Using Machine Learning to Recreate Signals from the Primary Visual Cortex of Mice

by

Markus Leira Asprusten

Thesis

for the degree of

Master of Science

Faculty of Mathematics and Natural Sciences
University of Oslo

June 2020

Abstract

Generative adversarial networks (GAN) have received much attention lately for its use with images and has been shown to be able to create extremely realistic images of different kinds of objects. Even though its use in images is popular, there has not been much study into using this type of generative method for other types of data. GANs replicate the distribution of a data set to produce realistic samples that are not in the data set. Being an adversarial method based on game theory, training a GAN can be difficult. If not tweaked correctly, the GAN can collapse and produce unrealistic results. A different kind of machine learning method called an autoencoder is more stable as it is based on replicating data instead of replicating the distribution of the data. Experimental biology needs large amounts of data, which is sometimes difficult to procure in sufficient amounts. Using generative methods in these instances have much potential. Proposed here is using an autoencoder to stabilize training of a GAN for use with biological signals. Autoencoders were shown to be stable, and replicated input data almost exactly. The GAN model was shown to perform better when pre-trained with an autoencoder. The samples produced by the GAN were not realistic as the model requires more training.

Contents

Part I

Theory

Chapter 1

Introduction

In experimental science, there is always a need for more data. Traditionally, this data could only come from one source: experiments. However, experimental data can be both difficult and expensive to obtain. Experiments can be challenging to set up, equipment can be expensive to procure, and it might also be difficult to find an ethically satisfactory way to do experiments. These problems are especially prevalent when working with live specimens in biology. Finding new ways to reduce the number of experiments needed is a good way of addressing these problems.

Using machine learning to generate new data based on already available data has gained popularity in the recent years. Perhaps most prominent the use of Generative Adversarial Networks (GANs) to create realistic images by training on real images[13].

GANs have not been subject to much research outside of its use with images. Although it has been used with electroencephalographic (EEG) brain signals[9], there have not been much use with other kinds of time series[9]. This thesis will study using GANs to replicate local field potentials (LFPs) to a life-like degree. To do this, the model needs to be very realistic and replicate as many important features as possible.

GANs are notoriously difficult to train. Autoencoders are more stable and easier to train but lack the same generative capabilities as a GAN-model. This thesis proposes using autoencoders to pre-train parts of the GAN-model. Doing so will hopefully speed up the training process and produce more realistic results.

1.1 Local Field Potential

The local field potential (LFP) is the lower frequency part of an extracellular potential, i.e., below around 500 Hz. The LFP is measured by inserting a probe into the extracellular space in the brain tissue [3]. The LFP is thought to contain information from the immediate area in the extracellular space and varies significantly based on location [3]. This is in contrast to electroencephalography (EEG), which is measured on the scalp [3] and is thought to be created by the combination of many sources after filtering through the various media between the brain tissue and the scalp [3].

The interest in LFPs has increased lately after new methods make it easier both to collect data and process it [6]. As the LFP-signal is created from a combination of many sources, the signal itself is ambiguous and difficult to interpret [6]. Despite this, the stability and ease of recording these signals [6] make them promising in use with prosthetic devices [6]. They are also thought of as good candidates for study into how the brain processes sensory information, motor planning, and higher cognitive processes [6].

Chapter 2

Machine Learning

Machine learning, and especially neural networks, has many applications and is finding its way into nearly all relevant fields. Machine learning is an extensive term that encompasses linear regression, complex neural networks, as well as many other methods. The common theme behind these is that the machine is supposed to find the solution to a problem by learning some parameters that best describe the problem, usually through an iterative process using gradients of a loss function. This process can be a potent tool as a machine can find patterns in large amounts of data. Deep neural networks have become particularly popular in the later years, and this thesis will look mostly at the use of deep neural networks with convolutional layers.

2.1 Convolutional Neural networks

Convolutional Neural Networks (CNN) has become a prevalent method in machine learning. It is based on many of the same principals as a densely connected neural network (DNN) but uses fewer variables. CNNs are frequently used in image analysis, as it can look for features in the input invariant of where in the input a feature is located. This is done by moving a filter, or kernel, over the input. The filter contains weights that are multiplied element-wise with a small portion of the input the same size as the filter. The result of this is then added together and becomes the first element in the output. The process is then repeated until the filter has passed over all of the input. This operation is, in essence, a scalar product between a part of the input and the filter. A 1-dimensional illustration of this shown in Figure 2.1, but 2-dimensional convolutional layers are also possible and frequently used with images. Convolutional layers can be expanded to n-dimensions and convolve over volumes or other higher dimensional data.

A one-dimensional convolution operation can be defined as

$$s_i = \sum_{j=0}^{k} S_{i+j} K_j, \tag{2.1}$$

where s is the output for the convolution, S is the input, and K is the filter (or kernel), and k is the size of the filter. Convolutional layers usually also apply a bias after the convolution operation. All in all, a convolutional layer need $c_1 \times c_2 \times k$ weights and c_2 biases, where c_1 is the number of channels in the input and c_2 is the number of channels in the output. This is because there is one filter going from each channel in the input to every channel in the output. Channels can correspond to many different things here. When used with images, channels often represent the different colours of the image, but it can also represent a temporal or spatial dimension. It is, in essence, just another dimension to the data. Strides, padding and dilation modify how the convolution is done. Padding will add zeros to the beginning and end of the input, which makes the edge elements affect more elements in the output. Stride affects how far the filter is moved across the input and, by doing so, decreases the size of the output and is useful for down-sampling. Dilation can also be used for down-sampling by changing the distance between the elements in the input that are multiplied with the filter.

If a DNN were to transform an input of size n into size m, it would need $n \times m$ weights. This is because every element in the input needs to be multiplied by different weights for each element in the output. It would also include m biases, which are added to the output. I.e. a DNN needs $(n+1) \times m$ variables to connect a layer with n elements to a layer with m elements. Because of this, larger inputs such as images will require many variables. Dense layers also usually do not have different weights for different channels. This means that the same weights are used for all channels. Using a convolutional network has the benefit of taking the relationship between different channels into account while reducing the number of resources needed to train the network. It can also work well when the position of a feature is not fixed between different samples of the input, as mentioned earlier. A common example of why this is helpful is its use in classification models. E.g. the layer can recognize a dog in an image regardless of where in the image the dog is located. The layer will normally recognize lower-level features such as edges, not high-level features such as an object being a dog.

More about convolutional layers can be read in the book by Goodfellow et al. on Deep Learning [7, p. 326] and in the Guide [5] by Dumoulin et al.

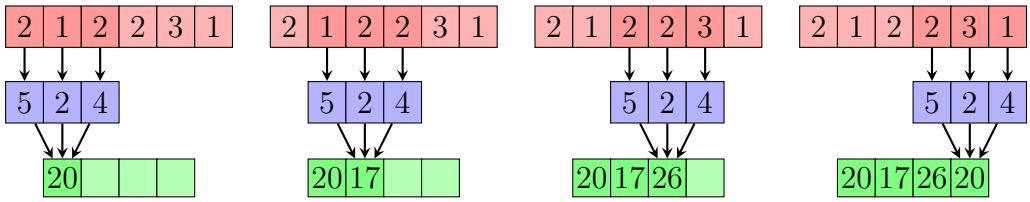

Figure 2.1: Illustration of a 1-dimensional convolutional layer.

2.1.1 Transposed convolutional layers

A transposed convolutional layer can be thought of as a combination of an up-sampling method and a convolution. It is almost exactly the opposite of a convolutional layer. Where a convolutional layer combines several inputted values with the help of the weights in the filter, the transposed convolutional layer expands a single value with the help of the weights in the filter [5]. This is done by multiplying one element in the input separately with all the weights in the filter and adding this number to different elements in the output. A one-dimensional transposed convolutional operation can be defined as

$$s_i = \sum_{j=\max(0,i-m+k)}^{\min(i+1,k)} S_{i-j} K_j, \tag{2.2}$$

where n is the size of the input, k is the size of the filter, and $m = n+k-1$ is the size out the output. An illustration of this operation can be seen in Figure 2.2. Stride, padding and dilution are reversed from normal convolutions. E.g. strides are done over the output, not the input. When comparing Figure 2.1 and 2.2, it is easier to see why it can be thought of as the opposite of a convolutional layer. The layer is sometimes incorrectly referred to as an inverse convolutional layer, but this is not correct as it is not a proper inverse. It is impossible to create an inverse convolutional layer because there is no way to create a general inverse scalar product.

Transposed convolution may be beneficial to traditional up-sampling, such as linear up-sampling because it combines two operations into one. It also has the benefit of reverting the shape of a convolutional layer when using the same filter size. This makes it easy to use in autoencoders, which are discussed more in section 2.6. A more detailed explanation of transposed convolutional layers can be found in the guide [5] by Dumoulin et al.

2.2 Batch normalization

One of the many problems faced when working with deep neural networks is internal covariate shift [11]. As the parameters of a given layer change, the dis-

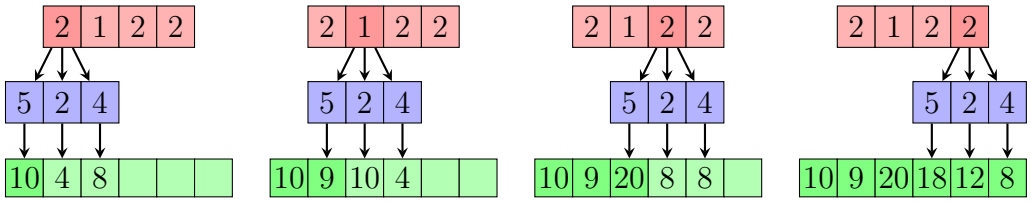

Figure 2.2: Illustration of a transposed convolutional layer.

tribution of the output from that layer change as well. This change in distribution makes it more challenging to train the next layer [11]. Using batch normalization (BN) reduces this problem by forcing the output from the layer into a certain distribution. This, in turn, allows for larger learning rates without the gradient exploding or vanishing [11]. The choice of initial weights is also not as critical when using BN [7, p. 314]. It also acts as a regularization layer which helps cut down on over-fitting [11].

A BN transformation includes two trainable parameters, γ and β. γ is a scaling factor, and β is a bias parameter that shifts the input. These parameters make BN different from a normal normalization and ensure that the BN transformation can represent an identity transform. Doing this ensures that the transformation does not change what the layer can represent [11], e.g. to make sure that transform does not disrupt a non-linearity.

The batch normalizing transform from an input x_i for a mini-batch of size n can be defined as

$$\mathrm{BN}_i(x_i) = \gamma \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta, \tag{2.3}$$

where $\mu_B = \frac{1}{n}\sum_{i=0}^{n} x_i$ is the mean of the batch, $\sigma_B^2 = \frac{1}{n}\sum_{i=0}^{n}(x_i - \mu_B)^2$ is the variance of the batch, and ϵ is a small constant added to avoid dividing by zero if the variance were to become zero. Note that the transformation is done separately for each channel in the input. This means that there are separate γ- and β-parameters for each channel [11].

Because batch normalization includes β as a bias parameter, there is no need to include bias parameters in the layer before the BN as this serves the same function.

2.3 Non-linearity and Activation functions

Neural networks include a non-linearity between the layers in the form of activation functions so that the model can represent more than a linear transformation. Without these functions, the model will never be able to represent anything other

than a linear transform no matter how many layers are added to the model. This makes the activation function a crucial part of any deep learning network.

One such non-linear activation function is the Rectified linear unit (ReLU), which can be defined as

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise.} \end{cases} \tag{2.4}$$

This activation breaks the linearity and lets the model predict non-linear functions. Allowing for negative values to survive through the activation has been shown to improve performance [17]. The Leaky-ReLU activation function allows negatives values through by multiplying them with a scaling factor. The name comes from the function "leaking" through a small part of the negative value. The function can be defined as

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ cx & \text{otherwise,} \end{cases} \tag{2.5}$$

where c is a scaling constant. c is normally kept in the open interval between 0 and 1, as $c = 0$ would be the same as a ReLU, and $c = 1$ would correspond to an identity transform and would no longer break the linearity.

2.4 Dropout

Regularization methods are important for many deep learning applications to avoid over-fitting the model [7, p. 224]. Dropout is one such regularization method that is easy to implement and does not require much computational power [7, p. 255]. It can be thought of as an approximation of Bootstrap aggregating (Bagging), in which multiple models are trained on the same data set. With the ever-increasing complexity of deep neural networks, bagging becomes impractical. Dropout emulates different models by randomly dropping different parameters in the network so that each element in the batch drops different parameters. This way, each element in the batch trains a different subset of the model. The probability of a specific parameter being dropped is independent of the other parameters, and also from the input.

2.5 Back-Propagation

The most crucial part of any trainable model is the method used for training. Back-propagation is the method used to determine how the model parameters

should be updated in neural networks. It works by calculating the derivative of all the trainable parameters with respect to some loss-function. The gradients are calculated by first doing a forward pass through the model. The gradients are then calculated by beginning at the back and differentiating the loss-function, and the using the chain rule to calculate the gradients for the parameters in previous layers.

A consequence of using back-propagation is that all operations used in the model have to be differentiable. This dependence on gradients makes deep neural networks vulnerable to both vanishing and exploding gradients.

After calculating the gradients, the model parameters are updated with an optimizer. The optimizer could be as easy as an Euler-method, but in practice, more complex optimizers are used. A popular choice is Adam, which is widely seen as a robust optimizer that is easy to implement and is computationally inexpensive [15][7, p. 305]. It can be described as a stochastic gradient descent methods that that takes ideas from both the AdaGrad and RMSProp methods by maintaining a separate adaptable learning rate for each parameter it should update [15], and calculating an average of previous gradients to use as weights for adapting these learning rates using estimations of the first and second-order moments of the gradients [15]. There are three initialization parameters of interest: the step size α [15], and the exponential decay rates for the first and second-order moment estimations β_1 and β_2 [15].

