University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering

Bachelor's thesis

Data Augmentation for Biological Signal Processing

Plzeň 2022 Václav Hrabík

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2021/2022

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Václav HRABÍK**
Osobní číslo:	**A19B0061P**
Studijní program:	**B0613A140015 Informatika a výpočetní technika**
Specializace:	**Informatika**
Téma práce:	**Rozšíření dat pro zpracování biologického signálu**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Seznamte se s koncepty umělých a impulzních neuronových sítí a jejich využitím pro klasifikaci biologických signálů.
2. Seznamte se s datovými kolekcemi, experimenty a klasifikátory používanými neuroinformatickou skupinou KIV pro elektroencefalografická data.
3. Na základě bodů 1 a 2 vyberte vhodnou datovou kolekci a klasifikátory (zahrnující jak klasické, tak impulzní neuronové sítě) pro další experimentování.
4. Navrhněte a implementujte metodu rozšíření vybrané datové kolekce z bodu 3.
5. Použijte vybrané klasifikátory z bodu 3 nad rozšířenou datovou kolekcí z bodu 4.
6. Porovnejte výsledky klasifikace nad původní a rozšířenou datovou kolekcí.

Rozsah bakalářské práce:	**doporuč. 30 s. původního textu**
Rozsah grafických prací:	**dle potřeby**
Forma zpracování bakalářské práce:	**tištěná/elektronická**
Jazyk zpracování:	**Angličtina**

Seznam doporučené literatury:

Dodá vedoucí bakalářské práce.

Vedoucí bakalářské práce: **Doc. Ing. Roman Mouček, Ph.D.**
Katedra informatiky a výpočetní techniky

Datum zadání bakalářské práce: **4. října 2021**
Termín odevzdání bakalářské práce: **5. května 2022**

L.S.

Doc. Ing. Miloš Železný, Ph.D.
děkan

Doc. Ing. Přemysl Brada, MSc., Ph.D.
vedoucí katedry

V Plzni dne 14. října 2021

Declaration

I hereby declare that this bachelor's thesis is completely my own work and that I used only the cited sources.

Plzeň, 5th May 2022

Václav Hrabík

Abstract

There have been a lot of attempts of human (animal) brain simulation. Analogue neural networks were the first major step. These neural networks have various sub-types. All these types work with continuous data but this data is not available every time. Spiking neural networks were developed for work with discrete data. Neural networks in general have big problems with learning. A dataset is necessary for learning. In many cases adding new samples into dataset is not any problem. In neural signals like Electroencephalography (EEG), it is a big problem to get new samples. Because of it, this thesis aims to augment an existing dataset in order to increase the accuracy of automatic recognition of P300 signals. This augmentation is done by adding the synthetic samples. The results show that augmentation is really possible and a functional solution.

Abstrakt

Snaha napodobit lidský (zvířecí) mozek existuje už dlouho. Prvním velkým krokem byly analogové neuronové sítě. Tyto sítě mají spoustu podtypů, které všechny pracují na principu spojitých dat, ale to není vždy úplně možné. Pro práci s diskrétními daty byly vyvinuty impulzivní neuronové sítě, které napodobují mozkové chování ještě lépe a pracují s diskrétními daty. Velký problém u neuronových sítí obecně je schopnost se učit. K tomu je zapotřebí množina dat. Ve spoustě oblastí je získávaní dat jednoduché. V oblasti Elektroencefalografických (EEG) dat je velmi obtížné získat data. Proto se tato práce zabývá umělým zvětšením již naměřené množiny dat za účelem zlepšení úspěšnosti automatického rozpoznávání P300 signálů. Toto zvětšení je provedeno přidáním umělých prvků. Výsledky ukazují, že tato metoda je možným a funkčním řešením.

Contents

1 Introduction **8**

2 Analogue and Spiking Neural Networks **9**

3 Tools for Working with Spiking Networks **11**
3.1 NEURON 11
3.2 Nengo 12
3.2.1 NengoDL 13
3.3 NEST 13
3.4 PyNN 13
3.5 Summary 14

4 Data Augmentation **15**
4.1 Noise Addition 15
4.2 Generative Adversarial Network 16
4.2.1 Generative Algorithm 16
4.2.2 How GANs Work 17
4.3 Sampling 18
4.3.1 Oversampling 19
4.3.2 Undersampling 19
4.4 Sliding Window 19
4.4.1 Without Overlapping 20
4.4.2 With Overlapping 20
4.5 Summary 21

5 Primary P300 dataset **23**
5.1 Chosen dataset 23
5.2 Used Format of Dataset 23

6 Applications of GAN on P300 Dataset **24**
6.1 Created GANs 24
6.2 Training of GAN 25
6.3 Implementation 26
6.4 Results 34
6.5 Summary 38

7 Conclusion **41**

Bibliography **42**

User Guide **45**

Setting up the Environment 45

Work with Project 47

1 Introduction

Analogue neural networks and their younger counterparts, spiking neural networks, are not new in classification tasks and overall in machine learning. The concepts of both analogue and spiking neural networks are based on simulations of human brain behaviour. In order to do it, the neural network needs its parameters. These parameters can be set manually but we are unlikely to set millions of parameters and do not make any mistakes. To set all parameters in the network, proper training is necessary. For training, a specific set of inputs called a dataset is necessary. The dataset is one of the key factors for the better performance of the neural network. Dataset is for neural network like knowledge for human brain. Like for us quality and quantity of knowledge are important. It is the same for neural networks.

At the University of West Bohemia there are already two successful experiments from Roman Kalivoda [9] and Václav Honzík [8]. These experiments use electroencephalography (EEG) dataset from the Guess the number experiment, the dataset contains samples of P300 signals. In these experiments P300 components are classified, accuracy of 63% was achieved on the classification task using this dataset. The main aim of this thesis is to improve the accuracy of these experiments by augmenting the dataset which they used.

Artificial neural networks are described in Chapter 2 and the most used frameworks for working with them in Chapter 3. The augmenting methods are introduced in Chapter 4; the used dataset in Chapter 5. Finally, experiments and their results are described in Chapter 6.

2 Analogue and Spiking Neural Networks

Artificial neural networks are a subset of machine learning and deep learning algorithms. The name of a neural network is like the structure taken from a human brain. They are trying to mimic human brain's behavior. As a human brain they are composed from nodes called "neurons". These neurons are combined into layers. Between layers there are links called "synapses". There are three main types of layers. First is the input layer. The input layer reads the input data. The second one is the hidden layer. There can be many but also none hidden layers. Number of hidden layers can differ from an experiment to another experiment. If in neural network are two or more hidden layers, then it is called the "deep" neural network. The last one is the output layer. The common structure of Artificial neural network is shown in Figure 2.1. The output layer provides the vector as an output. For example, in the recognition tasks this vector contains the numbers whose sum gives one. The highest value in that vector is the wanted outcome. [13]

Analogue neural networks (ANNs) are the one big group of neural networks. These neural networks have various sub-types. All these types work with continuous data but this data is not available every time.

Spiking neural networks (SNNs) are the latest generation of computer simulated neural networks. SNNs are folded from neurons and synapses and this resemblance to biology is no coincidence. SNNs are trying to simulate biological neural networks more precisely. For example, they have a lower response time, reduce power consumption or allow asynchronous calculation compared to their older related analog neural networks. [8]

Neurons in SNNs are different from neurons in ANNs. Still spiking neurons (see Figure 2.2) are grouped into layers and they are connected via synapses but here similarities end. Neurons in SNNs do not have an activation function but have a membrane potential. Neurons receive and share information via sequences of action potentials, also known as spike trains [25], which alter the membrane potential. Whenever a certain threshold voltage is exceeded, the neuron produces a spike (a stimulation) and the membrane potential is reset towards a defined baseline. This phenomenon is also commonly called as neuron firing.

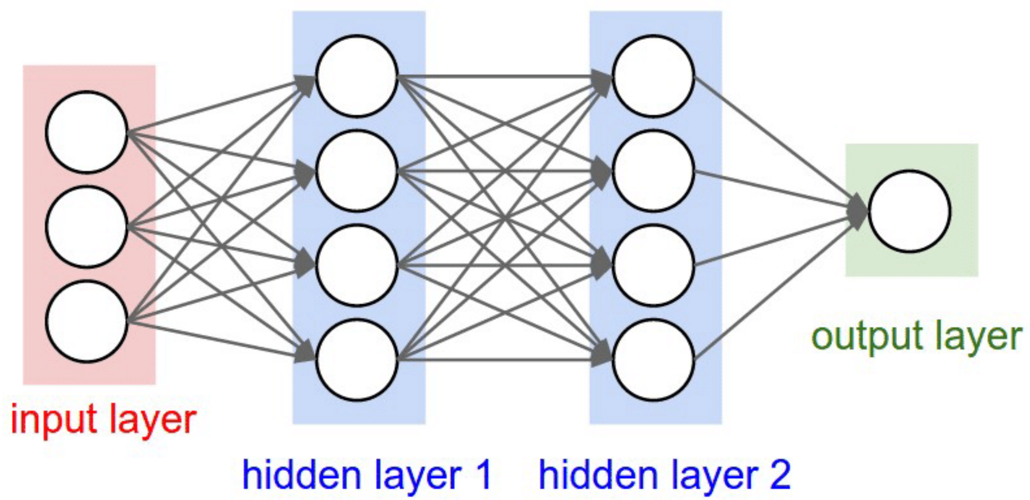

Figure 2.1: Example of ANN structure.The input layer is on the left side. In the middle are hidden layers. Output layer is on the right side. Source: [13]

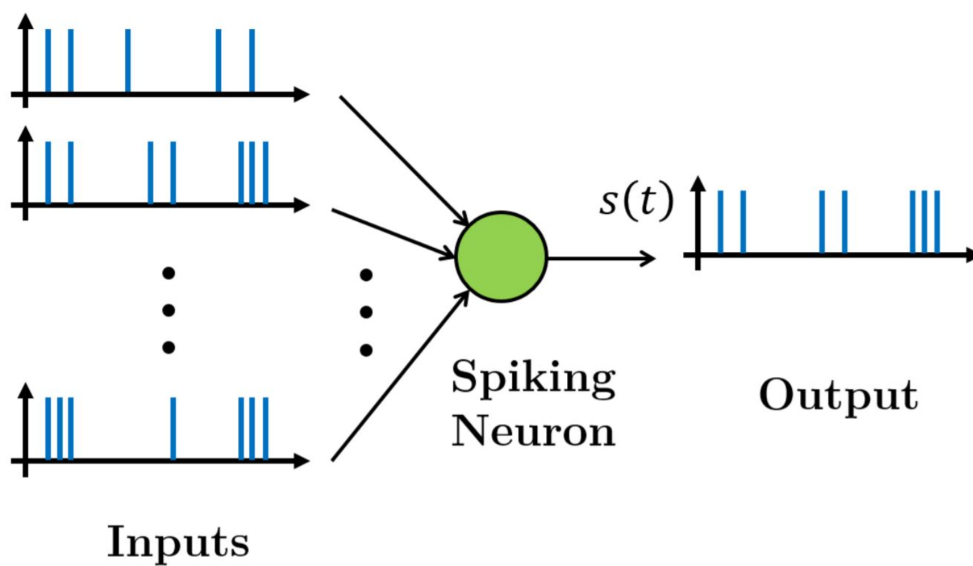

Figure 2.2: Spiking neuron with incoming spikes on the left side and outgoing spikes on the right side. Source: [2]

3 Tools for Working with Spiking Networks

In this part, the aim is to show the most used tools available for working with spiking neural networks. These tools are used for modeling and simulating spiking neural networks. Each tool specializes in a different area and can be made up of a huge framework with lots of gadgets. The main used programming language is Python for its simple syntax, portability and vast libraries support.

