UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Project Outside the Course Scope report

Hierarchical Normalizing Flow for ECG data

Mikolaj Mazurczyk

Supervisor: Oswin Krause

Submitted: January 29, 2023

Contents

Introduction

Recent years have shown many advancements in generative deep learning models, including the development of new architectures and training methods, constantly pushing boundaries of what is possible to achieve in machine learning. Key features of these models, motivating their recent popularity, include the ability to model highly complex dependencies of high dimensional data in an unsupervised manner, which is especially advantageous in the current big data world, where often human labeling and categorization are unfeasible due to abundance and complexity of the datasets. Therefore it is not surprising that these models have been applied to a wide range of fields, including computer vision, natural language processing, and speech synthesis. However, their use in the field of medicine is also rapidly growing, with a variety of applications that have the potential to revolutionize the way we diagnose and treat diseases. This project aims to further expand this research by applying a normalizing flow-based generative neural network to the electrocardiogram (ECG) record data, which would allow the generation of artificial datasets, as in (Thambawita et al. [2021]), but also thanks to the properties of flow models, one could normalize individual patients data, to perform outliers detection.

The key idea behind these models is to transform an arbitrary base distribution into a complex target distribution of the data through a series of invertible transformations. Since the mapping is direct, by a change of variables formula, one can use the flows for the target's density estimation, exact latent-variable inference, and exact log-likelihood evaluation. Moreover, if the base distribution is easy to sample from (such as multivariate normal) and the computational cost of invertible transformations is low, then sampling from the target distribution can be done very efficiently. In our implementation, we decided to further utilize those properties by making the architecture hierarchical, meaning that at particular points of mapping the target density to the base one, input is split into two parts: one that undergoes further transformations and another that is left out to the base distribution (see Figure 1). Using this multi-scale definition, we can potentially learn features on a global and local scale and later test whether the learned features correspond to any patient data. Thus, by checking the base distribution at different scales, we could get an insight into which features influence the target distribution the most. However, this poses a crucial constraint, which determined many of our design choices behind the flow's architecture: to preserve multi-scale latent variable semantics, at each level model can have access to only local information of the input, otherwise strongest and weakest features will influence the network at each scale, loosing their hierarchical interpretability. The report is organized as follows:

- In the Background section, I introduce the definition and basics of how flow models operate, along with the notation used throughout the report.

- General architecture contains the core description of the model, which remained about the same as we were developing the project.
- Then I introduce the Process of developing the final model, where I describe how the project advanced through time, finishing with the final model definition. The section also includes a description of the dataset and the test suite.
- Finally, the Conclusion summarizes the project's results and discusses potential future work.

Background

Notation

Bold symbols are used to indicate vectors (lowercase) and matrices (uppercase) where index i of vector $\mathbf{x}$ is marked in a subscript $\mathbf{x}_i$. Otherwise, variables are scalars. Inputs with channel and spatial dimensions are also written as bold lowercase symbols, and in that case, $\mathbf{x}_{i,j}$ denotes the element in the i-th channel and j-th index of the spatial dimension. It should be clear from the context when the lowercase bold symbol indicates vector and when it marks a two-dimensional ECG.

Normalizing Flows

In principle, the objective of the flow-based model is to create a direct mapping between two probability densities defined over continuous random variables using invertible transformations. Let $\mathbf{x}$ be a D-dimensional vector with unknown but well-behaved joint distribution p_x, further referred to as a *target distribution*. The idea behind the flow-based models is to pick an arbitrary (but usually simple) well-behaved *base distribution* p_z and find an invertible and differentiable transformation f, for which we have

$$\mathbf{x} = f(\mathbf{z}), \quad \text{where} \quad \mathbf{z} \sim p_z(z).$$

Then, by the change of variables formula, we get

$$p_x(\mathbf{x}) = p_z(\mathbf{z}) \left|\det J_f(\mathbf{z})\right|^{-1},$$

and thanks to f being invertible, the above is equivalent to

$$p_x(\mathbf{x}) = p_z\left(f^{-1}(\mathbf{x})\right) \left|\det J_{f^{-1}}(\mathbf{x})\right|,$$

where $J_{f^{-1}}(\mathbf{x})$ is the Jacobian of all partial derivatives of f^{-1}. The crucial characteristic of invertible and differentiable functions is that they can be

decomposed into multiple transformations $f = T_K \circ \cdots \circ T_1$ that hold the same properties as f. Then the inverse has a simple closed form

$$f^{-1} = (T_K \circ \cdots \circ T_1)^{-1} = T_1^{-1} \circ \cdots \circ T_K^{-1},$$

and its Jacobian determinant is

$$\det\left(J_{f^{-1}}(\mathbf{x})\right) = \det J_{T_1^{-1} \circ \cdots \circ T_K^{-1}}(\mathbf{x}) = \prod_{k=1}^{K} \det J_{f^{-1}}^{k}(\mathbf{x})$$

due to the chain rule, where

$$J_{f^{-1}}^{k}(\mathbf{x}) = \begin{cases} J_{T_k^{-1}}\left(\left(T_{k+1}^{-1} \circ \cdots \circ T_K^{-1}\right)(\mathbf{x})\right), & \text{if } 1 \leq k < K \\ J_{T_K^{-1}}(\mathbf{x}), & \text{if } k \equiv K \end{cases},$$

which can be interpreted as the Jacobian of transformation T_k^{-1} evaluated on the argument returned from chaining the preceding transformations, as going from $\mathbf{x}$ to $\mathbf{z}$.

Considering that f can be parametrized by θ, it becomes clear how normalizing flows fit in the scope of the deep learning optimization problem: given a dataset $\mathcal{D}$ of N elements $\mathbf{x}^{(i)}$ sampled with some unknown true distribution p_x, we try to approximate it with p_θ using a simple base distribution p_z and a neural network f with layers T_k, which is done by directly minimizing the negative log-likelihood

$$\begin{aligned} \mathcal{L}(\mathcal{D}) &= -\frac{1}{N}\sum_{i=1}^{N} \log p_\theta\left(\mathbf{x}^{(i)}\right) \\ &= -\frac{1}{N}\sum_{i=1}^{N} \log\left(p_z\left(f^{-1}\left(\mathbf{x}^{(\mathbf{i})}\right)\right)\left|\det J_{f^{-1}}\left(\mathbf{x}^{(\mathbf{i})}\right)\right|\right) \\ &= -\frac{1}{N}\sum_{i=1}^{N} \log p_z\left(f^{-1}\left(\mathbf{x}^{(i)}\right)\right) + \sum_{k=1}^{K} \log\left|\det J_{f^{-1}}^{k}\left(\mathbf{x}^{(i)}\right)\right|. \end{aligned}$$

From a practical point of view, this means that for training, inverses T_k^{-1} and their log-determinants must be efficiently computable. If one of the model's applications is sampling, it poses the same constraint on the T_k transformations.

To better understand how flow-based networks function, a simple instructive example might be in order. For the purpose of deriving the determinants of Jacobians and implementing the training routine, it is more natural to think of mapping from p_θ to p_z as a *forward* pass. Therefore the rest of this report is written in this view, contrary to the formulation in the paragraph above. Let $p_z = \mathcal{N}(\mathbf{0}, \mathbf{I})$, and f be a simple network consisting of two affine linear layers,

$$\text{Linear1}(\mathbf{x}) = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1, \qquad \text{Linear1}(\mathbf{x}) = \mathbf{W}_2\mathbf{x} + \mathbf{b}_2,$$

with Leaky ReLU

$$\text{LeakyReLU}\left(\mathbf{x}_{i,j}\right) = \begin{cases} \mathbf{x}_{i,j}, & \text{if } \mathbf{x}_{i,j} \geq 0 \\ \text{negative\_slope} \cdot \mathbf{x}_{i,j}, & \text{otherwise} \end{cases}$$

in between, as an activation function. If we use the notation

$$\begin{aligned} f(\mathbf{x}) &= \left(\text{Linear2} \circ \text{LeakyReLU} \circ \text{Linear1}\right)(\mathbf{x}) \\ &= \left(\text{Linear2} \circ \text{LeakyReLU}\right)\left(\mathbf{x}'\right) \\ &= \text{Linear2}\left(\mathbf{x}''\right) \\ &= \mathbf{z}, \end{aligned}$$

the loss has a form

$$\begin{aligned} \mathcal{L}(\mathbf{x}) = &- \log p_z(\mathbf{z}) - \log \left|\det J_{\text{Linear2}}\left(\mathbf{x}''\right)\right| - \log \left|\det J_{\text{LeakyReLU}}\left(\mathbf{x}'\right)\right| \\ &- \log \left|\det J_{\text{Linear1}}(\mathbf{x})\right| \end{aligned}$$

which in practice gives a raise to the following training loop: we collect the log-determinants of Jacobians as $\mathbf{x}$ gets transformed through consecutive layers, then we evaluate $p_z(\mathbf{z})$ (which in the case of this example is straightforward, as p_z is multivariate gaussian) and finally, the loss gets backpropagated through the network.

