

DORTMUND UNIVERSITY OF APPLIED
SCIENCES AND ARTS

DEPARTMENT OF COMPUTER SCIENCE

MASTER THESIS

Creation of general representation
of a local power grid as a basis for
an embedding of electrical devices

DOI

<https://doi.org/10.26205/opus-3795>

Ulf Müller-Baumgart

supervised by

Prof. Dr. Sebastian BAB

Prof. Dr. Andreas HARRER

Dortmund, 29 February 2024

Abstract

This work develops a general representation of a domestic electricity network in the form of an embedding. Unlabelled data from a smart meter is used to train neural networks (auto-encoders) in three stages, each focusing on specific properties. Stage 1 embeds a single electrical phase, stage 2 embeds a time series such as switch events, and stage 3 makes the embedding independent of a particular phase. The final assembly of all three stages is tested with labelled data to assess the quality of the embedding found, showing distinct clusters for different states of the domestic power grid, illustrating the good result.

Zusammenfassung

In dieser Arbeit wird eine allgemeine Darstellung eines Haushaltsstromnetzes in Form einer Kodierung (embedding) entwickelt. Rohdaten (unlabeled data) von einem intelligenten Stromzähler (smart meter) werden verwendet, um neuronale Netze (Autocoder) in drei Phasen zu trainieren, die sich jeweils auf bestimmte Eigenschaften konzentrieren. In Phase 1 wird eine einzelne elektrische Phase kodiert, in Phase 2 eine Zeitreihe wie z.B. Zustandsänderungen und in Phase 3 wird die Kodierung unabhängig von einer bestimmten elektrischen Phase gemacht. Die kombinierte Endversion aller drei Phasen wird mit Testdaten (labeled data) getestet, um die Qualität der gefundenen Kodierung zu bewerten. Das gute Ergebnis wird durch die gezeigten Datenwolken der verschiedenen Zustände des Hausstromnetzes veranschaulicht.

Contents

1	Introduction	7
2	Terms, concepts and technologies	8
2.1	Latent space and GANs	8
2.2	Auto-Encoder	11
2.3	Convolutional neural networks	14
3	Planning	16
3.1	Data collection	16
3.2	Stages of learning	17
3.2.1	Stage 1 - single phase	18
3.2.2	Stage 2 - phase window	18
3.2.3	Stage 3 - all phases invariant	19
4	Exploration	21
4.1	Dataset	21
4.2	Data precision	22
4.3	Data ranges	23
4.4	Size reduction	23
4.5	Redundant data	24
4.6	Visual analysis	24
5	Implementation	28
5.1	Stage 1	29

5.2	Stage 2	36
5.3	Stage 3	47
5.4	Final assembly	53
5.5	Real world training and use	54
6	Evaluation	57
7	Conclusion and future work	62
	Bibliography	65
A	Attached source code	72
B	Code	73
B.1	Docker container	73
B.2	Models	74
B.2.1	Stage 1	74
B.2.1.1	simple-latend3	74
B.2.1.2	simple-latend4	74
B.2.1.3	simple-latend3-optimizer_sgd	74
B.2.1.4	simple-latend3-all_sigmoid	75
B.2.1.5	simple-latend4-all_sigmoid	75
B.2.1.6	layer5-latend3	75
B.2.1.7	layer5-latend3-dropout0_2	76
B.2.2	Stage 2	76
B.2.2.1	window-in3-lat8-4layers	76
B.2.2.2	window-in3-lat8-6layers	77
B.2.2.3	window-in3-lat12-4layers	77
B.2.2.4	window-in3-lat12-6layers	78
B.2.2.5	window-in3-lat16-4layers	78
B.2.2.6	window-in3-lat16-6layers	79
B.2.2.7	window-in4-lat8-4layers	79
B.2.2.8	window-in4-lat8-6layers	80
B.2.2.9	window-in4-lat12-4layers	80

B.2.2.10	window-in4-lat12-6layers	81
B.2.2.11	window-in4-lat16-4layers	81
B.2.2.12	window-in4-lat16-6layers	82
B.2.2.13	window-in4-lat12-conv-mini	82
B.2.2.14	window-in4-lat16-conv-mini	83
B.2.2.15	window-in4-lat9-conv-mini	83
B.2.3	Stage 3	85
B.2.3.1	Graphs	85
B.2.3.2	stage1_lat4-stage2_lat9-stage3_lat32- 0_01_percent-conv	86
B.2.3.3	stage1_lat4-stage2_lat9-stage3_lat32- 0_1_percent-conv	86
B.2.3.4	stage1_lat4-stage2_lat9-stage3_lat32- 0_01_percent-conv-bigger	87
B.2.3.5	stage1_lat4-stage2_lat9-stage3_lat64- 0_01_percent-conv-bigger	88
B.2.3.6	stage1_lat4-stage2_lat9-stage3_lat32- 0_01_percent-dense	89
B.2.3.7	stage1_lat4-stage2_lat9-stage3_lat32- 0_01_percent-dense-bigger	89
B.2.3.8	stage1_lat4-stage2_lat9-stage3_lat32- 0_01_percent-dense-epoch100-bigger	90
B.2.3.9	stage1_lat4-stage2_lat9-stage3_lat32- 0_01_percent-dense-epoch100	90
B.2.3.10	stage1_lat4-stage2_lat9-stage3_lat32- 0_1_percent-dense-epoch20	91
B.2.3.11	stage1_lat4-stage2_lat9-stage3_lat32- 0_01_percent-dense-epoch100-bigger-wider	91
B.2.3.12	stage1_lat4-stage2_lat9-stage3_lat24- 0_01_percent-dense	92
B.2.3.13	stage1_lat4-stage2_lat9-stage3_lat16- 0_01_percent-dense	92

