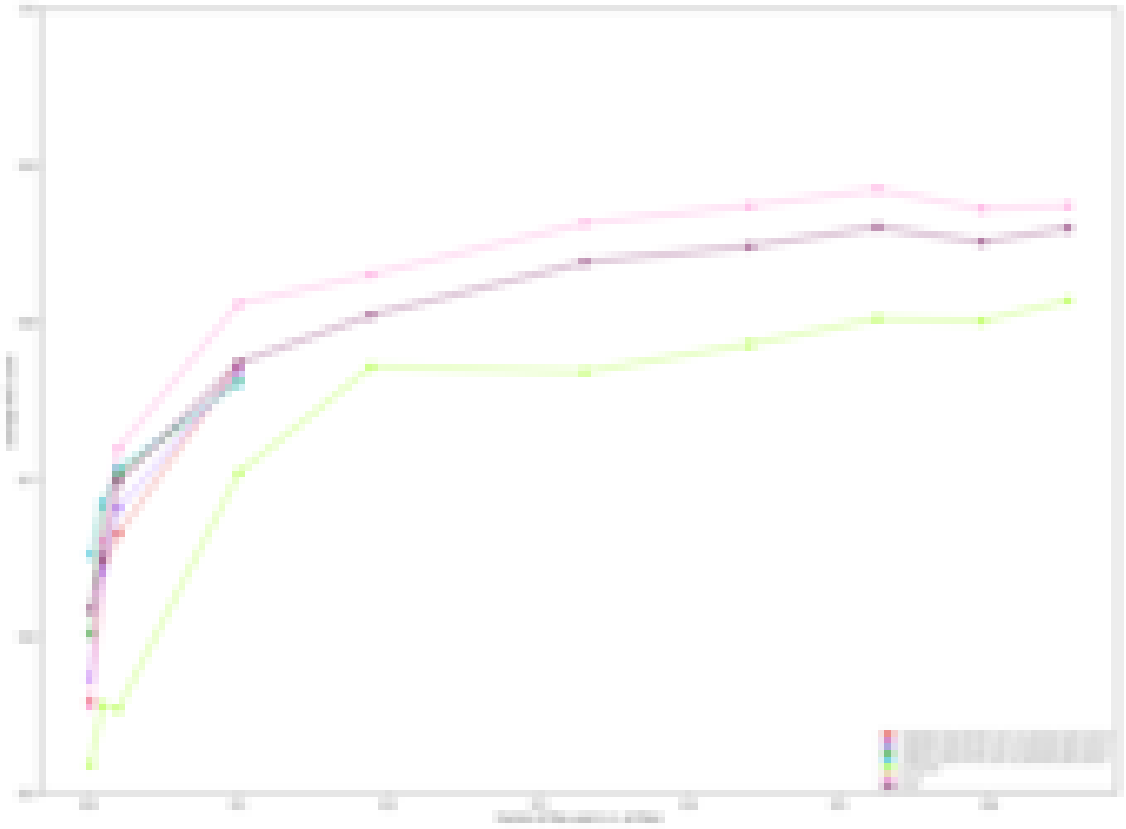


(b) Macro Average ROC-AUC score. Dashed Line: CPC models, Full Line: Baselines. Baseline-epochs: 120, CPC-epochs: 20

Figure 12: Test scores for different models trained with a fraction of data labels (more epochs)



(a) Micro average ROC-AUC score. Dashed Line: CPC models, Full Line: Baselines. Baseline-epochs: 120, CPC-epochs: 50



(b) Macro Average ROC-AUC score. Dashed Line: CPC models, Full Line: Baselines. Baseline-epochs: 120, CPC-epochs: 50

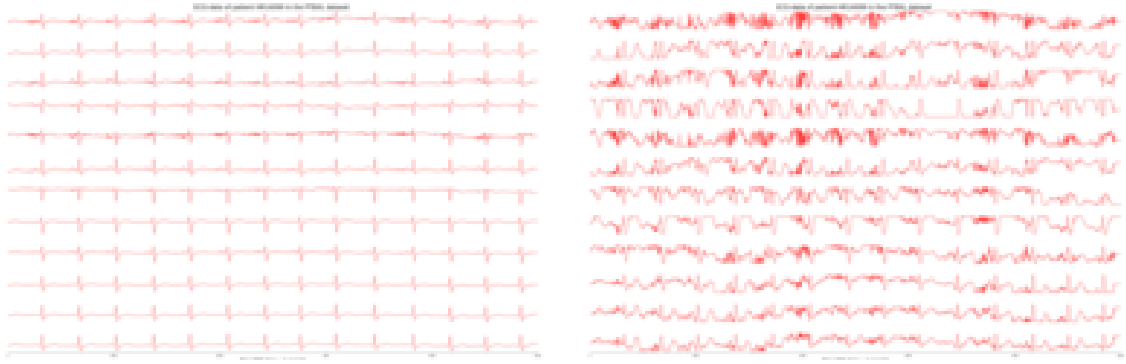
time efficiency. While all the baseline models needed retraining for every single training dataset with 120 epochs each, totaling to more than 24 hours wall clock training time, the CPC model only needed the initial 100 epochs **that is shared amongst all datasets**, plus 20 (or 50) additional training epochs, greatly improving training time by a factor of almost 6 (or >2). Apart from low label availability this of course also the case for different downstream tasks, which is a great opportunity for saving power and time or for neural net enthusiasts that do not have access to capable hardware.

5.1.5 Experiment: Training on Augmented Data

Under what circumstances is CPC especially efficient to use?

We wanted to test whether CPC and the Baseline models are able to correctly predict classes with heavily augmented data, which might be useful for very noisy or even partially broken data. For this we applied the "min-max (augmented)" normalization function on the ecg data and trained the models after. Figure 13 shows the augmented data.

Figure 13: Normalized ECG example from beginning Figure 1



(a) Correct Min-Max Normalization **(b)** Augmented Min-Max Normalization

The following tables show the results for training (and pretraining) multiple baseline and CPC-networks with different training settings on the augmented data. Note that for this section, we used slightly different downstream CPC networks compared to prior results (we refer to Figure 5 for their description).

Table 8a and table 8b show the ROC-AUC scores for all the trained baseline models, while table 9a and table 9b show the AUC scores for all trained cpc models. The left shows the results with class weights utilized in the loss calculation, while the right shows all scores for the same models without the class weights. We make this differentiation, because the macro average scores, which are normally improved by factoring in the class weights during training, were heavily degraded by making use of this method. We suspect the different normalization made the training process more unstable and the added class-weights, which alter the loss directly, contribute to that already unstable training.

Table 8: Baseline ROC-AUC scores

model	micro	macro
BL_v0_1	0.579	0.485
BL_v0_2	0.618	0.490
BL_v0_3	0.546	0.490
BL_v0	0.522	0.501
BL_v1	0.658	0.482
BL_v2	0.563	0.514
BL_v3	0.547	0.492
BL_v4	0.404	0.504
BL_v5	0.396	0.521
BL_v6	0.454	0.522
BL_v7	0.870	0.730
BL_v8	0.465	0.496
BL_v9	0.439	0.511
BL_v14	0.524	0.501
BL_v15	0.894	0.806
baseline_FCN	0.484	0.505
baseline_MLP	0.500	0.500
baseline_TCN_block	0.615	0.500
baseline_TCN_down	0.874	0.747
baseline_TCN_flatten	0.841	0.547
baseline_TCN_last	0.846	0.567
baseline_alex_v2	0.836	0.500
baseline_rnn_simplest_lstm	0.834	0.532

(a) Models trained with class weights in the loss function.

model	micro	macro
BL_v0_1	0.835	0.773
BL_v0_2	0.856	0.680
BL_v0_3	0.854	0.676
BL_v0	0.861	0.768
BL_v1	0.917	0.800
BL_v2	0.891	0.840
BL_v3	0.852	0.813
BL_v4	0.777	0.690
BL_v5	0.866	0.640
BL_v6	0.667	0.436
BL_v7	0.838	0.533
BL_v8	0.891	0.842
BL_v9	0.875	0.839
BL_v14	0.886	0.841
BL_v15	0.884	0.813
baseline_FCN	0.607	0.502
baseline_MLP	0.500	0.500
baseline_TCN_block	0.628	0.500
baseline_TCN_down	0.785	0.727
baseline_TCN_flatten	0.855	0.584
baseline_TCN_last	0.857	0.576
baseline_alex_v2	0.851	0.500
baseline_rnn_simplest_lstm	0.848	0.585

(b) Models trained without class weights in the loss function.

Table 9: CPC ROC-AUC scores

model	micro	macro
CPC | frozen | C | L | m:all | cnn	0.888	0.825
CPC | frozen | C | L | m:all | twolinear	0.889	0.822
CPC | frozen | C | L | m:same | cnn	0.904	0.849
CPC | frozen | C | L | m:same | twolinear	0.885	0.810
CPC | frozen | C | m:all | cnn	0.890	0.813
CPC | frozen | C | m:all | linear	0.914	0.810
CPC | frozen | C | m:all | twolinear	0.875	0.783
CPC | frozen | C | m:same | cnn	0.918	0.822
CPC | frozen | C | m:same | linear	0.918	0.834
CPC | frozen | C | m:same | twolinear	0.900	0.839
CPC | frozen | L | m:all | cnn	0.883	0.832
CPC | frozen | L | m:all | linear	0.910	0.844
CPC | frozen | L | m:all | twolinear	0.897	0.825
CPC | frozen | L | m:same | cnn	0.914	0.769
CPC | frozen | L | m:same | linear	0.841	0.708
CPC | frozen | L | m:same | twolinear	0.903	0.779
CPC | strided | frozen | C | L | m:all | cnn	0.909	0.780
CPC | strided | frozen | C | L | m:all | twolinear	0.776	0.545
CPC | strided | frozen | C | L | m:same | cnn	0.895	0.771
CPC | strided | frozen | C | L | m:same | twolinear	0.751	0.591
CPC | strided | frozen | C | m:all | cnn	0.921	0.792
CPC | strided | frozen | C | m:all | linear	0.921	0.812
CPC | strided | frozen | C | m:all | twolinear	0.911	0.795
CPC | strided | frozen | C | m:same | cnn	0.903	0.754
CPC | strided | frozen | C | m:same | linear	0.892	0.841
CPC | strided | frozen | C | m:same | twolinear	0.898	0.833
CPC | strided | frozen | L | m:all | cnn	0.898	0.836
CPC | strided | frozen | L | m:all | linear	0.884	0.806
CPC | strided | frozen | L | m:all | twolinear	0.898	0.838
CPC | strided | frozen | L | m:same | cnn	0.876	0.824
CPC | strided | frozen | L | m:same | linear	0.910	0.852
CPC | strided | frozen | L | m:same | twolinear	0.885	0.844

(a) Models trained with class weights in the loss function.

model	micro	macro
CPC | frozen | C | L | m:all | cnn	0.901	0.829
CPC | frozen | C | L | m:all | twolinear	0.838	0.808
CPC | frozen | C | L | m:same | cnn	0.886	0.843
CPC | frozen | C | L | m:same | twolinear	0.877	0.843
CPC | frozen | C | m:all | cnn	0.914	0.766
CPC | frozen | C | m:all | linear	0.921	0.812
CPC | frozen | C | m:all | twolinear	0.902	0.834
CPC | frozen | C | m:same | cnn	0.916	0.791
CPC | frozen | C | m:same | linear	0.911	0.795
CPC | frozen | C | m:same | twolinear	0.882	0.843
CPC | frozen | L | m:all | cnn	0.842	0.775
CPC | frozen | L | m:all | linear	0.769	0.540
CPC | frozen | L | m:all | twolinear	0.834	0.785
CPC | frozen | L | m:same | cnn	0.862	0.802
CPC | frozen | L | m:same | linear	0.781	0.595
CPC | frozen | L | m:same | twolinear	0.860	0.814
CPC | strided | frozen | C | L | m:all | cnn	0.867	0.823
CPC | strided | frozen | C | L | m:all | twolinear	0.865	0.817
CPC | strided | frozen | C | L | m:same | cnn	0.565	0.549
CPC | strided | frozen | C | L | m:same | twolinear	0.870	0.814
CPC | strided | frozen | C | m:all | cnn	0.917	0.822
CPC | strided | frozen | C | m:all | linear	0.921	0.792
CPC | strided | frozen | C | m:all | twolinear	0.914	0.842
CPC | strided | frozen | C | m:same | cnn	0.917	0.828
CPC | strided | frozen | C | m:same | linear	0.909	0.779
CPC | strided | frozen | C | m:same | twolinear	0.897	0.838
CPC | strided | frozen | L | m:all | cnn	0.858	0.799
CPC | strided | frozen | L | m:all | linear	0.902	0.775
CPC | strided | frozen | L | m:all | twolinear	0.822	0.795
CPC | strided | frozen | L | m:same | cnn	0.848	0.799
CPC | strided | frozen | L | m:same | linear	0.895	0.781
CPC | strided | frozen | L | m:same | twolinear	0.828	0.794

(b) Models trained without class weights in the loss function.

Legend:
m:all All latents are used as negative examples in the cpc loss. m:same Latents only from the same timestep are used as negative examples. linear/cnn/twolinear/latent_maximum The used downstream architecture. L Latent vectors are used for predicting the class labels. C Context vector is used for predicting the class labels. frozen The weights are not updated in the cpc network layers during downstream task. strided Disjoint window data encoder is used instead.