2.6 Autoencoder

An autoencoder is an example of an unsupervised machine learning method. Because it is unsupervised, it does not need any ground truth data to train on. This is because it trains on the input itself. The goal of the autoencoder is to create two separate models that, when working together, will reconstruct the original input. This is done in two parts. Firstly, an encoder reduces the dimensionality of the input. This could be done with several methods, as long as the method used is trainable in some way. After the input has been made smaller, it is then sent through a decoder. The decider then, in turn, expands the input into its original dimensionality. It is then compared to the original input, and the model itself should be trained by using a loss function that brings the output closer to the input. A basic diagram of an autoencoder can be seen in Figure 2.3.

By forcing the input into a lower-dimensional space, the idea is that the model will learn to extract only the features that are important to reconstructing the input, which is similar to methods such as Principal Component Analysis (PCA) [7, p. 45,145]. It is then possible to use the decoder and encoder independently

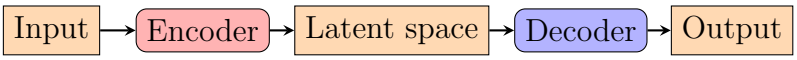

Figure 2.3: A diagram of an autoencoder.

for different purposes, e.g. as pre-trained parameters for other networks. This could make it possible to train a supervised model, even with a limited amount of ground truth data to train on.

More details on autoencoders can be found in the Deep Learning book[7, p. 499] by Goodfellow et al.

2.6.1 Loss function

Mean squared error is a loss function that calculates a distance between some predicted values and some observed values. This is an ideal metric for an autoencoder because it is easy to calculate. It is defined as

$$f(x) = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2 \tag{2.6}$$

where Y is the samples coming from the real data set, $\hat{Y}$ is the generated samples, and n is the number of samples. This number should go towards zero while optimizing the model.

2.7 Generative Adversarial Networks (GAN)

A Generative Adversarial Network (GAN) is another unsupervised machine learning method. It is similar to an autoencoder but has several key differences. The goal of the method is to train two separate networks to compete with each other. The two different parts are commonly called the generator and the discriminator. The generator generates samples given some random input, and the discriminator should then attempt to differentiate between generated samples and samples coming from the real data set [7, p. 696]. A basic diagram of this setup can be seen in Figure 2.4. This adversarial model draws inspiration from game theory, where two actors become better when they have to have to compete against each other [7, p. 696]. When used correctly, GANs can then create a natural-looking results [13][9].

The discriminator outputs a number comparable to a probability of how real it thinks the inputted sample is. It should be trained by maximizing this probability for the real input and minimizing it for the generated input. The generator should

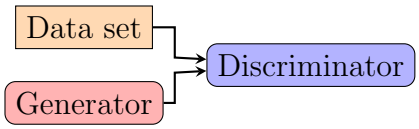

Figure 2.4: A simple diagram of a GAN-model.

then be trained on trying to fool the discriminator, i.e. move towards the weights that give the highest probability of the generated signal being real when passed through the discriminator.

One of the downsides of a GAN-model is that it is difficult to train properly. The discriminator experiences mode collapse notoriously easily. Mode collapse is the state when the discriminator only learn how to recognize a small subset of the features that make up a real input [9]. This hinders the generator from learning to recreate all the relevant features and gives an unrealistic result. A good way to counteract this is to pick stable hyperparameters.

2.7.1 Loss function

Hartmann et al. proposed a loss function using the Wasserstein distance in their 2018 paper [9]. It attempts to avoid the problem of vanishing gradients and build upon the method from Arjovsky et al. in their 2017 paper [1]. Using the Jensen-Shannon divergence, as proposed in the original paper on GANs [8], might lead to vanishing gradients when the model is trained to optimality [9][1]. Using the Wasserstein distance as a loss function reduces this problem as the quality of the gradients are proportional to the performance of the discriminator[1]. The goal of the GAN-model becomes minimizing the Wasserstein distance between the distribution of real data, $\mathbb{P}_r$, and the distribution of the generated data, $\mathbb{P}_\theta$. This can be defined as

$$\tilde{W}(\mathbb{P}_r, \mathbb{P}_\theta) = E_{x_r \sim \mathbb{P}_R}\left[D(x_r)\right] - E_{x_f \sim \mathbb{P}_\theta}\left[D(x_f)\right], \tag{2.7}$$

where $D(x)$ is K-Lipschitz continuous, x_f is drawn from $\mathbb{P}_\theta$, and x_r is drawn from $\mathbb{P}_r$. The Lipschitz continuity limits how fast a function can change [1]. To enforce this continuity, Hartmann et al. introduces a one-sided penalty,

$$P_1(\mathbb{P}_{\hat{x}}) = \lambda \cdot E_{\hat{x} \sim \mathbb{P}_{\hat{x}}}\left[\max\left(0, ||\nabla_{\hat{x}} D(\hat{x})||_2 - 1\right)^2\right], \tag{2.8}$$

where λ is a scaling factor for the penalty term, and the distribution $\mathbb{P}_{\hat{x}}$ contains all points on the straight line between the generated and the real samples, and $\hat{x}$ is drawn from this distribution [9]. This penalty is based on a stricter two-sided

penalty proposed by Arjovsky et al. [1]. After combining this, the loss function for the discriminator then becomes

$$L_c = -\tilde{W}(\mathbb{P}_r, \mathbb{P}_\theta) + \max\left(0, \tilde{W}(\mathbb{P}_r, \mathbb{P}_\theta)\right) \cdot P_1(\mathbb{P}_{\hat{x}}), \tag{2.9}$$

which the discriminator should minimiz e[9]. This has been shown to keep the gradients stable, even when the distance between the distributions is decreasing [1]. The generator should maximize $E_{x_f \sim \mathbb{P}_\theta}[D(x_f)]$ [9]. In other words, the discriminator should maximize the distance between the distributions, while the generator should maximize the output from the discriminator.

2.8 Model evaluation metrics

2.8.1 Inception score

A commonly used evaluation metric is the Inception score [2]. It seems highly correlated with human evaluation [2], but it does not detect mode collapse and it is sensitive to noise [9] and has several other problems [2]. One such problem is that it does not compare the generated samples to the samples from the real data set [2]. It is, nevertheless, a common metric used for generative methods [2].

The inception score is calculated by passing the samples through a pre-trained classifier network before calculation the entropy of the conditional label distribution, $p(y|x)$ [16]. This distribution is the output from the classifier network and says something about the probability of a certain class matching with the input. Images containing meaningful data should have a low entropy conditional label distribution [16]. The marginal distribution, $\int p(y|x = G(z))\mathrm{d}z$ [16], is a combination of a batch of conditional label distributions, and should have a high entropy. Salimans et al. combined these two entropies into one by defining the Inception score as

$$\mathrm{IS}(x) = e^{E_x[\mathrm{KL}(p(y|x)||p(y))]}, \tag{2.10}$$

where KL is the Kullback–Leibler divergence.

2.8.2 Fréchet inception distance

Heusel et al. proposed using the Fréchet inception distance (FID) [10], which has a few advantages to the Inception score. Namely, that FID is reactive to

mode collapse [9]. It does not, however, give any clues to whether the model is over-fitted or not[9][10]. Much like the inception score, it is calculated by passing all generated data and all the samples from the data set through a pre-trained classifier network. The FID is then Fréchet distance between the two sets after passing through the classifier. The Inception-v3 is used to extract features relevant for images [10]. The Fréchet distance[4, eq. 4] is defined as

$$d^2(\mathbb{P}_r, \mathbb{P}_\theta) = |\mu_r - \mu_x| + \mathrm{Tr}\left[\Sigma_r + \Sigma_x - 2\sqrt{\Sigma_r \Sigma_x}\right], \tag{2.11}$$

where μ_r and Σ_r is the mean and the covariance matrix of the real data set, and μ_x and Σ_x is the same for the generated samples. Tr denotes the trace [7, p. 44] of a matrix. This distance is also known as the Wasserstein-2 distance[10].

Part II

Method

Chapter 3

Experimental data

The data used in this thesis was collected from mice using a Neuropixel Probe while exposing the mice to different visual stimuli by the Allen Institute [12]. All the experimental data used is publicly available from the Allen Institute.[1] The measurements themselves consist of several channels, which corresponds to the physical location on the probe. This thesis looks exclusively at the measurements from the primary visual cortex of a single mouse.

As seen in Table 3.1, there are 118 natural images shown to the specimen during the experiment. The specimen is also shown other images and movies, but these are not used in this thesis. Each image is shown 50 times for 313 frames (0.25 seconds). This means that there are a total of 5900 images showings. The recording itself lasted for around 2 hours and 20 minutes, with a frequency of 1250 Hz. This makes a total of 12 081 284 data points per channel, which makes it possible to sample 12 080 971 different samples with length 313 from the experiment. Batches are drawn randomly from this data set when training the autoencoder.

80 % of the images are randomly picked to be training samples, while the other 20 % become test samples. This means that the GANs will be trained on 4700 samples, while the test set contains 1200 samples.

All samples are normalized by subtracting the mean and dividing by the standard deviation of each sample.

[1] https://allensdk.readthedocs.io/en/latest/visual_coding_neuropixels.html

Table 3.1: Information about the experiment conducted by the Allen Institute. The Session ID and Probe ID can be used to find the specific samples used in this thesis through their API.

Specimen name	Sst-IRES-Cre;Ai32-387858
Specimen sex	Male
Specimen age	122 days
Session ID	719161530
Probe ID	729445652
LFP sampling rate	1250 Hz
Number of images	118
Experiment date	8. January 2019
Experiment time	08:25:16 UTC

Chapter 4

Code Implementation

All code used in this thesis can be found on GitHub[1]. The code is made using PyTorch and should be used with CUDA, although running on CPU is also supported. All packages needed are listed in the Github repository.

4.1 Overview

To facilitate pre-training, the GAN will use the decoder from the autoencoder in the generator, and the encoder in the discriminator. This will hopefully make the GAN easier to train, as parts of the model should already have been trained to extract the important features of the input. Two different autoencoders will be used. One of them will train on the LFP-data, and the other will train on the images. The GAN will only use the encoder in the autoencoder for the images, as images generation is not a goal here.

All code is written in Python using Pytorch, which takes care of calculating gradients and backpropagating them through the network automatically. Adam will be used as an optimizer with $\alpha = 0.001$, $\beta_1 = 0$ and $\beta_2 = 0.999$. These values are the same as used by Hartmann et al. in their paper [9].

4.2 Signal autoencoder

The encoder consists of 9 layers of convolutional layers with varying kernel sizes and strides. The signal input is 313 samples long, with 22 different channels. As seen in Table 4.1, the encoder then reduces the signal down to an output of 115 samples with 10 channels. This is about 17 % of its original size.

[1] https://github.com/maraspr/LFPGAN

Table 4.1: Architecture for autoencoder for signals.

Signal encoder	Channels	Filter	Stride	Padding	Activation	Output shape
Layer 1 - Conv	22 to 20	10	1	3	ReLU/BN	310
Layer 2 - Conv	20 to 18	40	1	3	ReLU/BN	277
Layer 3 - Conv	18 to 16	25	1	4	ReLU/BN	261
Layer 4 - Conv	16 to 15	70	1	0	ReLU/BN	192
Layer 5 - Conv	15 to 14	24	1	0	ReLU/BN	169
Layer 6 - Conv	14 to 13	5	1	0	ReLU/BN	165
Layer 7 - Conv	13 to 12	13	1	0	ReLU/BN	153
Layer 8 - Conv	12 to 11	20	1	0	ReLU/BN	134
Layer 9 - Conv	11 to 10	20	1	0	ReLU/BN	115
Signal decoder	**Channels**	**Filter**	**Stride**	**Padding**	**Activation**	**Output shape**
Layer 1 - Transposed Conv	10 to 11	20	1	0	ReLU/BN	134
Layer 2 - Transposed Conv	11 to 12	20	1	0	ReLU/BN	153
Layer 3 - Transposed Conv	12 to 13	13	1	0	ReLU/BN	165
Layer 4 - Transposed Conv	13 to 14	5	1	0	ReLU/BN	169
Layer 5 - Transposed Conv	14 to 15	24	1	0	ReLU/BN	192
Layer 6 - Transposed Conv	15 to 16	70	1	0	ReLU/BN	261
Layer 7 - Transposed Conv	16 to 18	25	1	4	ReLU/BN	277
Layer 8 - Transposed Conv	18 to 20	40	1	3	ReLU/BN	310
Layer 9 - Transposed Conv	20 to 22	10	1	3	Linear/Normalization	313

Because the decoder should reconstruct the signal that the encoder has reduced down into a latent space, the decoder mirrors the encoder almost exactly. The only difference is the output channel, as the decoder does not output a channel with the standard deviation of the other channels. The decoder also uses transposed convolutional layers in place of the convolutional layers, and all the kernel sizes and channels are reversed. It outputs 22 channels of 313 samples, which are normalized by subtracting the mean of the sample and dividing by the standard deviation of the sample. The model is then back-propagated using the mean square error loss function.

The model is trained for 7 epochs with samples independent of images from the experiment. There are about 12 million samples in the data set, so this equates to 84 million showings.

4.3 Image autoencoder

The autoencoder for the images is quite similar to the autoencoder for the signals. The main difference is that images contain two-dimensional data, so the convolutional layers also need to be two-dimensional. The images are 1174 pixels wide and 918 pixels high, and they are black-and-white, so there is only one channel. The output shape from the encoder is 3×23, which is about 0.02 % of the original image size. After flattening, the output of the encoder is 2 channels with $3 \times 23 = 115$ samples. This makes it easy to concatenate this output to the

Table 4.2: Architecture of autoencoder for images.