B.2.3.14	stage1_lat4-stage2_lat9-stage3_lat12-	
	0_01_percent-dense	93
B.2.3.15	stage1_lat4-stage2_lat9-stage3_lat8-	
	0_01_percent-dense	93
C	GAN prompts	94
C.1	ChatGPT3.5	94
C.2	deepai.org	96

List of Figures

2.1	OpenAI Sora latent space longer calculations	9
2.2	Image of a CNN generated by ChatGPT and DeepAI	11
2.3	Auto-encoder principle	12
2.4	Auto-encoder principle training	13
2.5	CNN concept	14
4.1	Dataset duplicate analysis	25
4.2	Dataset power vs. reactive power	25
4.3	Dataset histogram of power	26
4.4	Dataset histogram of voltage	27
4.5	Dataset histogram of power factor	27
5.1	Auto-encoder Stage 1	29
5.2	Structure dataset for Stage 1	30
5.3	Auto-encoder Stage 1 latent space 3 example	31
5.4	Auto-encoder Stage 1 latent space 4 example	31
5.5	Result of the training of Stage 1	35
5.6	Structure dataset for Stage 2	37
5.7	Auto-encoder Stage 2	38
5.8	Stage 2 model definition	39
5.9	Result of the training of Stage 2	42
5.10	Structure dataset for Stage 3	48
5.11	Auto-encoder Stage 3	50
5.12	Encoder final version	54

6.1	Result illustrated as waterfall plot	58
6.2	Result illustrated with reduced dimensions	59
6.3	Raw data illustrated with reduced dimensions	60
6.4	Example to show operation with a kettle	61
7.1	Future extension of the system	63
7.2	Training of future extension	64
7.3	High dimensional Euclidean space with clusters of devices	64
B.1	Docker container	73
B.2	Docker container GPU cluster	73
B.3	Result of the training of Stage 3 part 1	85
B.4	Result of the training of Stage 3 part 2	85
B.5	Result of the training of Stage 3 part 3	86

Chapter 1

Introduction

Over the last years, the requirements for power meters have changed due to new needs arising from the challenges of the transition to renewable energy and the energy crisis caused by the Russian invasion of Ukraine.

In the past, meters were read on an annual basis. However, they have now become an integral part of the smart grid. With the increasing use of home automation systems, meters are not only responsible for measuring consumption but also for estimating the state of the household or identifying certain appliances. The traditional approach of using individual smart sockets to identify the status and consumption of appliances is costly, maintenance-intensive and does not scale.

The interfaces of modern smart meters output measurements every few seconds. These unlabelled data could become a valuable source for estimating the state of a household or individual appliances. This work aims to develop an embedded representation of the data to make the complex task of estimating the state of a household or disaggregating individual appliances from raw, unlabelled data much more feasible.

The upcoming chapters will describe the process of developing this embedded representation. The process involves planning, collecting, and preparing the required data, followed by the training. Throughout the research, various approaches were tested and the results are analysed and evaluated in the final chapters.

Chapter 2

Terms, concepts and technologies

This chapter provides explanations of the terms, concepts and technologies used or mentioned in the context of this research.

2.1 Latent space and GANs

A latent space is a high dimensional vector space where semantically similar data is near in distance to each other. It is an embedding of data where a focus to the meaning of the data is targeted and expressed within its position in the latent space.

A good example of a latent space is Sora published by OpenAI 'Video generation models as world simulators' [BPH⁺24] a few days before on 15.02.2024. It shows the possibilities of a GAN (generative adversarial networks), a content generator framework, which makes use of latent spaces. The created system is able to generate videos which have temporal coherence and object coherence and can be created from text prompts as explained below.

Image data from videos is brought to a latent space based on the work of [RBL⁺22]. The videos are brought from pixel space to a temporally and spatially latent space [BPH⁺24]. Afterwards, they are embedded.

The system employs an encoder and a decoder. The encoder brings video data

from the pixel space to this latent space (embedding) and a decoder brings it back from the latent space to the pixel space. The decoder generates videos this way.