We see that CPC overall performs the best in terms of ROC-AUC score, although labels have not been used as often as in the baseline models. Interesting to see is also that the baseline models cannot cope well with the class weights added to the loss function: Most models cannot confidently surpass even an ROC-AUC area of 0.5, which is the threshold for random guessing. However we admit that the performance drop for the baseline models with the class weights added is suspiciously high and might suggest other factors such as non ideal training settings, that affect weighted models only, are the root of this problem.

Having said this, all CPC models, with only few exceptions, do not see this degradation in performance, meaning they fit all classes in the augmented dataset. As a result we observe better macro average values in some cases with the weights added to the loss function. The overall badly performing baselines and the few CPC models that completely fail to classify the augmented data, suggest that the training for this task is extremely

Table 10: Average CPC ROC-AUC scores, no pretraining

model	micro	macro
CPC | strided | unfrozen | C | m:all | cnn	0.889	0.864
CPC | strided | unfrozen | C | m:all | linear	0.808	0.482
CPC | strided | unfrozen | L | m:all | cnn	0.868	0.825
CPC | strided | unfrozen | L | m:all | linear	0.874	0.815
CPC | unfrozen | C | m:all | cnn	0.828	0.598
CPC | unfrozen | C | m:all | linear	0.788	0.642
CPC | unfrozen | L | m:all | cnn	0.824	0.499
CPC | unfrozen | L | m:all | linear	0.772	0.536

(a) No class weights were used in loss function.

model	micro	macro
CPC | strided | unfrozen | C | m:all | cnn	0.918	0.833
CPC | strided | unfrozen | C | m:all | linear	0.802	0.498
CPC | strided | unfrozen | L | m:all | cnn	0.904	0.824
CPC | strided | unfrozen | L | m:all | linear	0.901	0.798
CPC | unfrozen | C | m:all | cnn	0.820	0.498
CPC | unfrozen | C | m:all | linear	0.803	0.497
CPC | unfrozen | L | m:all | cnn	0.820	0.500
CPC | unfrozen | L | m:all | linear	0.708	0.570

(b) Class weights were used in loss function.

ROC-AUC scores for CPC models trained without pretraining, trained on the downstream task for 120 epochs. Legend: (like in Figure 9a)

unstable.

To determine whether this difference in performance is caused by the pretraining step or the architecture itself we additionally train a few new CPC models fully supervised without the pretraining step for 120 epochs. Comparing Figure 10 to Figure 9, the non-strided version of the model that used the context, did not work as good as the same model that was pretrained with the CPC-loss, especially the macro average degraded substantially. These results are in line with the baseline results in the stability experiment already observed. We argue that with heavily augmented/noised data, the CPC pretraining step is crucial to correctly learn the latent representations in the CPC architecture.

Interestingly, the "strided" CPC models do not exhibit such a big performance gap compared to the pretrained models and work well for both micro- and macro-average scores. We hypothesize that, even without pretraining, the "strided" architecture is easier to train due to fewer latent vectors being calculated, making it easier for downstream task networks.

Table 11a and Table 11b show yet another way of training the CPC architecture: pretrain the representations for 100 epochs and then, when training on the downstream task, do not freeze the weights in the CPC network layers:

Table 11: CPC ROC-AUC scores, with all layers updated in downstream training

model	micro	macro
CPC | strided | unfrozen | C | m:all | cnn	0.826	0.789
CPC | strided | unfrozen | L | m:all | cnn	0.818	0.752
CPC | unfrozen | C | m:all | cnn	0.818	0.731
CPC | unfrozen | L | m:all | cnn	0.819	0.499

(a) No class-weights are used.

model	micro	macro
CPC | strided | unfrozen | C | m:all | cnn	0.822	0.498
CPC | strided | unfrozen | L | m:all | cnn	0.821	0.498
CPC | unfrozen | C | m:all | cnn	0.820	0.476
CPC | unfrozen | L | m:all | cnn	0.821	0.520

(b) Class-weights are used.

ROC-AUC scores for CPC models trained with pretraining for 100 epochs and training on the downstream task for 20 epochs with the CPC weights also updated. Legend: (like in Figure 9a)

At first one would assume this would lead to the best overall prediction accuracy, because the CPC pretraining is utilized in order to train the autoregressive architecture and then all weights are fine-tuned to optimize the classification loss. Yet we observe the opposite: while the micro-average ROC-AUC scores are still in line with the other models, the macro-average scores are the overall worst, at least in the training with added class weights. We assume the cpc objective lead to a specific latent representation that were in an opposite direction of the downstream loss function and acted like a bad weight initialization. With more training epochs, we assume that the model will converge towards the results of Figure 10b, assuming the model can get out of the suboptimal local minimum. A different explanation is that the added flexibility allowed the model to more easily overfit on the training data.

Although the CPC models outperformed all baseline architectures (Figure 8 vs. Figure 9), which suggests that the pretraining step seems to help train the models, we cannot confidently say that pretraining is always helpful for augmented data because the strided CPC networks still perform well in a setting with no pretraining at all (see Figure 10), while some CPC models could not be succesfully trained even with pretraining. Since strided and non-strided models only vary in architecture and not how they are trained, it is possible that difference in the architecture lead to good performance and not the pretraining per se. Nevertheless we suspect that the CPC pretraining leads more easily to overall good representations if the data is difficult to classify, and that too much freedom during the difficult downstream task leads to overfitting or unstable training, which can be followed from Figure 9 vs. Figure 11. We suspect the strided models get around overfitting because of easier model convergence. In summary we argue that the stability experiment results are inconclusive but they hint at that CPC models and their pretraining make finding the latent features easier, in unstable training environments.

5.1.6 Additional changes

Can the original architecture be improved?

Next we take a look at the additional changes we made to the CPC architecture to allow a more flexible use or increase performance. In some cases we relaxed our downstream epoch limit from 20 to 40, because we are not directly compare to baseline networks and in order to more thoroughly test an architectures capabilities. We note that this increases the difficulty of comparing those models to the ones that only trained for 20 epochs above, but if the performance is still lower when trained for 40 epochs, we are more certain that this specific change in the architecture was a step in the wrong direction.

Additionally please note that for certain categories like "Downstream Model", we will only display the models with the highest macro average scores. For example a model that has been trained with `cpc_downstream_latent_average` vs `cpc_downstream_latent_maximum` with otherwise equal settings, will only show the better model with the associated scores. Also all models use `cpc_intersect_manylatents` (see Paragraph 2.2.1.2) and — besides their one changed module — still the standard encoder-, context- and predictor network `cpc_encoder_v0`, `cpc_autoregressive_v0` and `cpc_predictor_v0`. This restriction is necessary since some models require to be trained with

Table 12: CPC ROC-AUC scores, 40 Downstream Epochs

model	Freeze CPC	strided	CPC Sampling Mode	Downstream Epochs	Downstream Model	micro	macro
CPC	False	False	crossentropy	40.000	cpc_downstream_latent_average	**0.930**	**0.842**
CPC	False	False	crossentropy	40.000	cpc_downstream_latent_maximum	0.925	0.830
CPC	False	False	crossentropy	40.000	cpc_downstream_twolinear_v2	0.911	0.830
CPC	False	True	crossentropy	40.000	cpc_downstream_latent_average	0.905	0.768
CPC	False	True	crossentropy	40.000	cpc_downstream_latent_maximum	0.878	0.802
CPC	False	True	crossentropy	40.000	cpc_downstream_twolinear_v2	0.915	0.801
CPC	True	False	crossentropy	40.000	cpc_downstream_latent_average	0.923	0.824
CPC	True	False	crossentropy	40.000	cpc_downstream_latent_maximum	0.926	0.827
CPC	True	False	crossentropy	40.000	cpc_downstream_twolinear_v2	0.929	0.838
CPC	True	True	crossentropy	40.000	cpc_downstream_latent_average	0.908	0.772
CPC	True	True	crossentropy	40.000	cpc_downstream_latent_maximum	0.915	0.804
CPC	True	True	crossentropy	40.000	cpc_downstream_twolinear_v2	0.917	0.801

ROC-AUC scores for CPC models trained with pretraining for 100 epochs and training on the downstream task for 40 epochs.

Table 13: CPC ROC-AUC scores: normalized latents

model	Freeze CPC	strided	normalizes latents	CPC Sampling Mode	Downstream Epochs	Downstream Model	micro	macro
CPC	False	False	True	crossentropy	20.000	cpc_downstream_twolinear_v2	0.873	0.659
CPC	False	False	True	crossentropy	40.000	cpc_downstream_latent_average	0.905	0.774
CPC	False	True	True	crossentropy	20.000	cpc_downstream_twolinear_v2	0.869	0.750
CPC	False	True	True	crossentropy	40.000	cpc_downstream_latent_maximum	**0.916**	**0.799**
CPC	True	False	True	crossentropy	20.000	cpc_downstream_twolinear_v2	0.845	0.690
CPC	True	False	True	crossentropy	40.000	cpc_downstream_twolinear_v2	0.887	0.697
CPC	True	True	True	crossentropy	20.000	cpc_downstream_twolinear_v2	0.879	0.754
CPC	True	True	True	crossentropy	40.000	cpc_downstream_twolinear_v2	0.902	0.753

ROC-AUC scores for CPC models trained with pretraining for 100 epochs and training on the downstream task for 20 or 40 epochs (both frozen/updated). Latent vectors are always normalized

`cpc_intersect_manylatents` in order to function.

Table 12 can be used as a reference for what is possible using all standard modules with 40 epochs (with `cpc_intersect_manylatents`).

5.1.6.1 Normalized Latent Vectors Table 13 shows the results for models that were trained with normalized latent vectors (see Paragraph 2.2.1.3 for details). We see that even with more epochs the scores are overall lower, for both micro and macro average, indicating this change most likely is not beneficial. We suspect that normalizing the vectors restricted the model too much.

5.1.6.2 Encoder like Baseline_v8 In Table 14 we see the results for models that were trained with `cpc_encoder_likev8`, which is a clone of the baseline model `BL_v8`. `cpc_encoder_likev8` outputs 147 latent vectors as opposed to 26 like the standard architecture. We report lower scores overall.

Nevertheless we want to mention that generating more latent vectors is a trait which is useful in some tasks, as for example later on in Subsubsection 5.2.6.

5.1.6.3 Hidden State Context Network We look at the results of our networks where `cpc_autoregressive_hidden` is used, which uses the hidden state as a context for the

Table 14: CPC ROC-AUC scores: `cpc_encoder_likev8`

model	Freeze CPC	strided	CPC Sampling Mode	Encoder	Downstream Epochs	Downstream Model	micro	macro
CPC	False	False	crossentropy	cpc_encoder_likev8	20.000	cpc_downstream_twolinear_v2	0.853	0.771
CPC	True	False	crossentropy	cpc_encoder_likev8	20.000	cpc_downstream_twolinear_v2	**0.873**	**0.787**

ROC-AUC scores for CPC models trained with pretraining for 100 epochs and training on the downstream task for 20 or 40 epochs (both frozen/updated). `cpc_encoder_likev8` is always used as an encoder

Table 15: CPC ROC-AUC scores: `cpc_autoregressive_hidden`

model	Freeze CPC	strided	CPC Sampling Mode	Autoregressive	Downstream Epochs	Downstream Model	micro	macro
CPC	False	False	crossentropy	cpc_autoregressive_hidden	40.000	cpc_downstream_latent_maximum	**0.927**	**0.851**
CPC	True	False	crossentropy	cpc_autoregressive_hidden	40.000	cpc_downstream_twolinear_v2	0.885	0.797

ROC-AUC scores for CPC models trained with pretraining for 100 epochs and training on the downstream task for 20 or 40 epochs (both frozen/updated). `cpc_autoregressive_hidden` is always used as context network

next step latent predictions. The model that was trained for 40 epochs with all weights updated yielded the best results, even surpassing the macro-average score from the best model in the comparison Table 12. Interestingly the score got degraded for the model with the weights frozen, making this context network useful only situational.

5.1.6.4 No Context Network For the network that did not use the context for predicting future latents, and instead a TCN trained to predict next-step latents directly, the results are overall a bit lower. Despite not displayed here, we mention that the CPC accuracy during pretraining was higher than for any other model, with exemption of models trained with the 'same' sampling method and all standard settings, indicating that the respresentations were more easily distinguishable for the different entries in the batch. Although the results were slightly less accurate, we note that it is still interesting to see

Table 16: CPC ROC-AUC scores: `cpc_predictor_nocontext`

model	Freeze CPC	strided	CPC Sampling Mode	Predictor	Downstream Epochs	Downstream Model	micro	macro
CPC	False	False	crossentropy-nocontext	cpc_predictor_nocontext	20.000	cpc_downstream_twolinear_v2	0.873	0.762
CPC	False	False	crossentropy-nocontext	cpc_predictor_nocontext	40.000	cpc_downstream_latent_maximum	**0.913**	**0.831**
CPC	True	False	crossentropy-nocontext	cpc_predictor_nocontext	20.000	cpc_downstream_twolinear_v2	0.881	0.764
CPC	True	False	crossentropy-nocontext	cpc_predictor_nocontext	40.000	cpc_downstream_twolinear_v2	0.911	0.803

ROC-AUC scores for CPC models trained with pretraining for 100 epochs and training on the downstream task for 20 or 40 epochs (both frozen/updated). `cpc_predictor_nocontext` is always used as the latent predictor

that the context network + single linear layers for predicting future latents is not a given and can be exchanged for different architectures.