Four examples of simulators and one example of a modeler of spiking neural networks were chosen for further description. These frameworks are most used among all communities and have the biggest number of contributors, see Table 3.1. [8]

3.1 NEURON

NEURON [17] is a simulator of neurons and neural networks. It supports building, managing and using a great number of models, and it works for experimental data best. NEURON can be run on all platforms like MSWin (98 and upper), LINUX, etc., even on parallel hardware like Beuwulf cluster or IBM Blue Gene.

NEURON has a computational engine that uses special algorithms with high efficiency. This high efficiency can be achieved by the structure of the equations, which describe neuronal properties. It has several functions for easy control of simulation and a simple graphic output that is easy to understand. NEURON is designed to let users deal directly with familiar neuroscience concepts.

The main goal of NEURON is to help users address high-level neuroscience research questions without being distracted by any low-level mathematical or computational issues. For that purpose, NEURON has several gadgets. The first of them is that the user can utilize the neural syntax, which is made of well known neural idioms. This means users do not need any kinetic schemes or differential equations in the form of statements. The second gadget is an integrator-independent model specification. The user can choose between three integration methods which can increase the accuracy or run fast. This depends on a practical situation and empirical

experiences. The last gadget is that the user can delay or even skip the explicit specification of the spatial and temporal discretization.

NEURON has its programming language called "hoc". It uses the C-like syntax. However, NEURON offers a another option which is the Python programming language.

3.2 Nengo

Nengo [14] is another simulator of neurons, learning rules, optimization methods and much more. It is optimized for building and running both analogue neural networks and spiking neural networks, and it supports various neural simulators or neuromorphic hardware. Nengo is running completely on the Python programming language. Nengo can be run on all platforms like MSWin (98 and upper), LINUX, etc.

Nengo can be considered more pragmatic than the previous mentioned NEURON. Nengo allows the creator to replace the manual setting of weights or the usage of learning rules with the function for their computing. In Figure 3.1 Nengo GUI is shown in action. On the left side, there are graphs, and on the right side, there is a source code.

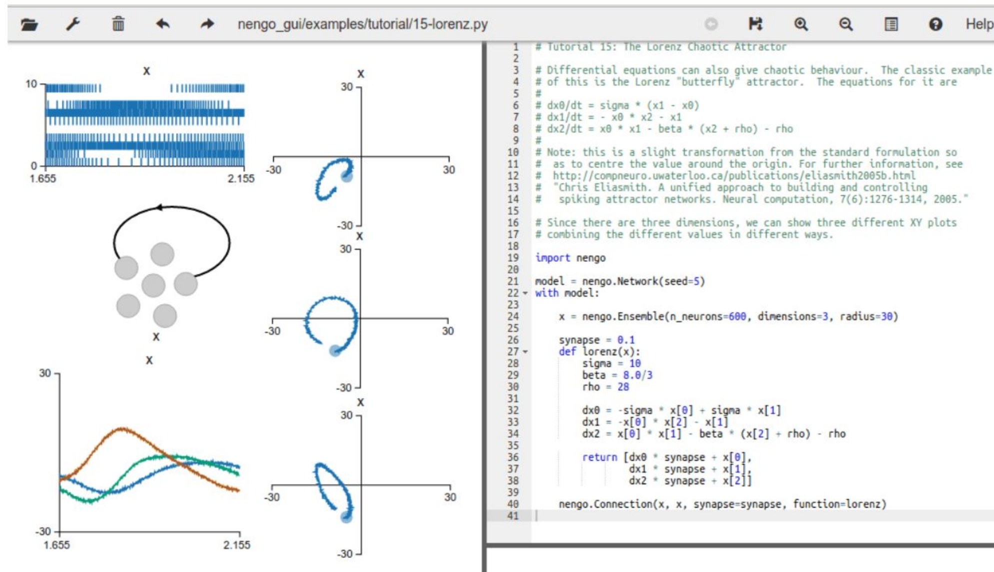

Figure 3.1: Nengo graphical user interface. On the left side, there are graphs showing the action of the source code, and on the right side, there is a source code. Source: [14]

3.2.1 NengoDL

NengoDL [15] is a specialized simulator for models from the Nengo modeller. The models of networks created in the Nengo modeller are inputs to NengoDL. NengoDL uses TensorFlow as an underlying computational framework; the user can use exactly the same code for models as in the Nengo simulator. The only change is in the different Simulator class, which is used to execute the model. NengoDL is not a copy of Nengo. It also brings numerous ad-dons to do the simulation, such as faster simulation speed, inserting TensorFlow code directly into a Nengo model and optimizing the parameters of a model through deep learning the training methods.

3.3 NEST

NEST [4] (NEural Simulation Tool) is the most used simulator according to Table 3.1. NEST is a type of simulator that focuses on the spiking neural networks model as a whole rather than on modeling neurons. NEST supports over 50 models of neurons and over ten models of synapses. NEST can use the interpreted programming language Python. NEST is supported by PyNN, and their combination is called "PyNEST". It can also stand on its own because NEST is implemented in the C++ programming language. NEST runs on UNIX-like systems, from MacBooks to BlueGene supercomputers. NEST is refereed in over 520 papers as the simulator which was used. [16]

3.4 PyNN

PyNN [3] is a tool for building neuronal network models independently of the used simulator. The source code of models is written in PyNN and then it runs on independent supported simulators like previous mentioned NEURON and NEST. PyNN supports high-level abstraction but still gives an option to access a single neuron when necessary.

In PyNN, there are predefined sets of neurons, synapses and synaptic plasticity models made for all supported simulator platforms. In the case of connectivity algorithms, PyNN has a set of commonly-used algorithms or provides an option to use different algorithms by writing them in the Python source code or making them by using the Connection Set Algebra [7] library. If the modelled neural network is used only on one simulator, it does not have to use sets of the supported neuron, synapse and synaptic plasticity

models. It can use any model that can be made via the PyNN powerful high-level interface.

3.5 Summary

In this chapter, there are presented the newest and the most promising frameworks - NEURON [17], Nengo [14], NengoDL [15], NEST [4], and PyNN [3]. All these frameworks have their own simulator of neural networks except PyNN, which is only a high-end API. In PyNN, users can model the neural network and then transform that model into one of the supported simulators mentioned previously - NEURON and NEST. The remaining frameworks, Nengo, NengoDL and NEST, are aiming for a more pragmatic view of neural networks. The main focus of NEST is on models of neural networks as a whole rather than on single parts of neural networks. Nengo and NendoDL are two frameworks which differ in the simulator class which executes the model. Nengo uses its simulator, but NengoDL uses the TensorFlow framework as a simulator.

NEST is the most used and evolving simulator in the recent year in all the statistics gathered in Table 3.1. The second and the third in the number of releases are NengoDL and Nengo. Nengo is second in the number of stars and forks as well. These two mentioned, Nengo and NengoDL, will be further used in this thesis work.

Platform	Starts	Forks	Releases	Contributors	Languages
NEURON	207	81	9	38	C++, C, Python
Nengo	684	165	20	32	Python
NengoDL	70	14	29	11	Python
NEST	44.7k	4.9k	62	288	TypeScript
PyNN	212	113	2	34	Python

Table 3.1: The comparison of frameworks. Data gathered of their GitHub's.

4 Data Augmentation

In neuroscience, there are often problems with data. In one case, there is only a little data for proper neural network training, and in the other case, data cause overfitting and accuracy losses. One solution for this problem is gathering more data on the topic. However, for example, when observing the human brain by the electroencephalography (EEG) method, it is very long and hard work to extend the dataset. For these purposes, scientists are trying to improve the existing datasets by augmentation. Data augmentation takes the existing dataset and adds new elements to it.

The concept of data augmentation first appeared in 2015, according to [10]. Before 2015 some techniques enhanced data, but these were called other names and did not form a consistent category of methods. More interesting is that since 2015 the number of scientific papers which use data augmentations as a category of methods has been increasing each year. For example, according to [10], there were 53 scientific papers where a process of data augmentation was used on EEG datasets. From 2018 to 2019, it was 37 out of 53, and in 2019 alone, there were 21 papers. In this Chapter are presented the most used and known methods of augmenting.

4.1 Noise Addition

The general aim of this method is to blur the original data. There are different types of noise like Gaussian, Poisson, "salt and pepper", etc., which represent different types of distribution of changed parts of the original data. Each of these types has different parameters that it controls, for instance, a mean value and standard deviation for Gaussian noise. This method takes one data element (for example, an image) and makes a copy for further work so that the original image remains unchanged. Then the method adds noise to this image. This noise must have the same shape as the original image and follow the selected division that determines how much individual pixels will change.

For example, taking the matrix $\begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{pmatrix}$, the method will create a noise matrix: $\begin{pmatrix} 0.010 & -0.010 & -0.016 \\ -0.034 & 0.092 & -0.075 \\ 0.106 & 0.047 & -0.034 \end{pmatrix}$ which was done using Gaussian distri-

bution with parameters $\mu = 0, \sigma = 0.1$. Then the method adds the noise to the original and the final matrix is $\begin{pmatrix} 1.010 & 0.990 & 0.984 \\ 1.966 & 2.092 & 1.925 \\ 3.0106 & 3.047 & 2.966 \end{pmatrix}$. This whole procedure can be seen in Figure 4.1.

This procedure is possible to use not only for images, but also for raw signals with a few changes. A signal can be converted to sequences of images, and then this method can perform the same procedure with these images as before. Once this is done, the images can be converted back to signals [10].

Figure 4.1: An example of noise addition in the image from the left (original) image to the right image. Source: [20]

To augment EEG data is even simpler than to augment an image. The image is a matrix, but the signal is a vector. This means that this method can augment signals as well as images. As well as in the previous example with the matrix, this procedure can be applied here too.

4.2 Generative Adversarial Network

The generative adversarial network, shortly GAN, is composed of two neural networks working one against the other, thus adversarially. The main goal of this network is to generate new synthetic instances of the data. GANs are used in image, video and voice generation. GANs' usability is wide because they can learn to mimic any distribution of data in any domain: images, music, speech etc.

4.2.1 Generative Algorithm

GAN is a generative algorithm. To understand the generative algorithm, we need to know the contrast with a discriminative algorithm. The big contrast is how these algorithms classify the input data.