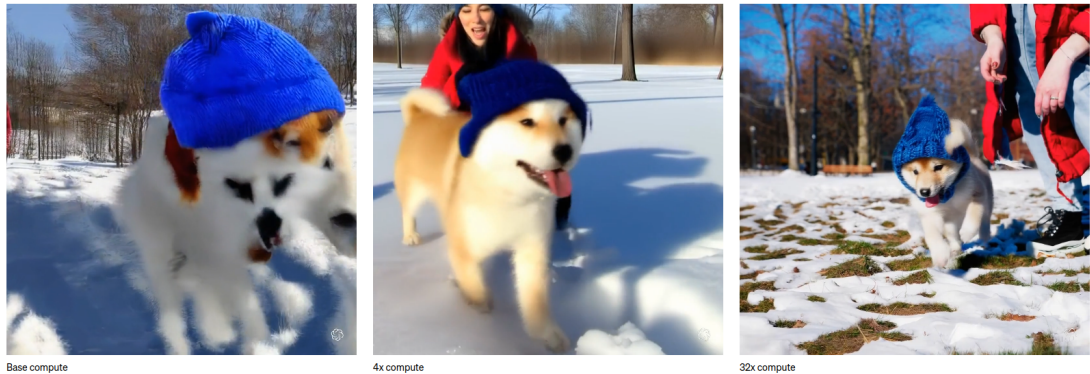


Figure 2.1: The image shows that the concepts are encoded in the latent space. The concepts like snow, dog and poodle hat, exist in every version of the video. The more computation is applied (left to right) the finer the concepts are expressed.[BPH⁺24]

The operations on the data in the latent space can be controlled with text prompts similar to systems like ChatGPT.

To understand the concept of embedding, one can think of every vector in the latent space as a representation of a complex situation in a video. Similar concepts create similar vectors close to each other.

A concept of a situation could be 'snow, dog, person, timestamp 1.3 seconds'. In simple terms, the video is controlled and created by creating and moving the vectors around in the latent space and bringing the results to the pixel space.

The control of the process comes from the fact that a vector in a certain region is surrounded by other vectors of very similar meaning but slightly different properties (variations in concepts).

Videos with text descriptions bring the connection of text prompts and vectors of concepts of situations in a video. The videos are transformed into a concept which the authors call 'patches', similar to the concept of tokens in languages. This way, the creation process can be controlled with text prompts (e.g. what concept is shown).

Sora brings improved temporal coherence. This means that the video adjusts the scene slightly between consecutive frames. For example, the video shows a dog in frame 1, 2 and 3 moving instead of a dog in frame 1, a tree in frame 2 and a person in frame 3, or different dogs in each frame. This becomes possible, as the representation of the output in a latent space encodes semantics to differentiate between 'a dog' and 'this specific dog, timestamp 1.3 seconds'.

Sora brings better object coherence. An object in frame 1 of the resulting video could be occluded or disappear in frame 2, and would remain the same when it reappears in frame 3. For example, if a car would drive in front of a dog, the same dog would appear as soon as the car passes.

To illustrate this, imagine a latent space with the axes x , y , z and opacity. The axes x , y , z encode the position in a room from 0 to 1, the opacity encodes if the transparency from 0 for invisible to 1 completely visible. If 4 objects of a video would be embedded in this latent space one of the objects could be visible in one frame with an opacity of 1, invisible in the next frame with an opacity of 0 and again visible in the third frame with an opacity of 1. The object would seem to disappear in the resulting video but the encoding would keep the object persistent in the scene. This latent space should have a good object coherence in the results. It is a bit like younger children have to understand this concept and learn that objects don't disappear even if they are not visible because they are occluded.

The effect of the meaning of vectors and the precision of concepts can be seen in figure 2.1 *OpenAI Sora latent space longer calculations* on page 9 where the main concept improves as more computation is applied to refine the concept in the latent space as the results are created from random noise and refined over time. It is easy to see that the main concepts exist in every version but the longer the computation goes on the finer the concepts get expressed. It is not quantitative qualities such as noise, resolution, etc. that improve, but rather semantics such as concepts, details, etc. of the image.

All of this demonstrates the strength of latent spaces where the encoding of certain properties, such as semantics, in regions of the space becomes possible.

The image shown in figure 2.2 *Image of a CNN generated by ChatGPT and DeepAI* on page 11 is generated with two GANs (ChatGPT [Ope24], DeepAI [Dee24]) to illustrate a simple experiment with a GAN in real life. ChatGPT is used to embed the question about the concept of convolutional neural networks with a text prompt and the GAN extends the text token by token with the tokens of highest probability. The result looks like an answer. ChatGPT is used to generate an input for a DeepAI image-generating GAN. The resulting image displays diagram fragments without any meaningful output. The input needs to be fine-tuned for the output to serve the intended purpose. See the appendix in section C *GAN prompts* on page 94 for the prompts.

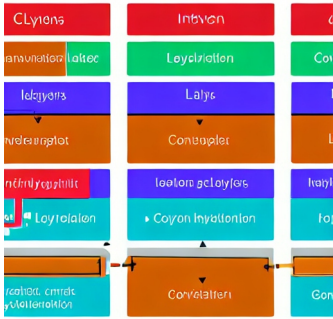


Figure 2.2: In an attempt to illustrate a CNN a GAN (DeepAI) [Dee24] prompted by another GAN large language model (ChatGPT) [Ope24] was used (full prompt in appendix). The illustration seems to show some abstract layers without any sensible information.