5.2 Qualitative

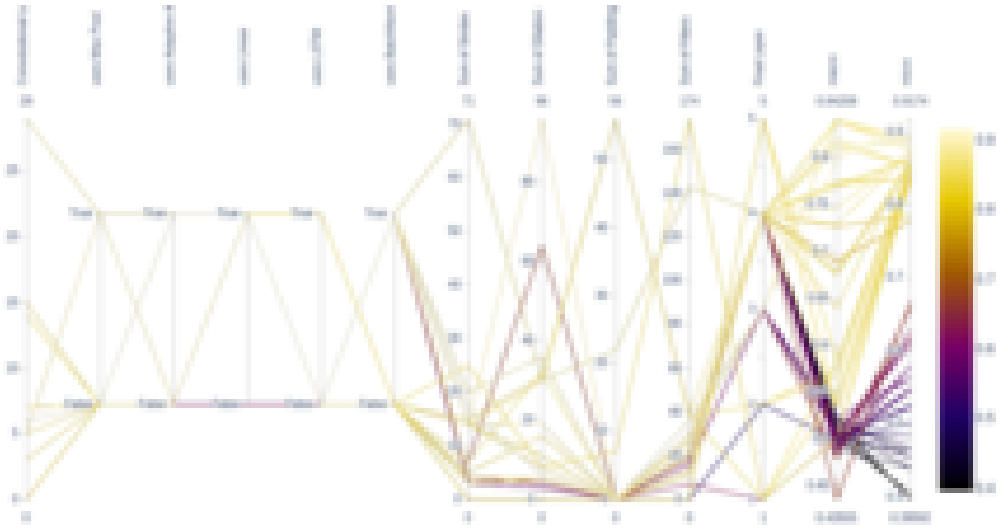
5.2.1 Finding the best Hyperparameters with Parallel Coordinates

What are the best settings/hyperparameters for CPC?

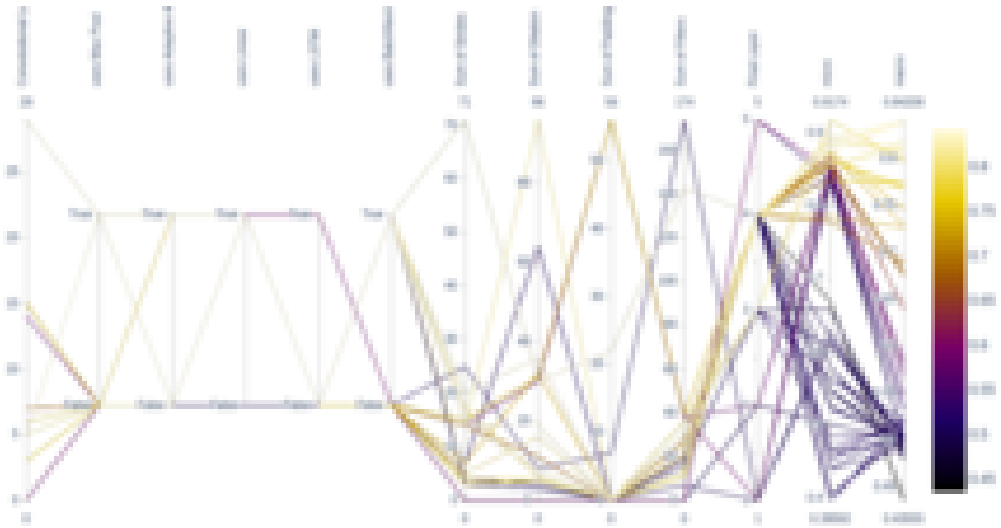
To better evaluate what model specific baseline hyperparameters lead to a good test performance we make use of a so called "Parallel Coordinates" plot which can visualize high-dimensional datasets. It consists of one x-axis that uses a categorical label instead of a number at each step with no specific order and multiple differently ranged y-axes, one for each categorical label. Every datasample is drawn as line in a color specific to one of the datasamples attributes. In short, parallel coordinate plots visualize tables by using the column-headers as x-axis and draw each row as a line with a color specific to one selected column.

Since we want to figure out what hyper parameters influence our micro and macro ROC-AUC score in a positive way, both will be used individually as color in the plot, while some of the hyperparameters will be shown on the x-axis. We summarize list-like hyperparameters such as multiple strides or kernel-sizes as a sum in order to visualize them. For strides especially, a product could be used instead to deliver more accurate results, which is due to the fact that strides have a multiplcative impact on input data, while kernel-sizes are closer to additive. The widely different values however made the plot unreadable with big products in our case. The resulting plot can be seen in Figure 14a and Figure 14b. While it is difficult to judge most hyperparamters with confidence based on this plot, we can see that the final layer choice seems to have a big impact on performance in our baseline models. There, approach 4 — which downsamples the models output with a convolutional layer before feeding it to the final linear layer — seems to achieve the overall best performance.

Figure 14: Parallel Coordinates Plot for all trained baseline networks from Figure 8a.



(a) The color is the **micro** ROC-AUC score



(b) The color is the **macro** ROC-AUC score

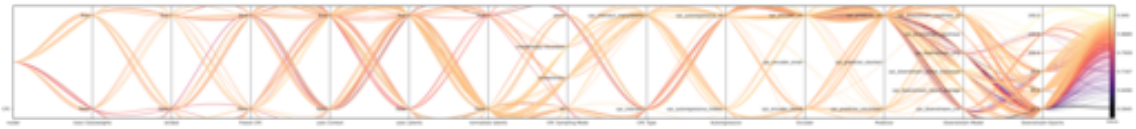
For the trained CPC models we use a different implementation obtained from Github[5], where the lines are curved to better visualize binary values. Figure 15b and Figure 15a show the parallel coordinate plots for all of our trained CPC models. This also includes experiments like "low label availability".

Although more CPC models have been evaluated in the CPC parallel coordinates plots, they are less conclusive regarding model hyperparameters and performance relationship,

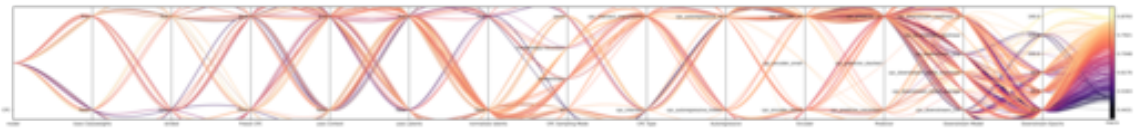
[5] **https://github.com/jraine/parallel-coordinates-plot-dataframe/blob/master/parallel_plot.py**.

because there are both good and bad AUC-scores for most hyperparameters. An exemption of this might be normalizing the latent vectors during training, which results in a worse macro AUC score. Additionally it seems to be beneficial to utilize the context vector, which can be again seen in the macro AUC score.

Figure 15: Parallel Coordinates Plot for all trained baseline networks from Figure 8a.



(a) The color is the **micro** ROC-AUC score



(b) The color is the **macro** ROC-AUC score

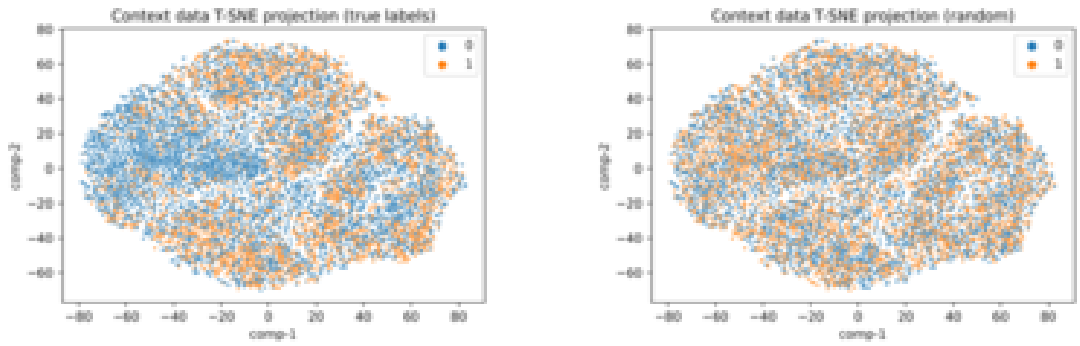
5.2.2 Latent/Context t-SNE representations

Does CPC learn visually verifiable useful latent representations?

In an attempt to visually verify the usefulness of the learned latent data representations we separately perform a dimensionality reduction from the 128 valued latent vectors and the 256 valued context vectors into a two dimensional space. This is done by taking one of the above trained CPC networks (with the settings use_weights, strided, unfrozen, cpc_downstream_cnn in this case) and saving all latent and context vectors for the whole dataset. Afterwards we use t-SNE [35] to embed the vectors into two dimensions, the results are visualized in a scatter plot where the embedded dimensions are used as x- and y-axis. Each sample is colored according to its label in the data. Since the data has multiple possible labels per sample, which makes distinct class coloring difficult, we opted to use binary coloring — if class x is present in the data assign 1, otherwise assign 0 to the datapoint. This is done for all classes.

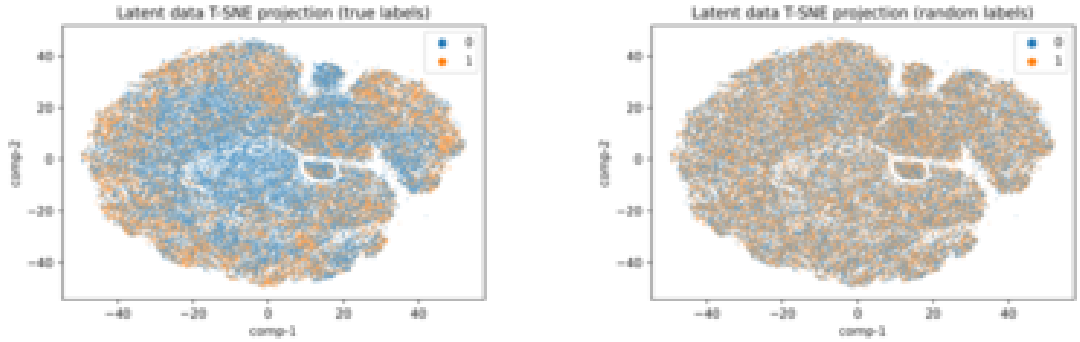
In order to be able to better judge if the colored embeddings have meaning, we added a scatter plot with randomized labels as well. An example for this can be seen in Figure 16 which shows all context vector embeddings, with the labels for class 426783006 (normal sinus rhythm). Although the data points were not fully separated, a clear distinction to the random plot can still be made (see mostly blue big area on the left). The same goes for Figure 17 which shows all latent vector embeddings instead. There the positive labels seems to form a border around the data, the negative labels are mostly centered in the middle.

Figure 16: Two dimensional t-SNE embeddings of all context vectors



t-SNE embedding of all context vectors (one for each patient). Left: Data points are colorized according to their true labels. Right: The colors are randomly assigned.

Figure 17: Two dimensional t-SNE embeddings of all latent vectors



t-SNE embeddings of all latent vectors (nine vectors per patient). Left: Data points are colorized according to their true labels. Right: The colors are randomly assigned.

Figure 18 and Figure 19 show the embeddings with the respective point coloring for 64 of 67 classes, denoted in the upper right of each scatter plot. The plots are ordered by their ROC-AUC-score from top-left to bottom-right, each score visualized as background color. Especially for smaller classes (fewer yellow dots) it is difficult to judge how well the data is clustered, but there are cases were clusters are clearly distinguishable. Take for example classes 427084000 (sinus tachycardia) and 426177001 (sinus bradycardia) in Figure 18, which are on the opposite site in the embedding and also have a high ROC-AUC-score. Interestingly sinus tachycardia means that the heart beat is faster than usual (over 100bpm [36]) and sinus bradycardia that the heart beats slower than usual (under 60bpm [37]), which means the classes are in fact conflicting and very unlikely to occur together as final diagnosis on the same patient. The same classes are also fairly spatially divided with the latent embeddings Figure 19, however not as clearly as with the context embeddings. This might be due to the fact that a patient diagnosed with sinus tachycardia might still have a few slow beats or vice versa with sinus bradycardia. Also since the tested model relies on the context vector for the final prediction, the classes need to

be more distinguishable in the context vector space than the latent vector space, which could explain the better spatial division. Additionally since nine times more latent data points are generated, the two dimensional embedding could be harder to find for t-SNE.

In contrast to spatially divided clusters there also seem to be class clusters which are more or less equivalent. In Figure 19 take for example classes 6374002 (Bundle Branch Block) and 164909002 (Left Bundle Branch Block), which look very similar apart of their sizes, and are also very similar from a medical standpoint.