Image encoder	Channels	Filter	Stride	Padding	Activation	Output shape
Layer 1 - Conv	1 to 4	30×30	2	30	ReLU/BN	475×603
Layer 2 - Conv	4 to 2	20×20	2	0	ReLU/BN	228×292
Layer 3 - Conv	2 to 4	30×30	1	0	ReLU/BN	119×263
Layer 4 - Conv	4 to 5	70×70	1	0	ReLU/BN	130×194
Layer 5 - Conv	5 to 3	16×30	1	0	ReLU/BN	115×165
Layer 6 - Conv	3 to 2	30×30	1	0	ReLU/BN	86×136
Layer 7 - Conv	2 to 3	15×15	1	0	ReLU/BN	72×122
Layer 8 - Conv	3 to 6	5×5	2	0	ReLU/BN	34×59
Layer 9 - Conv	6 to 5	20×20	1	0	ReLU/BN	15×40
Layer 10 - Conv	5 to 4	10×10	1	0	ReLU/BN	6×31
Layer 11 - Conv	4 to 2	2×9	1	0	ReLU/BN	5×23
Image decoder	Channels	Filter	Stride	Padding	Activation	Output shape
Layer 1 - Transposed Conv	2 to 4	2×9	1	0	ReLU/BN	6×31
Layer 2 - Transposed Conv	4 to 5	10×10	1	0	ReLU/BN	15×40
Layer 3 - Transposed Conv	5 to 6	20×20	1	0	ReLU/BN	34×59
Layer 4 - Transposed Conv	6 to 3	5×5	2	0	ReLU/BN	72×122
Layer 5 - Transposed Conv	3 to 2	15×15	1	0	ReLU/BN	86×136
Layer 6 - Transposed Conv	2 to 3	30×30	1	0	ReLU/BN	115×165
Layer 7 - Transposed Conv	3 to 5	16×30	1	0	ReLU/BN	130×194
Layer 8 - Transposed Conv	5 to 4	70×70	1	0	ReLU/BN	119×263
Layer 9 - Transposed Conv	4 to 2	30×30	1	0	ReLU/BN	228×292
Layer 10 - Transposed Conv	2 to 4	20×20	2	0	ReLU/BN	475×603
Layer 11 - Transposed Conv	4 to 1	30×30	2	30	Sigmoid/BN/Upsample	918×1174

output of the signal encoder. The decoder then reconstructs the image to the best of its ability. As with the signal autoencoder, the network is then trained on the mean square error between the real image and the image reconstructed by the network. The architecture of this network can be found in Table 4.2.

The model was then trained for 10 000 epochs. There are only 118 images in the data set, so this equates to 1180000 showings.

4.4 GAN

The generator consists of two main parts: an image encoder and a signal decoder. An image is fed into the image encoder, and the output from this is fed into the signal decoder. In addition to this, there is a network injecting a random input into the latent space between the image decoder and the signal decoder. This is to increase the entropy in the generated samples so that the same image as input can produce different samples as output. The architecture used can be seen on the left side of Figure 4.1. The architecture of the random input-network can be seen in Table 4.3.

The discriminator consists of a signal encoder and an image encoder which is

Table 4.3: Architecture of network for random input in the generator.

Randnet	Channels	Filter	Stride	Padding	Activation	Output shape
Layer 1 - Conv	3 to 4	10	1	3	ReLU/BN	310
Layer 2 - Conv	4 to 5	30	1	3	ReLU/BN	277
Layer 3 - Conv	5 to 4	25	1	4	ReLU/BN	261
Layer 4 - Conv	4 to 3	70	1	0	ReLU/BN	192
Layer 5 - Conv	3 to 5	24	1	0	ReLU/BN	169
Layer 6 - Conv	5 to 6	5	1	0	ReLU/BN	165
Layer 7 - Conv	6 to 5	13	1	0	ReLU/BN	153
Layer 8 - Conv	5 to 7	20	1	0	ReLU/BN	134
Layer 9 - Conv	7 to 10	20	1	0	ReLU/BN	115

run in parallel. The output from each of these are concatenated and fed into 8 convolutional layers (see Table 4.4), and a single densely connected layer that brings the output down to one sample in one channel. Finally, the discriminator returns the mean of these values across the batches. The discriminator represents the Lipschitz $D(x)$-function described in subsection 2.7.1, and the loss function described there is implemented by back-propagating three times. $D(x)$ represents the expected value from the distribution. This value should be maximized for samples from the real data set, and minimized for samples coming from the generator. By back-propagating the output with respect to a tensor pointed in the negative direction, the gradients will point towards where the output will be maximized instead of minimized. This is useful when training the generator, as the generator is trained to maximize the output from the discriminator. Similarly, with the discriminator, the output from the generator should be minimized, while the output from the real data set should be maximized.

To enforce the Lipschitz continuity, the discriminator is back-propagated once again, but this time with respect to the penalty given in Equation 2.8, which in this case equates to $\lambda(r + g)^2$ where r is the output from the discriminator given a real input, and g is the output from the discriminator given a generated input.

Due to hardware and time constraints, the model is trained for 200 epochs. There are 4700 samples in the train set, so this equates to 940000 showings. The image encoder adds a significant extra demand on resources. The model is therefore also trained without the image encoder for 4000 epochs. All this is done both with and without pre-trained weights from the autoencoder.

Table 4.4: Architecture of network for combination layers in the discriminator.

Combnet	Channels	Filter	Stride	Padding	Activation	Output shape
Layer 1 - Conv	10 to 10	15	1	0	ReLU/Dropout/BN	101
Layer 2 - Conv	10 to 8	26	1	0	ReLU/Dropout/BN	76
Layer 3 - Conv	8 to 6	30	1	0	ReLU/Dropout/BN	47
Layer 4 - Conv	6 to 5	2	1	0	ReLU/Dropout/BN	46
Layer 5 - Conv	5 to 6	5	1	0	ReLU/Dropout/BN	42
Layer 6 - Conv	6 to 4	4	1	0	ReLU/Dropout/BN	39
Layer 7 - Conv	4 to 2	2	1	0	ReLU/Dropout/BN	38
Layer 8 - Conv	2 to 1	15	1	0	ReLU/Dropout/BN	24
Layer 9 - Linear	N/A	N/A	N/A	N/A	Linear	1

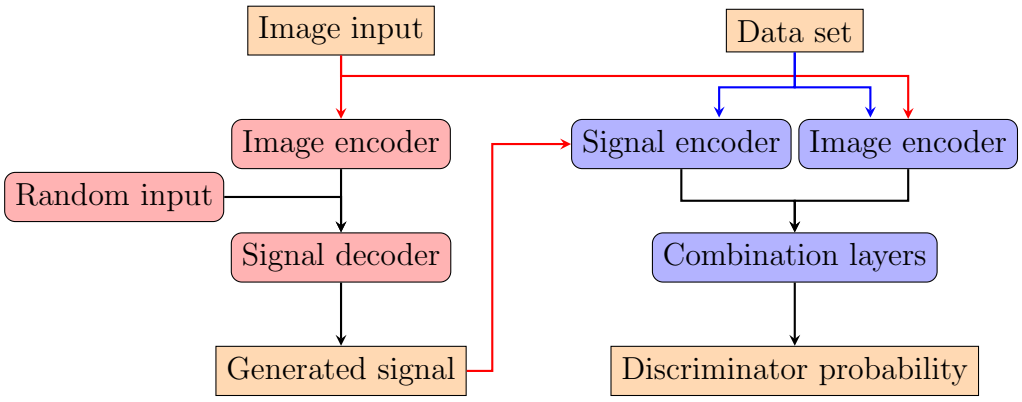

Figure 4.1: The Implementation of the GAN-model using the Signal encoder and decoder from Table 4.1 and the image encoder from Table 4.2.

4.5 Model evaluation metrics

The Fréchet inception distance is calculated by using a Pytorch port[2] of the program from the original paper [10]. This library requires images as inputs, so the signals need to be stored in images. This can be done by using the channel dimension as image height and the signal dimension as image width (or vice versa). The Inception-v3 model is trained on images and should extract vision-related features. This is not optimal when dealing with time series, but it makes the score somewhat comparable to other networks. The Fréchet distance without the inception model is also provided.

The inception score is calculated using a Pytorch library[3] in a similar fashion to the Fréchet inception distance.

[2] https://github.com/mseitzer/pytorch-fid
[3] https://github.com/sbarratt/inception-score-pytorch

Part III

Results

Chapter 5

Results

5.1 Pre-training

The signal Auto-Encoder (see Table 4.1) has been trained on all available LFP-data for seven epochs. This also includes samples where the specimen is seeing a blank screen, or there is something on the screen that is not natural images. The signals are recreated with a high degree of accuracy, as seen in Figure 5.1. The figure shows samples from the real data set on the right, and samples recreated with the autoencoder on the left. There are some small differences, but these seem to be minor.

This phenomenon is also seen in Figure 5.2 where there are some peaks in the mean frequency spectrum of the real samples which are not present in the generated samples.[1]

Figure 5.3 shows that the distribution of the real and generated data are more or less identical. Perhaps more interesting, is how the autoencoder replicates signals from the data set with natural images. Figure 5.4 shows that the distribution from the autoencoder follows the real data very accurately. Note also in Figure 5.6 that the network seems to learn lower frequency features before the higher frequency features of the data set.

Figure 5.7 shows images reconstructed by an autoencoder trained on images (see Table 4.2). The generated images on the right are reconstructions of the images on the left after passing through the image Auto-Encoder. As seen, the network performs relatively well with high contrast images, but become more blurry when

[1]There is some clipping in the frequency distribution plots. This is caused by the lower edge of the standard deviation area being below zero. Since the frequency spectrum is a non-negative value, this does not make sense. As these are logarithmic plots, the lower edge of the distribution is clipped to the lowest amplitude value for that channel for convenience.

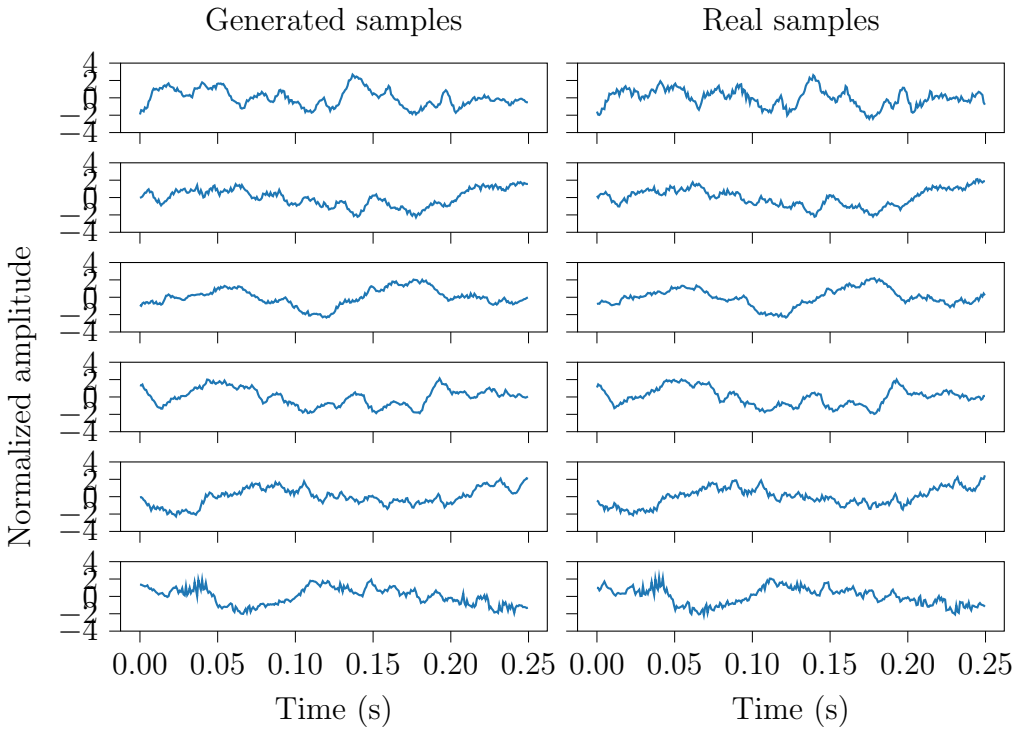

Figure 5.1: Randomly picked samples passed through the Signal Auto-Encoder. The samples on the left are recreations of the samples on the right.