2.2 Auto-Encoder

The concept of auto-encoders is based on the training of neural networks. Auto-encoders open up the possibility to learn an encoding of unlabelled data for unsupervised learning. The concept is often used in GANs.

An auto-encoder consists of two neural networks that work together during training. The first network is the encoder, which transforms the input data into an encoding of limited size. The dimensionality gets reduced, which could be seen as a type of 'compression' to a dense semantic vector. The decoder then recreates

the original input data as good as possible from this encoding (see figure 2.3 *Auto-encoder principle* on page 12).

The learning process tries to reduce the error between the input data given to the encoder and the result which the decoder is able to reconstruct from the encoding created by the encoder.

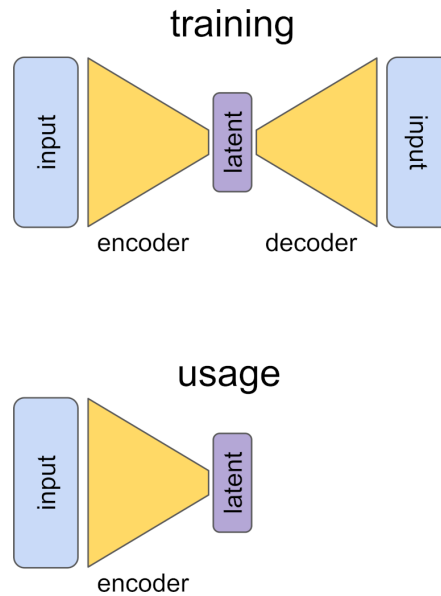


Figure 2.3: An auto-encoder consists of two neural networks, the encoder and the decoder connected to each other. The interconnected 'middle' is of reduced size to enforce a process of learning of a compressed, semantic vector space: the latent space. After the training, the encoder is used to encode the data into the latent space. The decoder is not needed after the training process and could be discarded.

The goal of the training is that the encoder and decoder find some kind of meaningful understanding or 'language' to formulate an input, the 'encoding', and learn how to express a more complex data input from that, done by the decoder.

After a successful training, the decoder is not needed anymore in most cases. The encoder would only be used to transform all data into the encoded form, called the latent space.

An auto-encoder, as seen in figure 2.4 *Auto-encoder principle training* on page 13, that has been trained on facial data shows poor results at first (first row). As

training progresses, the encoding or latent vector tends to contain more specific and helpful semantic data, resulting in better suited decoder results (middle row). Following further training, the error may fall below a certain threshold or remain around a fixed value. This convergence might be sufficient, and is an indicator that the training was successful if low enough.

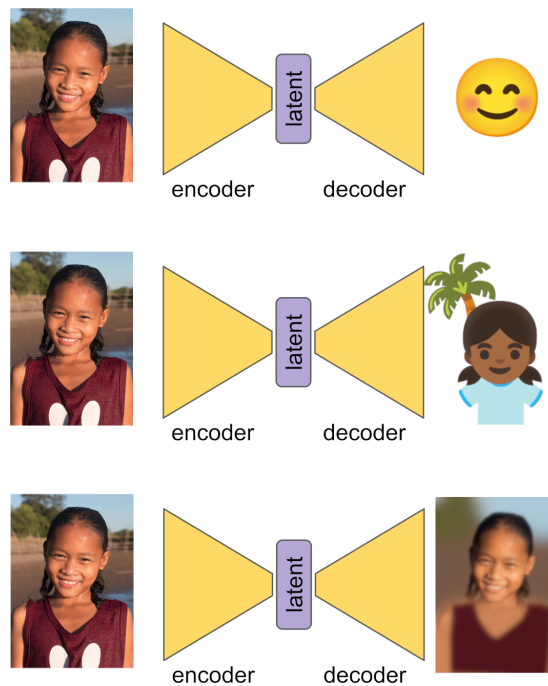


Figure 2.4: The picture illustrates an example of a learning process of an auto-encoder. The input pattern with face data applied in an early stage of the training leads to poor results of the decoder (top row). The results improve during the training and the latent vector (between encoder and decoder) tends to get rich in semantic expression (middle row). If the error of the results of the decoder gets below a certain threshold, it is expected that the encoder is trained sufficiently [Wik23].

The encoder would be ready to use to transform input data to this encoding space: the latent space.

The properties of the latent space depend on the input data, type of training and parameters like the error function.

Latent spaces can be used with GANs, were an encoder transforms the images

of two faces into a latent space. The properties (like axis) in the latent space could provide the possibility to transform a property of the face in the first image (e.g. smiling woman) to the face in the second image (e.g. sad looking man). This way, it could be possible to regenerate the second image with some of the properties (such as smiling) of the first image. The training could enforce certain properties of the latent space depending on the use case.