So the only aspect left to specify is how the log determinants of transformations are computed, and for completeness of this example, I will also provide their inverses. In the case of Leaky ReLU, the calculations are straightforward, as it has a simple inverse

$$\text{LeakyReLU}^{-1}\left(\mathbf{x}_{i,j}\right) = \begin{cases} \mathbf{x}_i, & \text{if } \mathbf{x}_i \geq 0 \\ \text{negative\_slope}^{-1} \cdot \mathbf{x}_i, & \text{otherwise} \end{cases}$$

and the log-determinant

$$\log\left(\text{negative\_slope}^{k}\right) = k \cdot \log\left(\text{negative\_slope}\right) \;\Big|\; k = \sum_{i=1}^{c}\sum_{j=1}^{l} \mathbb{1}\left[\mathbf{x}_{i,j} < 0\right]$$

since its Jacobian is a diagonal matrix with entries equal to either 0 or negative slope, depending on the input value. At first, the derivations for the linear layer may also seem trivial, as its inverse can be essentially written as

$$\mathbf{x} = \mathbf{W}^{-\mathbf{1}}\left(\mathbf{z} - \mathbf{b}\right),$$

which also naturally produces the Jacobian's log-determinant of form

$$\log\left(\left|\det\left(\mathbf{W}\right)\right|\right).$$

However, $\mathbf{W}$ might not be invertible, and even if it is, in general case, computing its inverse and determinant takes $\mathcal{O}\left(c^3\right)$ time. To alleviate those issues,

one of the approaches (which we also used in our implementation) is to initialize $\mathbf{W}$ as a random orthogonal matrix, which ensures that it has a determinant of either 1 or -1, and then factorize it using the PLU decomposition, where all the elements of the diagonal of matrix $\mathbf{L}$ are set to 1. Finally, we save the sign of the determinant, and we keep the logarithm of the diagonal $\mathbf{u}_{\log}$ of the upper triangular matrix $\mathbf{U}$ as a vector of trainable parameters (meaning $\mathbf{u}_{\log} = \log\left(\operatorname{diag}\left(|\mathbf{U}|\right)\right)$).

During the forward pass, we reconstruct the weight matrix in the following way

$$\mathbf{W} = \mathbf{PL}\left(\mathbf{U} \pm \operatorname{diag}\left(\mathbf{u}\right)\right),$$

where the elements on the diagonal of $\mathbf{L}$ are set to 1 and the elements on the diagonal of $\mathbf{U}$ are set to 0, and $\mathbf{u} = \exp\left(\mathbf{u}_{\log}\right)$. Then, the log-determinant is easy to obtain

$$\begin{aligned}
\log\left(\left|\det\left(W\right)\right|\right) &= \log\left(\left|\det\left(\mathbf{PL}\left(\mathbf{U} \pm \operatorname{diag}\left(\mathbf{u}\right)\right)\right)\right|\right) \\
&= \log\left(\left|\det\left(\mathbf{P}\right)\det\left(\mathbf{L}\right)\det\left(U \pm \operatorname{diag}\left(\mathbf{u}\right)\right)\right|\right) \\
&= \log\left(\left|(\pm 1)\cdot 1\cdot\left(\pm e^{\operatorname{sum}\left(\mathbf{u}_{\log}\right)}\right)\right|\right) \\
&= \operatorname{sum}\left(\mathbf{u}_{\log}\right),
\end{aligned}$$

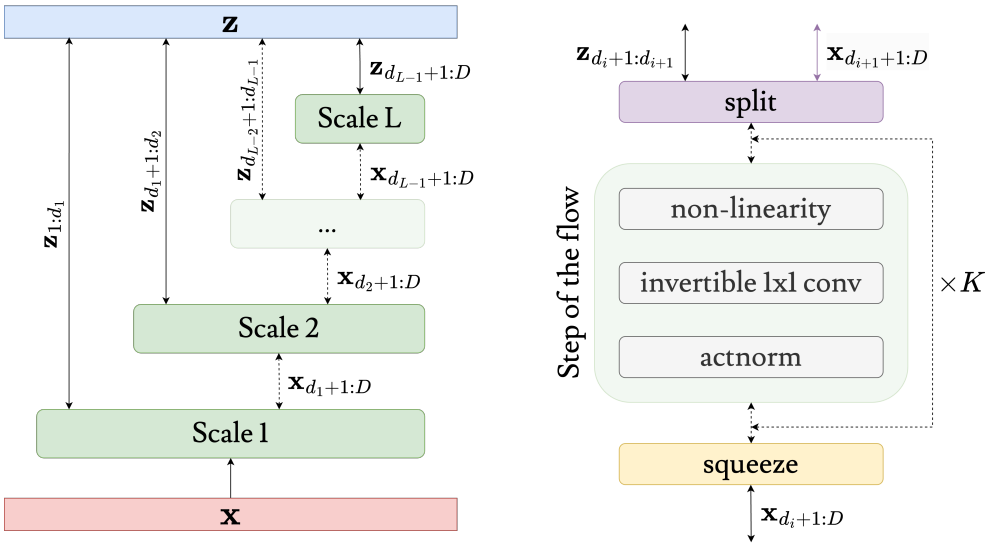
thus also ensuring that $\mathbf{W}$ is invertible since $e^{\operatorname{sum}\left(\mathbf{u}_{\log}\right)} \neq 0$. The inverse $\mathbf{W}^{-1}$ also can be easily calculated by solving the lower/upper triangular systems by forward/backward substitution in $\mathcal{O}\left(c^2\right)$ time and by taking $\mathbf{P}^{-1} = \mathbf{P}^T$.

General architecture

The model architecture builds upon the work of (Kingma and Dhariwal [2018]), where the authors introduced the Generative Flow with Invertible 1×1 Convolutions, coined Glow (which, in turn, extends the architecture of Real NVP (Dinh et al. [2016])). It is a hierarchical flow-based neural network designed to be run on high-resolution image datasets which especially fits the needs of our project: the multi-scale architecture and support for multiple channel inputs. Another crucial reason behind this choice was the introduction of the invertible 1×1 convolution, which is discussed in more detail further in this section.

Hierarchical flow

The flow consists of L scales, which can be changed as a hyperparameter of the network. Each scale in the forward pass first changes the dimensionality of the input, runs it through multiple steps of the flow (where the actual learning of the model happens), and finally splits it into two parts: one, which goes directly into the base distribution, and the other undergoing further training as it is passed to the successive scale.



(a) Multi-scale architecture.

(b) Scale's composition.

Figure 1: Visualization of the model's structure on the global (left) and local levels (right). For simplicity, $\mathbf{x}$ and $\mathbf{z}$ are depicted as vectors where $d_i{+}1 : d_{i+1}$ subscripts denote their non-overlapping partitioning, and D marks the spatial size of both $\mathbf{x}$ and $\mathbf{z}$.

Squeeze operation

The squeeze operation is responsible for trading the signal length for the number of channels. Specifically, for an input of shape $c \times d$ (where c denotes the number of channels and d marks the signal length/spatial dimension), it takes every second element of spatial dimension per channel, concatenates them, and adds as an additional channel, leading to the output of shape $2c \times \frac{d}{2}$. Visualization of this process is shown in Figure 3. In both invertible 1×1 convolution and actnorm layers, the number of trainable parameters increases with the number of channels (and it is independent of the size of spatial dimension) hence squeeze operation effectively improves their expressive power at higher levels of the flow.