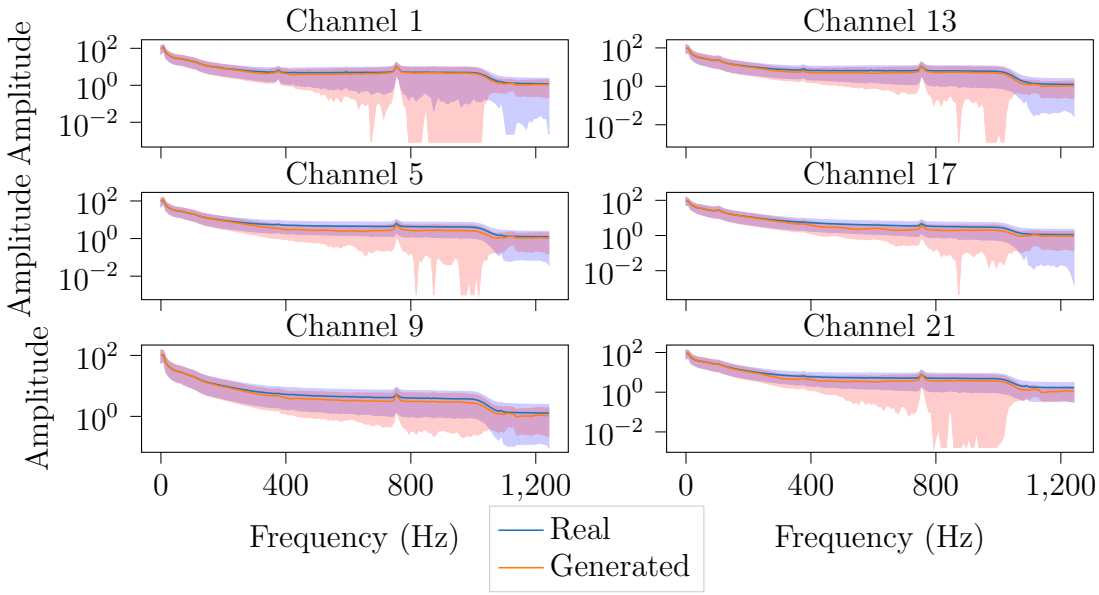

Figure 5.2: Distribution of frequency spectrum from generated signals from an autoencoder trained on all samples from the data set for 7 epochs. The lower limit of the standard deviation is clipped to the lowest amplitude value in the data set.

there are more nuances in the image, although it is still usually possible to make out the images, especially when compared to the real images.

5.2 GAN without pre-training

Figure 5.8 shows a generated distribution that does not represent the real data set except perhaps at the very centre of the signal. After 200 epochs the model has failed to learn any high-frequency features, as seen in Figure 5.9. None of the metrics shown in Table 5.1 show any evidence of the model collapsing.

5.3 GAN with Auto-Encoder

The distribution of the GAN with parts pre-trained with an autoencoder is seen in Figure 5.10. The central parts of the distribution seem to follow the real distribution, but the edges have quite clear artefacts. The frequency distribution in Figure 5.11 corroborate this story. It shows that the frequency spectrum of the generated samples deviates from the distribution of the real samples, especially in the highest frequencies. The FID and Fréchet distance in Table 5.1 is lowest for the pre-trained GAN.

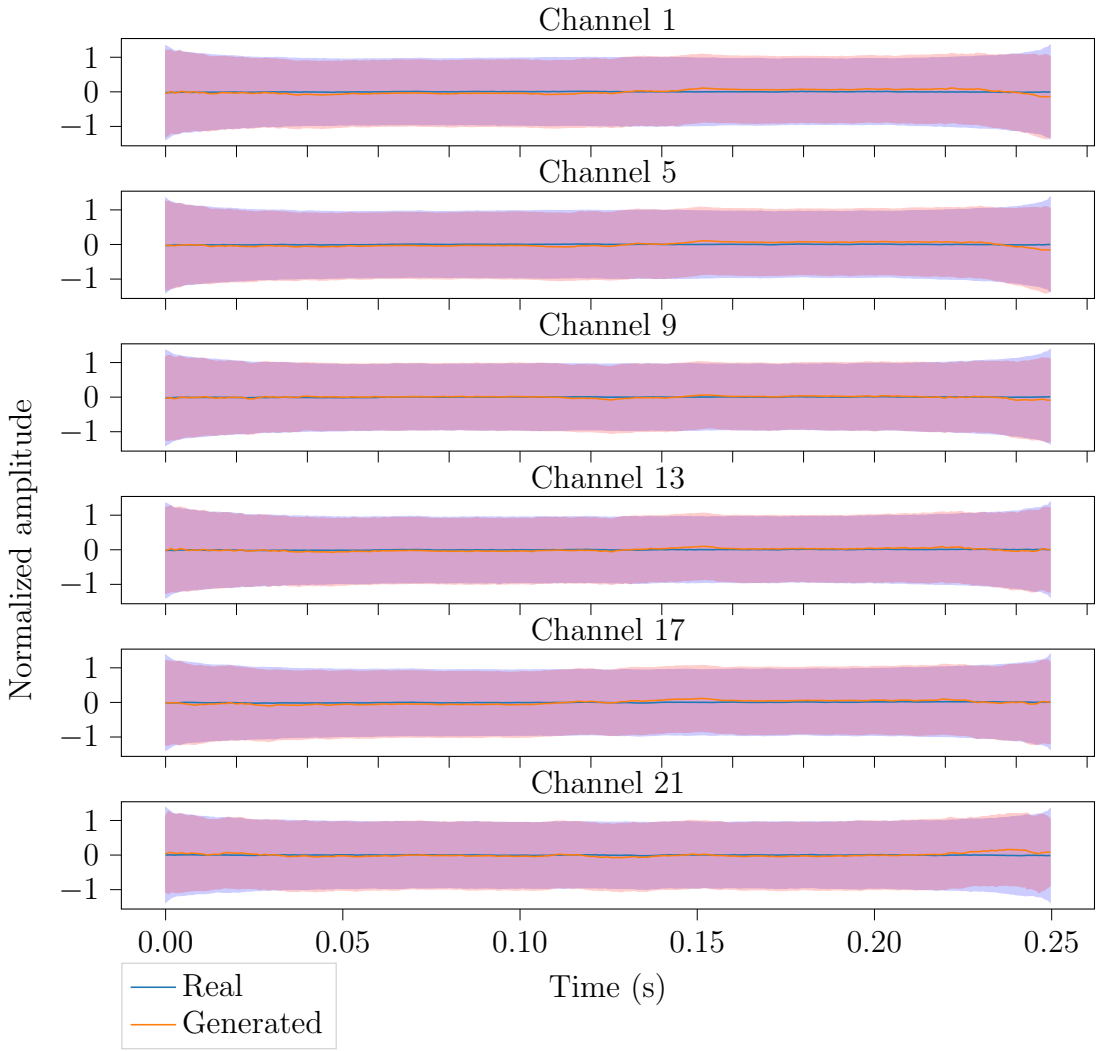

Figure 5.3: Distribution of samples passed through an autoencoder that has trained on all LFP-data for 7 epochs.

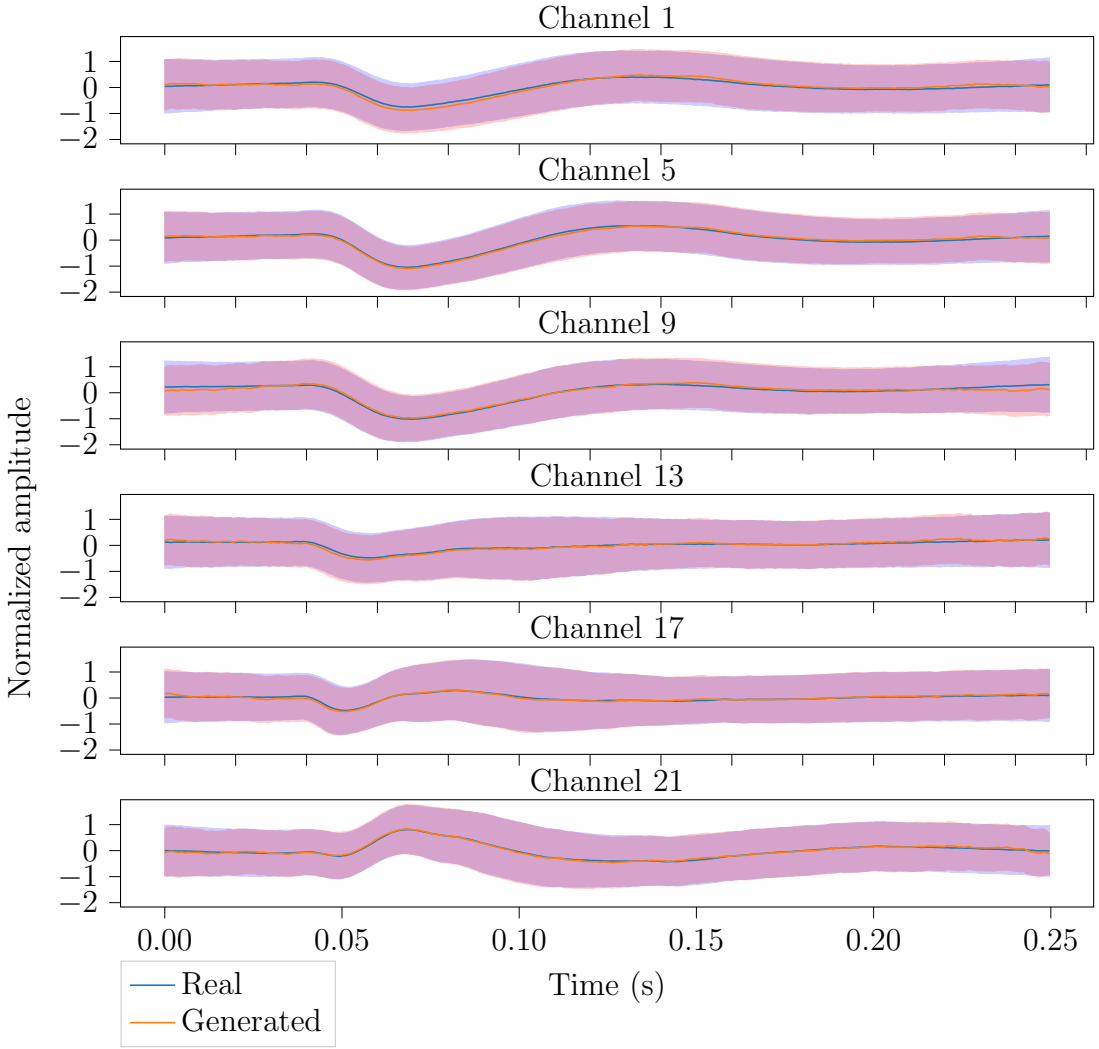

Figure 5.4: Distribution samples from autoencoder replicating LFP-signals corresponding to different natural images.

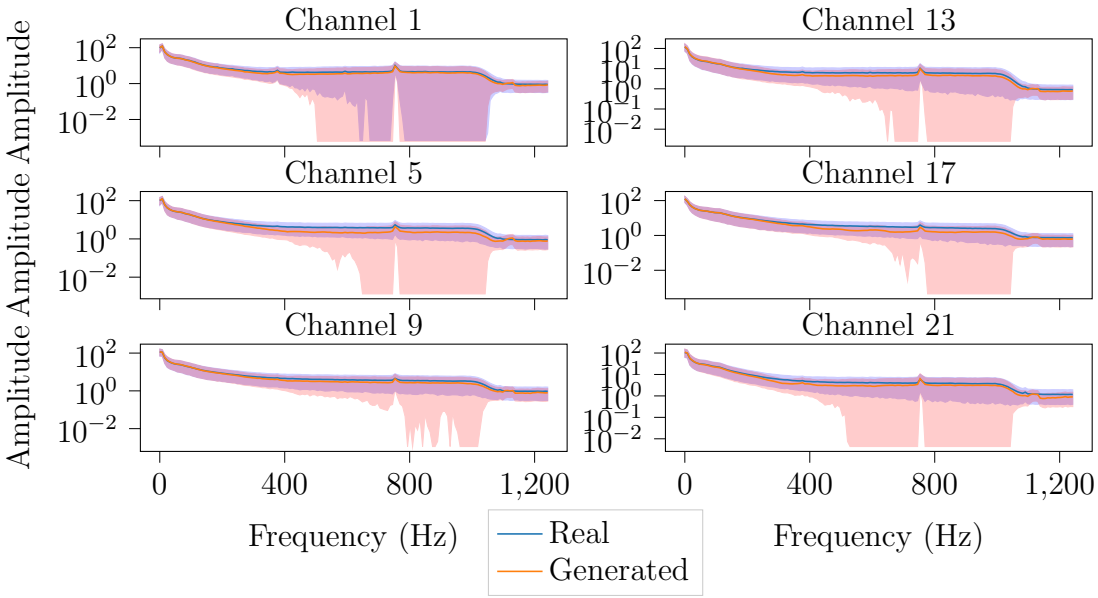

Figure 5.5: Frequency distribution from autoencoder replicating samples from data set with of LFP-signals corresponding to different natural images. The lower limit of the standard deviation is clipped to the lowest amplitude value in the data set.

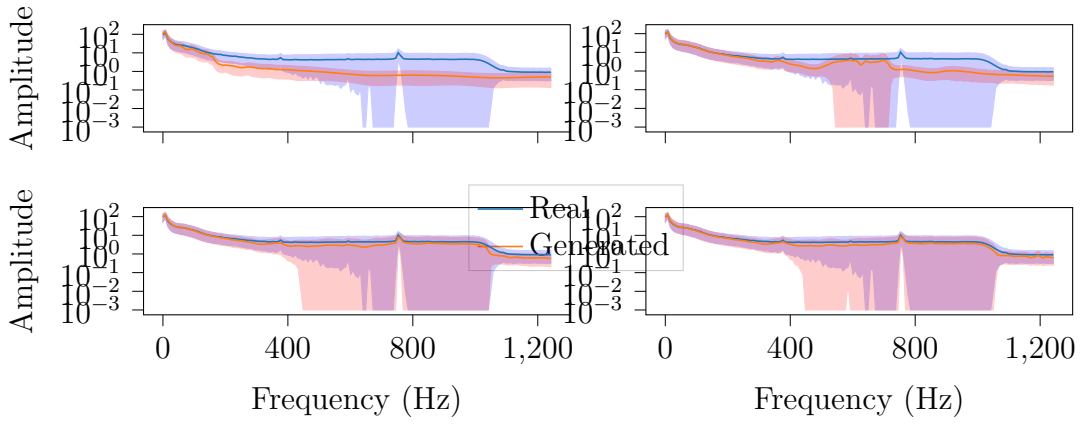

Figure 5.6: Evolution of frequency spectrum during the training of autoencoder. Upper left is trained for 2 % of an epoch, upper right trained for 30 % of an epoch, lower left for 60 & of an epoch, and the lower right has been trained for one epoch. The lower limit of the standard deviation is clipped to the lowest amplitude value in the data set.