The discriminative algorithm tries to predict a correct label from symptoms in the data. For instance, symptoms will be words in the *email* and labels will be *spam* or *nospam*. The discriminative algorithm tries to assign the right label to the words in the *email*. Considering symptoms $\mathbf{x}$ and labels $\mathbf{y}$, $P(y|x)$ means the probability of $\mathbf{y}$ given $\mathbf{x}$. In the case of the *email*, this is the probability that an email is spam given to the words it contains.

On the other hand, generative algorithms like GAN are attempting an inverted approach. They assume the label and answer the question: How likely are these symptoms? In the word of the previous example, a generative algorithm assumes that the email is spam, and then it ascertains what must be in the email to be spam. Since generative algorithms are inverted to discriminative ones, mathematical expressions are also inverted. They capture the formulation $P(x|y)$-the probability of $\mathbf{x}$ given $\mathbf{y}$.

4.2.2 How GANs Work

GAN is composed of two neural networks. One of them is a *generator* which creates new instances of data, and the other one is a *discriminator* which evaluates data for authenticity. The main task of the discriminator is to recognize the given data. If the sample is from the original dataset, the discriminator must say that the sample is authentic. On the other hand, the generator generates new synthetic samples and feeds the discriminator with them. The generator wants its samples to be authentic - even though they are fake. The task of the generator is to create new passable data. The aim of the discriminator is to reveal the synthetic samples which are from the generator. These samples are considered fake.

The process has these steps:

1. The generator is fed by a random number.
2. The generator creates a synthetic sample from the random number.
3. The new sample is given to the discriminator with the data stream from the original dataset.
4. The discriminator returns the number from 0 to 1; 0 means the synthetic sample is recognized as fake. 1 means that the discriminator thinks the synthetic sample is from the original dataset.

Both neural networks go against each other. The generator creates new synthetic samples, and the discriminator recognizes these samples as fake. Both learn each other's preferences and try to be better than the other.

The best is the state where both neural networks end up on the same 'skill level'. In this state, both have the same chance of winning. The generator wins if it can create new samples, and the discriminator is then unable to distinguish these samples from the original data. The discriminator wins if it can recognize the fake samples from the original data. If the generator ends up training on a higher 'skill level', it will generate samples which can go through the discriminator as the original data, but they will be too different from the original data. On the contrary, if the discriminator ends up training on a higher 'skill level', it retains almost everything that the generator creates. The augmenting will take more time because the generator must create more elements to pass the discriminator requirements. [18]

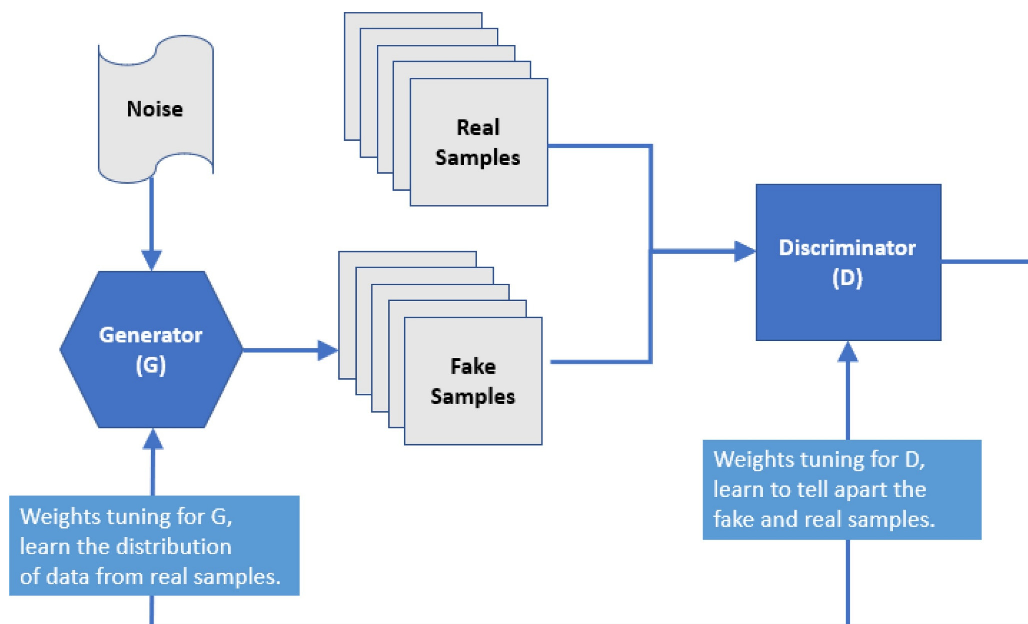

Figure 4.2: A diagram of a Generative Adversarial Network. Source: [10]

4.3 Sampling

Sampling is a technique to balance the dataset. Datasets can have classes with high differences in numbers of elements; one or a few classes have a greater number of elements than the other classes in the dataset. This unbalance in the dataset can cause accuracy losses. By sampling, the dataset can be augmented in two ways by using an oversampling or undersampling technique. [23]

4.3.1 Oversampling

This method selects an element in the minority class and makes a copy of that element for the minority class. The selection algorithm can be arbitrary, but common practice is to have an algorithm which chooses elements randomly; the algorithm does not need to know almost anything about the dataset. One element can be chosen multiple times, which adds multiple copies of that element to the dataset. In Figure 4.3, an example of oversampling method on the data is shown. Oversampling will stop when the differences in the numbers of elements of the classes are near zero. [23]

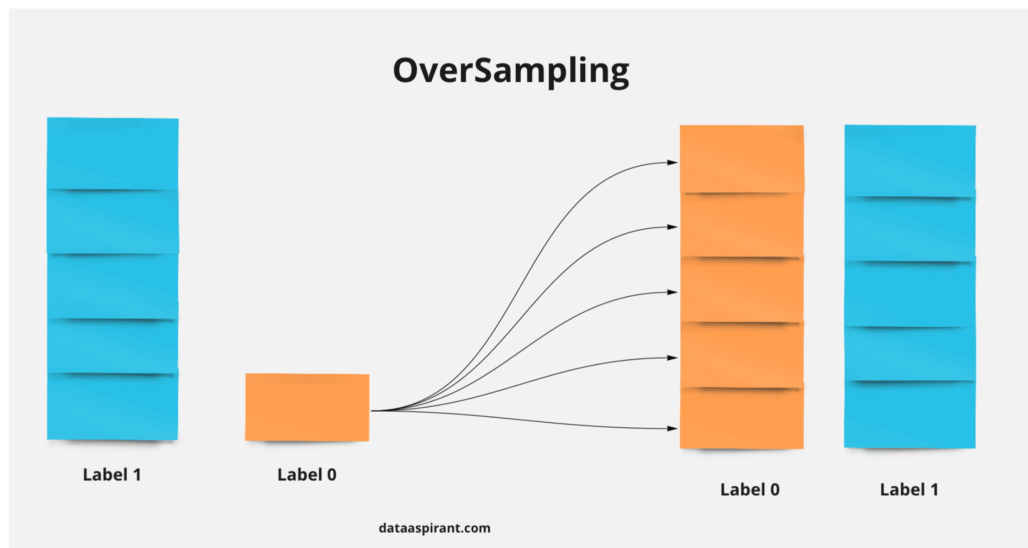

Figure 4.3: An example of oversampling on data. Source: [21]

4.3.2 Undersampling

Undersampling has the opposite path than oversampling. Undersampling removes elements of a majority class in order to make the classes equal. The selection algorithm can be arbitrary as well in oversampling. In Figure 4.4, an example of the undersampling method on the data is shown. [23]

4.4 Sliding Window

The sliding window is a technique that divides the data elements into smaller parts. All these parts are of the same size. The size of the window can be computed from the data to cover all of them, or it can be set by the user. If selected by the user, this method may not cover all the data entirely.

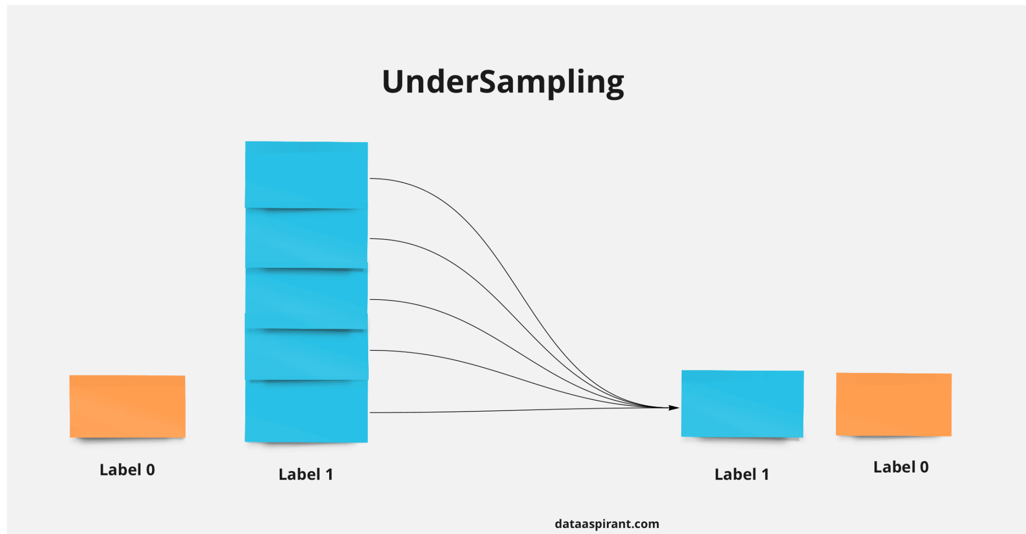

Figure 4.4: An example of undersampling on data. Source: [22]

An example can be seen in Figure 4.5. The choice of size can be difficult because there is no easy solution for that. In [10] there are several studies which examine the sliding window further. These studies report on windows from size 1 s to 5 s. This choice of size needs to be tested for every dataset and learning algorithm separately.

There are two basic options of how to do that. One option is the sliding window without overlapping, and the other is the sliding window with overlapping. [10]

4.4.1 Without Overlapping

The sliding window without overlapping is a segmentation of the data into smaller parts where one goes after the other - see Figure 4.5 (a). Here the data is a signal in time. This method splits this signal into three smaller signals which have the size of 5 s each. Then these windows can be processed separately. [24]

4.4.2 With Overlapping

The sliding window with overlapping is a segmentation of the data as well, but in this case, the part does not end where the other part begins - see Figure 4.5 (b). Here this method splits the data into four smaller segments. The overlap, in this case, is 2 s, in other words, 40 % of the window size. This overlap can be set from 100 % to 0 %. The overlap 0 % is the sliding window without overlapping. The overlap of 100 % means that the method

will start on the same point over and over again, and it will never stop. Both these extreme cases will do nothing useful if we want to use overlapping. In the other case, between 0 % and 100 %, the method can bring improvements to the dataset. [5]

How much is an overlap? This question has the same difficulty as the size of a window. As well, there is no simple solution. In [10] there were reported achievements with overlapping from 50 % to 87.5 %. Again this choice needs to be tested for every dataset, learning algorithm and the window size separately.