Splitting

Each scale, except for the last, after squeezing the input and passing it through multiple steps of the flow, splits it into two parts. We decided to split the datapoint into two equally shaped parts, $\mathbf{x}$ and $\mathbf{z}$, where the number of channels in both outputs stays the same, at the expense of dividing the signal's lengths

by two. Further, based on $\mathbf{x}$, a constant for each element of $\mathbf{z}$ is computed and then subtracted from it. It follows the idea that since $\mathbf{x}$ and $\mathbf{z}$ are expected to be similar, deducting an interpolation of elements of $\mathbf{x}$ from $\mathbf{z}$ would help the network to map $\mathbf{z}$ to the base distribution, which in our case was $\mathcal{N}(\mathbf{0}, \mathbf{I})$. The way we split the inputs and the details of calculating the constants for the interpolation changed as the project progressed. Therefore they are discussed more exhaustively in the Process of developing the final model, with visualization shown in Figure 3.

Nevertheless, it implies that at each hierarchical level, the input of shape $c \times d$ is transformed into two parts, both shaped $2c \times \frac{d}{4}$ (firs by squeezing, and then by splitting). Therefore, if the model takes the initial ECG of shape $c' \times d'$, then on the last scale, it transforms it into a $\left(2^L c'\right) \times \left(d'/2^{2L-1}\right)$ tensor, meaning we need to ensure that $d' \equiv 0 \pmod{2^{2L-1}}$ holds. Since the ECG signal is mainly flat at its tails, we decided to truncate the minimal required number of elements from both ends so that the condition holds.

Step of the flow

Finally, the step of the flow is the part of the network with trainable parameters. On each scale, those steps are chained together K times, where K is one of the network's hyperparameters. The block consists of three layers, applied consecutively to the input: activation normalization, invertible 1×1 convolution, and a layer responsible for introducing non-linearity.

Actnorm

In (Kingma and Dhariwal [2018]), the authors introduced the activation normalization layer (actnorm in short), which performs affine transformation of the inputs using a separate scale $\mathbf{s}_i$ and bias $\mathbf{b}_i$ parameter per channel $\mathbf{x}_i$. The normalization aspect comes from the fact that the parameters in the first forward pass of the training are initialized using minibatch's per channel mean $\tilde{\mu}_i$ and variance $\tilde{\sigma}_i^2$. Therefore in the first training's pass, input is transformed in the following way

$$\forall_{i \in \{1,\dots,c\}} \mathbf{x}_i \to (\mathbf{x}_i + \mathbf{b}_i) \odot \mathbf{s}_i \;\Big|\; \mathbf{b}_i = -\tilde{\mu}_i,\ \mathbf{s}_i = 1/\sqrt{\tilde{\sigma}_i^2 + \epsilon},$$

effectively rescaling it to channel-wise zero mean and unit variance. During initialization, the $\epsilon = 10^{-6}$ is added to the variance for numerical stability. After setting initial values, $\mathbf{s}$ and $\mathbf{b}$ are treated as trainable parameters. Naturally, the inverse operation has a form of

$$\forall_{i \in \{1,\dots,c\}} \mathbf{z}_i \to \mathbf{z}_i \odot \mathbf{s}_i - \mathbf{b}_i,$$

and since each element is scaled independently, the Jacobian is a diagonal matrix with each scale parameter occurring d times, resulting in the log-

determinant

$$d \cdot \text{sum}\left(\log |\mathbf{s}|\right).$$

Invertible 1x1 convolution

The main improvement that the authors of Glow introduced compared to Real NVP is the invertible 1×1 convolutional layer. The main advantage of this approach in the scope of our project is that the layer learns across-channel dependencies, using only local information of the spatial dimension, allowing us to model more and more global features as the input passes through the scales. Since our data has only a single spatial dimension, c filters with kernels of size one are convolved with the data point, producing the output of the same shape.

Given the square $c \times c$ weight matrix $\mathbf{W}$ and the input of a form

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_{1,1} & \dots & \mathbf{x}_{1,d} \\ \vdots & \ddots & \vdots \\ \mathbf{x}_{c,1} & \dots & \mathbf{x}_{c,d} \end{bmatrix}$$

the transformation and its inverse can be essentially written as

$$\mathbf{W}\mathbf{x} = \mathbf{z}, \quad \mathbf{x} = \mathbf{W}^{-\mathbf{1}}\mathbf{z},$$

which also naturally produces the Jacobian's log-determinant of form

$$d \cdot \log |\det \mathbf{W}|.$$

We ensured the invertibility of $\mathbf{W}$ and efficient computation of its inverse and log-determinant in the same way as described in the Background section.

Non-linearity

Thus far, all layers produce a linear combination of the inputs, so naturally, what is left is to introduce an activation function to model the non-linear dependencies in the data. For simplicity, initially, we chose Leaky ReLU for this task, but after running into the numerical stability issues, we again took inspiration from Glow and Real NVP by using the affine coupling layer. Both of the approaches are discussed in more detail in the next section.

Process of developing the final model

Dataset

Since the project didn't reach satisfactory enough results, the experiments were only limited to the artificially generated dataset using a GAN (Goodfellow et al. [2014]) model developed by (Thambawita et al. [2021]). Specifically,

our architecture was trained on 150,000 1.2-s median representative beats obtained from the full 8-lead 10s ECGs. The whole dataset was normalized channel-wise to zero mean and unit variance to improve the training process (examples can be seen in the Appendix A).

Initially, ECGs were loaded directly from files every time a minibatch was processed, but as it turned out, even when utilizing 32 CPUs to run concurrently 32 processes responsible for loading the data, it persisted as a bottleneck of training's efficiency. Therefore, we decided to save the whole dataset as PyTorch's tensor using the pickle protocol, which in turn increased the memory usage, but to an acceptable extent. In most experiments, we kept 10% of the dataset for validation and used minibatches of size 512.

Tests

To examine the correctness of the implementation, we developed a test suite using the `pytest`[1] library. Specifically, we tested:

1. the invertibility of the transformations by checking if $T^{-1}\left(T\left(\mathbf{x}\right)\right) \approx \mathbf{x}$ for the whole model and each layer separately,

2. whether log determinants are computed correctly by comparing the determinant of automatically calculated Jacobian of flow's transformations with the values directly calculated by the implementation,

3. whether the empirical mean and variance of the whole dataset mapped to the base distribution by the trained flow match the $\mathcal{N}\left(\mathbf{0}, \mathbf{I}\right)$.

In the next section, I will refer to each test by the number corresponding to its place in the above list.