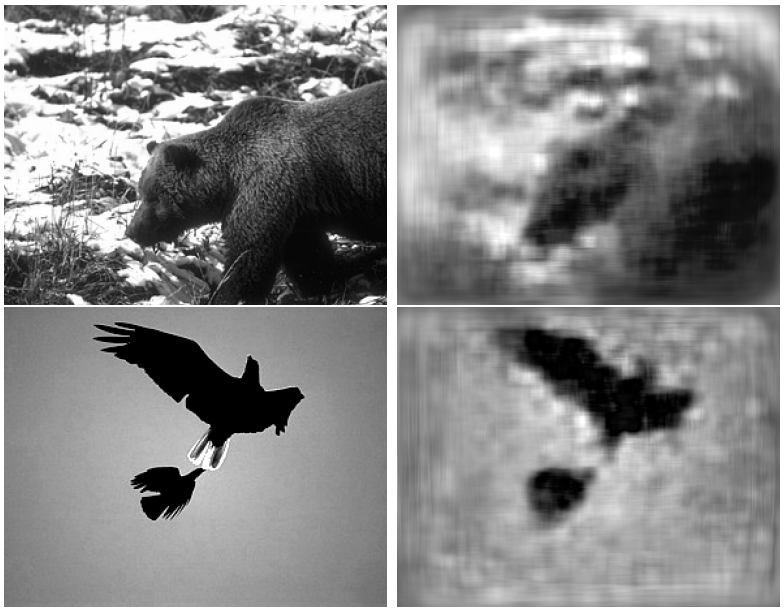

Figure 5.7: Images recreated using image autoencoder

Table 5.1: Metrics for GANs with or without pre-training. Values for the data set and Gaussian noise with same mean and variance as the data are also added.

	Train set			Test set		
Network	IS	FID	Fréchet Distance	IS	FID	Fréchet Distance
GAN	1.98	315.54	5789.57	1.93	320.69	6250.28
Pre-trained GAN	1.17	42.94	2548.33	1.19	37.59	2126.82
Noise	1.12	349.34	9343.42	1.12	351.49	10295.38
Data set	1.28	0	0	1.29	0	0

5.4 Results for different images

Figure 5.12 shows the distribution of samples generated with the pre-trained GAN from a single image from the training set. As with the distribution over the whole data set, it generally follows the mean and variance of the real samples, but it deviates more locally. This is true to higher degree for Figure 5.13.

More figures for individual images can be found in appendix B.

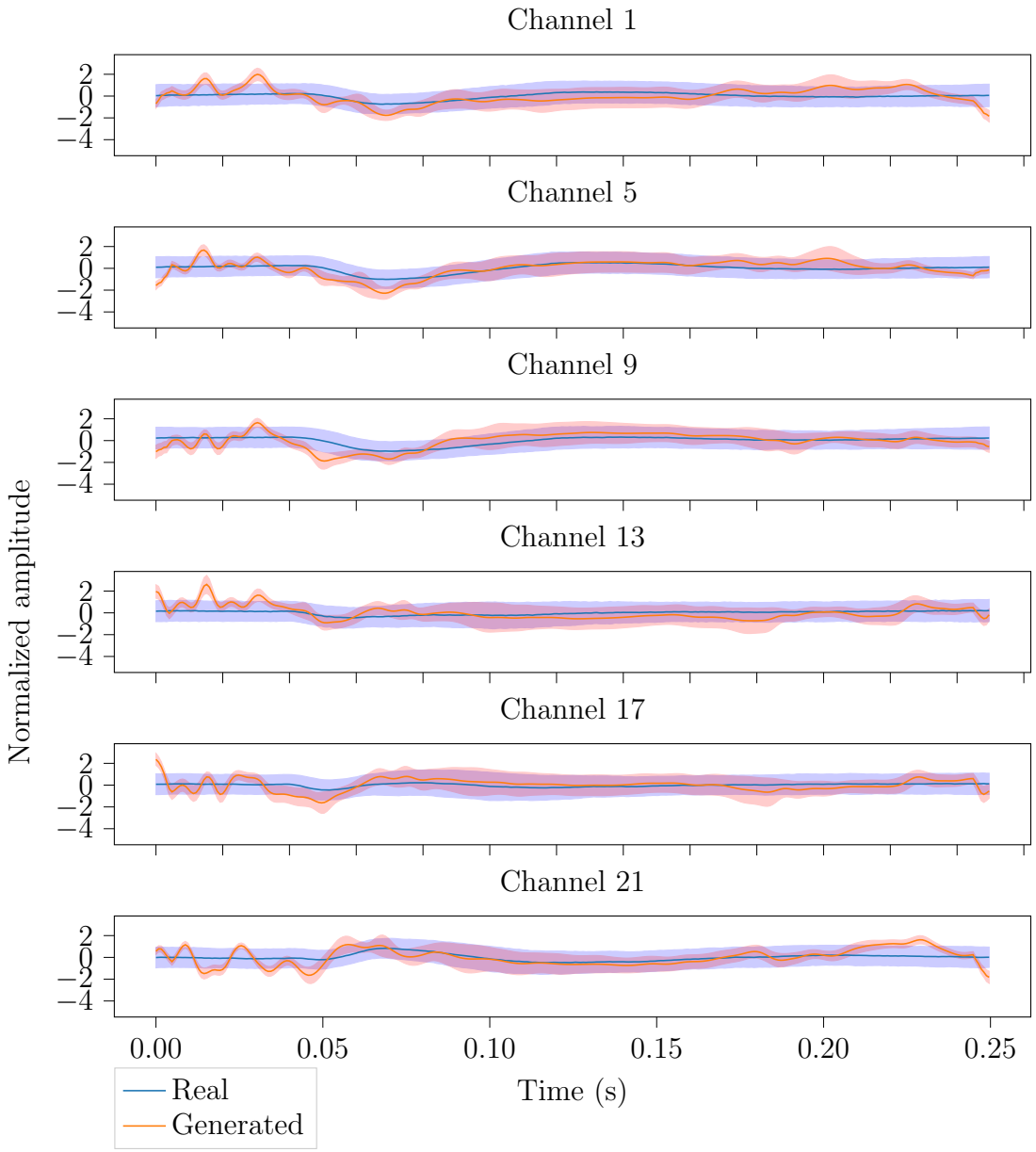

Figure 5.8: Distribution of samples generated by GAN-model with no parts pre-trained with autoencoder.

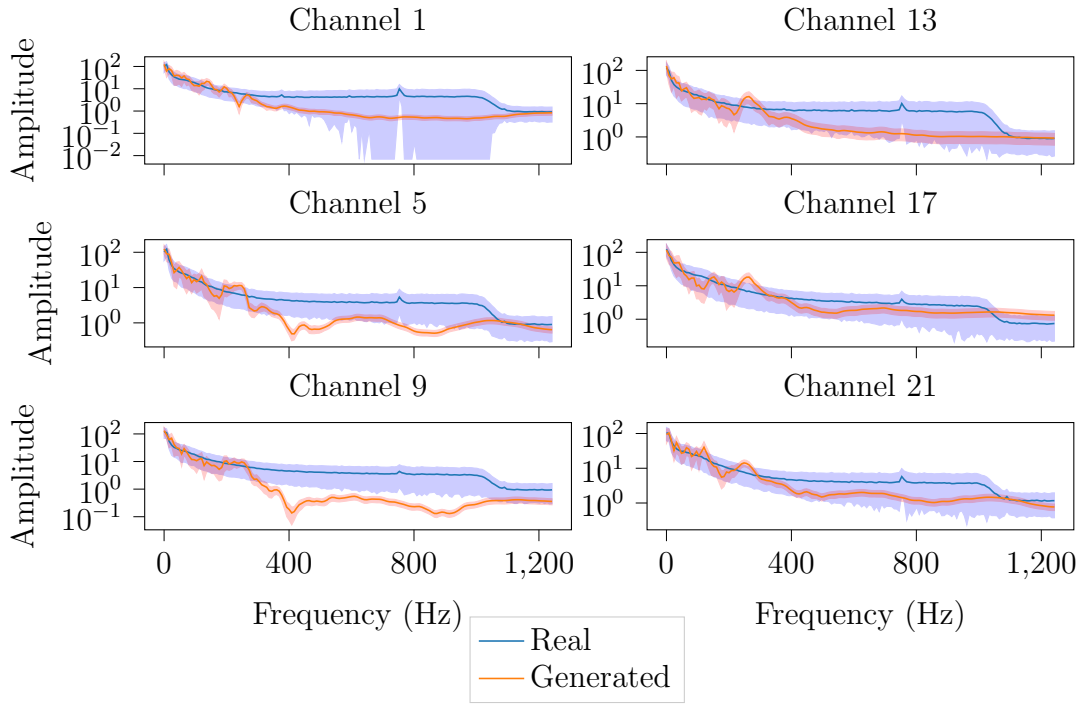

Figure 5.9: Distribution of frequency spectrum of samples generated by GAN-model with no parts pre-trained with autoencoder. The lower limit of the standard deviation is clipped to the lowest amplitude value in the data set

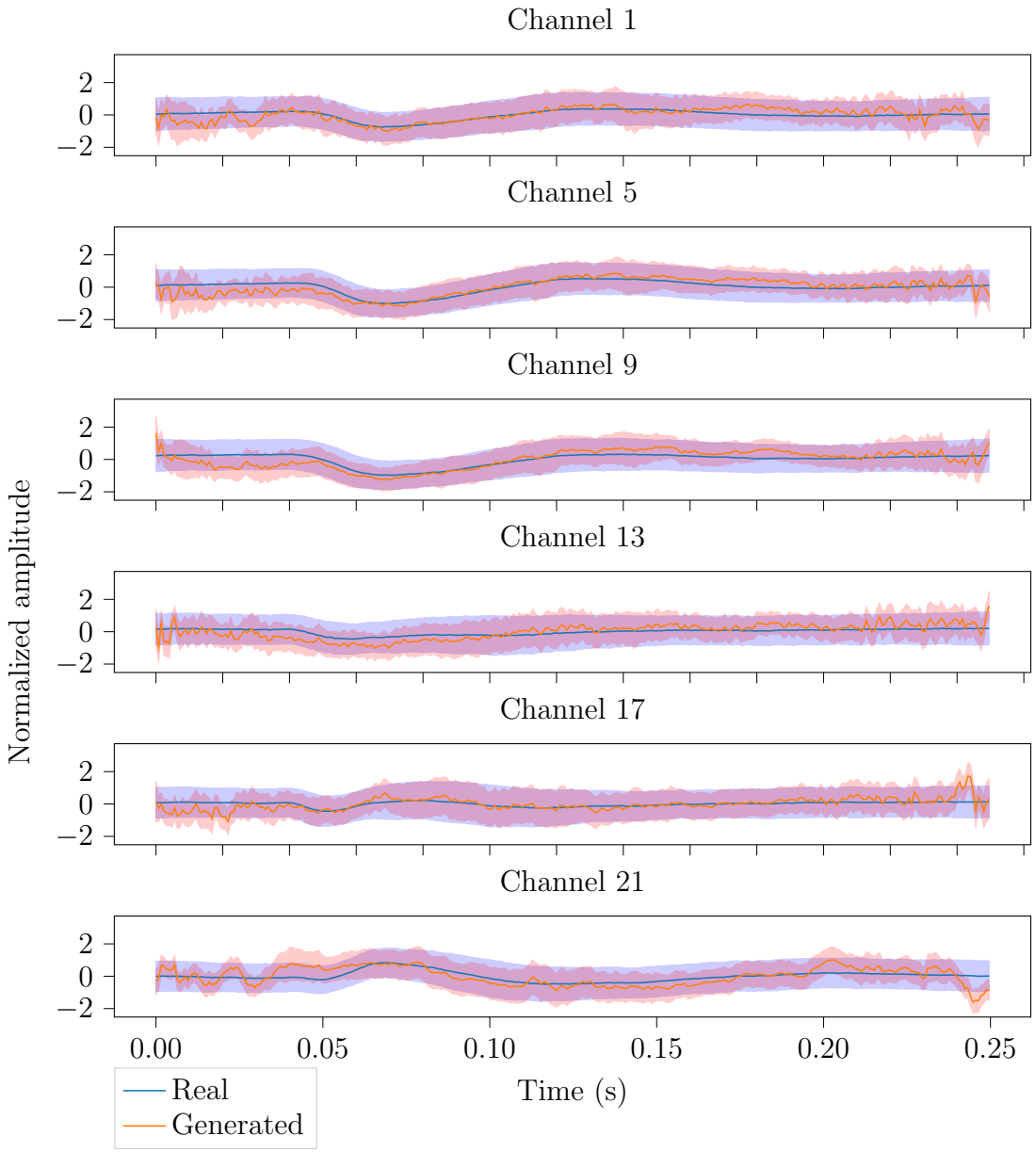

Figure 5.10: Distribution of samples from GAN-model pre-trained with an autoencoder.