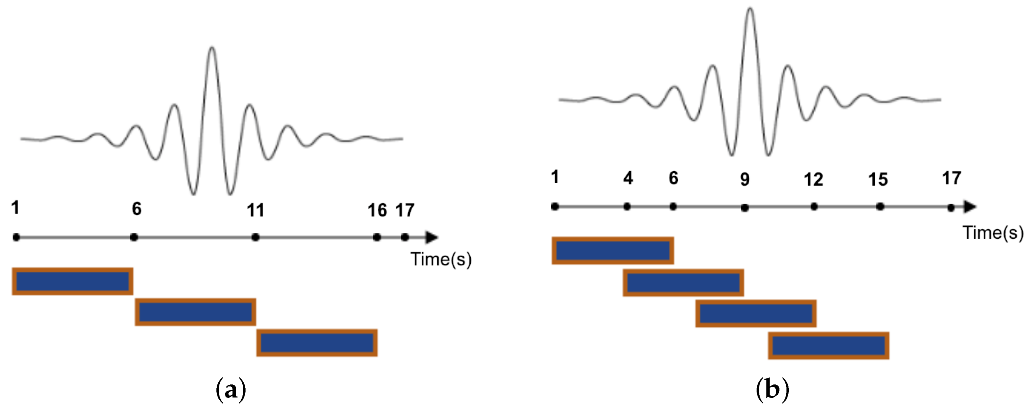

Figure 4.5: Example 5 s sliding windows. (a) The sliding window without overlapping. (b) The sliding window with an overlap of 2 s. Source: [6]

4.5 Summary

In this chapter, there were presented methods to augment datasets. Those methods were noise addition, GAN, oversampling, undersampling, sliding window without overlapping and sliding window with overlapping. Table 4.1 shows the usage of those methods which work with EEG datasets in scientific papers by January 2020. There can be seen that the small numbers of papers using each of the method do not differ significantly. Nevertheless, GAN is used in the most scientific papers and the sliding window without overlapping is used the least.

Method	Number of uses
Noise addition	12
GAN	14
Oversampling	11
Undersampling	8
Sliding window without overlapping	6
Sliding window with overlapping	7

Table 4.1: The number of augmentation methods which work with EEG datasets as published in scientific papers by January 2020. Source: [10]

5 Primary P300 dataset

There is a research group at the University of West Bohemia, the Czech Republic. This group exists here since the year 2008. They have created 30 experimental datasets and have done approximately 1000 experiments on them. For the purpose of sharing data inside group and into other groups EEG/ERP Portal was created. There are stored all datasets which were created by the group. We chose to augment the biggest dataset which is yet published. [11]

5.1 Chosen dataset

The dataset which this thesis aims to augment is P300-based from the research group on the University of West Bohemia [12]. This dataset Guess the number is an experiment where children guess the number. In this experiment a participant is tasked with choosing one number from the range of 1 to 9. Afterwards there are shown numbers from the range of 1 to 9 in a random order. This display of digits is recorded in a form of EEG signal and the number selected by the participant is guessed from the EEG signal. P300 dataset is composed of 250 samples from children at school age. Each sample has electroencephalographic data from three different channels (Fz, Cz, Pz) and stimuli markers. The length of each sample is 1500 ms. To these samples metadata about the participants were supplemented (gender, age and various interesting additional information).

5.2 Used Format of Dataset

The dataset in a raw format is hard to used effectively in programming. For this purpose, dataset was prepared into a more efficient format. All the samples were cut from start to length 1200 ms. After this cut all the samples was divided into two classes. The one class called "target" contains samples which represent P300 stimulus. The other class called "non target" includes samples which are not P300 stimuli. This all is in one MATLAB file. This pre-processed dataset can be found on this web site: https://dataverse.harvard.edu/dataset.xhtml.

6 Applications of GAN on P300 Dataset

In the chapters above, there were presented spiking neural networks (2) and tools we use to work with them (3). In chapter (4), there were shown methods to augment the data for training neural networks. After evaluating all these methods, GAN supported by a sliding window were chosen because these methods are mostly used among the scientific community - see Table 4.1. For testing the augmented datasets, the spiking neural network developed by Václav Honzík [8] was chosen because it uses a dataset described in the previous chapter (5).

The previous work of Václav Honzík [8] reached the best accuracy on the P300 dataset (using a spiking neural network) around 63%. The aim of this work is to improve its accuracy. The same aim should be reached with the work of Roman Kalivoda [9] who created a first analogue neural network to classify the P300 dataset.

Primary dataset	Augmented method
P300	GAN and Sliding window

Table 6.1: The chosen dataset for augmenting [12] and the methods to augment the dataset.

6.1 Created GANs

A discriminator and a generator are necessary to create GAN. Two GANs were created with the same discriminator and two different generators.

The discriminator has four-layer architecture. The input layer has a vector of 3600 neurons which represents three channels and a 1200 ms signal. Then there are two dense layers followed by LeakyReLU functions. The output layer has one neuron. If the last layer is activated, the discriminator recognizes an image as fake. The whole structure can be seen in Figure 6.1. In this structure, the discriminator has 8 million trainable parameters.

The first model of the generator was created from scratch with a help of an example from [18]. This article also describes how to make a generator; the example of the generator produces samples of the MNIST dataset.

The model I have developed consists of three layers with two activation layers. The input layer contains a randomly generated vector followed by the LeakyReLU function. Then there are two dense layers with one LeakyReLU function between them. The output layer reshapes the previous layer to a different shape which we want to mimic, see Figure 6.2. In this structure, the first generator has 12.7 million trainable parameters.

The second model of the generator was created from the first one by adding one more Dense layer. Another upgrade was done by adding BatchNormalization functions after each LeakyReLU function. These additions lead to a four-layer structure displayed in Figure 6.3 with two different functions between each layer. Surprisingly by adding one layer and some adjustments to the shape of layers, this structure has only 8 million trainable parameters, which is a huge difference.

6.2 Training of GAN

The training of GAN needs two main parameters. These parameters are the same for the entire training. The first one is the number of epochs. Epochs are iterations of training cycles. Here 50,000 epochs were chosen for training. This number was set experimentally. The number of 30,000 epochs was take from [18] and there testing starts. But the samples were highly different from the original ones, the number of epochs was increased. However, the number of epochs above 50,000 did not get significant improvements in created samples. The next parameter is the batch size. The batch size is the number of samples going to the discriminator in each iteration. Here the batch size was set to 32 samples. This number originates from [18] as well. Testing different batch sizes did not get any improvements. Half of the samples of batch are from the generator, and the other half are from the input dataset.

To train one model of the GAN, two generators and two discriminators are needed. This is caused by the dataset. The dataset is split into two different classes as it is described in Chapter 5. One couple of generators and discriminators can learn one type of signal. For training one model there is the need to have two couples of generators and discriminators. To get two models there will be four couples of generators and discriminators.

After training one model of GAN is done, generators are saved and discriminators are erased. Generators are saved to generate new samples of data. In Figures 6.4 and 6.5 there are shown examples of signals from the original P300 dataset. Generated signals from the first model are shown in

Figures 6.6 and 6.7. The first image shows the simulated target class, and the second image shows the non target class. Signals from the second model are displayed in Figures 6.8 and 6.9. As well, this image shows signals from target and non-target classes.

6.3 Implementation

The project is written in Python version 3.8.5. The GitHub repository is available at https://github.com/Hrabikv/Data Augmentation. There are five Python files and three text files available. Other files are only git specific or from the development environment.

The text file *README.txt* is the same as the user guide at the end of this bachelor thesis. The *requirements.txt* file defines a minimum of required libraries for the project. The most used libraries are Keras [1], TensorFlow [26], and NumPy [19]. The configuration file *config.txt* contains adjustable parameters for the project. The majority of these parameters are used for work with trained models. For example, one parameter determines how many new samples will be added to the original dataset. All parameters have their description in same file *config.txt*.

Python files are segmented by activity what each of them doing. In files *disriminator.py* and *generator.py*, there are definitions of models described above. In the file *DataWork.py*, there are two methods and one class named FileWorker. The first method merges two arrays into one. The second method loads parameters from the configuration file *config.txt*. Classes responsibility is to load input dataset, apply filter on samples from dataset and save augmented dataset. Dataset is ready for better work as it is written in Chapter 5. But in dataset there are damaged samples. For filtering these damaged samples a filter is apply. Threshold of filter was set to $\pm 100\mu V$. This filter is applied in preparation of data in the class FileWorker. Next file is *GAN.py*. This file starts with supporting methods. Only method for the averaging of signal needs highlight. This method gets two parameters. The one is array of generated samples and the other one is size of averaging window. In the method there are created new samples by averaging input samples. After those methods there is the second class of project named GAN. This class in constructor define its discriminator and generator. Other methods are predict, train, save data image, save model and load model. Predict method generate number of new samples. Number is passed as argument. Returns generated samples. Train method takes care of training the generator and discriminator. Save data image method

saving continuous generated data images. Save and load data methods are here for save model after training and load model for multiple use. Last file is *main.py*. This file is entering point of project. There are all previous methods called.

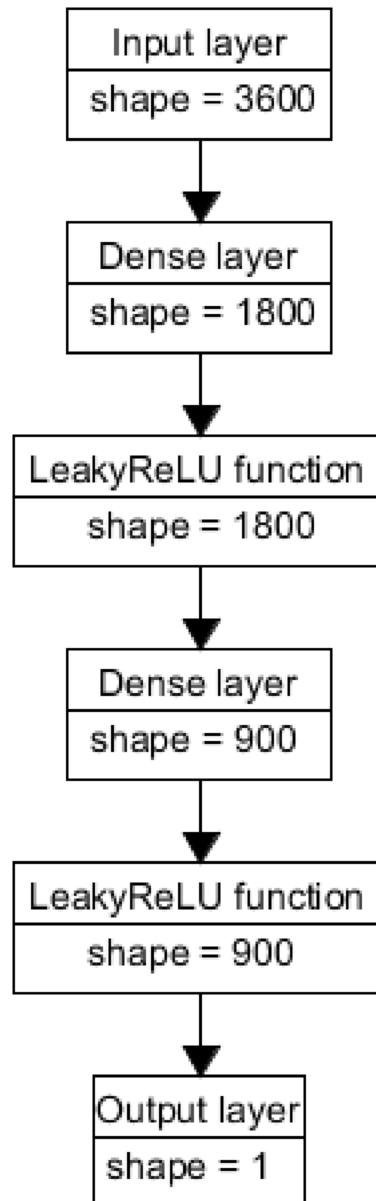

Figure 6.1: GAN - discriminator neural network.