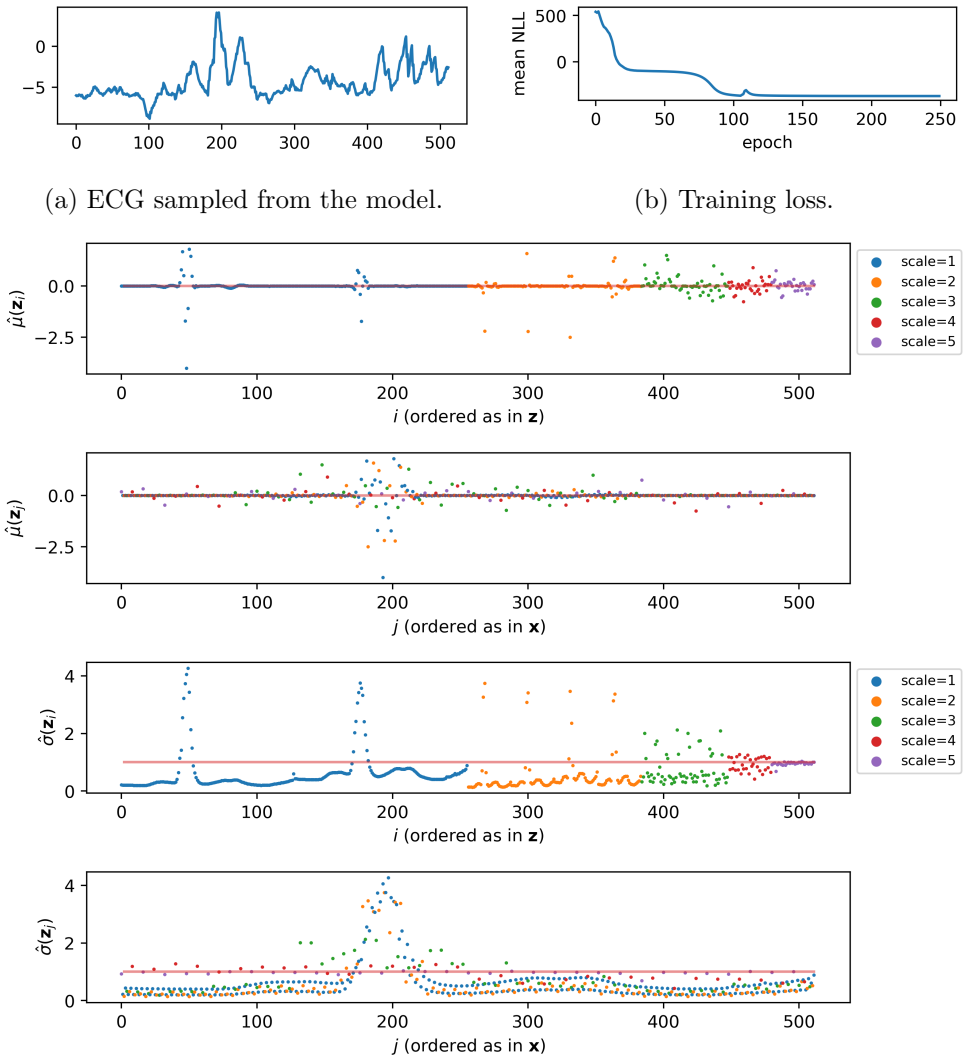
Iteration I

Since ReLU is a standard activation function in modern neural networks, initially, we used its invertible analog, the Leaky ReLU, where its inverse and log-determinant follow the same derivations as shown in the Background section. For the interpolation step in the splitting part of the network, we decided to initially use a simple approach by subtracting the mean of the adjacent elements of the spatial dimension of the part that undergoes further training from the part that is being directly mapped to the base distribution. Otherwise, the model's structure remained the same as described in General architecture.

At this point, we run into issues with numerical stability. When tested separately, every layer passed both tests (1) and (2) on all configurations. However, when evaluated on the whole flow with negative slope values ≤ 0.1,

[1] `https://docs.pytest.org/en/7.2.x/`

only instances with $K \leq 4$ passed the tests. Training statistics of the model, with different configurations of the parameters, also confirmed the results: for larger K values and smaller values of negative slope, the training was very unstable, where often, during the initial steps, the loss was growing instead of decreasing. After those initial trials, we also realized that L needs to have a fixed value, dependent on the data. Specifically, since the components of our base distribution are independent, flow needs to learn how to transform them into dependent ones, as clearly, components of our target distribution are correlated. However, that would be impossible if some of the input elements were never combined by the network, and since we restrict spatial elements from influencing each other, it means that at the last scale, the spatial dimension of each channel has to equal one. Thus in the further experiments, we fixed $L = 5$ since median ECGs have a length of 600, which fulfills the above constraint (how dimensions change is discussed in more detail in the Splitting section). We also incorporate a negative slope equal to 0.2 and the number of steps between 2 and 8. For this setting, learning rates of $10^{-4}/10^{-5}$ proved to work well, and for simplicity, we used only a single (first) lead of the median ECGs.



(a) ECG sampled from the model.

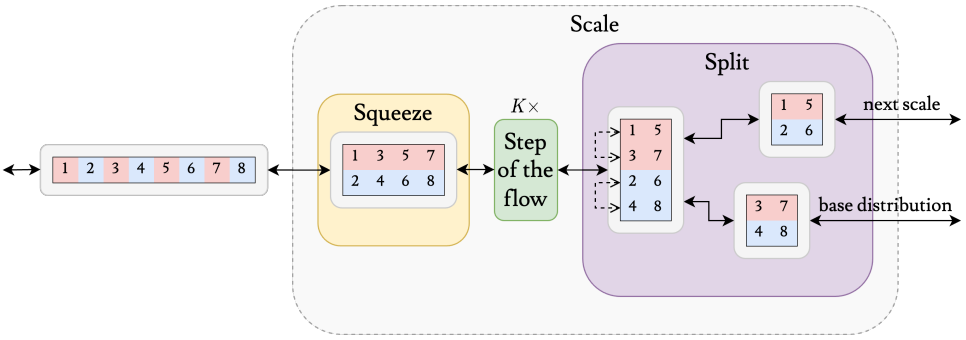
(b) Training loss.

(c) Visualisation of the test (3).

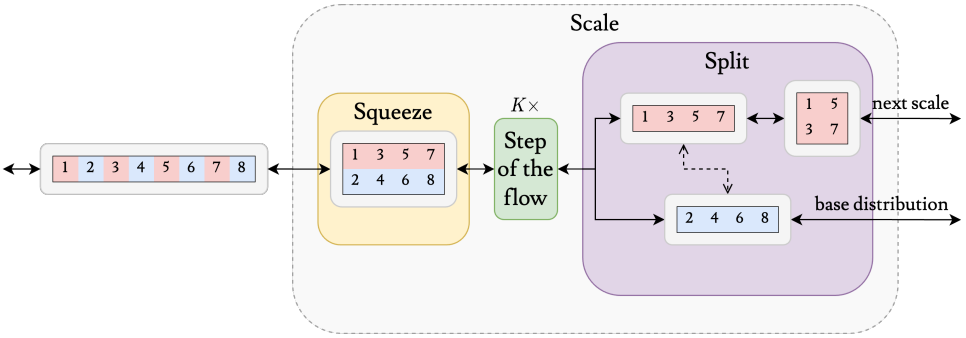
Figure 2: Results of training the Iteration I model, where $L = 5$, $K = 2$, with the learning rate set to 10^{-4} and LeakyReLU's negative slope set to 0.2. Figure (c) depicts the visualization of mean $\hat{\mu}$ and standard deviation $\hat{\sigma}$ obtained from the test (3), first ordered by scale (as they appear in $\mathbf{z}$ [base distribution]) and then by how they appear in $\mathbf{x}$ (ECG).

Iteration II

As Figure 2 shows, the results were far from adequate. However, a great insight came from the visualization of the test (3) (Figure 2c), which resulted in two main modifications.



(a) As done in the Iteration I.



(b) As done in the Iteration II.

Figure 3: Visualization of how input is reshaped by the flow's scale at different iterations. Dashed arrows mark pairs of vectors, where the interpolation of elements of the vector that undergoes further training is subtracted from the one mapped to the base distribution (from the perspective of the forward pass).

First, we can see that components mapped to the base distribution at higher scales are not evenly spaced when re-ordered as they appear in the initial ECG. After examination, we realized that the way input was split and reshaped was incorrect. Specifically, in Iteration I, input was squeezed from shape $c \times d$ to $2c \times \frac{d}{2}$, and then we would take every second spatial element of the $2c \times \frac{d}{2}$-shaped tensor when splitting, which meant that we were "clustering" together even and odd spatial elements. The following scales propagated it even further, producing unexpected results in the test (3). Therefore we changed the implementation of the splitting layer to take every second channel

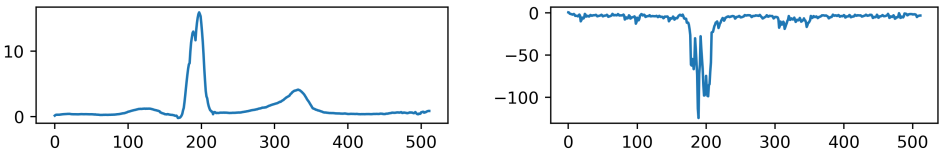
of $2c \times \frac{d}{2}$-shaped output when splitting, and then another squeeze operation is performed on the part that goes further through the model so that the number of channels still increases after each scale of the flow. Since visual explanations are much more instructive in this case, I provide a graphical representation of the described change in Figure 3.