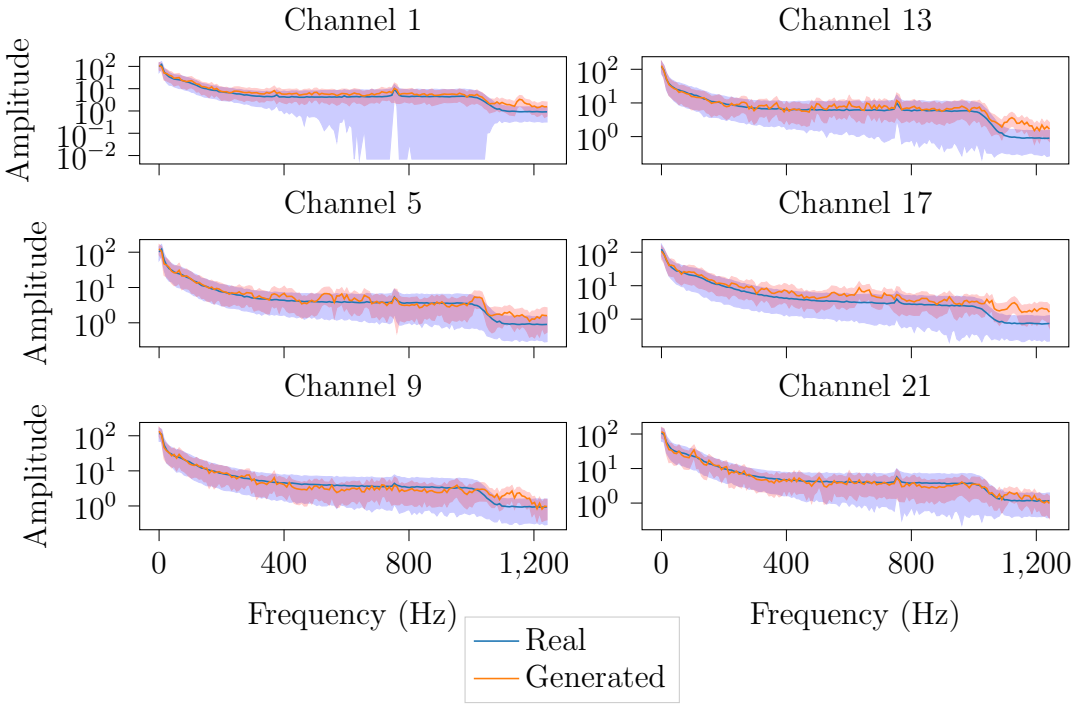

Figure 5.11: Distribution of frequency spectrum of samples from GAN-model pre-trained with an autoencoder. The lower limit of the standard deviation is clipped to the lowest amplitude value in the data set.

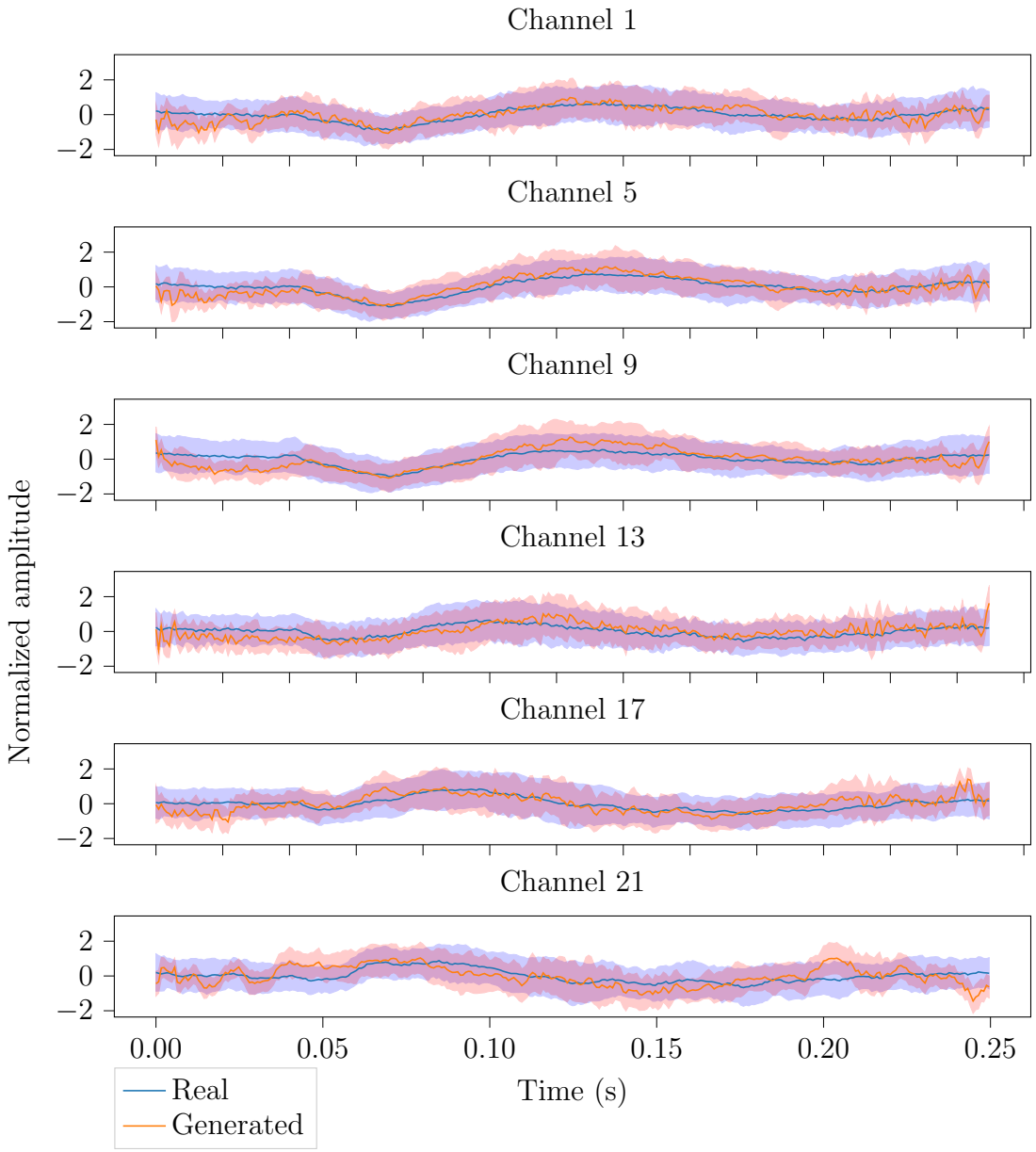

Figure 5.12: Distribution of samples from pre-trained GAN-model for single image from training data set.

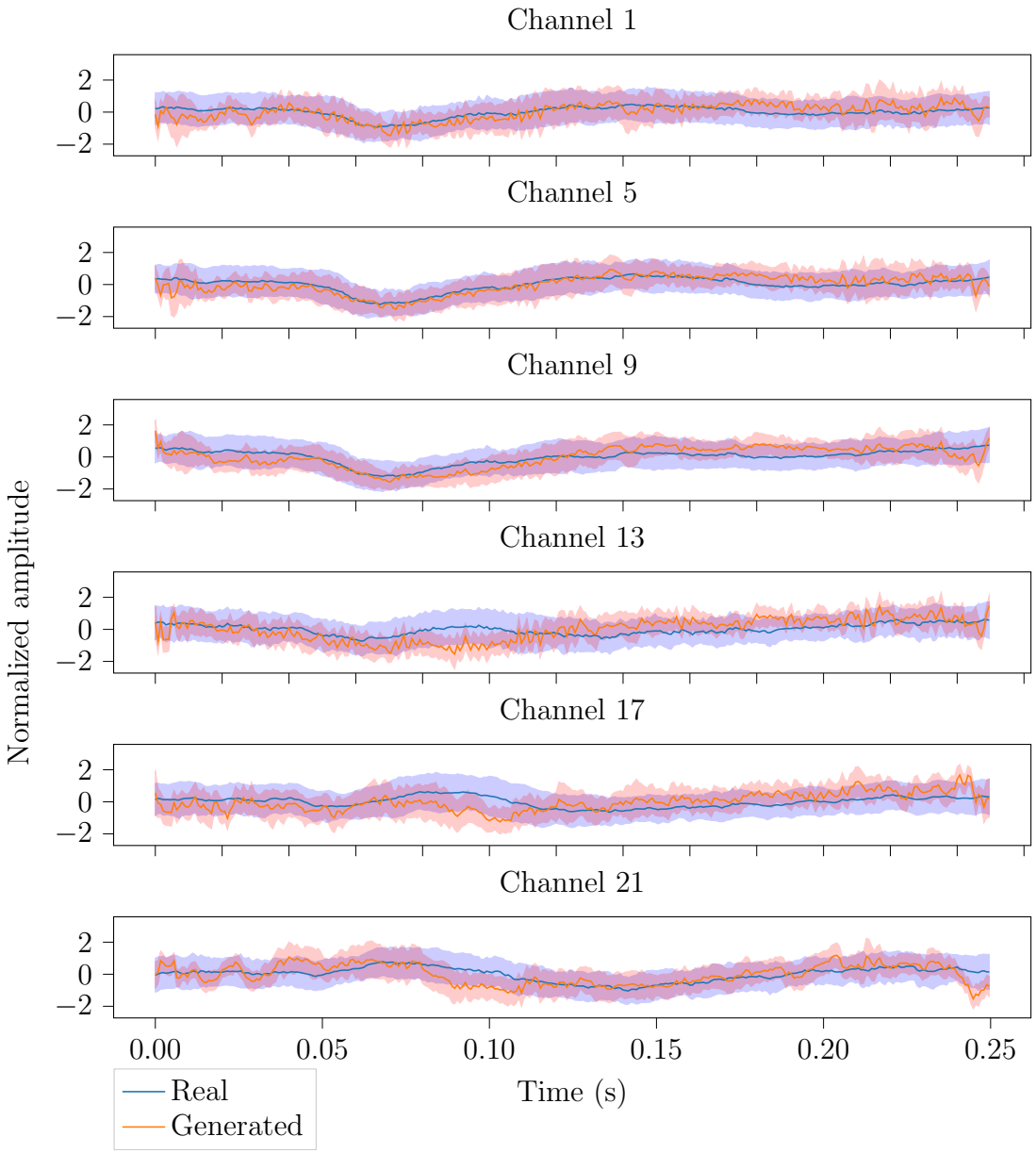

Figure 5.13: Distribution of samples from pre-trained GAN-model for single image from test data set.

Part IV

Discussion, Conclusion and Future Work

Chapter 6

Discussion

Figure 5.1 shows that an autoencoder is very good at feature extraction, and manages to replicate the inputted signals almost exactly. This indicates that the signals are compressible, and all of their features can be sensibly compressed into the latent space of the autoencoder. This latent space seems to be able to contain all possible samples from the distribution and indicates that the chosen architecture is capable of producing the correct distribution.

Figure 5.10 shows a model that has learned some general features of the distribution but still deviates from the real distribution. This seems especially true at the edges. This could be an artefact of the transposed convolutional layer, as fewer weights have more impact on each element near the edge. This is not necessarily a problem, as the length of the time-series is arbitrary. I.e. the edges of the generated signals can just be cut to remove these artefacts. More concerning, however, is the deviations near the centre of the signal, although, more training should remove these.

Figure 5.10 and Figure 5.8 seem to indicate that pre-training the model using an autoencoder speeds up the training process, and helps the model produce accurate high-frequency features. This is seen more clearly in Figure 5.9 and Figure 5.11. The model without pre-training did not produce any high-frequency features in the same amount of epochs as the pre-trained model did. It seems that the high-frequency features take the most time to train. This is corroborated by the evolution training the autoencoder, as seen in Figure 5.6. The FID (Table 5.1) also indicate that the pre-trained network is a better approximation of the real data. Comparing the FID with the random noise in the table indicates that the model without pre-training performed little better than random noise.

Ultimately, the number of epochs trained was too low. A better result may have been produced if the model could have been trained for a longer period, but this was not possible due to time and hardware constraints. Using transposed

convolutional layers instead of linear interpolation (such as in [9]) made the model less flexible, and contributed to larger model sizes and longer training times. It is also more difficult to get the model to expand to the desired size. Their use should, therefore, be weighed up against their weaknesses.

Chapter 7

Conclusion

Pre-training parts of a GAN-model seem to stabilize the training process and help the model converge faster. Autoencoders are much more stable than GANs and are excellent candidates for doing such pre-training. GANs show much promise in generating realistic LFP-signals, but require much training and tweaking to perform well.

7.1 Future work

The stability of an autoencoder over a GAN model might make a Variational autoencoder a better candidate for generative models such as this. Progressively growing GANs have been found to improve the stability and quality of GAN networks [14]. Adapting this architecture in a way to work with pre-training with autoencoders might increase the quality and stability further.

It seems that the model is on the right track. Training for a longer period might yield great results. It might be a good idea to attempt to reduce the model more so that it is less computationally expensive to train. The images do not necessarily need to be sent in with their original dimensionality. Downscaling the images would reduce the size of the network and the number of operations needed, hopefully reducing the time used for training.

A classifier model should be trained and made publicly available for time-series LFP-data so that it can act as a standard metric for generative methods in this area.

Acknowledgements

I want to thank my supervisors Alexander Johannes Stasik, Gaute Einevoll and Morten Hjort-Jensen. This thesis would not have been possible without them. I would also like to thank the High-Performance Computing (HPC) group at the University of Oslo IT division for access to computational hardware.

Bibliography

[1] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan. *arXiv preprint arXiv:1701.07875*, 2017.

[2] Shane Barratt and Rishi Sharma. A note on the inception score. *arXiv preprint arXiv:1801.01973*, 2018.

[3] A. Destexhe and C. Bedard. Local field potential. *Scholarpedia*, 8(8):10713, 2013. revision #137113.

[4] DC Dowson and BV Landau. The fréchet distance between multivariate normal distributions. *Journal of multivariate analysis*, 12(3):450–455, 1982.