2.3 Convolutional neural networks

Convolutional neural networks bring, among other things, the capability to apply specific functions (convolutions) to the input data. These functions could filter or group, sharpen or soften certain aspects of the input. This can be used to take the position of the input or regional information of the input into account.

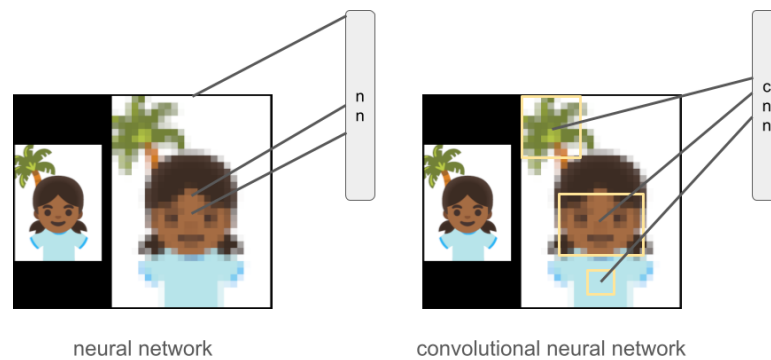


Figure 2.5: Convolutional neural networks are able to apply different types of functions to input data. The type of function is learned during training. The regional information of the pixel of a face is used to illustrate the regional character of the pixel in this example. Whilst on the left the input consists of isolated pixels, the convolutional network on the right is able to apply a function to the input to preserve the regional information.

The pixel of an image encodes regional information. If given as an input to a neural network, a pixel of the first row could be quite far away from a pixel in the fifth row, depending on the way the input of the neural network is structured. Patterns which might appear in the image (e.g. a face) could be hard to learn for a neural network because of this.

Certain filters or similar operations could be used to extract or amplify such information. Convolutional neural networks bring the possibility to learn such operations over the course of the training.

A simple example of the described concept can be seen in figure 2.5 *CNN concept* on page 14. A convolutional neural network is able to apply a learned local aggregation function. The normal neural network would receive the same pixel information on input neurons located far away from each other, which might lead to poorer results.

Chapter 3

Planning

In this chapter the planning of the data processing is done.

As the project is not based on a ready to use dataset like the very popular MNIST [Den12] or COCO [LMB⁺14], data collection, processing and aggregation into a dataset is part of the planning.

The concepts of the learning process are described and different stages of the process are planned to enforce the focus on certain aspects of the training aspect.

3.1 Data collection

The original plan was to use an unmodified Shelly 3EM to collect data every second. The plan was to transfer the data via MQTT to an InfluxDB where the data would be stored and later extracted to create the dataset. The concept was changed as it turned out that the setup became more and more complex as more parts like the Telegraf software (connector between MQTT and InfluxDB) were needed. The setup reached a point where it became very fragile and the collection crashed at least once every 48 hours on average. As this resulted in many gaps in the raw data, the approach was changed.

The required data will be collected using a Shelly 3EM smart metering device, which is capable of monitoring all three phases of a household. The device is

equipped with custom Open Source firmware Tasmota for enhanced flexibility and reliability. The data is transmitted via MQTT to a Raspberry Pi and stored in simple CSV files. To minimise the risk of data loss or corruption, a new file is created every hour.

Due to the nature of the setup, which involves a real household with consumer-grade hardware, there may be small gaps in the data resulting from updates, reboots, network problems, or power outages. The plan is to address this issue during the training phase. Such errors are expected to be acceptable, as they would occur in regular real-world setups as well.

Each data point includes a timestamp, as well as the total exported and imported consumption of the current and the previous day for:

- power (W)
- apparent power (VA)
- reactive power (var)
- power factor ('pf')
- frequency (Hz)
- voltage (V)
- current (A)

for each of the three phases. Data samples are taken every two seconds, as this is the fastest rate at which the firmware can reliably provide this data without requiring extensive customisation.

3.2 Stages of learning

The primary goal is to find a universal embedding for devices which could allow device disaggregation and device clustering at some point. The embedding must remain consistent over time and across sockets and phases. Whether a device, such as a toaster, is used in a kitchen or an office should not make a difference

in its representation. To accomplish this, a staged setup will be implemented to facilitate the learning process.

The stages are used to refine specific aspects during training and focus on particular goals at different learning steps.

A stage is considered adequately trained if the training process produces satisfactory results. Due to limited resources, it may be necessary to accept lower quality results if the process demonstrates feasibility, as otherwise the training would take too long.

The plan does not involve using the embedding for tasks such as clustering or categorisation, as this may exceed the scope of the project.

3.2.1 Stage 1 - single phase

In Stage 1, the focus is on creating a universal and abstract representation of a single data point.

To accomplish this, an auto-encoder is trained using all available data to generate a good general representation for a single phase. The encode-decoder (auto-encoder) takes a vector consisting of $[W, VA, var, pf, Hz, V, A]$ as input and is expected to output the same vector. Synthetic training data is also incorporated by adding random noise to the input, but not to the output. This is expected to make the encoding more robust and stable in a real-world setup.