The second key observation is that for the lower scales, the part of the input that is left out during the split matches the base distribution much worse compared to the higher hierarchical levels. It is especially true for the variance, where its distribution fits more the distribution of ECG-s rather than the base one. To solve this issue, we tried applying the K' steps of the flow to the part that is left out after splitting. However, even though it improved the results of the test (3), the training was even more unstable, as adding more layers increased the numerical instability of the network.

Therefore to mitigate those issues, we decided to apply just a single actnorm layer once instead of multiple steps of the flow. However, as each element might have different distribution when being mapped to the base distribution, we increased the number of parameters in the actnorm layer so that each element has a separate scale $\mathbf{s}_{i,j}$ and bias $\mathbf{b}_{i,j}$, thus changing the layer's log-determinant to just

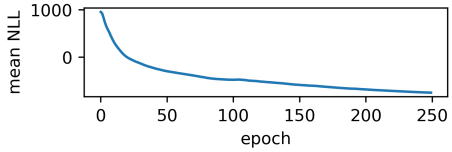
$$\sum_{i=1}^{c} \sum_{j=1}^{d} \log\left(|\mathbf{s}_{i,j}|\right).$$

This also means that now the parameters are initialized using element-wise mean and variance of the minibatch. We applied this change to all actnorm layers in the network.

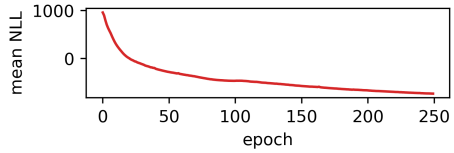


(a) ECG sampled from the model.

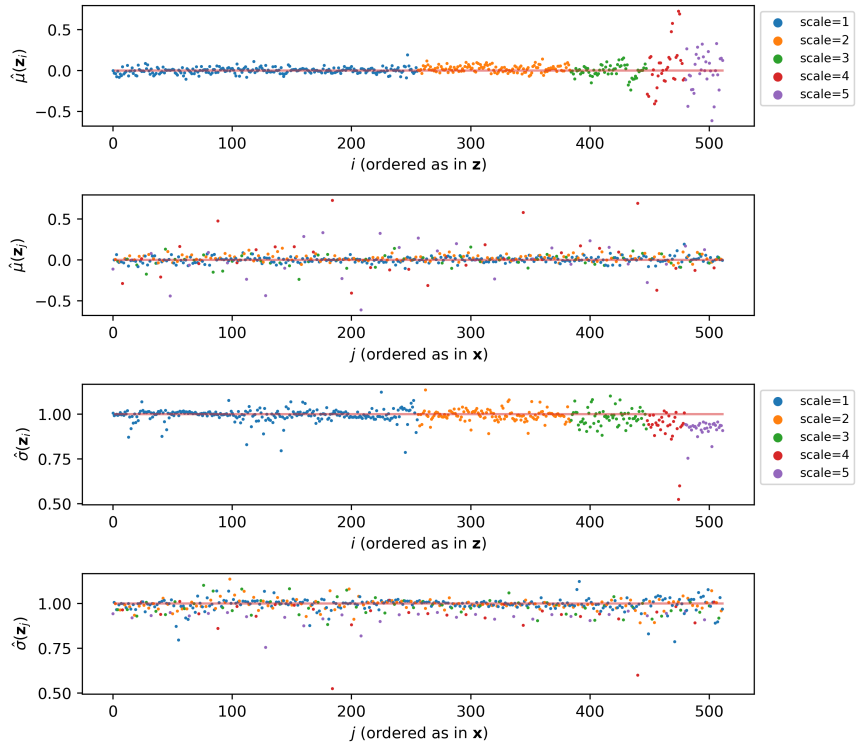
(b) ECG sampled from the model.



(c) Training loss.



(d) Validation loss.



(e) Visualisation of the test (3).

Figure 4: Results of training the Iteration II model with single actnorm after splitting, where $L = 5$, $K = 4$, with the learning rate set to 10^{-4} and LeakyReLU's negative slope set to 0.2. We can observe considerable improvement compared to the Figure 2, however, even when sometimes samples resemble the proper ECG as in (a), they are noisy and the amplitude is completely off. Still, more often than not model generates samples looking closer to the one in (b).

Final model

After multiple attempts of trying to get the model defined in the Iteration II to work, we still could not improve upon the results presented in Figure 4. Since the results suggest that it is due to issues with numerical stability, and tests (1) and (2) indicate that Leaky ReLU might be the main factor causing this issue, in our final attempt, we decided to replace it with a much more powerful Affine Coupling Layer introduced in NICE (Dinh et al. [2014]), which was also incorporated in Real NVP (Dinh et al. [2016]) and Glow(Kingma and Dhariwal [2018]). The main advantage of the layer is that it allows the application of an arbitrarily complex neural network to the input while still ensuring that the inverse operation and Jacobian's log-determinant are easy to calculate.

Given a D dimensional input $\mathbf{x}$, we first split it into two non-overlapping parts, $\mathbf{x}_{1:d}$ and $\mathbf{x}_{d+1:D}$, where $d < D$. Then $\mathbf{x}_{1:d}$ is run through a neural network `NN`, producing scale $\mathbf{s}$ and bias $\mathbf{b}$ parameters which are used to transform $\mathbf{x}_{d+1:D}$ element-wise to $\mathbf{x}'_{d+1:D} = \mathbf{x}_{d+1:D} \odot \mathbf{s} + \mathbf{b}$. Finally, we keep the left part as it is $\mathbf{x}'_{1:d} = \mathbf{x}_{1:d}$, and we concatenate it back with the transformed $\mathbf{x}'_{d+1:D}$ obtaining output $\mathbf{x}'$. The transformation has the following partial derivatives

- naturally $\frac{\partial \mathbf{x}'_{1:d}}{\partial \mathbf{x}^T_{1:d}} = \mathbf{I}_d$ and $\frac{\partial \mathbf{x}'_{1:d}}{\partial \mathbf{x}^T_{d+1:D}} = \mathbf{0}$, since $\mathbf{x}'_{1:d} = \mathbf{x}_{1:d}$,
- $\frac{\partial \mathbf{x}'_{d+1:D}}{\partial \mathbf{x}^T_{d+1:D}} = \operatorname{diag}(\mathbf{s})$ because $\mathbf{s}$ is independent from $\mathbf{x}_{d+1:D}$, so it can be treated as a scaling constant,

and we can ignore calculating $\frac{\partial \mathbf{x}'_{d+1:D}}{\partial \mathbf{x}^T_{1:d}}$ since the Jacobian is a triangular matrix

$$\left[\begin{array}{c|c} \frac{\partial \mathbf{x}'_{1:d}}{\partial \mathbf{x}^T_{1:D}} & \frac{\partial \mathbf{x}'_{1:d}}{\partial \mathbf{x}^T_{d+1:D}} \\ \hline \frac{\partial \mathbf{x}'_{d+1:D}}{\partial \mathbf{x}^T_{1:d}} & \frac{\partial \mathbf{x}'_{d+1:D}}{\partial \mathbf{x}^T_{d+1:D}} \end{array}\right] = \left[\begin{array}{c|c} \mathbf{I}_d & \mathbf{0} \\ \hline \frac{\partial \mathbf{x}'_{d+1:D}}{\partial \mathbf{x}^T_{1:d}} & \operatorname{diag}(\mathbf{s}) \end{array}\right]$$

with a simple log-determinant

$$\sum_{i=1}^{D-d} \log(\mathbf{s}_i).$$

The inverse operation is then straightforward: since $\mathbf{x}'_{1:d} = \mathbf{x}_{1:d}$, we can run it through the network to obtain identical $\mathbf{s}$ and $\mathbf{b}$, which are used to reverse the linear transformation $\mathbf{x}_{d+1:D} = (\mathbf{x}'_{d+1:D} - \mathbf{b}) \odot \mathbf{s}$, and by concatenating, we end up back with $\mathbf{x}$. It is worth noting that in the Glow implementation, and also ours, before obtaining $\mathbf{s}$, the output of the `NN` is first run through the sigmoid function to bound the gradients[2], which improves the numerical stability of the network. We found that, at least in our case, it was impossible

	in channels	out channels	kernel size	padding
Conv1d #1	c	v	3	1
Conv1d #2	v	v	1	0
Conv1d #3	v	$2c$	3	1

Table 1: Structure of the `NN` used in the affine coupling layer. Here c denotes the number of input channels, and v marks the number of filters that varied between the experiments. Final convolution outputs $2c$ channels since the output is then split channel-wise in half producing, $\mathbf{s}$ and $\mathbf{b}$. Also, ReLU activation is applied between convolutions #1/#2 and #2/#3.

to train the model without this modification. For the `NN`, we used a simple convolutional neural network, which is described in more detail in Table 1.