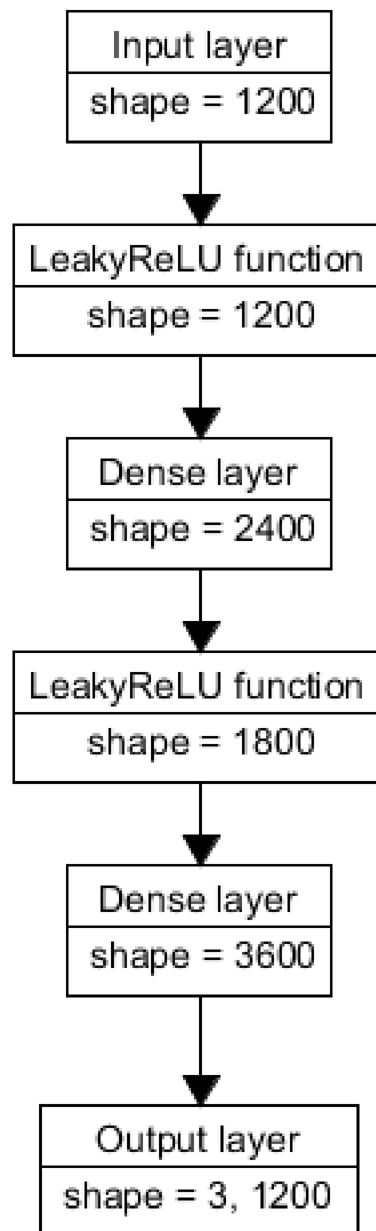

Figure 6.2: GAN - generator, the first model.

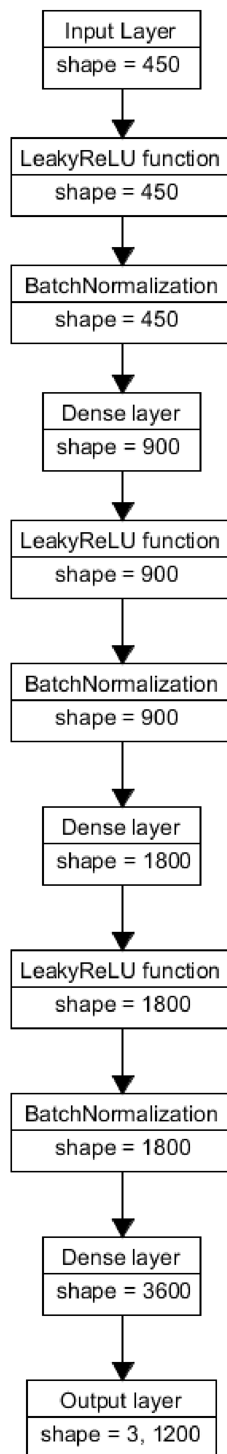

Figure 6.3: GAN - generator, the second model.

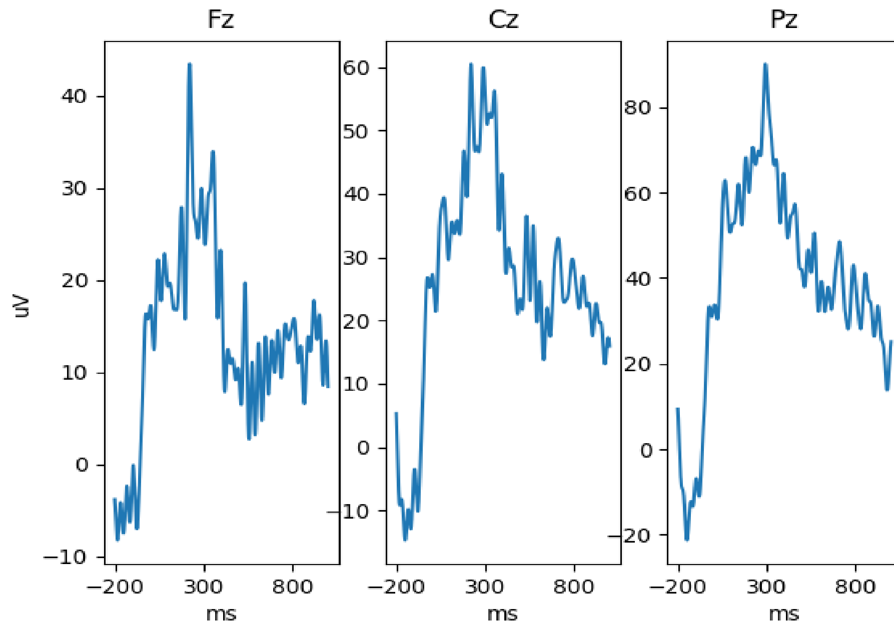

Figure 6.4: A sample of the P300 dataset from the target stimulus class.

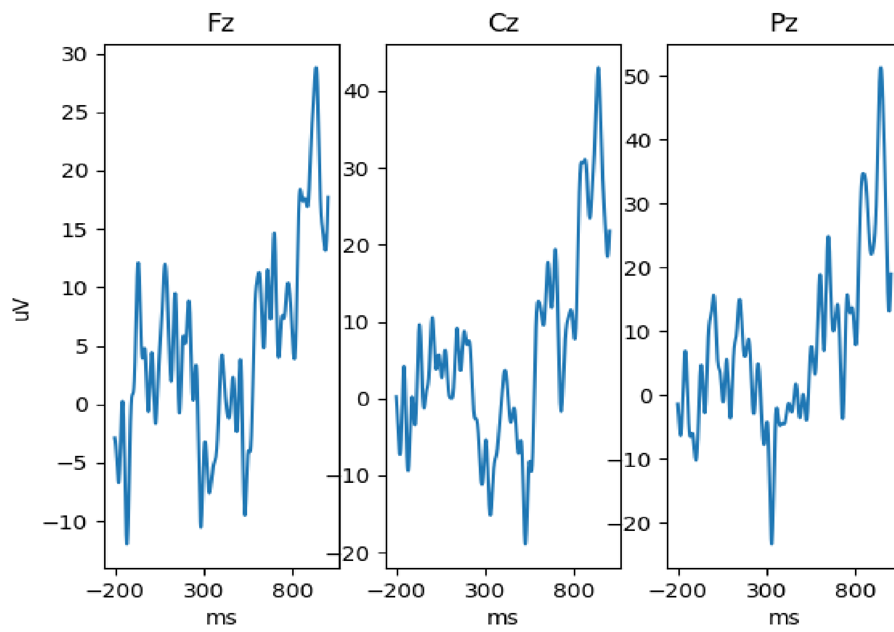

Figure 6.5: A sample of the P300 dataset from the non target stimulus class.

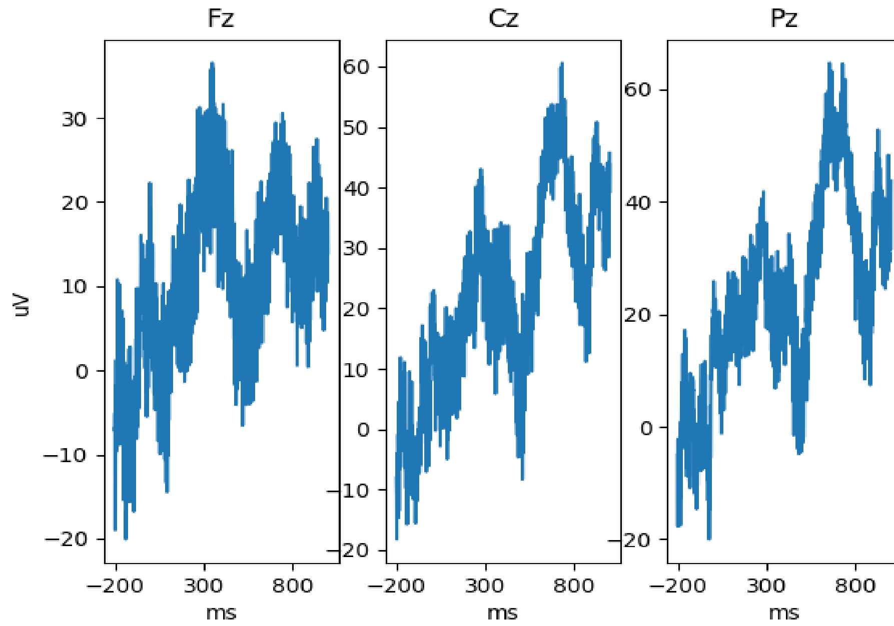

Figure 6.6: A sample of the generated signal from the target stimulus class. The first model of the generator was used.

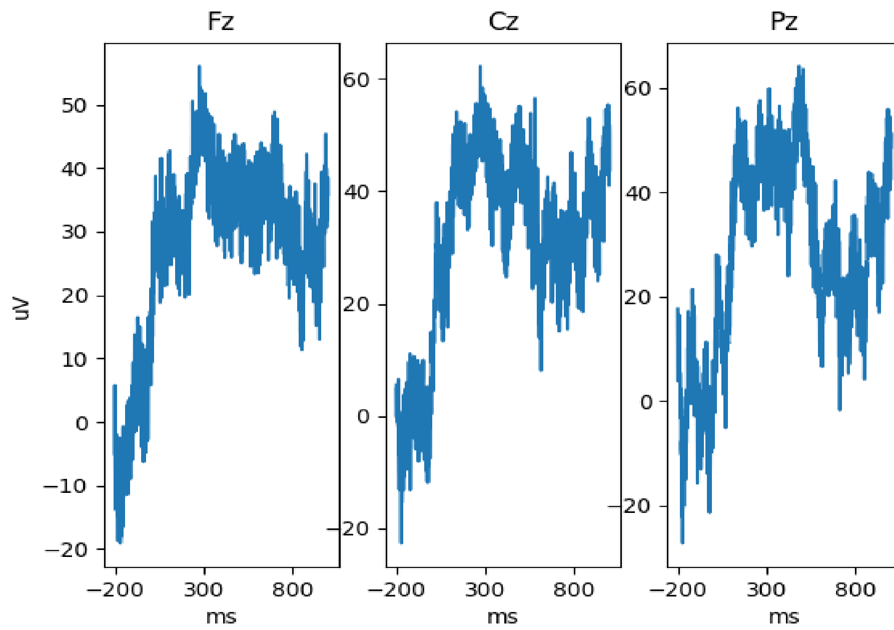

Figure 6.7: A sample of the generated signal from the non target stimulus class. The first model of the generator was used.