3.2.2 Stage 2 - phase window

Stage 2 focuses on creating a universal and abstract embedding of the current state of a single phase, using the embedding of Stage 1. The input consists of a window of 16 consecutive raw data samples encoded by the Stage 1 encoder. This accounts for approximately the last 30 seconds of data.

The raw data is checked and filtered prior to training to prevent any gaps that may arise due to the setup. Only samples with time deltas of approximately 2 seconds are utilised.

The second stage uses another auto-encoder to find an embedding which represents the current state (with history) of a single phase. In a first step, the input and output data is the same.

In the next step, synthetic training data is generated, similar to Stage 1, using random noise. The aim is to determine whether this approach produces more robust and improved results, or if the embedding does not improve with the noisier version of the already encoded data.

3.2.3 Stage 3 - all phases invariant

In Stage 3, the focus is on creating a universal, abstract and invariant embedding of a single state of all provided phases (three or less). The embedding from Stage 2 is used to represent the state of each respective phase. All phases are fed into an auto-encoder, as in the previous stages.

The goal of this training is to get a representation which is invariant of the phase. For instance, a toaster should be represented equally whether it is connected to phase A, B, or C.

Another target are devices like electric ovens, heat pumps, hot water boilers and the like which are connected to more than one phase. The representation should be independent of the phase configuration as much as possible. For example, an oven connected with its clamps 1, 2 and 3 in a configuration A, B, C should be represented as equally as possible if connected in a configuration A, C, B.

If data of less than three phases is provided for some reason (single phase system or failure) the system should provide an embedding that represents the available data as closely as possible to a system with more data. For instance, if one of the three phases is missing, the embedding should still be usable with the remaining phases. This approach ensures compatibility of data with different configurations. For example, if a toaster is represented on a single-phase system, it should be comparable to the same toaster on a three-phase system.

In order to achieve this, the training data will be structured as follows (permuted) to cover all possible combinations:

- $A B C \Rightarrow A B C$
- $A C B \Rightarrow A B C$
- ...
- $A B C \Rightarrow B C A$
- $A C B \Rightarrow B C A$
- ...
- $A 0 0 \Rightarrow A 0 0$
- $0 A 0 \Rightarrow A 0 0$
- $0 0 A \Rightarrow A 0 0$
- $A B 0 \Rightarrow A B 0$
- $0 A B \Rightarrow A B 0$
- $A 0 B \Rightarrow A B 0$
- ...

As in previous steps, synthetic data with noise is tested to see if this leads to better results.

The exponential to factorial growth of the dataset when permuted might become problematic. This might lead to a limited training setup with constrained computation capabilities.

Chapter 4

Exploration

This chapter covers the creation of the dataset, based on the previous chapter section 3.1 *Data collection* on page 16. The raw data is analysed, prepared and transformed into a machine-learning-friendly format known as pandas data frame.

4.1 Dataset

The data from a household was collected using the setup described in section 3.1 *Data collection* on page 16. The data was written to CSV files except for minor short outages.

The dataset used in this work was created by concatenating all CSV files collected up to November 16th, 2023, starting at '2023-09-23 13:19:02' and ending at '2023-11-16 15:28:28'. There are some gaps in the dataset due to setup problems and short power outages.

The time frame contains 2,191,434 measurements and covers 4,673,366 seconds or around 54.1 days. On average, this equates to 2.13 seconds per data point, including gaps. As the data was taken every 2 seconds (sometimes 3 seconds) the gaps seem small enough to be ignored at this point. However, it is important to address these gaps as soon as a window function is applied to the dataset.

The column for Hertz was dropped as it was constant at 50Hz.

4.2 Data precision

Most of the columns are assigned data types that use an enormous amount of precision when the data is imported into Pandas without any prior definition. Timestamps are defined in nanoseconds and most numbers end up as floats with a precision of 64 bits.

To reduce memory usage and possibly speed up training in later steps, an initial data exploration is performed.

Time stamps are reduced to a precision of seconds, and all power related values (reactive power (rpower) and apparent power (apower)) are cast to integers with a precision of 16 bits, which is more than sufficient and allows values up to 65 kW, not expected to be exceeded in households.

The voltage is reduced to an integer of 8 bits, as legal requirements state that the grid voltage needs to stay in the range of 230V with a maximum of +10% and -6%. As this means that the maximum value is expected to be 253V, an integer of 8 bits (maximum value 255) is more than sufficient.

By reducing the power factors and currents to 32-bit floats, there is still enough precision to represent the values accurately.

timestamp	datetime64[s]
power 1/2/3	int16
apower 1/2/3	int16
rpower 1/2/3	int16
power factor 1/2/3	float32
voltage 1/2/3	int8
current 1/2/3	float32

The precision of the resulting data types still exceeds those provided by the Shelly 3EM.

4.3 Data ranges

To preprocess the data and make it usable for training, it is necessary to scale the different features. This ensures that values such as a large power value with hundreds of Watts and a small current value in fractions of Ampere, are scaled to a similar range, making the nature and range of the values more obvious.