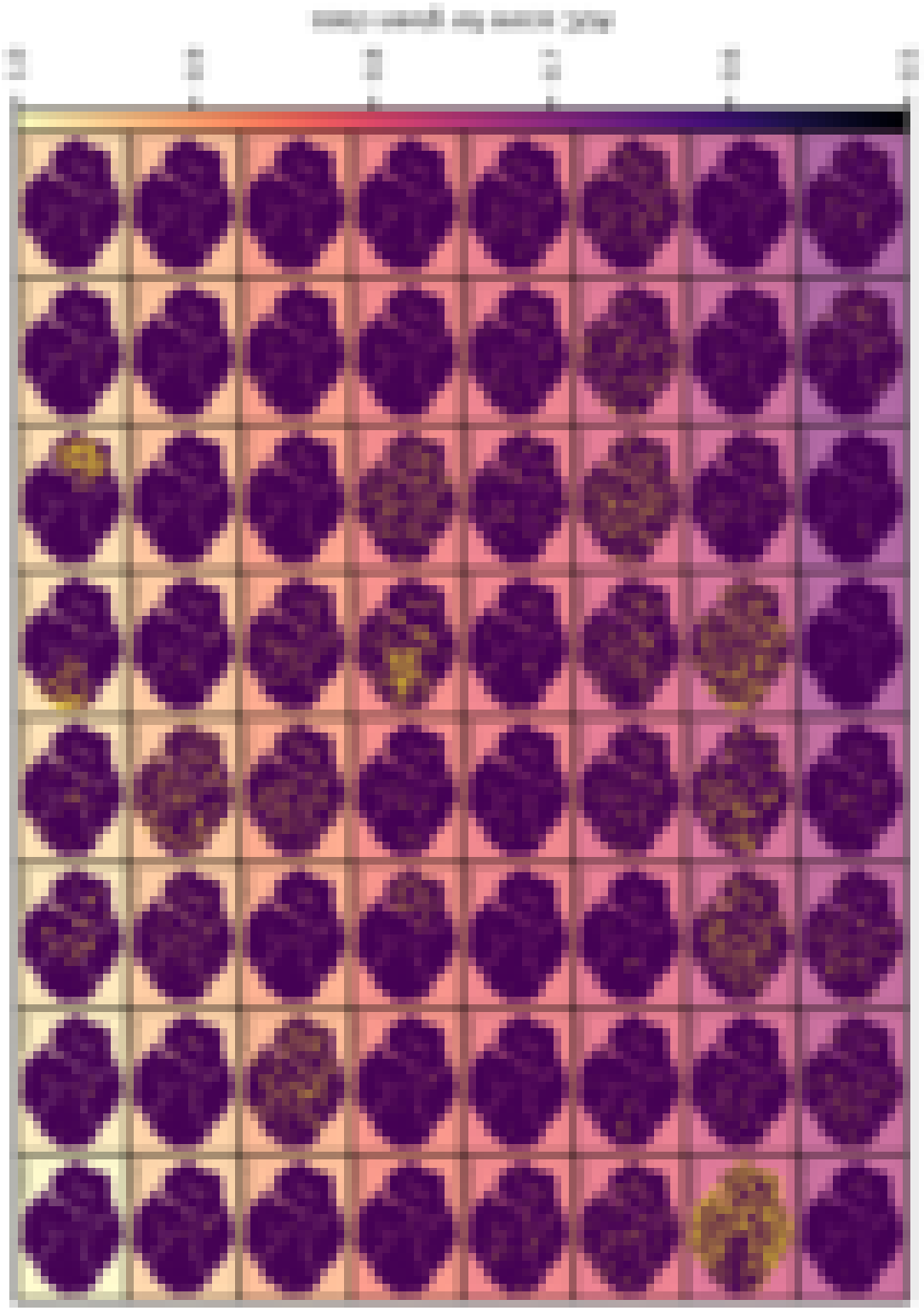


Figure 18: t-SNE embedding of all context vectors (one for each patient) for 64 of 67 classes. Legend: number in top right denotes class name, yellow dot means class name is part of the recording's labels. Background color is the AUC score per class, sorted descending

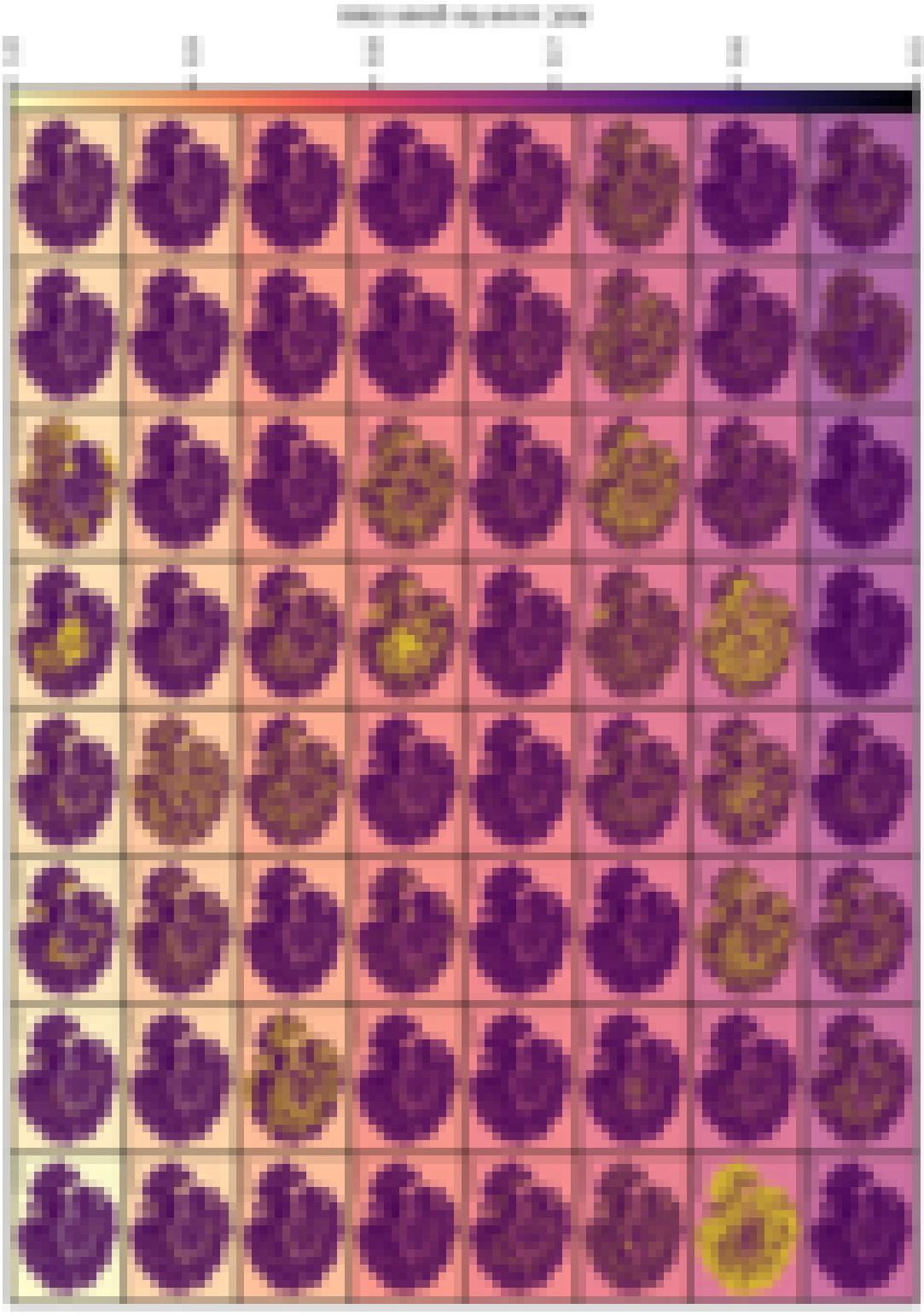


Figure 19: t-SNE embedding of all latent vectors (nine for each patient) for 64 of 67 classes. Legend: number in top right denotes class name, yellow dot means class name is part of the recording's labels. Background color is the AUC score per class, sorted descending

Figure 20: t-SNE embedding of all context vectors (one for each patient) for 64 of 67 classes. Context obtained with frozen model. Legend: number in top right denotes class name, yellow dot means class name is part of the recording's labels. Background color is the AUC score per class, sorted descending

Figure 21: t-SNE embedding of all latent vectors (26 for each patient) for 64 of 67 classes. Latents obtained with frozen model. Legend: number in top right denotes class name, yellow dot means class name is part of the recording's labels. Background color is the AUC score per class, sorted descending.

We compare prior results of a strided model where all weights are updated, with the t-SNE embeddings of a non-strided network that was trained with frozen weights. As a result the learned latent representations obtained by only the CPC loss are formed completely independent of labeled data. The t-SNE embeddings will show how/if the latents are dividable into clusters and if the highlighted classes are comparably separable.

Figure 20 shows the t-SNE embedding of the context vectors, while Figure 21 shows the embedding for all latent vectors. Comparing the different model's context plots, we notice that the true samples of that specific class (yellow dots) seem to be less grouped and more spread throughout the plot. The distinction we made in the first plot between class sinus tachycardia and sinus bradycardia cannot be made anymore. We suspect that the training with updated weights is important for building a context that is focused on predicting class labels, rather than the next latent steps.

On the other side, for the latent embedding we observe that there seems to be a way finer clustering in general, which are however not restricted to class labels. We argue that the latent representations formed in the model with frozen weights have distinct properties partly independent of their class label in the data, which is not surprising as the CPC loss is minimized without labels. The class labels seem to appear predominantly in few selected clusters which suggests that each class has its own defining properties.

In conclusion the context vectors of the strided model with updated weights seems to be more suited for classification, while the latent representations of the frozen non-strided model appear to be of higher quality and more capable of identifying unique data properties, which are focused on specfic classes only sporadically. This suggests that CPC finds general latent representations which encode the data's properties and that during the downstream task those representations get fine tuned for classification. In the case of the strided model those representations were less separable in a lower dimension, where It's not entirely clear if these properties got imperfectly "mixed" into classes, or if the strided model's latent representations have less distinct properties in general.

5.2.3 Transforming Predictions Scores with Model Thresholds

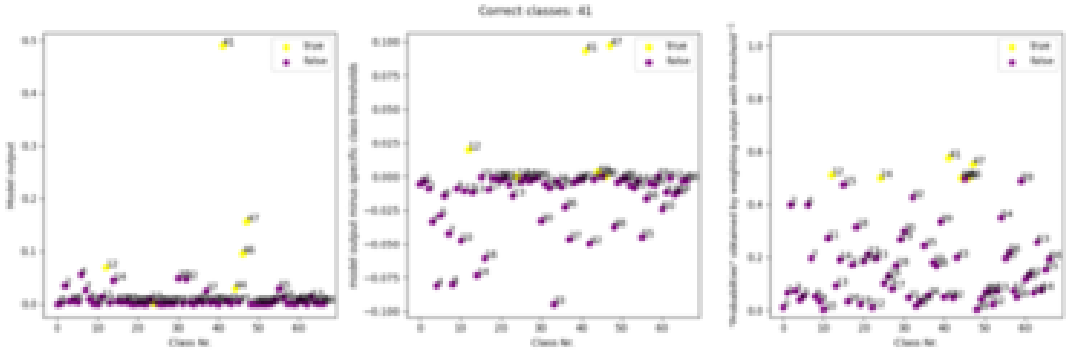
After predictions for the ECG-data have been made it is important to visualize them correctly, to give an idea of what class labels are considered true Let p be a prediction score vector for n classes $p = (s_1, \ldots, s_n)$ with $\forall i \leq n : s_i \in [0, 1]$ and t be a threshold score vector for n classes $t = (t_1, \ldots, t_n)$ with $\forall i \leq n : t_i \in [0, 1]$ that decides if class i is considered true (Equation 22). We define the reweighted probability scores $p_r = (r_1, \ldots, r_i, \ldots, r_n)$ as:

$$r_i := \begin{cases} (\frac{s_i - t_i}{t_i} + 1)/2, & \text{if } s_i < t_i \\ (\frac{s_i - t_i}{1 - t_i} + 1)/2, & \text{if } s_i \geq t_i \end{cases} \tag{25}$$

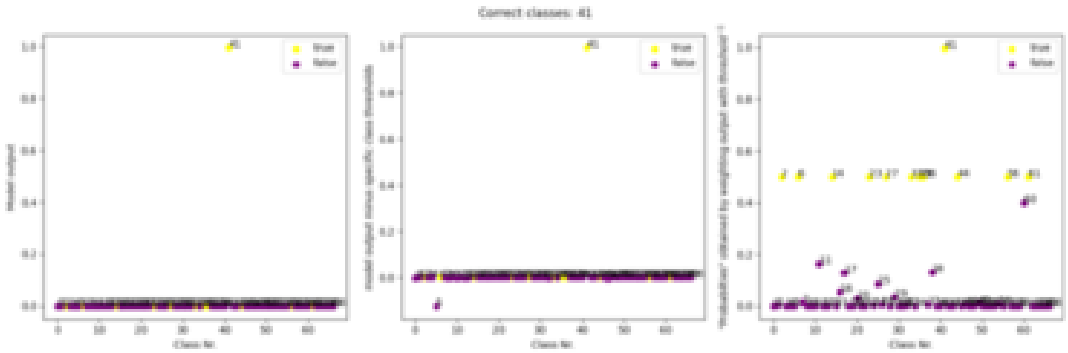
The reweighting function can be explained as splitting scores into positive and negative predictions where we stretch the space below the threshold $[0, t_i[$ to $[0, 0.5[$ and the space above and equal to threshold t_i: $[t_i, 1] \mapsto [0.5, 1]$. This has the advantage that predictions mean the same across all classes and models: We moved the decision boundary from $\forall i, t_i \mapsto 0.5$ In Figure 22 and Figure 23 we see examples for the process of transforming model outputs to reweighted probabilities.