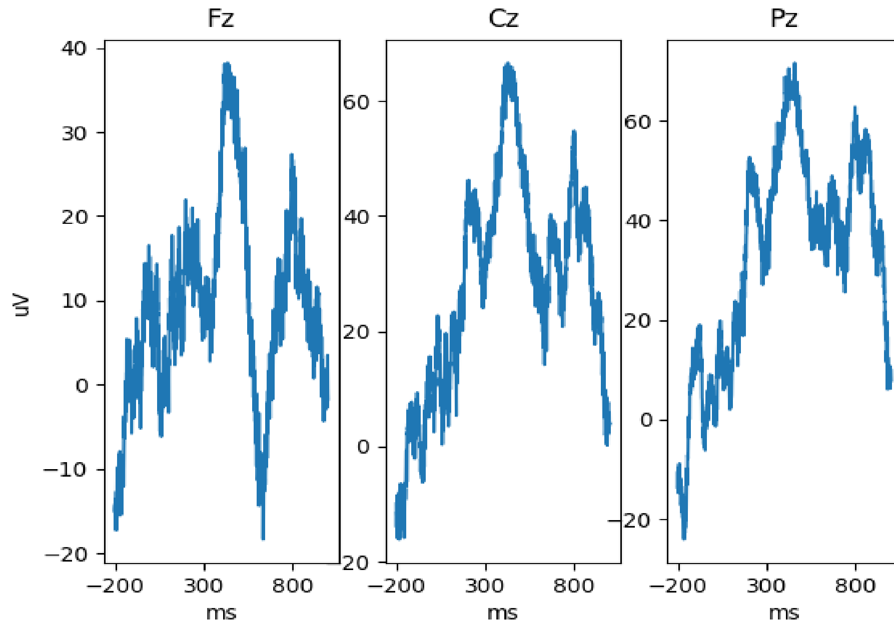

Figure 6.8: A sample of the generated signal from the target stimulus class. The second model of the generator was used.

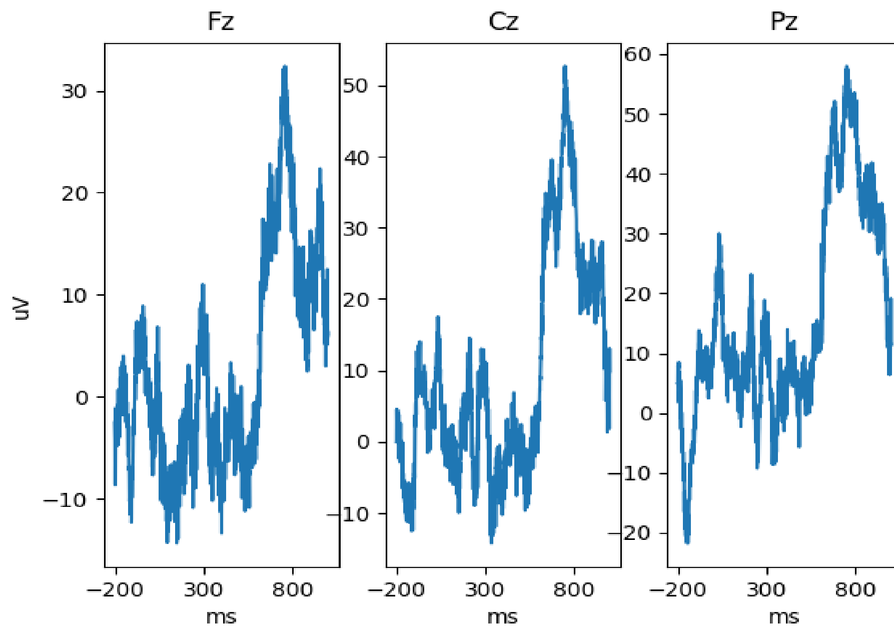

Figure 6.9: A sample of the generated signal from the non target stimulus class. The second model of the generator was used.

6.4 Results

When the models were created and put through the training procedures, it was time to create the augmented datasets. There is no exact number of how much augment or where the limits of augmentation are. Because of it, we decided to test several options to describe the ways of augmenting. These options differ in two parameters. One parameter is the expansion of the dataset. This parameter is labelled in percentages. For instance, 100% is a dataset without new samples, and 200% is a dataset of double size. The other parameter is an averaging window. This parameter describes how many generated samples will be averaged to create a new sample. For each model of the generator, tests were run in a parameter range from 150% to 400% of size and several different sizes of the sliding window. The size has steps long 50% in this range. The window has the sizes of 1, 5, 20 and 50 of samples. In total, 48 different augmented datasets were created (24 from each model).

There are five different neural networks which were created by Roman Kalivoda [9] and Václav Honzík [8]. In Roman Kalivoda [9] work an analogue neural network was created. This network is labeled as ANN in training and testing phases over augmented datasets. Václav Honzík continued in this topic. He created four new spiking neural networks. They are different in parameters called time-steps, scaling and synapse. Parameters settings are described in Table 6.2. The labels in the last column are used in training and testing phases over augmented datasets.

Neural network	Time-steps	Scaling	Synapse	Label
Spiking	50	1000	0.010	SNN 1
Spiking	50	1000	None	SNN 2
Spiking	50	1	0.010	SNN 3
Spiking	50	1	None	SNN 4

Table 6.2: Description of parameters of used spiking neural networks

In the first batch, there were datasets with the window size 1 through both models and all capacities. In total, 12 different datasets were created. In Table 6.3 there are results of the first model. The results of the second model are in Table 6.4. These settings of parameters cause a growing trend in accuracy across almost all types of neural networks. Only SNN 3 and SNN 4 in the second model has stagnation trend.

In the second batch, there were datasets with the window size 5 through both models and all capacities. In total, 12 different datasets were created.

Type\Size	150%	200%	250%	300%	350%	400%
ANN	0,7208	0,7926	0,8348	0,8594	0,8828	**0,8955**
SNN 1	0,7211	0,7917	0,8345	0,8590	0,8827	**0,8952**
SNN 2	0,7208	0,7918	0,8342	0,8590	0,8827	**0,8951**
SNN 3	0,5200	0,5233	0,5291	0,5289	0,5309	**0,5364**
SNN 4	0,5198	0,5233	0,5267	0,5305	0,5353	**0,5365**

Table 6.3: The average accuracy of the 1st model with the window size of 1

Type\Size	150%	200%	250%	300%	350%	400%
ANN	0,7247	0,7925	0,8334	0,8596	0,8835	**0,8987**
SNN 1	0,7245	0,7914	0,8327	0,8594	0,8833	**0,8985**
SNN 2	0,7243	0,7915	0,8334	0,8589	0,8836	**0,8988**
SNN 3	0,5278	**0,5264**	0,5235	0,5249	0,5229	0,5262
SNN 4	0,5279	**0,5300**	0,5277	0,5253	0,5264	0,5296

Table 6.4: Average accuracy of the 2nd model with the window size of 1

In Table 6.5 there are results of the first model. The results of the second model are in Table 6.6. As a previous batch, this one shows improvement of 0,6% in accuracy across ANN, SNN 1 and SNN 2 and 2% improvement in SNN 3 and SNN 4. Here is shown a bigger growth in accuracy compared to the previous batch. Again, stagnation trend is shown in results of SNN 3 from the second model. All other neural networks have a growing trend. The same number, which is last column of ANN, SNN 1 and SNN 2 is caused by rounding error because all these results are different only in digits below this rounding.

Type\Size	150%	200%	250%	300%	350%	400%
ANN	0,7439	0,8088	0,8455	0,8685	0,8904	**0,9041**
SNN 1	0,7436	0,8083	0,8457	0,8684	0,8899	**0,9041**
SNN 2	0,7442	0,8081	0,8459	0,8684	0,8901	**0,9041**
SNN 3	0,5224	0,5290	0,5325	0,5400	0,5433	**0,5546**
SNN 4	0,5259	0,5307	0,5421	0,5396	0,5438	**0,5498**

Table 6.5: The average accuracy of the 1st model with the window size of 5

In the third batch, there were datasets with the window size of 20 for both models and all capacities. In total, 12 different datasets were created. In Table 6.7 there are results of the first model. The results of the second model are in Table 6.8. As previous batches, this one shows improvement in accuracy across all types of neural networks. Here is shown growth from

Type\Size	150%	200%	250%	300%	350%	400%
ANN	0,7417	0,8068	0,8446	0,8675	0,8896	**0,9041**
SNN 1	0,7419	0,8065	0,8449	0,8667	0,8891	**0,9041**
SNN 2	0,7412	0,8063	0,8451	0,8672	0,8891	**0,9042**
SNN 3	0,5281	0,5288	0,5270	**0,5325**	0,5286	0,5317
SNN 4	0,5264	0,5309	0,5299	0,5276	0,5229	**0,5353**

Table 6.6: The average accuracy of the 2nd model with the window size of 5

0,05% to 0,56% in accuracy from the previous batches.

Type\Size	150%	200%	250%	300%	350%	400%
ANN	0,7475	0,8069	0,8499	0,8683	0,8941	**0,9049**
SNN 1	0,7470	0,8071	0,8497	0,8683	0,8935	**0,9047**
SNN 2	0,7473	0,8069	0,8497	0,8684	0,8933	**0,9046**
SNN 3	0,5183	0,5180	0,5284	0,5386	0,5435	**0,5508**
SNN 4	0,5228	0,5214	0,5380	0,5478	0,5522	**0,5554**

Table 6.7: The average accuracy of the 1st model with the window size of 20

Type\Size	150%	200%	250%	300%	350%	400%
ANN	0,7480	0,8099	0,8491	0,8672	0,8952	**0,9072**
SNN 1	0,7474	0,8099	0,8490	0,8673	0,8947	**0,9071**
SNN 2	0,7475	0,8099	0,8490	0,8671	0,8946	**0,9069**
SNN 3	0,5288	0,5439	0,5473	0,5453	0,5478	**0,5586**
SNN 4	0,5269	0,5341	0,5412	0,5456	0,5501	**0,5527**

Table 6.8: The average accuracy of the 2nd model with the window size of 20

In the last batch, there were datasets with the window size of 50 for both models and all capacities. In total, 12 different datasets were created. This batch was added after the previous ones were completed to test the extreme case of window size. In Table 6.9 there are results of the first model. The results of the second model are in Table 6.10. These extreme settings are very interesting. The accuracy has the best starting values on 150% of capacity from all batches. Its values are 75% for ANN, SNN 1 and SNN 2. However, as the size was higher, accuracy start losing its advantage. At the size of 350%, this batch lost against the previous batch by almost 0,5%.

Type\Size	150%	200%	250%	300%	350%	400%
ANN	0,7522	0,8060	0,8497	0,8702	0,8893	**0,9040**
SNN 1	0,7528	0,8054	0,8499	0,8700	0,8891	**0,9037**
SNN 2	0,7527	0,8056	0,8495	0,8700	0,8892	**0,9038**
SNN 3	0,5191	0,5194	0,5306	0,5359	0,5492	**0,5509**
SNN 4	0,5169	0,5258	0,5428	0,5470	0,5527	**0,5579**

Table 6.9: The average accuracy of the 1st model with the window size of 50

Type\Size	150%	200%	250%	300%	350%	400%
ANN	0,7530	0,8091	0,8490	0,8700	0,8907	**0,9049**
SNN 1	0,7530	0,8090	0,8490	0,8699	0,8905	**0,9047**
SNN 2	0,7534	0,8093	0,8490	0,8697	0,8905	**0,9049**
SNN 3	0,5433	0,5418	0,5539	0,5476	0,5593	**0,5664**
SNN 4	0,5392	0,5405	0,5452	0,5522	**0,5682**	0,5484

Table 6.10: The average accuracy of the 2nd model with the window size of 50

6.5 Summary

In the first section, a description of the created models can be found. Their training process is given in the second section. In the third section, the implementation was described. Finally, all results were presented.