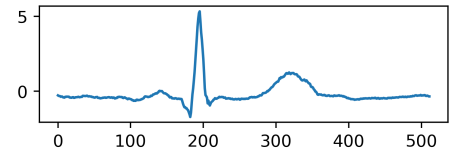
Now, what is left to specify, is how exactly the input is split in the coupling layer. In NICE, the input is partitioned in half along the channel dimension, and after transformation, two parts are concatenated back, again along the channel dimensions. However, in RealNVP, the authors argued that in the case of multi-scale architecture, it would leave some components unchanged. Therefore they incorporate splitting in a checkerboard manner (as they work with two-dimensional inputs) and use an altering pattern of which half is passed through `NN` between the layers. However, Glow implementation returns to the way inputs were split in NICE, motivating the choice by introducing 1×1 invertible convolution, which, as they argue, is a generalization of permutation, and therefore hard-coding it is not necessary. However, as there are no guarantees that the invertible convolution will learn the permutation, we tested both Real NVP and Glow approaches, where in our case, the splitting was done as in the splitting part of the network (i.e. by taking every second channel), and the pattern was alternating at each consecutive step of the flow (meaning that if one coupling layer passed even numbered channels to the `NN`, then the next one used odd). In our case, the alternating approach proved to work better, so we also applied it in our final experiments in Figures 5 and 7, and the results of Glow's method applied to our setting are shown in Appendix B. Notably, our experiments employ much smaller values of K than the authors present in Glow, which might account for why in our case invertible 1×1 convolutions were unable to learn the permutation.

To make our model even more expressive, we also replaced the mean interpolation in the splitting part of the network with the affine coupling layer, where $\mathbf{s}$ and $\mathbf{b}$ are obtained using the half that is passed to successive scales, and the learned values are used to rescale the other half that is mapped to the base distribution.

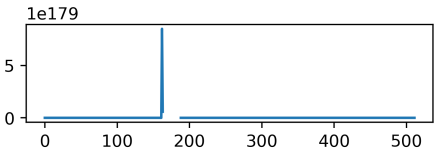
In this setting, we noted a significant leap in every aspect of the network's performance. For the first time, the implementation passes tests (1) and (2)

[2] https://github.com/openai/glow/issues/62

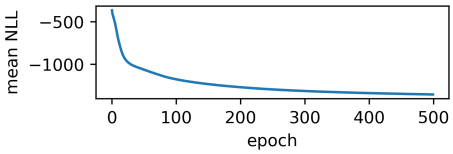
for all configurations, even for K values as high as 32. In the final evaluation, we trained two networks, one on a single lead and the second on full eight leads, and in both cases, test (3) showed that the model captured variance almost perfectly, while the mean is only slightly off, mainly for deeper scales. Now, models more consistently produce samples resembling proper ECG-s (Figure 5a and 6a), and the more distorted samples (Figure 6b) were a minority across our sampled data, although the model occasionally produces extraordinarily wrong outputs, as in Figure 5b. Nevertheless, under closer inspection, we can see that even the best samples are still noisy and therefore do not contain a quality of acceptable ECG signal. We tried experimenting with different settings of hyperparameters, but the results did not improve upon what is presented in this section. Since we were getting close to the project submission deadline, we decided to keep that architecture as the final design of our flow.



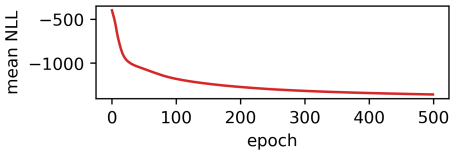
(a) ECG sampled from the model.



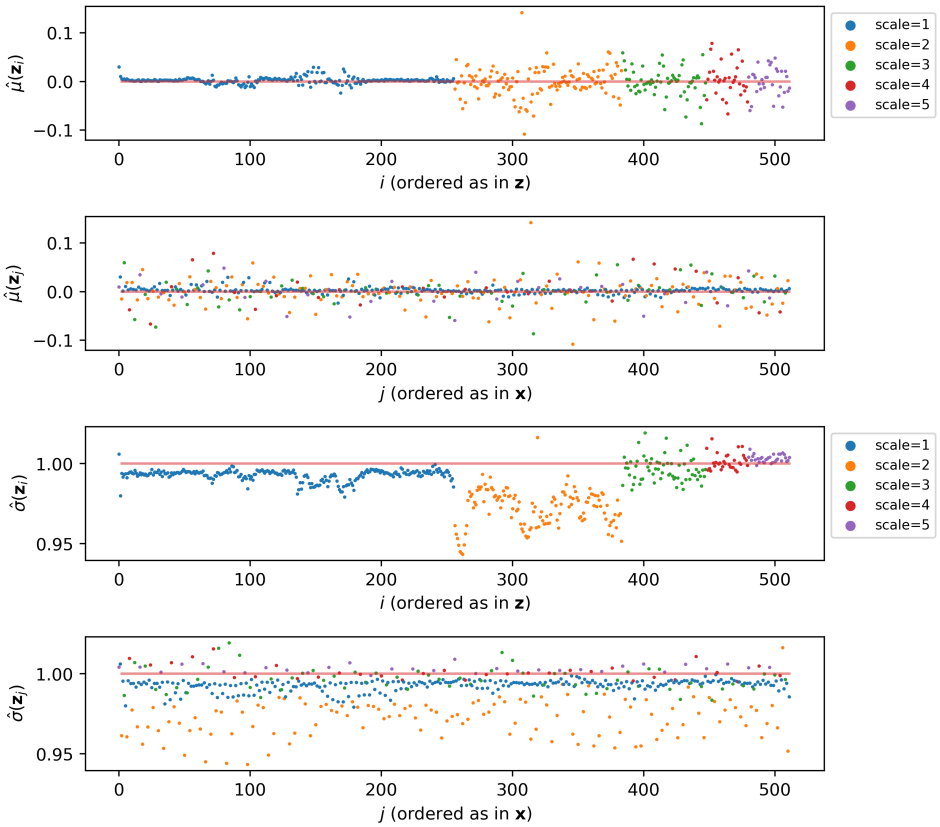
(b) ECG sampled from the model.



(c) Training loss.