5.2.4 Hand-picked Samples: Prediction Score Scatterplots

Figure 22: Scatterplots showing from left to right: exact model predictions, model predictions minus specific class thresholds, reweighted prediction probabilities



(a) CPC Model



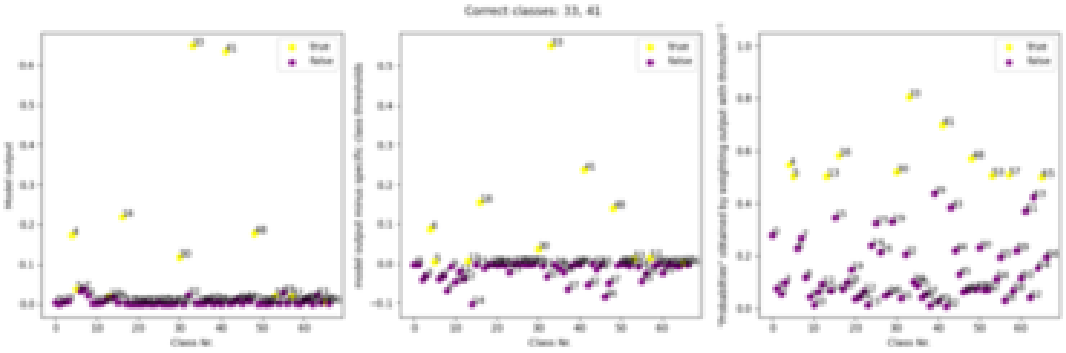
(b) Baseline Model (BL_v14)

Looking at the leftmost plot in Figure 22, we see why reweighting probabilities is important for readability: the unchanged model outputs for class 44 is considered true by the model but has a lower probability output than for example class 6 which is considered false. We substract the class specific thresholds and obtain a more readable output. Last but not least we transform the differences into real probabilities and are even able to judge how sure a model is about specific predictions: the closer to 0 or 1 $\implies$ prediction certainty high.

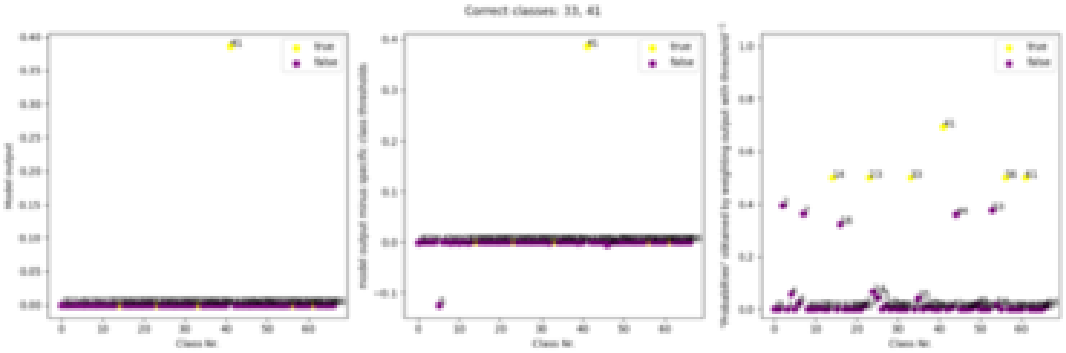
For the `BL_v14` model predictions we observe that only the correct class 41 has a high probability of being true while the other classes falsely classified as true barely go above 0.5. The CPC model predictions are less confident in general, however class 41 has still the highest probability. At this place we also want to point out a flaw of scores that are based on binary predictions: In this specific case CPC would have the higher score, as fewer classes are wrongly classified as true. Nevertheless we make the argument that in this case the predictions of the `BL_v14` are higher quality because of the models high certainty prediction of the correct class.

Figure 23 shows yet another prediction example — this time we would argue that the

Figure 23: Scatterplots showing from left to right: exact model predictions, model predictions minus specific class thresholds, reweighted prediction probabilities



(a) CPC Model



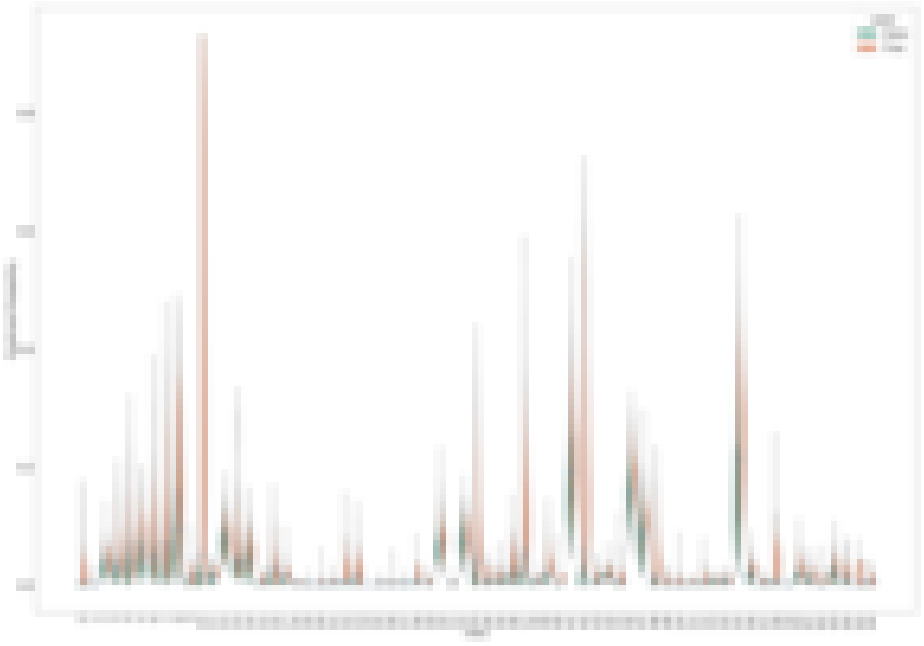
(b) Baseline Model (BL_v14)

CPC predictions are slightly better, at least concerning classes labeled "true". Although more classes are classified as true erroneously, both 33 and 41 have been predicted with the highest certainty. Compare this to The baseline predictions where 41 is fairly certain, but 33 is just as unlikely as erroneously classified classes.

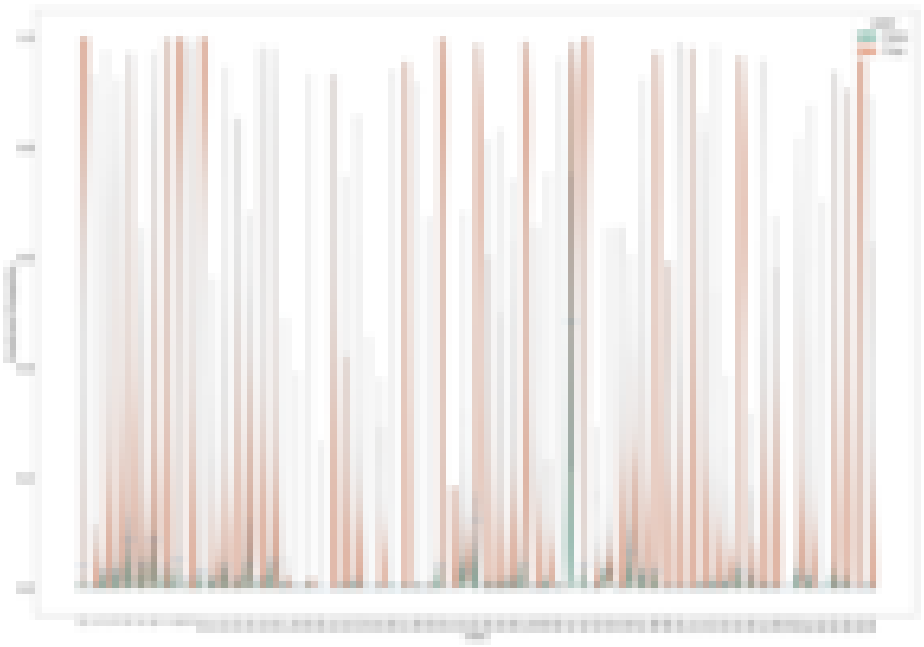
5.2.5 All Samples: Violinplots

Below in Figure 24 we include the raw prediction score data visualized as a violinplot, which show the prediction distributions for all classes. We opted to visualize both True and False prediction distributions with the same width, which removes the information about the group sizes, but increases readability. Noticeable is that even without the reweighting of model predictions, probabilities in the baseline model are more evenly distributed within $[0, 1]$, indicating that the sigmoid was able to more precisely predict the extrema 0 and 1. This is not implicitly better since thresholds get selected to divide the probabilties into true and false anyways and as long as the false and true label predictions are more or less separate from each other, results are equally correct.

Figure 24: Scatterplot showing all exact model predictions, divided into groups depending on their ground truth label.



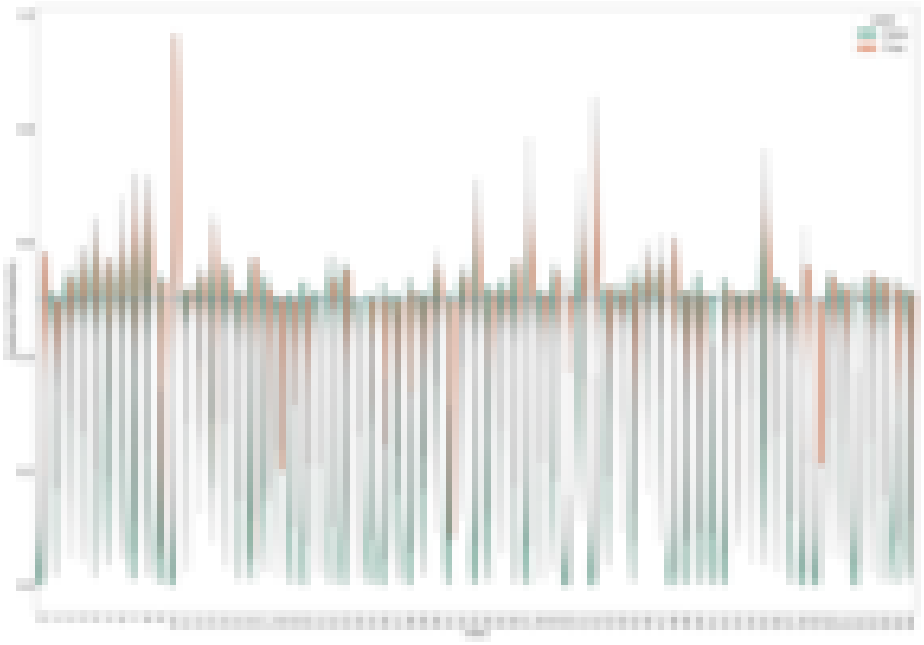
(a) Violinplot for a CPC model; unfrozen during Downstream Training. Horizontal lines show model specific decision thresholds



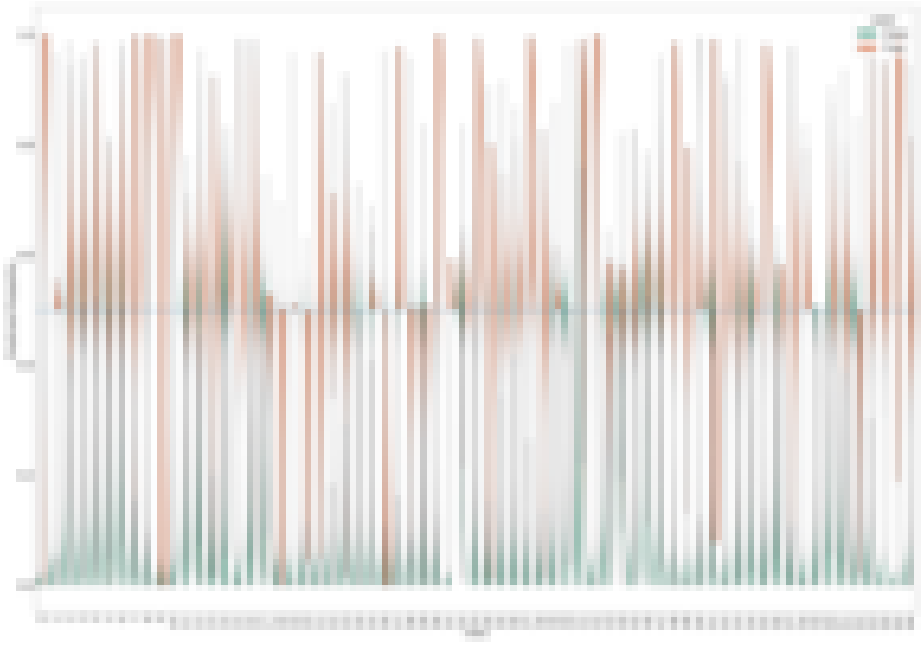
(b) Violinplot for Baseline (v14) model. Horizontal lines show model specific decision thresholds

Yet again, with the model thresholds added in Figure **??**, the groups are more or less visibly split into true and false samples, with some exceptions like class 41 (sinus rythm), where true samples get classified as false or false samples as true fairly often. The comparison between CPC model and baseline model is difficult, but we notice that "False" classes are more oftenly wrongly classified as true for the CPC model. Interesting to see is that both models failed to predict class 36 correctly, where the baseline model even predicted samples as false with high certainty ($p = 0$). On the other side these plots also show strengths of specific models: Class 31 is correctly split into true and false samples for the baseline model but not the CPC model, whereas class 62 is correctly classified by CPC but not the baseline. Looking at the class distribution in Figure 3 we observe that class 62 is a small class only available in select datasets, again hinting at CPC's ability of needing less labeled data.