All described results were made in normal test environments. The dataset was split in two groups. The first group has 75% of size, and it was labelled as the "training group". The other group has the remaining 25% of size, and it was labelled as the "testing group".

In Table 6.11 there are results of the dataset without any augmenting. Results are the achieved accuracy by neural network on dataset. This accuracy is what to we want enhance. Also, in Table 6.11 there are the best results from each batch. There are taken the accuracy from the capacity of 400% because these are the highest values from each test. From this comparison, several interesting outcomes can be seen. One of the best datasets created is the second model and window size 20. This row is highlighted in Table 6.11. It is the best dataset because the accuracy is the highest of all the tested datasets. There is the best gain in total. The other outcome from Table 6.11 is the limit of learning. From the data in Table 6.11 and data from the previous section, it seems that the limit is just above 90% of accuracy for ANN, SNN 1 and SNN 2. The other finding is that size of window do not increases accuracy that much as amount of added data. For better perspective all data in Table 6.11 are graphically shown in Figure 6.10.

Neural network	ANN	SNN 1	SNN 2	SNN 3	SNN 4
original dataset	0,6334	0,6343	0,6335	0,5197	0,5219
1st model, size 1	0,8955	0,8952	0,8951	0,5364	0,5365
2nd model, size 1	0,8987	0,8985	0,8988	0,5262	0,5296
1st model, size 5	0,9041	0,9041	0,9041	0,5546	0,5498
2nd model, size 5	0,9041	0,9041	0,9042	0,5317	0,5353
1st model, size 20	0,9049	0,9047	0,9046	0,5508	0,5554
2nd model, size 20	**0,9072**	**0,9071**	**0,9069**	**0,5586**	**0,5527**
1st model, size 50	0,9040	0,9037	0,9038	0,5509	0,5579
2nd model, size 50	0,9049	0,9047	0,9049	0,5664	0,5484

Table 6.11: The accuracy of the input dataset compared to the best accuracy of each combination of model and window size

There are some interesting accuracy gains. One of them when the second model and windows size of 50 are used. This gain can be seen in Table 6.8.

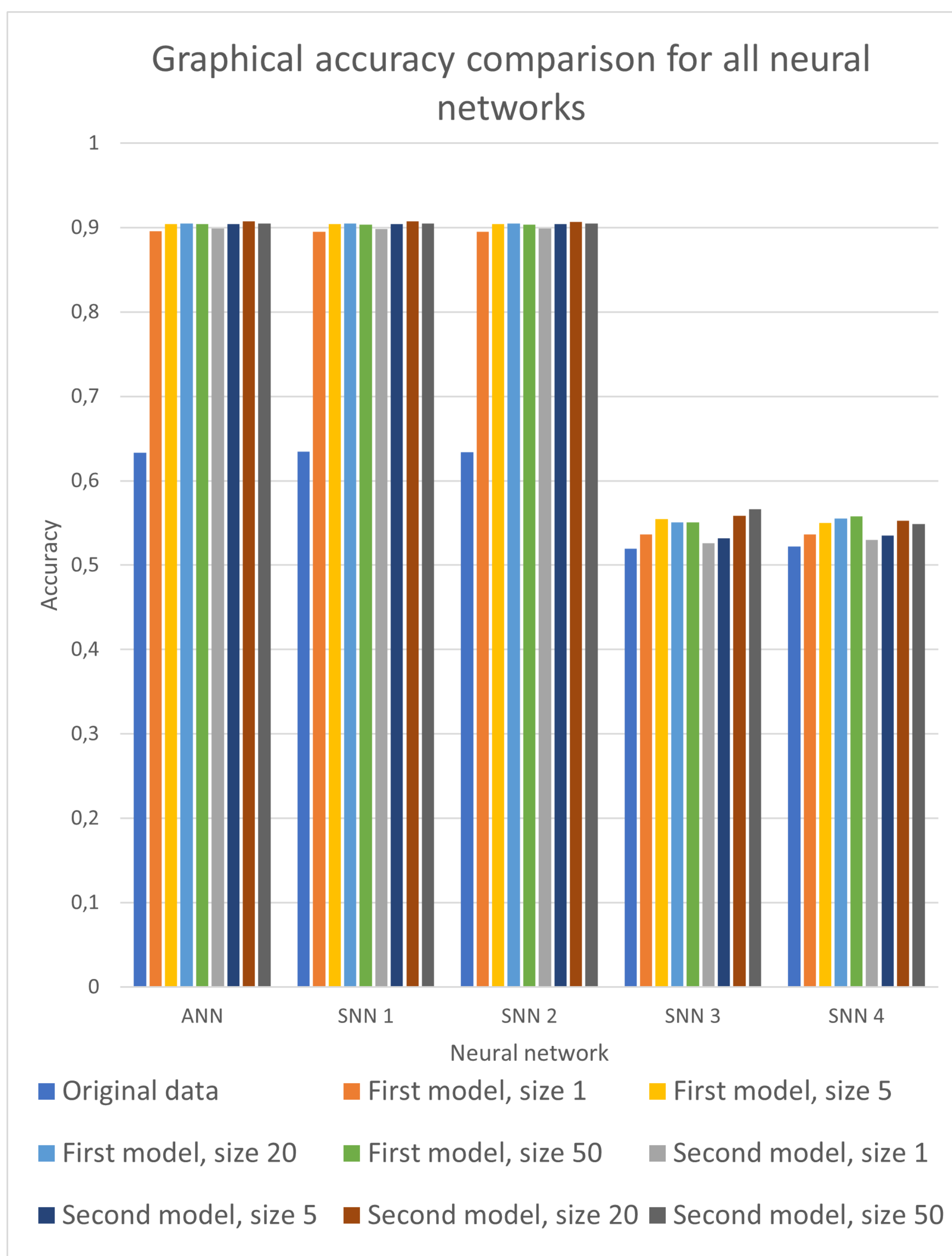

Figure 6.10: The graphical representation of Table 6.11.

The first column is the most interesting one. These values represent the highest jump in accuracy gain. It is an almost 12% jump from 63,3-63,4% to 75,3% for ANN, SNN 1 and SNN 2. This jump is interesting even more in terms of training the models, creating augmented datasets and testing them. This all can take huge computational resources. It shows that weaker computers can manage to run a decent neural network with a high accuracy.

In comparison with the other GANs, the created GAN with both models is doing a great job. In the article [10], it was written that the the average accuracy gain by GANs was 5,7%. It is worth mentioning that the numbers from the article can be misleading because the most researched neural networks there have decent accuracy from the beginning. Decent means from 75% to 85% base accuracy. The other difference is that different dataset were used.

After all previous tests were performed we came with idea to test different approach. In this approach, there were taken two datasets. One dataset was an augmented dataset which was the best dataset from Table 6.11. This dataset was created by the second model with 20 window size on a maximum 400% capacity. This dataset was labelled as "training". The other dataset was the original one without any augmentation. This dataset was labelled as "testing". The result of this test is in the middle column in Table 6.12. Accuracy gain is not as great as in previous test cases. The accuracy gain of 3% on ANN, SNN 1 and SNN 2.

Neural network	Original accuracy	Augmented accuracy	Only augmented dataset
ANN	0,6334	0,6639	0,9072
SNN 1	0,6343	0,6641	0,9071
SNN 2	0,6335	0,6641	0,9069
SNN 3	0,5197	0,5204	0,5586
SNN 4	0,5219	0,5205	0,5527

Table 6.12: There is the comparison of the original accuracy to the augmented accuracy. In the column named "original accuracy" there are results of the original dataset as training and testing group. In the column named "augmented accuracy" there are results of the original dataset only as testing group. In the last column there is the accuracy of test, where only the augmented dataset was involved.

7 Conclusion

In Chapter 2 there were slightly described neural networks, and in Chapter 3 there were introduced several frameworks for working with spiking neural networks. After that in Chapter 4 there were shown the most used augmenting techniques. Then the work and experiments done by the neuroinformatics group at the University of West Bohemia were described in Chapter 5. Created project and achieved results were presented in Chapter 6.

The created datasets and the experiments performed are the main contribution of this work. They can all increase the accuracy of tested neural networks. Some of the neural networks are worse than the others, but this is not because of the augmenting itself. In Table 6.11 there are shown results of the original dataset and the best dataset from each created model. The results are incredible because of the increase, which is almost 27% in the three out of five tested neural networks. These results are achieved only with one dataset involved. This dataset was split into two groups. One being the training group with 75% of original size. The other one being testing group with 25% of original size. However, if we involve two datasets - one being the augmented dataset as training group and the other one being the original one as testing group - results are different (see Table 6.12). Between the original accuracy and this test, there is difference 3%. It is not much, but it shows that , augmenting can improve the performance of neural networks in EEG recognition tasks.

The augmenting techniques need further testing. In this thesis, only GAN was tested because it was the most used technique according to Table 4.1. The other methods looks promising as GAN and they can may perform as good as GAN according to [10].

The source codes for this thesis are available on a GitHub repository: https://github.com/Hrabikv/Data Augmentation.

Bibliography

[1] M. M.-R. Aakash Nain, Sayak Paul. Keras [online], April 2022. URL https://keras.io/. Visited on: April 2022.

[2] N. Anwani and B. Rajendran. Training multilayer spiking neural networks using normad based spatio-temporal error backpropagation [online], July 2019. URL https://arxiv.org/pdf/1811.10678.pdf. Visited on: November 2022.

[3] A. Davison, D. Brüderle, J. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger. Pynn: a common interface for neuronal network simulators [online]. *Frontiers in Neuroinformatics*, 2:11, January 2009. ISSN 1662-5196. doi: 10.3389/neuro.11.011.2008. URL https://www.frontiersin.org/article/10.3389/neuro.11.011.2008. Visited on: November 2022.

[4] R. Deepu, S. Spreizer, G. Trensch, D. Terhorst, S. B. Vennemo, J. Mitchell, C. Linssen, H. Mørk, A. Morrison, J. M. Eppler, N. L. Kamiji, R. de Schepper, I. Kitayama, A. Kurth, A. Morales-Gregorio, P. Nagendra Babu, and H. E. Plesser. Nest 3.1 [online], September 2021. URL https://doi.org/10.5281/zenodo.5508805. Visited on: November 2022.

[5] A. Dehghani, O. Sarbishei, T. Glatard, and E. Shihab. A quantitative comparison of overlapping and non-overlapping sliding windows for human activity recognition using inertial sensors [online]. *Sensors*, 19, 11 2019. doi: 10.3390/s19225026. Visited on: February 2022.

[6] A. Dehghani, O. Sarbishei, T. Glatard, and E. Shihab. 5 s sliding windows. [online], November 2019. URL https://www.researchgate.net/publication/337357650_A_Quantitative_Comparison_of_Overlapping_and_Non-Overlapping_Sliding_Windows_for_Human_Activity_Recognition_Using_Inertial_Sensors. Visited on: February 2022.