[5] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

[6] Gaute T Einevoll, Christoph Kayser, Nikos K Logothetis, and Stefano Panzeri. Modelling and analysis of local field potentials for studying the function of cortical circuits. *Nature Reviews Neuroscience*, 14(11):770–785, 2013.

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016. http://www.deeplearningbook.org.

[8] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[9] Kay Gregor Hartmann, Robin Tibor Schirrmeister, and Tonio Ball. Eeg-gan: Generative adversarial networks for electroencephalograhic (eeg) brain signals. *arXiv preprint arXiv:1806.01875*, 2018.

[10] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. In *Advances in neural information processing systems*, pages 6626–6637, 2017.

[11] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[12] James J Jun, Nicholas A Steinmetz, Joshua H Siegle, Daniel J Denman, Marius Bauza, Brian Barbarits, Albert K Lee, Costas A Anastassiou, Alexandru Andrei, Çağatay Aydın, et al. Fully integrated silicon probes for high-density recording of neural activity. *Nature*, 551(7679):232–236, 2017.

[13] T. Karras, S. Laine, and T. Aila. A style-based generator architecture for generative adversarial networks. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4396–4405, June 2019.

[14] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196*, 2017.

[15] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[16] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. In *Advances in neural information processing systems*, pages 2234–2242, 2016.

[17] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.

Appendices

Appendix A

Correlation between channels of different models

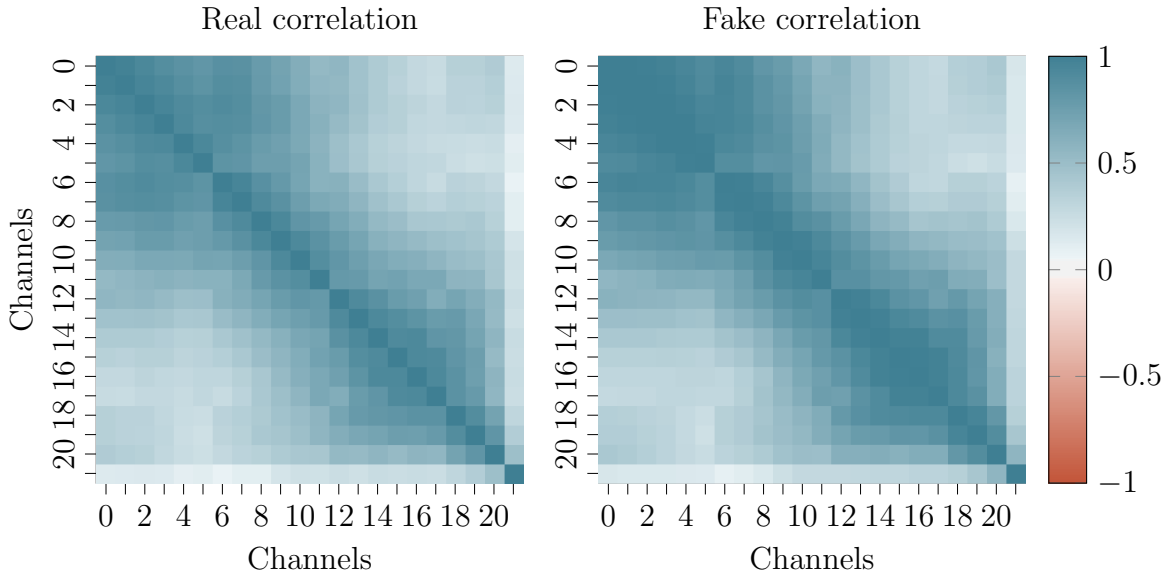

Figure A.1: Correlation between channels for samples passed through autoencoder

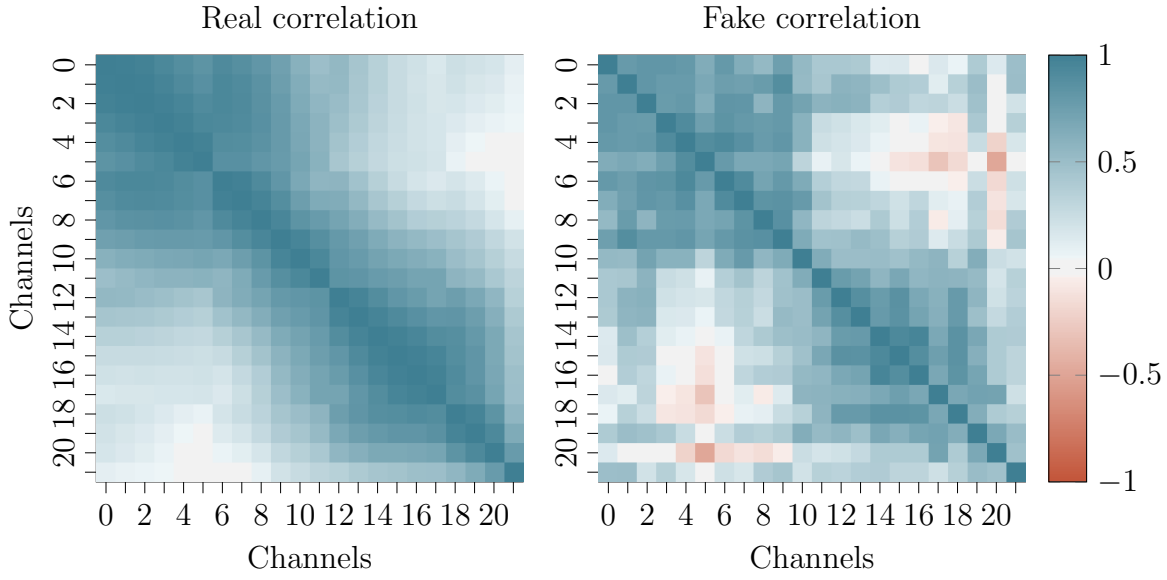

Figure A.2: Correlation between channels for generated samples from GAN with no pre-training.

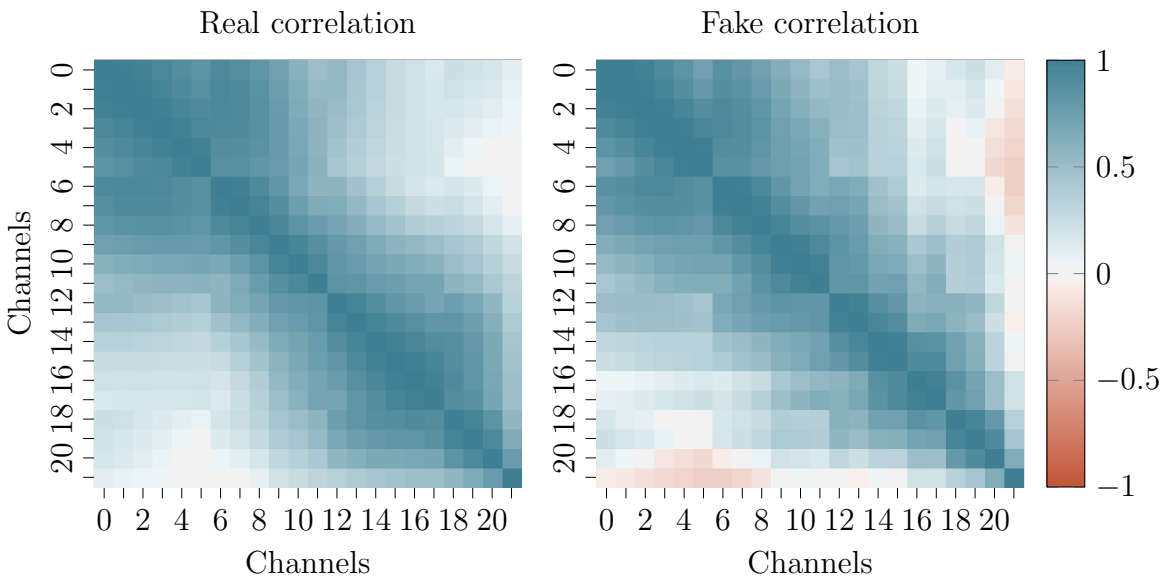

Figure A.3: Correlation between channels for generated samples from pre-trained GAN.

Appendix B

Distributions for different images

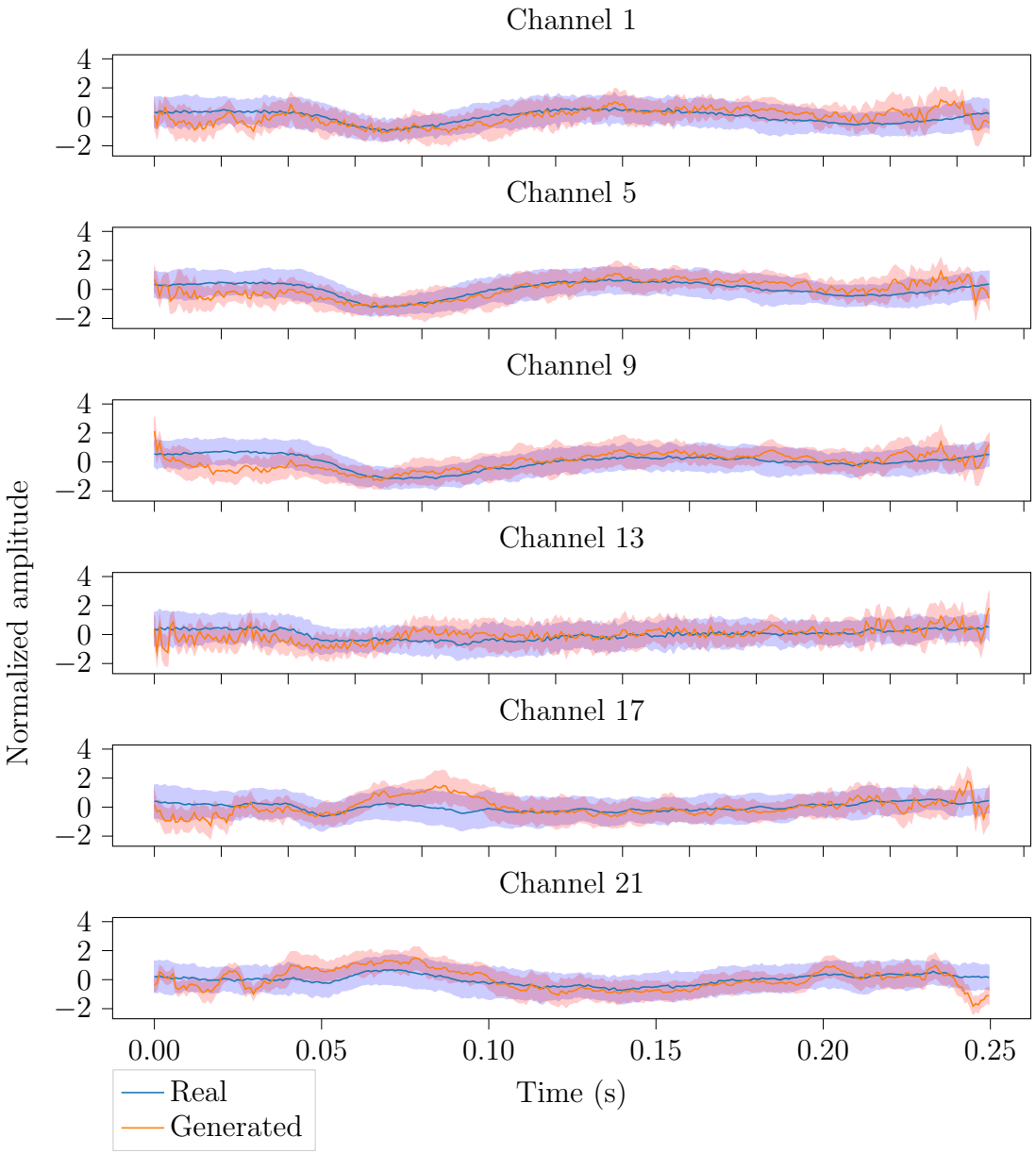

Figure B.1: Distribution of samples from pre-trained GAN-model for image 25 in training set.

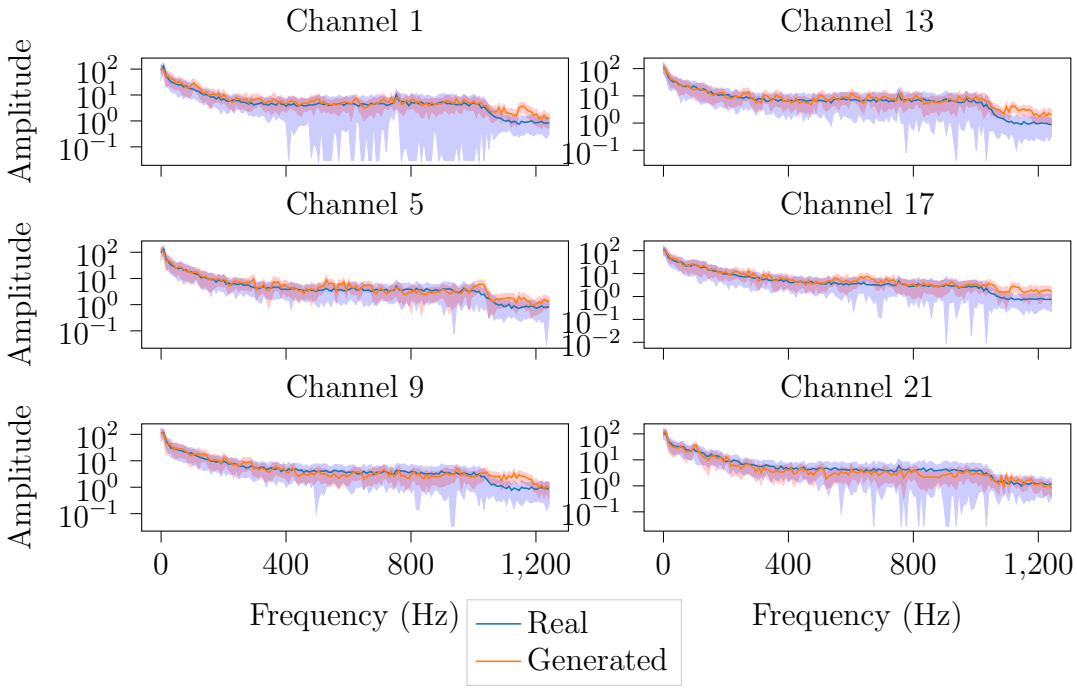

Figure B.2: Distribution of frequency spectrum for samples from pre-trained GAN-model for image 25 in training set.