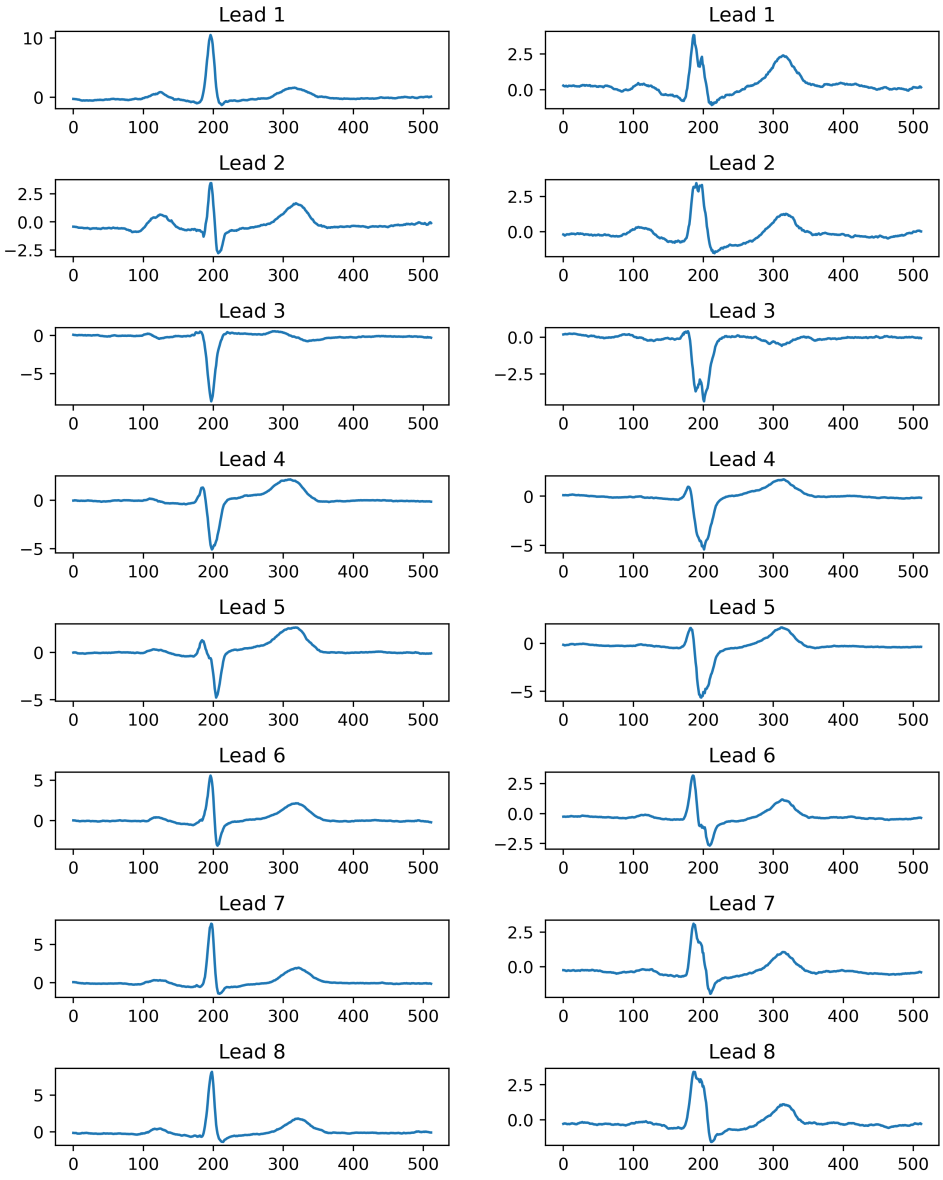


(d) Validation loss.



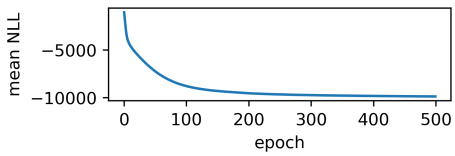
(e) Visualisation of the test (3).

Figure 5: Results of training the Final model on a single (first) lead ECGs, where $L = 5$, $K = 4$, $v = 32$ (see Table 1), with the learning rate set to 10^{-5}.

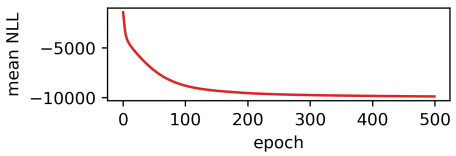


(a) ECG sampled from the model. (b) ECG sampled from the model.

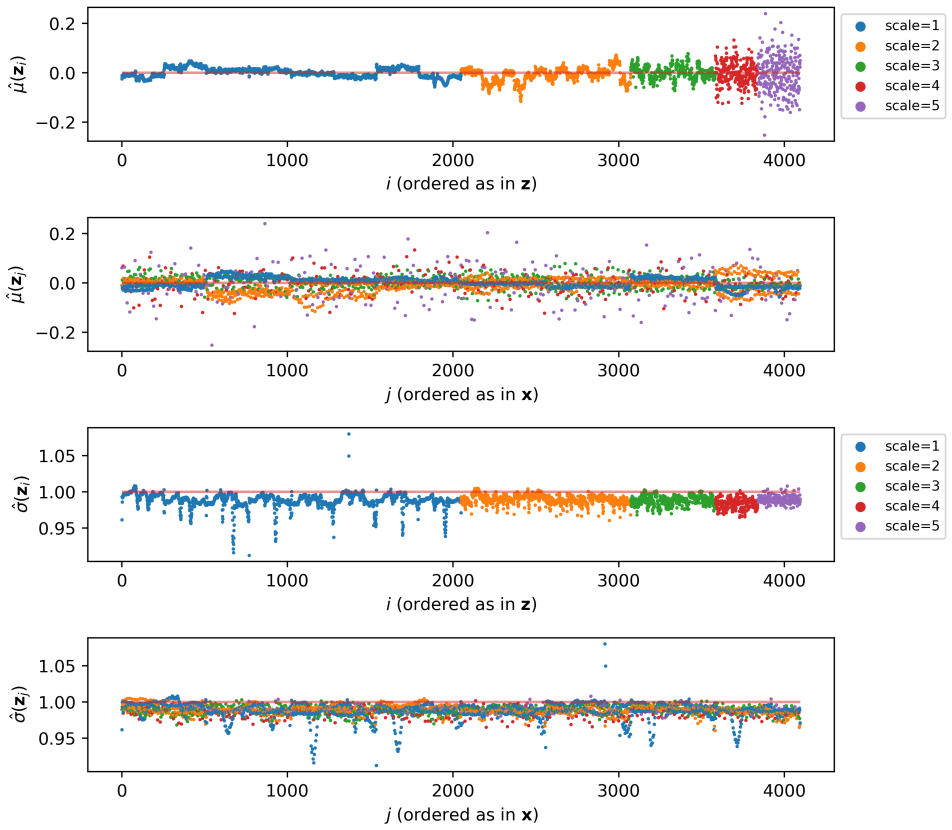
Figure 6: Example instances of ECGs sampled from the same model as in Figure 7.



(a) Training loss.



(b) Validation loss.



(c) Visualisation of the test (3).

Figure 7: Results of training the Final model on eight lead median ECGs, where $L = 5$, $K = 2$, $v = 128$ (see Table 1), with the learning rate set to 10^{-5}.

Conclusion

This project has presented a detailed implementation and evaluation of a Hierarchical Normalizing Flow Network, along with the iterative process of improving the model. The proposed network was able to effectively model the complex distribution of the dataset used in the experiments and managed to generate synthetic samples that were visually similar to the input data. I have also shown the mathematical foundations of normalizing flows, with derivations of inverts and log-determinants of all model's layers, and presented motivations behind our design choices, given the project's initial assumptions.

However, since the samples were not of the expected quality, we didn't achieve the project's initial goal of running the models on real patients' data to examine if the network could learn features at different scales. Compared to other generative models, like GANs, flow-based generative models have so far gained little attention in the research community and even less in the industry. Hence there is a general lack of out-of-the-box solutions that proved to work well in practice, which meant that we had to implement and evaluate many functionalities from scratch. That, in turn, prolonged the process of developing the implementation.

Despite these challenges, the results of this project demonstrate the potential of normalizing flows as a powerful tool for generative modeling and highlight the importance of further research in this area. The proposed model can be used as a starting point for future work and can be extended to other datasets and applications. Majority of the source code used to produce the results discussed in the report is available at GitHub (`https://github.com/mikmaz/ecg-norm-flow`).

Bibliography

Laurent Dinh, David Krueger, and Yoshua Bengio. Nice: Non-linear independent components estimation, 2014. URL https://arxiv.org/abs/1410.8516.

Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using real NVP. *CoRR*, abs/1605.08803, 2016. URL http://arxiv.org/abs/1605.08803.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014. URL https://proceedings.neurips.cc/paper/2014/file/5ca3e9b122f61f8f06494c97b1afccf3-Paper.pdf.

Diederik P. Kingma and Prafulla Dhariwal. Glow: Generative flow with invertible 1x1 convolutions, 2018. URL https://arxiv.org/abs/1807.03039.

Vajira Thambawita, Jonas L. Isaksen, Steven A. Hicks, Jonas Ghouse, Gustav Ahlberg, Allan Linneberg, Niels Grarup, Christina Ellervik, Morten Salling Olesen, Torben Hansen, Claus Graff, Niels-Henrik Holstein-Rathlou, Inga Strümke, Hugo L. Hammer, Mary M. Maleckar, Pål Halvorsen, Michael A. Riegler, and Jørgen K. Kanters. Deepfake electrocardiograms using generative adversarial networks are the beginning of the end for privacy issues in medicine. *Scientific Reports*, 11(1):21896, Nov 2021. ISSN 2045-2322. doi: 10.1038/s41598-021-01295-2. URL https://doi.org/10.1038/s41598-021-01295-2.