[7] M. Djurfeldt. The python implementation of the connection-set algebra [online], 2012. URL https://github.com/INCF/csa/. Visited on: November 2022.

[8] V. Honzík. Use of spiking neural networks [online], 2021. URL http://hdl.handle.net/11025/44220. Thesis, University of West Bohemia.

[9] R. Kalivoda. Extension of neural network architecture [online], 2020. URL http://hdl.handle.net/11025/41790. Thesis, University of West Bohemia.

[10] E. Lashgari, D. Liang, and U. Maoz. Data augmentation for deep-learning-based electroencephalography [online]. *Journal of Neuroscience Methods*, 346:108885, December 2020. ISSN 0165-0270. doi: https://doi.org/10.1016/j.jneumeth.2020.108885. URL https://www.sciencedirect.com/science/article/pii/S0165027020303083. Visited on: February 2022.

[11] R. Mouček, P. Brůha, P. Jezek, P. Mautner, J. Novotny, V. Papez, T. Prokop, T. Řondík, J. Štěbeták, and L. Vareka. Software and hardware infrastructure for research in electrophysiology. *Frontiers in Neuroinformatics*, 8, 2014. ISSN 1662-5196. doi: 10.3389/fninf.2014.00020. URL https://www.frontiersin.org/article/10.3389/fninf.2014.00020.

[12] R. Mouček, L. Vařeka, T. Prokop, and J. Štěbeták. Event-related potential data from a guess the number brain-computer interface experiment on school children [online]. *Scientific Data*, 4, March 2017. doi: 10.1038/sdata.2016.121. URL https://doi.org/10.1038/sdata.2016.121. Visited on: February 2022.

[13] M. Musiol. Speeding up deep learning computational aspects of machine learning, 01 2016. URL https://www.researchgate.net/figure/A-general-model-of-a-deep-neural-network-It-consists-of-an\-input-layer-some-here-two_fig1_308414212.

[14] Nengo. What is nengo? [online], 2020. URL https://www.nengo.ai/. Visited on: November 2022.

[15] NengoDL. Deep learning integration for nengo [online], 2020. URL https://www.nengo.ai/nengo-dl/introduction.html. Visited on: December 2022.

[16] NEST. The neural simulation technology initiative [online], 2016. URL https://www.nest-simulator.org/publications/. Visited on: November 2022.

[17] NEURON. What is neuron? [online], 2020. URL https://neuron.yale.edu/neuron/what_is_neuron. Visited on: November 2022.

[18] C. Nicholson. A beginner's guide to generative adversarial networks (gans) [online], 2019. URL

https://wiki.pathmind.com/generative-adversarial-network-gan. Visited on: February 2022.

[19] NumPy. Numpy [online], April 2022. URL https://numpy.org/. Visited on: April 2022.

[20] J. Pavlovicova. image whith noise [online], Jul 2011. URL https://www.researchgate.net/figure/Examples-of-images-modified-by-Gaussian-noise-Gaussian\-noise-was-applied-on-each-image_fig7_221913964. Visited on: February 2022.

[21] S. Polamuri. Oversampling example [online], August 2020. URL https://dataaspirant.com/10-oversampling/. Visited on: February 2022.

[22] S. Polamuri. Undersampling example [online], August 2020. URL https://dataaspirant.com/17-undersampling/. Visited on: February 2022.

[23] K. Pykes. Oversampling and undersampling [online], September 2020. URL https://towardsdatascience.com/oversampling-and-undersampling-5e2bbaf56dcf. Visited on: February 2022.

[24] QuanticDev. Sliding window technique [online], November 2018. URL https://quanticdev.com/algorithms/dynamic-programming/sliding-window/. Visited on: February 2022.

[25] A. Tavanaei, M. Ghodrati, S. R. Kheradpisheh, T. Masquelier, and A. Maida. Deep learning in spiking neural networks [online]. *Neural Networks*, 111:47–63, March 2019. doi: https://doi.org/10.1016/j.neunet.2018.12.002. URL https://www.sciencedirect.com/science/article/pii/S0893608018303332. Visited on: November 2022.

[26] TensorFlow. Tensorflow [online], April 2022. URL https://www.tensorflow.org/. Visited on: April 2022.

User Guide

This chapter is a guide describing setting up the environment and creating an augmented dataset. The project is written in Python version 3.8.5. The project was tested on two different laptops with Windows 10.

First, you need all the files from the GitHub repository: https://github.com/Hrabikv/Data Augmentation. You can download it via Git or by clicking **Code -> Download ZIP**. After downloading is finished, extract all the files from the zip into a folder where you want to work (the best option is a folder with **do not have any** non-ASCII character in the path).

Setting up the Environment

When you have prepared the folder with the project, it will need the environment. If you are familiar with Python enough, install all necessary modules from **requirements.txt**.

Otherwise, follow these steps:

1. Download Anaconda3: https://www.anaconda.com/ and install it. Make sure that it is added in PATH variables in Windows. The install process can do it for you if you check off one checkbox in the install procedure.

2. Open the command line. In the folder with the project, write "cmd" into the navigation line. It is shown in Figure 7.1.

3. Now, create a new environment using the command:

   ```
   conda create -n name of your environment python=3.8.5
   ```

   "name of your environment" replace with the name that you want. This will create an empty environment that can be used to install dependencies.

4. Open the created environment by command:

   ```
   conda activate name of your environment
   ```

   This will switch you into the conda environment.

```
pip install -r requirements.txt
```

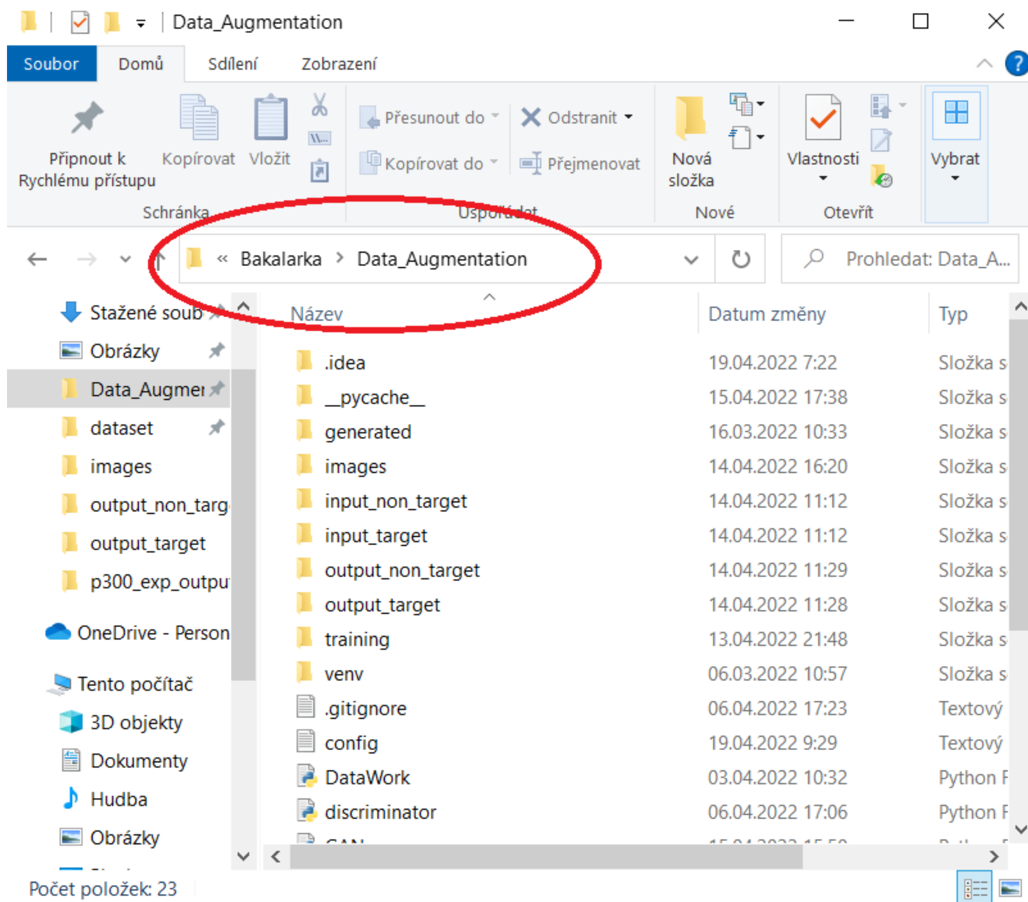

Figure 7.1: Example how to open the Command line in the folder in Windows 10.

5. To install dependencies write this command:

   This will install all module into the conda environment.

Now, everything is ready to run the project.

Work with Project

Before the project is run, you will need a dataset. A dataset can be downloaded on https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:1 0.7910/ DVN/G9RRLN. There is a file named "VarekaGTNEpochs.mat". After downloading, move this file into the folder with the other project files.

The project is run by command:

```
python main.py
```

that must be written in a prepared environment.

The project has a lot of parameters in **config.txt**. There are all parameters necessary for running the project. Do not delete any flags of parameters. On the next page is content of **config.txt**. There are parameters which need specific values. For example, there are first two parameters which indicates model which we want to train or used for augmenting. These specific values are described above each parameter. If there are not described any specific values, these parameters have no exact limitations. For example, there are parameters with flags **"-tg"** and **"-ng"** which need files with trained GANs. Theses files got default name after training but you can renamed them if you want. Because of it you must here select specific files. Same it is for next two parameters **"-p"** and **"-w"** which set two main parameters for augmenting. **"-p"** described size of created dataset in percents. **"-w"** described size of window in number of samples. Last three parameters have again specific values. In this case they are only switches whatever or not we want print examples of generated data during training, print input data or print final created data.

The training process is marked by flag **"-t"**. This flag have three specific values. There are described in **config.txt** below in the first section.

The augmenting process needs five parameters. The first parameter is **"-m"** which determines used model. The second and the third parameter are for path to trained model of GAN. Their flags are **"-tg"** for target GAN and **"-ng"** for non target GAN. The fourth parameter with flag **"-p"** determines size of augmented dataset. The last parameter **"-w"** determines size of averaging window. These parameters are in the middle section page below.

This is content of config.txt. There are described all necessary parameters for the project.

```
# Parameters needed for training:
# parameter for training of new model 1/2/n
# 1 - first model
# 2 - second model
# n - without training
-t n
################################################
# Parameters needed for augmenting:
# model which you want used for augmenting 1/2/n
# 1 - first model
# 2 - second model
# n - without generating
-m n
# file of target GAN for augmenting
-tg target_gan_mk_2.h5
# file of non target GAN for augmenting
-ng non_target_gan_mk_2.h5
# parameter to determine how much you want augment dataset
# it is set in percents
-p 150
# parameter which determines averaging window
-w 1
################################################
# Optimal parameters
# saving training progress as images T(True)/F(False)
-e F
# save input signals as images T(True)/F(False)
-gi F
# save output signals as images T(True)/F(False)
-go T
```