In the raw data, the currents were mostly below 20A. To scale the data range with enough room for larger values without losing too much precision, a maximal current of 50A was declared as the upper limit. This should cover even situations in households with electric ovens, cooking plates, water heaters and washing machines running at the same time.

This lead to 11500W at 230V. Even with a very low (out of specification) voltage of 200V a load of 10 kW ($200V \cdot 50A$) is representable. This is expected to be sufficient.

power 1/2/3	0 W to 11 500 W
apower 1/2/3	0 W to 11 500 W
rpower 1/2/3	0 W to 11 500 W
power factor 1/2/3	0 to 1
voltage 1/2/3	0 V to 255 V
current 1/2/3	0 A to 50 A

4.4 Size reduction

Due to the adjustment of precision and data types, the semantic information of the data remains unchanged, but a massive reduction in size is achieved.

As power, apower and rpower get the size of 16 bits, their size is reduced to 25%. The power factor and current are reduced by 50% and the voltage is reduced to 12.5% of its original size.

This reduces the data size by more than 5 times to an impressive 18.75% of its original size if the timestamp is not taken into account.

Reducing the size of the dataset is likely to speed up the learning process as it

requires less memory. At least it is good practice in machine learning to analyse and prepare a dataset before training this way.

4.5 Redundant data

To prevent unnecessary training and over-fitting of the dataset, it is important to ensure that it is free of redundant data. The dataset could contain long periods of identical data such as night hours with very low load.

Over a period of 54 days, from 23.09.2023 to 16.11.2023, a total of 2,191,434 measurements were taken, with 106,557 double entries, representing 4.86% of redundant data. As the power meter data is taken as a continuous stream of measurements, a small amount of redundancy is expected and accepted. In the later stages of the learning process redundant data points are necessary to create a continuous, windowed dataset.

As the amount of redundant data is small, it is expected that it will not slow down the learning process or lead to overfitting. Therefore, the redundant data is ignored.

4.6 Visual analysis

After the raw data has been scaled, transformed, checked and stored as a dataset in the form of a Pandas DataFrame, qualitative analysis is performed.

Based on my previous research a good method of analysis was found in 'Noninvasive appliance load monitoring' [Har92]. A space of power and reactive power is plotted in in figure 4.2 *Dataset power vs. reactive power* on page 25. The three phases with their loads are shown and the different patterns show the different types of household appliances on the phases.

As the data represents a phase rather than an individual appliance, the pattern represents the combined overlaid information of all the appliances driven by that phase.

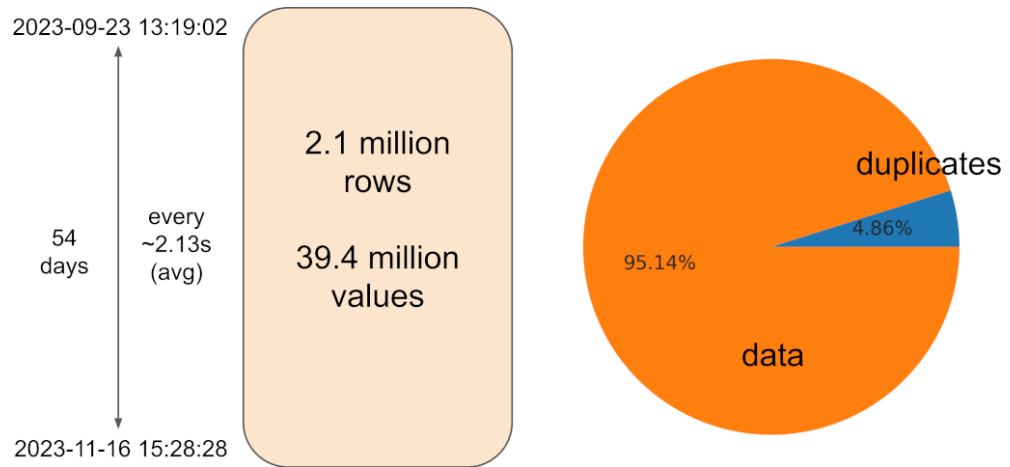


Figure 4.1: The dataset consists of 2.1 million rows of data with a timestamps taken around every 2 seconds in 54 days. The analysis revealed that there is less than 5% of redundancy in the data (e.g. night hours with very low activity). It is likely that this does not harm the initial training and the redundant data is needed in the later dataset of windowed time series.