With the help of these plots one could create a model ensemble which utilizes different networks for each class, creating a more powerful and accurate prediction model.



(a) Violinplot for CPC model; unfrozen during Downstream Training. Probabilities reweighted as per Equation 25



(b) Violinplot for Baseline (v14) model. Probabilities reweighted as per Equation 25

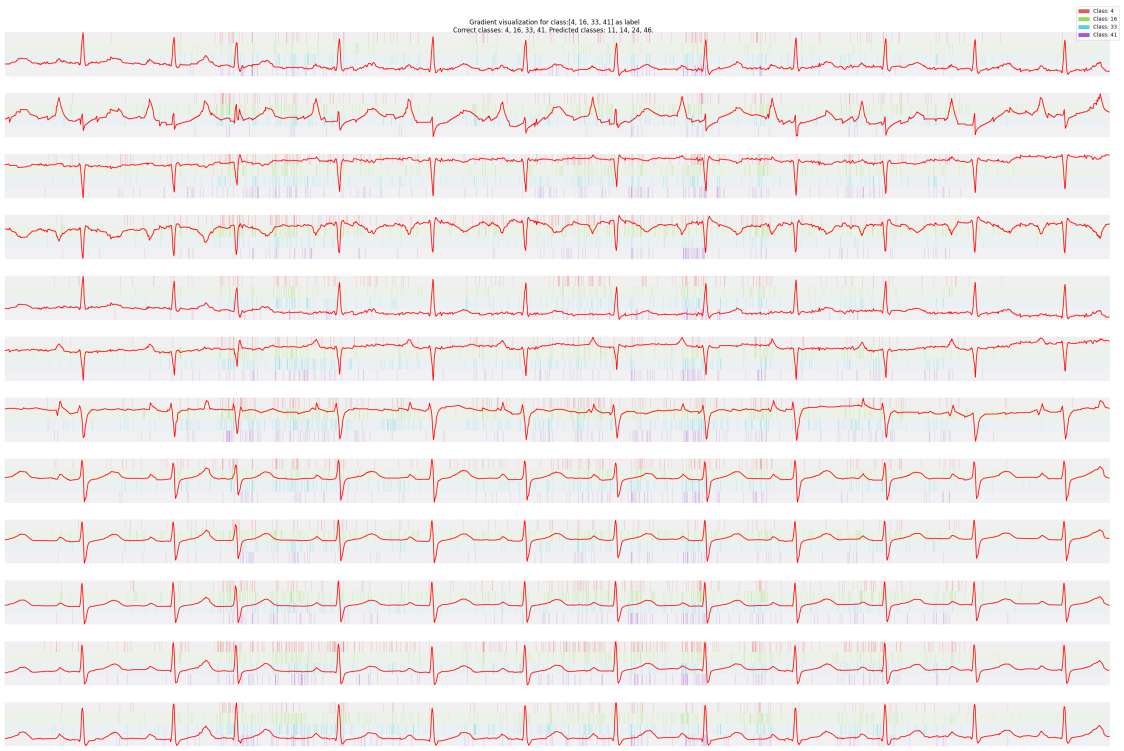
5.2.6 Explaining predicted classes in input data

As a bonus, having predicted a diagnostic label, it might be of great interest to also highlight where the model assumes the predicted classes are located at in the input data. We try to accomplish this task by performing a feed-forward of the input data through the network and than calculate the loss between the prediction and the target y^c — an all-zero valued vector ($\sim$ all diagnostic labels have a probability of 0) besides a specified target class c , which is set to one (class probability 1). As of yet we use the available ground truth labels, which are, if not available, to be replaced with the binary predictions obtained with the model's thresholds. The gradients are calculated by backpropagation with respect to the input data X. Once the gradient matrices $\frac{\partial y^c}{\partial X}$ has been obtained, we either take their absolute values to visualize areas that have an impact in general or apply a ReLU activation to highlight areas that positively influence the target's class prediction. The latter idea is based on the Grad-CAM paper [4], the procedure in general is called guided backpropagation [38]. We normalize the matrices individually to $[0, 1]$ and assign colors for each class, which increase in opacity for high values and become transparent below a certain threshold (e.g. < 0.2, which varies depending on model). We can see what values in the data would have to change in order to decrease the loss: High gradient values $\sim$ high influence on the predicted classes; low gradient values $\sim$ less influence on the predicted classes. To show multiple classes/colors in the same spots we divide the ecg channels into the number of present classes and color each division separately. The final result with multiple patients as example can be seen in Figure 26 for the `BL_v8` model and in Figure 27 for a CPC Model.

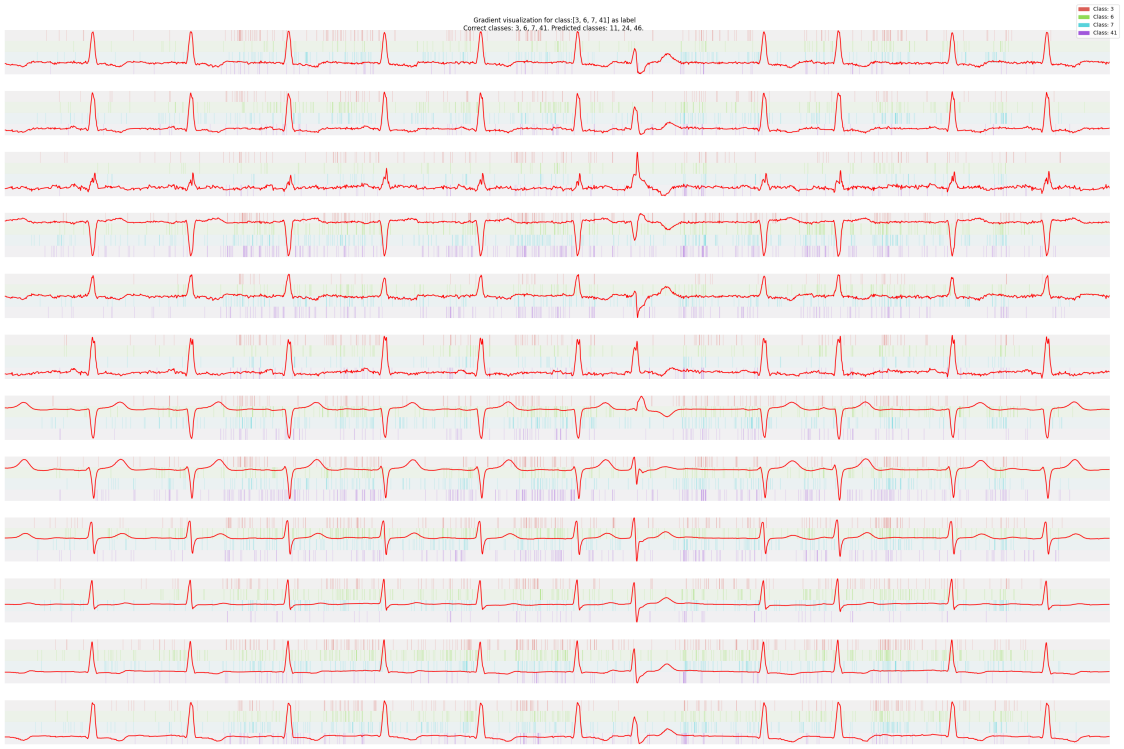
Interestingly the CPC model's biggest gradient values are less spread throughout the input which could potentially lead to marking interesting spots with a higher precision. On the other hand locations might not get marked although classes are present there. The markings in the baseline model are almost everywhere, making class judgments more difficult.

Additionally to our approach of calculating $\frac{\partial y^c}{\partial X}$, we tested Grad-CAM [4] which takes the gradient $\frac{\partial y^c}{\partial A^k}$ of the convolutional feature maps A^k in a selected layer and calculates $L^c_{\mathrm{Grad-CAM}} = ReLU\left(\sum_k \alpha^c_k A^k\right)$, where $\alpha^c_k = \frac{1}{ij}\sum_i \sum_j \frac{\partial y^c}{\partial A^k_{i,j}}$ is the mean/ global average pooling of feature map A^k. $L^c_{\mathrm{Grad-CAM}}$ can thus be seen as the sum of all feature maps in a specified layer, weighted according to their importance in predicting a specified class c, which results in a coarse heatmap where the model assumes the class to be located. One big upside compared to our localization approach is that the locations are directly generated from later layers, where most class features are encoded in. The downside however is that the heatmap size is dependent on the individual architecture and not the input data. This leads to few values available for localization in some cases. Additionally, in contrast to our approach, all channel information get lost, meaning eg. ecg-channel v1 and ecg-channel v3 are colored the exact same, although a specific class can only be recognized in one of them.

The coarse heatmap is then rescaled to match the input size, which is 4500 in our case. Since the layers feature map channels get summarized into a single value, no spots depending on input channels can be recovered and all input channels share the same areas of interest. This is to say that even if a certain class is only visible in a specific ecg channel



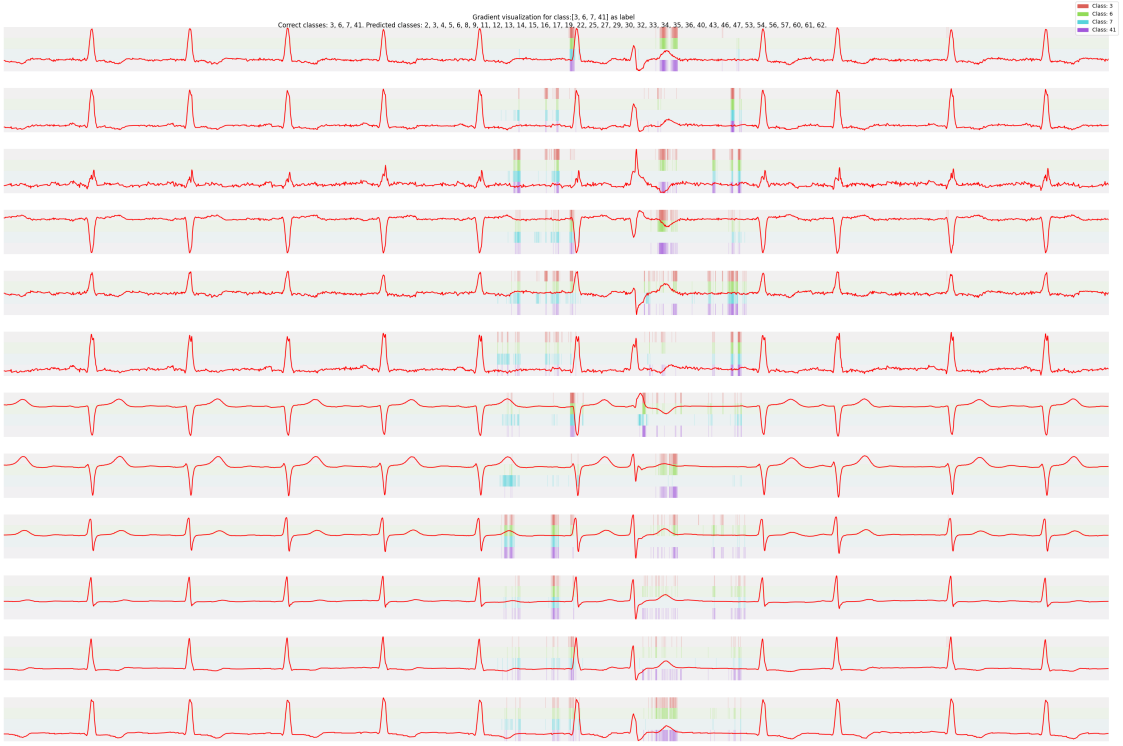
(a) ECG example HR07448



(b) ECG example HR16414

Figure 26: `BL_v8` model gradients. Absolute gradient values utilized. Colored vertical bars/areas show "interesting/controversial" spots in the input data.

(a) ECG example HR07448



(b) ECG example HR16414

Figure 27: CPC model (with `encoder_v0`) gradients. Absolute gradient values utilized. Colored vertical bars/areas show "interesting/controversial" spots in the input data.