Appendix A

Example ECGs from the dataset

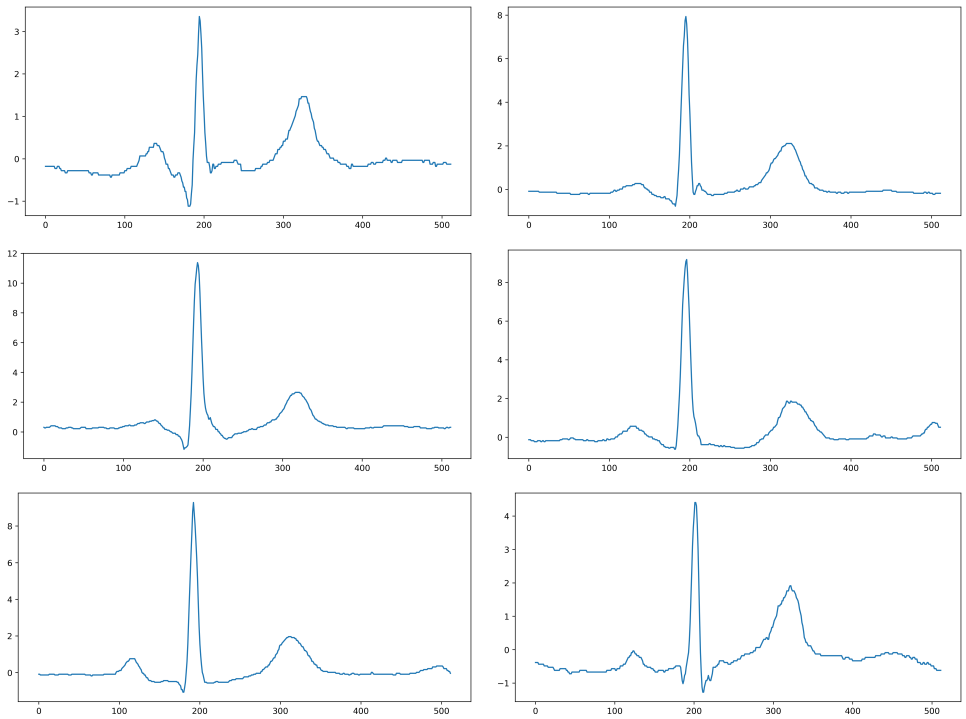


Figure 1: A closer look at the first leads of four example ECGs from the dataset.

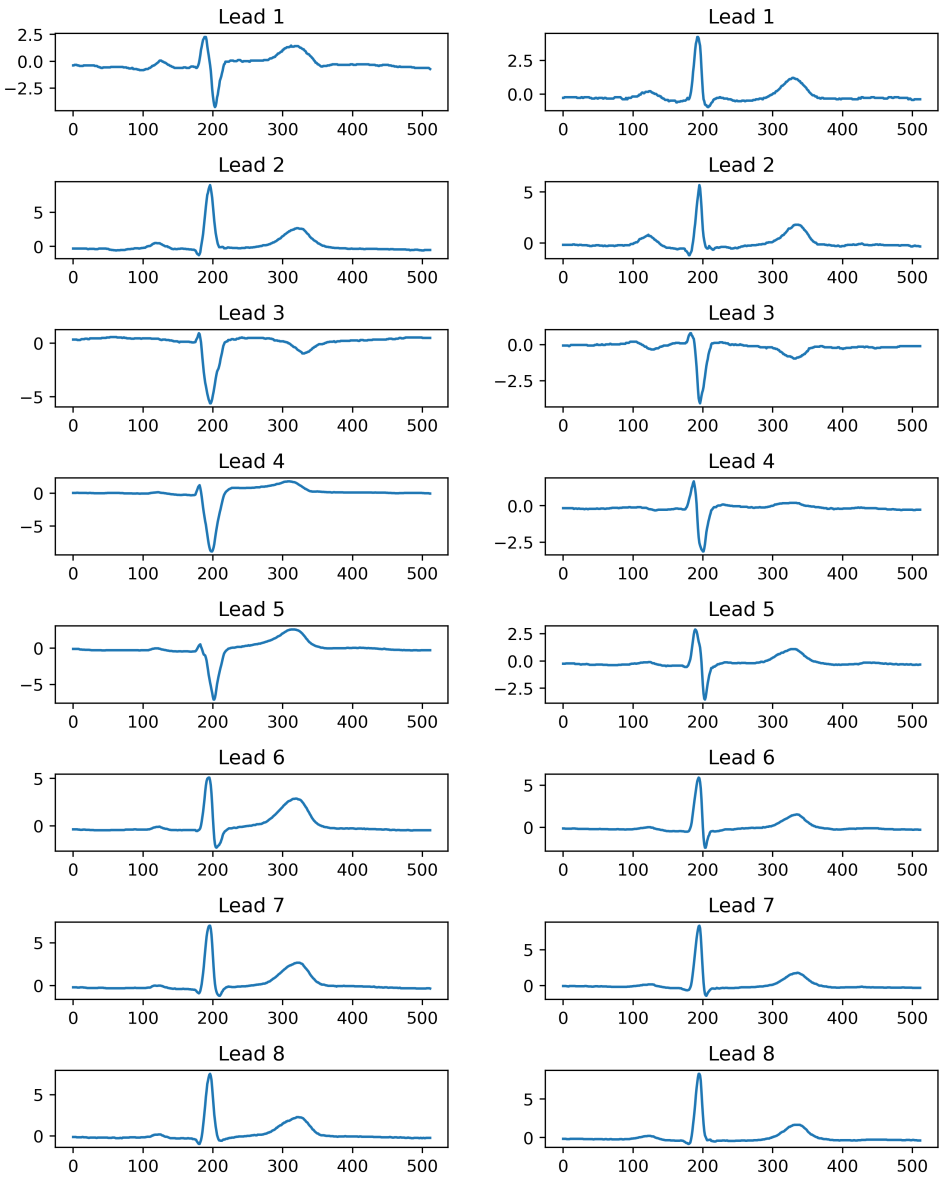
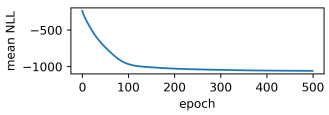


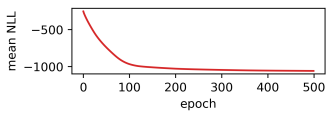
Figure 2: Example of two eight lead median ECGs from the dataset.

Appendix B

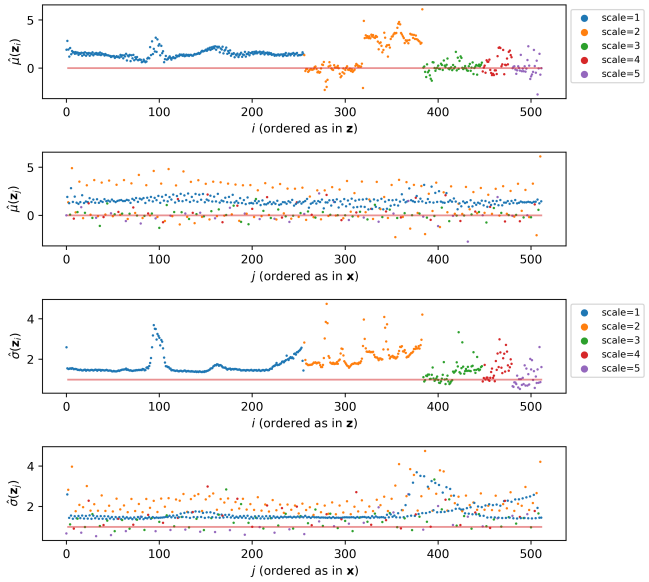
Affine Coupling Layer without alternating pattern



(a) Training loss.



(b) Validation loss.



(c) Visualisation of the test (3).

Figure 1: Results of training the Final model, but without alternating the way inputs are passed to the affine coupling layer. The hyperparameters were $L = 5$, $K = 4$, $v = 32$ (see Table 1), with the learning rate set to 10^{-5}.