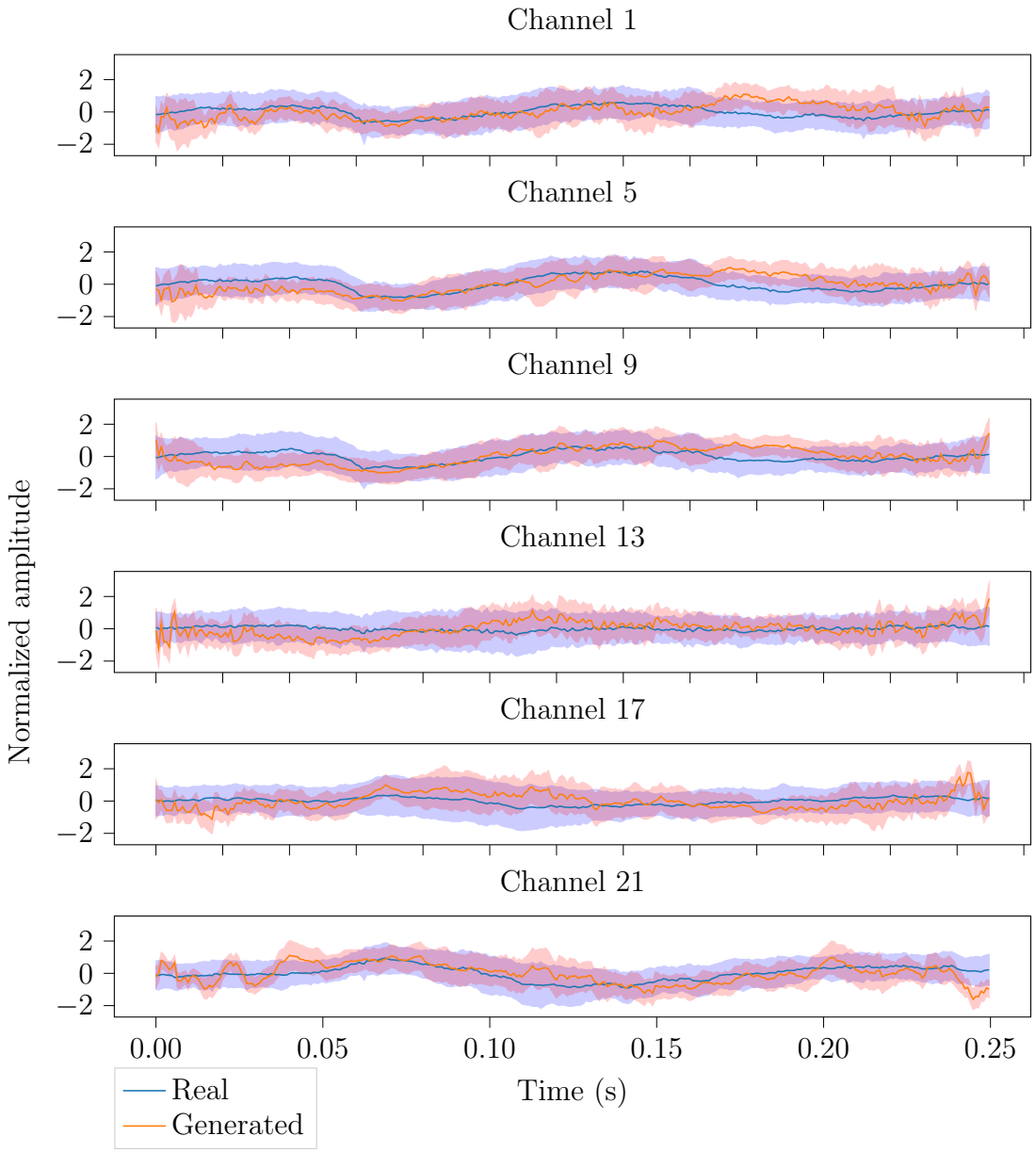

Figure B.3: Distribution of samples from pre-trained GAN-model for image 43 in test set.

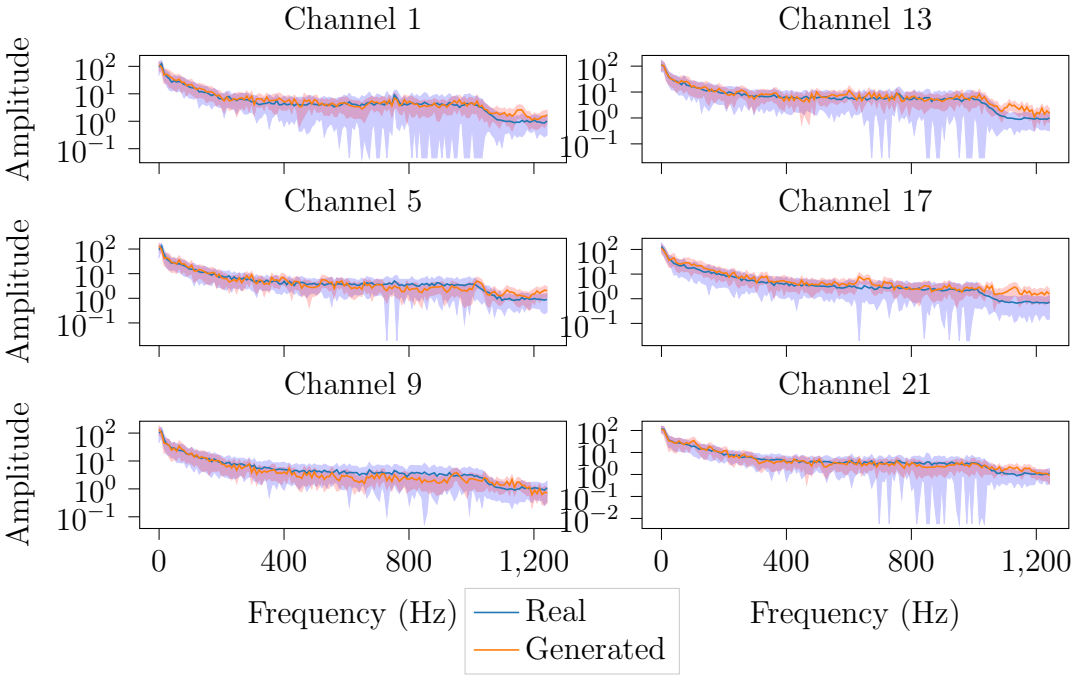

Figure B.4: Distribution of frequency spectrum for samples from pre-trained GAN-model for image 43 in test set.

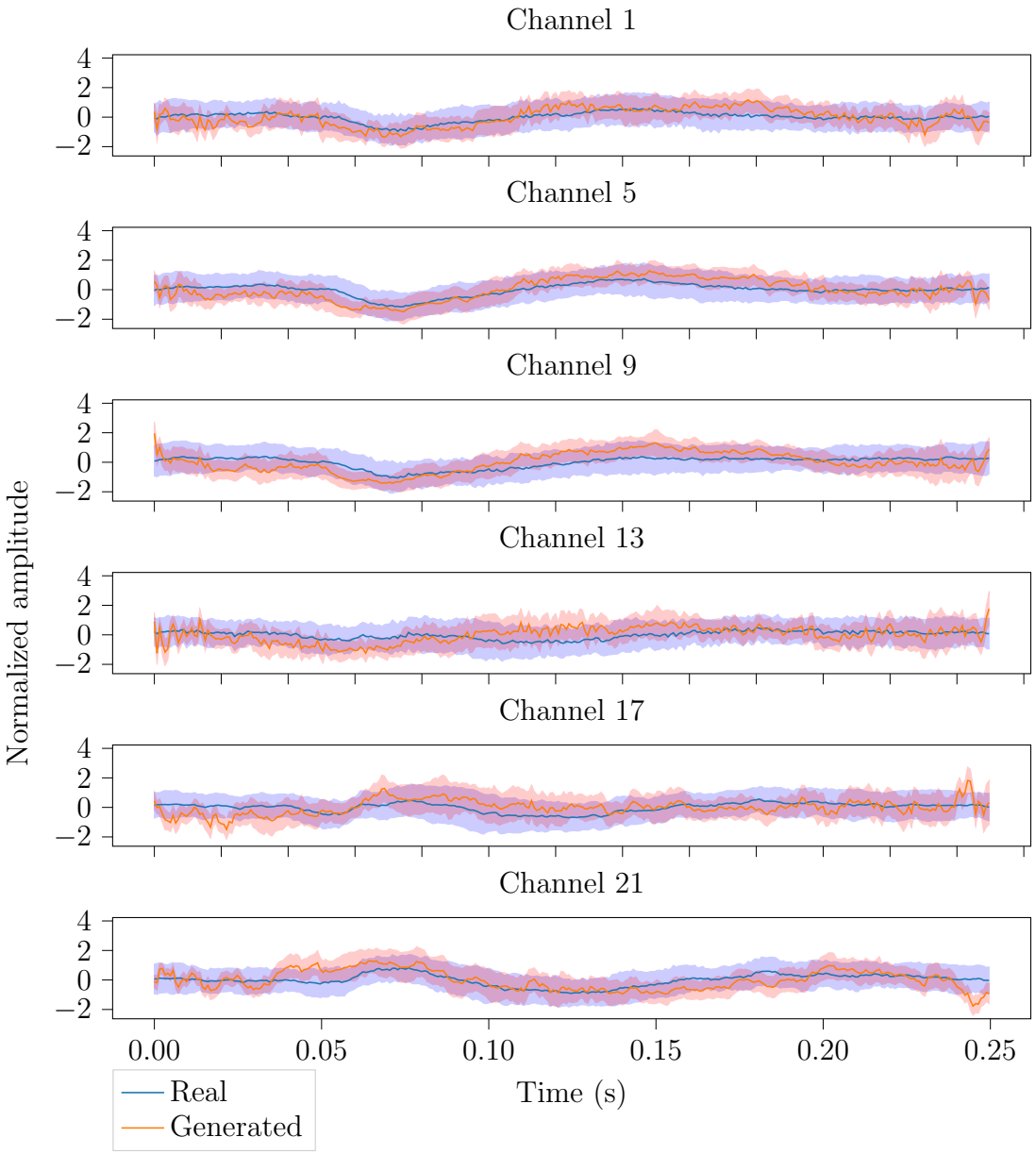

Figure B.5: Distribution of samples from pre-trained GAN-model for image 116 in test set.

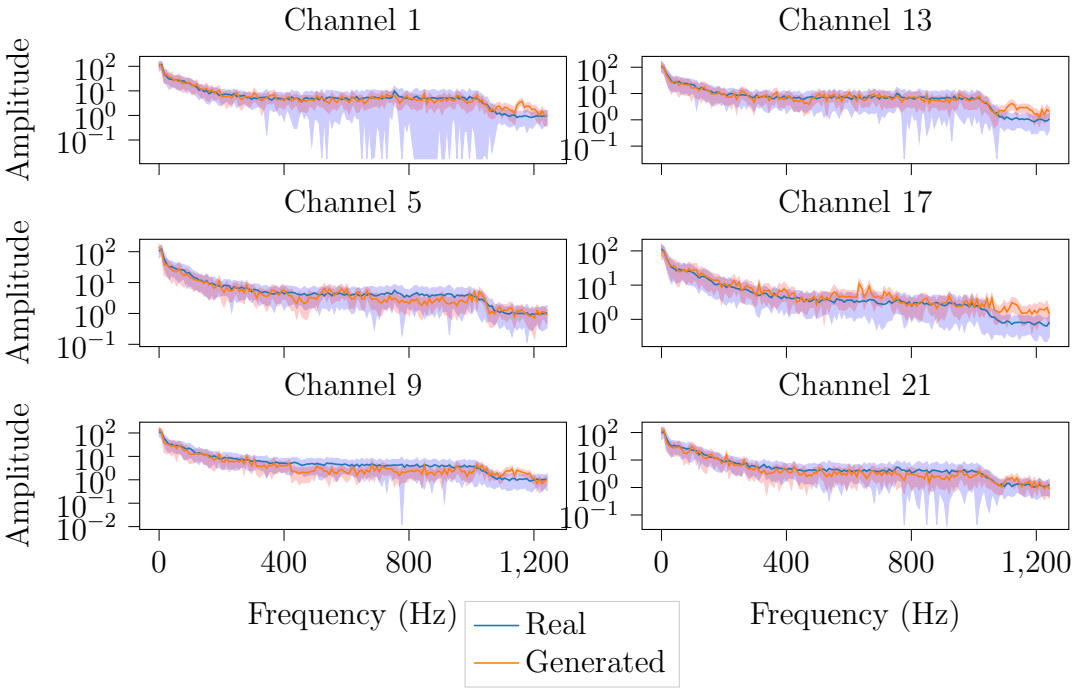

Figure B.6: Distribution of frequency spectrum for samples from pre-trained GAN-model for image 116 in test set.