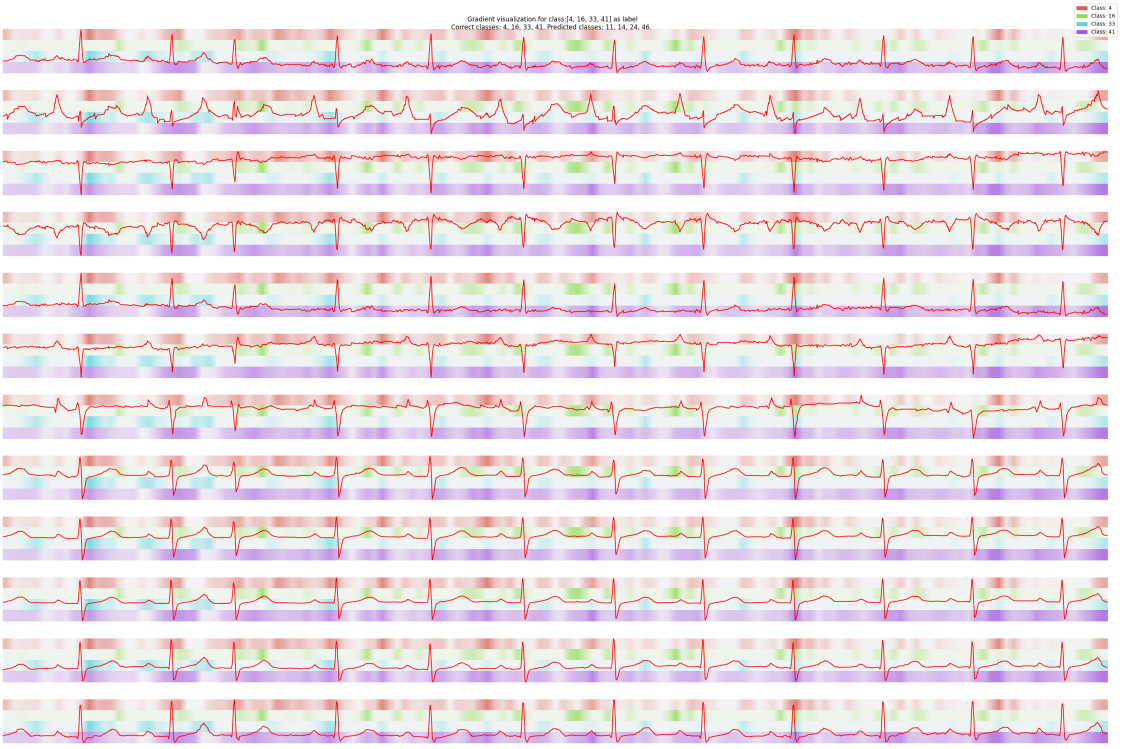


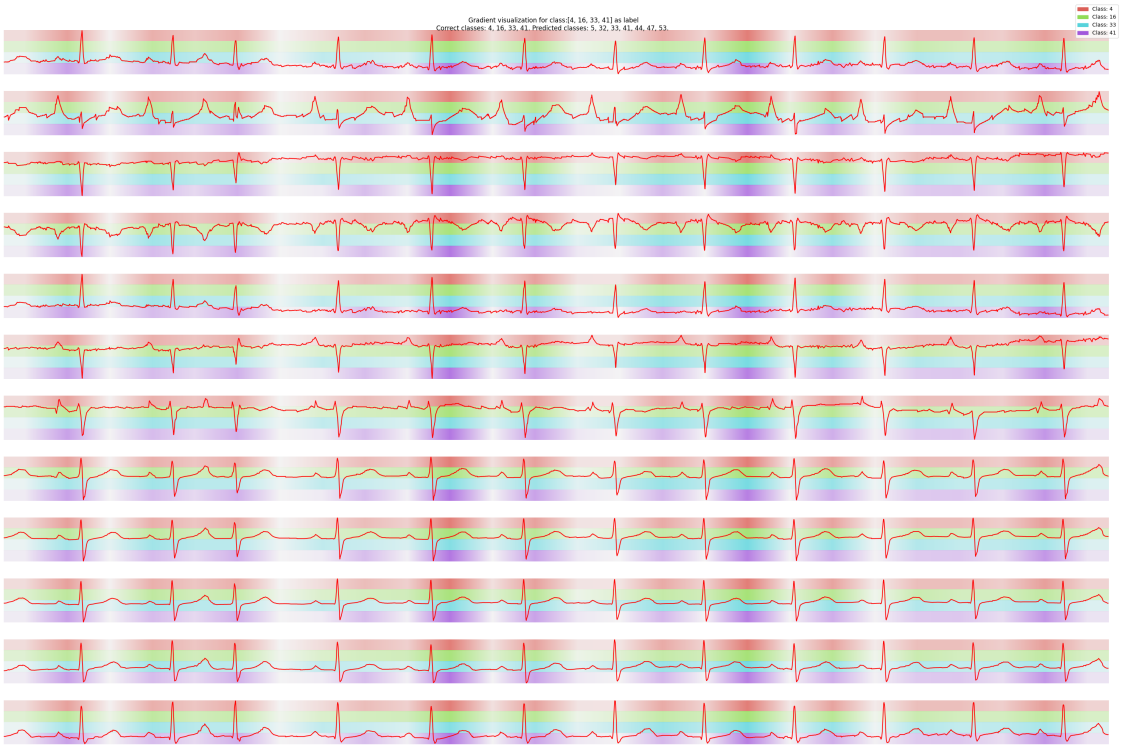
Figure 28: ECG examples, for the BL_v8 Model, Grad-Cam technique used. 147 feature map values stretched to input length of 4500.

all others will be colorized as well, which is a clear downside to our approach. Furthermore the heat map's size is dependent on the downsampling factor in the data dimension of the model architecture: For our `encoder_v0` CPC model only 1×26 values are generated, which allows for a rough localization of classes in the input data only. The BL_v8 model generates 1×147 values which results in much finer localization in the input data and thus more accurate results. The `TCN_down` architecture is probably the most interesting one since it makes use of heavy padding and no strides, making it output 4500 values (same as input) even in the later layers. We expect the best results there.

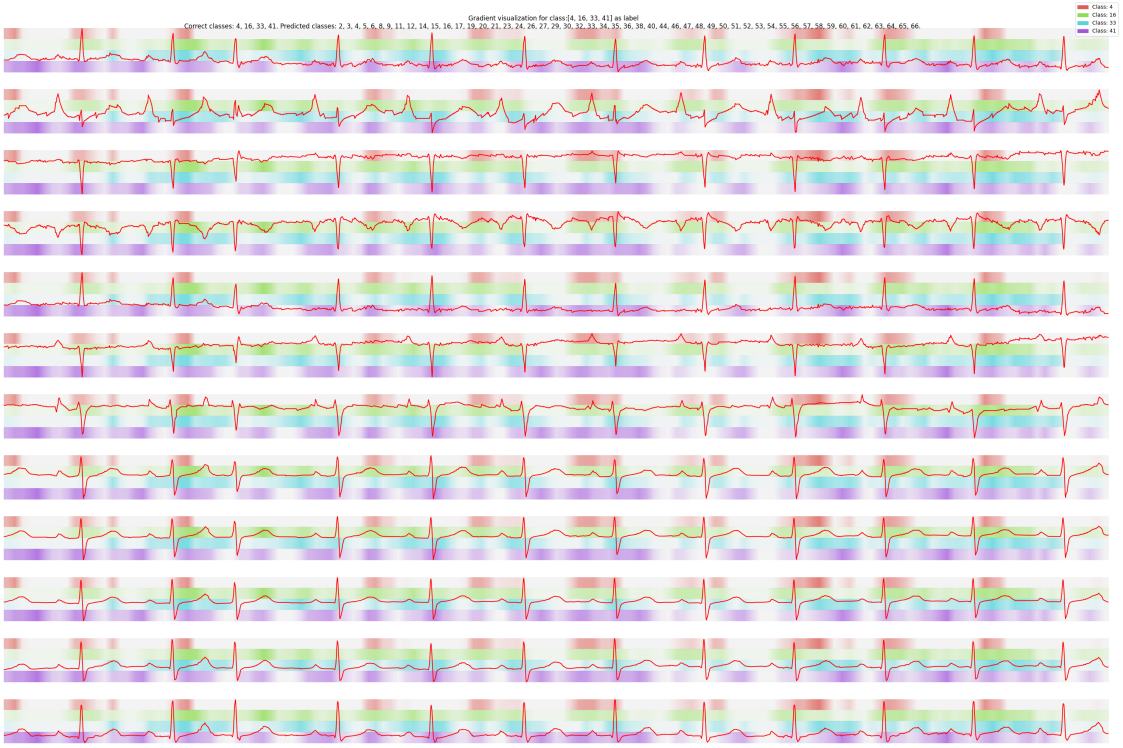
In Figure 28 we see that Grad-Cam created spots in the data that might be able to successfully explain the classes position in the input data. For example the normal sinus rhythm (no. 41) can be found in regular intervals everywhere in the data as expected, while "Left Axis Deviation" (class 33) can only be found at a single spot. "Mycardial Infarction" (class 4) is assumed at 2-3 spots.

For CPC model Figure 29a we see that the grad-cam results are not very localized and all over the place, also the different classes do not seem to differ between one another. We compare to a CPC model that is trained with a different encoder to check whether the poor performance is always present: Figure 29b. We observe similar results as with the baseline model.

The `TCN` baseline model in Figure 30 has a very clear localization of classes, which is probably mostly due to the big output size even in later layers, since the `v8` network has a similar macro average score.



(a) `encoder_v0` used. 26 feature map values stretched to input length of 4500.



(b) `encoder_likev8` used. 147 feature map values stretched to input length of 4500.

Figure 29: ECG examples, for the CPC Model, Grad-Cam technique used.

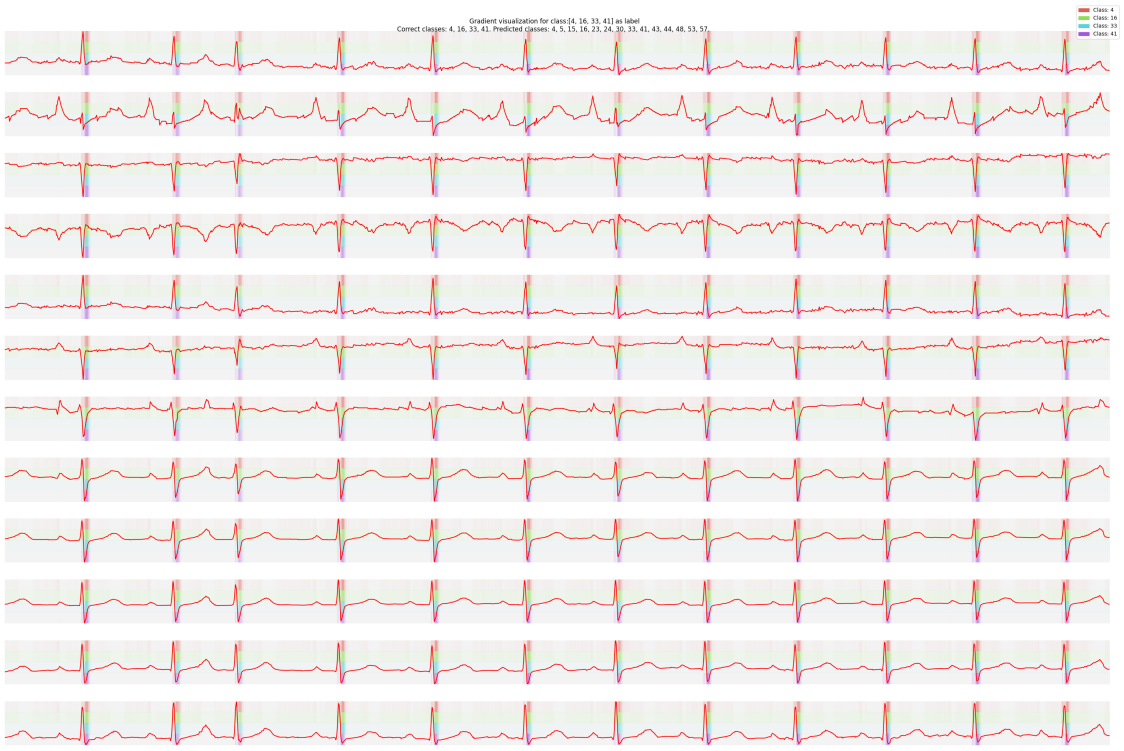


Figure 30: ECG examples, for the TCN (down) architecture, Grad-Cam technique used. 4500 feature map values 'stretched' to input length of 4500.

Last but not least we asked a cardiologist to rate the class markings in the grad-cam images and we were given the following comments:

- In Figure 28 the sinus rhythm (class 41) is off place in some places and generally too spread out. When labeling it you would pick the center of the QRS-points
- Figure 29a is marking too much, no specific locations are clearly recognizable.
- In Figure 29b the sinus rhythm is again off beat in some places but otherwise recognizable. The myocardial infarction is diagnosed mostly by looking at the S-T interval, which is also marked here. Left axis deviation is a "turned-around" QRS complex, so the localization seems okay, but should only be marked in selected channels (eg. channel 3).
- Figure 30 perfectly shows the sinus rhythm center at the QRS complex but disregards the P-wave completely. The myocardial infarction is not diagnosed by looking at the QRS complex, making these markings probably incorrect. Left axis deviation is visible in some channels only, but all got marked.

When asked to pick a "best-performing" figure they selected Figure 29b.

Especially the fact that some classes are only visible in certain channels shows that the grad-cam technique is intended for use on images where different data channels encode

color but have no independent role in localizing different classes. Calculating the gradient towards the input does not share this drawback but the gradient is not as suited for localizing class labels. We tried "Guided-Grad-Cam" which in our case struggled to display anything at all, since guided backpropagation and gradient cam are multiplied to receive the final output, the image was almost always mostly without colors due to values close to 0.

6 Discussion and Future work

In a fully supervised setting, our baselines were able to produce slightly better prediction results compared to our Contrastive Predictive Coding models. Once we handicapped CPC by splitting the training into a pretraining and downstream training phase, with the same number of epochs given in total, the advance for the baseline models increased even further. Yet the CPC results were still above average.

Thus, when labels are available in large enough quantities, supervised learning methods, that have seen much research in the past, are probably easier to optimize and are able to produce better results as a consequence.

In environments where only few labels are given, which we tried to emulate in our "Low Label Availability" experiment in Subsubsection 5.1.4, CPC showed better results compared to baseline models, when the least labels were given, after allowing sufficient training epochs. This is an indicator that CPC pretraining can improve classification results in fields where unlabeled data is readily available, while high quality labels are rare and only obtainable with high costs and effort — which is the case in many fields of modern medicine.

Furthermore our augmented/noised data experiment in Subsubsection 5.1.5 suggests that CPC pretraining increases training stability in the downstream task for unbalanced datasets, because both baseline- and the CPC networks that were trained fully supervised, mostly failed to classify the data satisfactory.

An additional strong benefit of CPC is time efficiency. Downstream tasks, where one has to train models with the same input data, but different labels, benefit of the shared weights obtained during pretraining. Often CPC can yield good results with only few downstream epochs, reducing the necessary total training time drastically. This especially showed during training on the different train splits in "Low Label Availability" experiment: while the baselines were trained completely from ground up for every split, needing more than 24 hours wall clock time, the CPC models were trained in a fraction of that time.

CPC allows for many architectural choices, for example the encoder network and many hyperparameters like latent-size, context-size, number of summarized latents, number of predicted latents and the latent sampling method to name just a few. This makes it very flexible and applicable to many data modalities, as it does not make explicit demands on label shapes and does not require pseudo-labels like many other representation learning algorithms mentioned in the introduction, making it a strong contender for universal representation learning. This also shows in the original paper, where CPC was applied to multiple data modalities, such as images, timeseries or even reinforcement learning[1]. However the great flexibility might also bear the risk of finding only suboptimal network settings for your specific task. This can also be seen in our implementation Subsubsection 5.1.6, where we showed that a context network is not even required to learn useful representations, but also that changing the encoder resulted in worse classification results, albeit giving more flexibility to the user.

Last but not least we applied grad-cam [4] to selected ECG samples, where a CPC network using our `cpc_encoder_likev8` encoder module, was voted the visually best

performing network by a cardiologist.

In conclusion we infer that Contrastive Predictive Coding is an overall good method for classifying ECG-data, but for the pure classification task our baseline models that were trained purely supervised performed better and should be used instead. CPC networks' strengths can be seen when only few labels are available or the data is noisy, but we had to artificially create these environments for the ECG data to outperform the baseline models. CPC yielded the best results in our colorization task, which are however still far from perfect.

Future work might include the search for additional representation learning methods that also cope well with timeseries/sequential/non-image data. This includes the aforementioned altered CPC architectures, but also unrelated representation learning networks like VQ-VAE or more simple encoder-decoder architectures.

Additionally the findings of [39] show that the original CPC has a high bias and needs a lot of negative samples to correctly approach the mutual information value. They introduce a revised CPC called α-CPC that uses weights on both positive and negative samples. According to their formulation the original CPC then becomes a special case of α-CPC. They also introduced a method called "Multi-label Contrastive Predictive Coding" where multiple positive samples are used to greatly decrease the theoretically needed amount of negative samples. Their conducted experiments showed a superior mutual information estimation and slightly better accuracy overall. We tried to reimplement α-CPC but our networks failed to decrease the loss satisfactory. Correctly reproducing their findings could potentially lead to better results in our experiments.

References

[1] Aäron van den Oord, Yazhe Li, and Oriol Vinyals. "Representation Learning with Contrastive Predictive Coding". In: *CoRR* abs/1807.03748 (2018). arXiv: 1807.03748.

[2] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. *An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling*. 2018. arXiv: 1803.01271 [cs.LG].

[3] Fisher Yu and Vladlen Koltun. *Multi-Scale Context Aggregation by Dilated Convolutions*. 2016. arXiv: 1511.07122 [cs.CV].

[4] Ramprasaath R. Selvaraju, Abhishek Das, Ramakrishna Vedantam, Michael Cogswell, Devi Parikh, and Dhruv Batra. "Grad-CAM: Why did you say that? Visual Explanations from Deep Networks via Gradient-based Localization". In: *CoRR* abs/1610.02391 (2016). arXiv: 1610.02391.

[5] *Electrocardiography - Wikipedia*. Jan. 14, 2021. URL: https://en.wikipedia.org/wiki/Electrocardiography (visited on 01/14/2021).

[6] Cables and Sensors EU. *12-Lead ECG Placement Guide with Illustrations*. Sept. 28, 2021. URL: https://www.cablesandsensors.eu/pages/12-lead-ecg-placement-guide-with-illustrations (visited on 09/28/2021).

[7] *ImageNet*. Jan. 14, 2021. URL: http://www.image-net.org/ (visited on 01/14/2021).

[8] *StackOverflow Question*. Jan. 14, 2021. URL: https://stackoverflow.com/a/42979315/3620718 (visited on 01/14/2021).

[9] Ralf-Dieter Bousseljot, D Kreiseler, and A Schnabel. *The PTB Diagnostic ECG Database*. 2004.

[10] Patrick Wagner, Nils Strodthoff, Ralf-Dieter Bousseljot, Wojciech Samek, and Tobias Schaeffter. *PTB-XL, a large publicly available electrocardiography dataset*. 2020.

[11] *Georgia 12-Lead ECG Challenge Database | Kaggle*. Mar. 15, 2021. URL: https://www.kaggle.com/bjoernjostein/georgia-12lead-ecg-challenge-database (visited on 03/15/2021).

[12] Jianwei Zheng, Jianming Zhang, Sidy Danioko, Hai Yao, Hangyuan Guo, and Cyril Rakovski. "A 12-lead electrocardiogram database for arrhythmia research covering more than 10, 000 patients". In: *Scientific Data* 7.1 (Feb. 2020).

[13] Erick A Perez Alday, Annie Gu, Amit J Shah, Chad Robichaux, An-Kwok Ian Wong, Chengyu Liu, Feifei Liu, Ali Bahrami Rad, Andoni Elola, Salman Seyedi, Qiao Li, Ashish Sharma, Gari D Clifford, and Matthew A Reyna. "Classification of 12-lead ECGs: the PhysioNet/Computing in Cardiology Challenge 2020". In: *Physiological Measurement* 41.12 (Jan. 2021), p. 124003.

[14] Ary L. Goldberger, Luis A. N. Amaral, Leon Glass, Jeffrey M. Hausdorff, Plamen Ch. Ivanov, Roger G. Mark, Joseph E. Mietus, George B. Moody, Chung-Kang Peng, and H. Eugene Stanley. "PhysioBank, PhysioToolkit, and PhysioNet". In: *Circulation* 101.23 (June 2000).

[15] *SNOMED CT - Classes | NCBO BioPortal*. Mar. 16, 2021. URL: https://bioportal.bioontology.org/ontologies/SNOMEDCT/?p=classes&conceptid=root (visited on 03/16/2021).

[16] *I now call it "self-supervised learning"*. June 26, 2021. URL: https://www.facebook.com/722677142/posts/10155934004262143/ (visited on 06/26/2021).

[17] Mehdi Noroozi and Paolo Favaro. *Unsupervised Learning of Visual Representations by Solving Jigsaw Puzzles*. 2017. arXiv: 1603.09246 [cs.CV].

[18] Spyros Gidaris, Praveer Singh, and Nikos Komodakis. *Unsupervised Representation Learning by Predicting Image Rotations*. 2018. arXiv: 1803.07728 [cs.CV].

[19] Carl Doersch, Abhinav Gupta, and Alexei A. Efros. *Unsupervised Visual Representation Learning by Context Prediction*. 2016. arXiv: 1505.05192 [cs.CV].

[20] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. *A Simple Framework for Contrastive Learning of Visual Representations*. 2020. arXiv: 2002.05709 [cs.LG].

[21] Cheng-I Lai. "Contrastive Predictive Coding Based Feature for Automatic Speaker Verification". In: *arXiv preprint arXiv:1904.01575* (2019).

[22] Michael Gutmann and Aapo Hyvärinen. "Noise-contrastive estimation: A new estimation principle for unnormalized statistical models". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. PMLR, 13–15 May 2010, pp. 297–304.

[23] *Mutual information - Wikipedia*. Mar. 25, 2021. URL: https://en.wikipedia.org/wiki/Mutual_information (visited on 03/31/2021).

[24] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL].

[25] Olivier J. Hénaff, Aravind Srinivas, Jeffrey De Fauw, Ali Razavi, Carl Doersch, S. M. Ali Eslami, and Aaron van den Oord. *Data-Efficient Image Recognition with Contrastive Predictive Coding*. 2020. arXiv: 1905.09272 [cs.CV].

[26] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG].

[27] Zhiguang Wang, Weizhong Yan, and Tim Oates. *Time Series Classification from Scratch with Deep Neural Networks: A Strong Baseline*. 2016. arXiv: 1611.06455 [cs.LG].

[28] *geekfeiw/Multi-Scale-1D-ResNet: pytorch code of multi scale 1d resnet, we hope it will help your research*. May 26, 2021. URL: https://github.com/geekfeiw/Multi-Scale-1D-ResNet (visited on 05/26/2021).

[29] *alexnet.py pytorch*. June 26, 2021. URL: https://github.com/pytorch/vision/blob/master/torchvision/models/alexnet.py (visited on 06/26/2021).

[30] Mohammad Kachuee, Shayan Fazeli, and Majid Sarrafzadeh. "ECG Heartbeat Classification: A Deep Transferable Representation". In: *2018 IEEE International Conference on Healthcare Informatics (ICHI)* (June 2018).

[31] Stefan Harmeling, Guido Dornhege, David Tax, Frank Meinecke, and Klaus-Robert Müller. "From outliers to prototypes: Ordering data". In: *Neurocomputing* 69.13–15 (Aug. 2006), pp. 1608–1618.

[32] *Brier Score: Definition, Examples - Statistics How To*. Feb. 9, 2022. URL: https://www.statisticshowto.com/brier-score/ (visited on 02/16/2022).

[33] Will Koehrsen. *Beyond Accuracy: Precision and Recall | by Will Koehrsen | Towards Data Science*. Mar. 22, 2021. URL: https://towardsdatascience.com/beyond-accuracy-precision-and-recall-3da06bea9f6c (visited on 03/22/2021).

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

[35] Laurens van der Maaten and Geoffrey Hinton. "Visualizing Data using t-SNE". In: *Journal of Machine Learning Research* 9.86 (2008), pp. 2579–2605.

[36] *Inappropriate Sinus Tachycardia | Cedars-Sinai*. Sept. 7, 2021. URL: https://www.cedars-sinai.org/health-library/diseases-and-conditions/i/inappropriate-sinus-tachycardia.html (visited on 09/07/2021).

[37] *Sinus Bradycardia | Cedars-Sinai*. Sept. 7, 2021. URL: https://www.cedars-sinai.org/health-library/diseases-and-conditions/s/sinus-bradycardia.html (visited on 09/07/2021).

[38] Renu Khandelwal. *How to Visually Explain any CNN based Models | by Renu Khandelwal | Towards Data Science*. Jan. 18, 2022. URL: https://towardsdatascience.com/how-to-visually-explain-any-cnn-based-models-80e0975ce57 (visited on 01/18/2022).

[39] Jiaming Song and Stefano Ermon. *Multi-label Contrastive Predictive Coding*. 2020. arXiv: 2007.09852 [cs.LG].

[40] Jacob Gildenblat and contributors. *PyTorch library for CAM methods*. https://github.com/jacobgil/pytorch-grad-cam. 2021.

[41] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. *Representation Learning with Contrastive Predictive Coding*. 2019. arXiv: 1807.03748 [cs.LG].

[42] *Entropy (information theory) - Wikipedia*. Mar. 29, 2021. URL: https://en.wikipedia.org/wiki/Entropy_(information_theory) (visited on 03/31/2021).

[43] *Classification of 12-lead ECGs: the PhysioNet/Computing in Cardiology Challenge 2020 | PhysioNet/CinC Challenges*. Mar. 15, 2021. URL: https://physionetchallenges.org/2020/ (visited on 03/16/2021).

[44] Jan Chorowski, Grzegorz Ciesielski, Jarosław Dzikowski, Adrian Łańcucki, Ricard Marxer, Mateusz Opala, Piotr Pusz, Paweł Rychlikowski, and Michał Stypułkowski. *Aligned Contrastive Predictive Coding*. 2021. arXiv: 2104.11946 [cs.LG].

[45] Patrick Wagner, Nils Strodthoff, Ralf-Dieter Bousseljot, Dieter Kreiseler, Fatima I. Lunze, Wojciech Samek, and Tobias Schaeffter. "PTB-XL, a large publicly available electrocardiography dataset". In: *Scientific Data* 7.1 (May 2020).

[46] Ary L. Goldberger, Luis A. N. Amaral, Leon Glass, Jeffrey M. Hausdorff, Plamen Ch. Ivanov, Roger G. Mark, Joseph E. Mietus, George B. Moody, Chung-Kang Peng, and H. Eugene Stanley. "PhysioBank, PhysioToolkit, and PhysioNet". In: *Circulation* 101.23 (June 2000).

[47] R. Bousseljot, D. Kreiseler, and A. Schnabel. "Nutzung der EKG-Signaldatenbank CARDIODAT der PTB über das Internet". In: *Biomedizinische Technik/Biomedical Engineering* (July 2009), pp. 317–318.

[48] *MIT-LCP/wfdb-python: Native Python WFDB package*. Jan. 11, 2021. URL: https://github.com/MIT-LCP/wfdb-python (visited on 01/11/2021).

[49] *Github Masterarbeit*. June 26, 2021. URL: https://github.com/Lullatsch/ecg-cpc (visited on 06/26/2021).

List of Figures

List of Tables