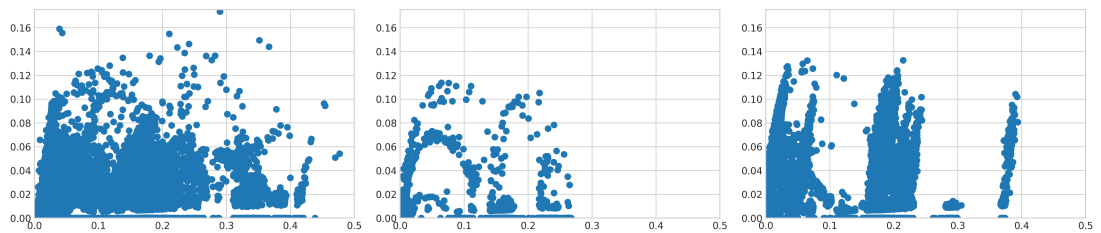


Figure 4.2: The illustration displays the scaled data of the three phases, with power on the X-axis and reactive power on the Y-axis. The different load patterns are clearly identifiable and should be distinctive enough to give sufficient results with the proposed leaning setup. (method: [Har92])

The fact that there are clear, distinct patterns even in this mixed data set makes it likely that deeper meaning can be extracted in later training.

The lines at the bottom of the x-axis show situations with a good power factor (very low rpower) on all three phases. The three stripes on phase 3 show three groups of low, medium and high power states with different power factors. These three groups also appear in phases 1 and 2, although in less clear patterns. A deeper interpretation would exceed the scope of this analysis.

If the plots had shown less distinct patterns, this might have been a reason for further data selection.

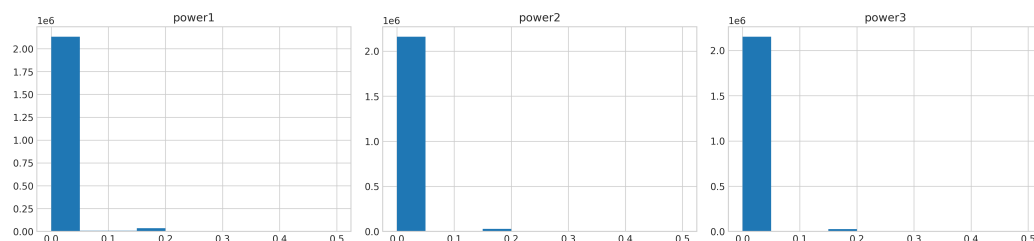


Figure 4.3: The histogram of power for the three phases shows that most values are small. This is to be expected as the night hours and many hours of the day are expected to have low consumption with little or no activity. The histogram shows the amount of values per class on the x-axis and value of the class on the y-axis.

The histograms of power, as shown in figure 4.3 *Dataset histogram of power* on page 26, reveal that most of the household appliances draw a small amount of power, and high power devices run for short durations. There are only a few high power devices in the household, such as the oven or cooking plate. These results are expected and match the households devices and their usage.

The scale of the X-axis of both graphs (rpower vs power, histogram power) shows that all values are below 50% of the scale. There is plenty of room for appliances with much higher power consumption.

The voltage is expected to stay within the regulatory specifications. The histograms in figure 4.4 *Dataset histogram of voltage* on page 27 show that all values stay well within these limits. The comparatively large fluctuation in voltage is

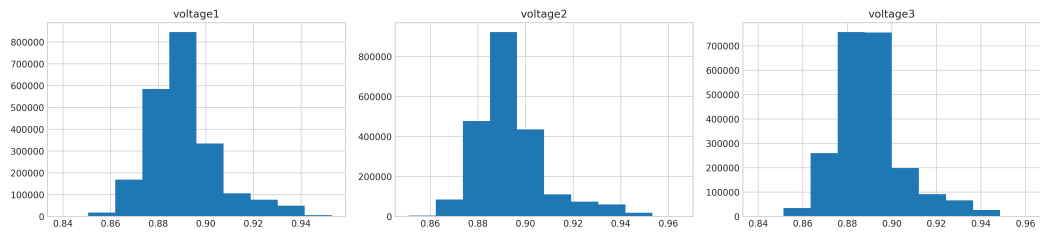


Figure 4.4: The histogram of the voltages of the three phases shows a very similar distribution. This is to be expected as the voltage has to stay within strict limits according to the regulations. The local solar grid may contribute some variation and data from other households might show less fluctuation over the day. The histogram shows the amount of values per class on the x-axis and value of the class on the y-axis.

most likely due to a local solar photovoltaic system installed on the roof. As local regenerative energy production is on the rise, it seems to be a perfect test setup.

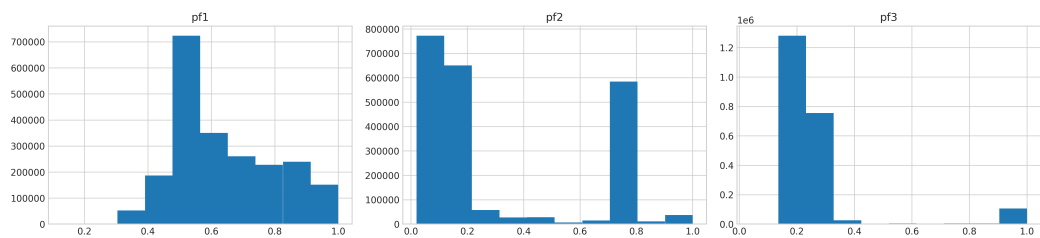


Figure 4.5: The histogram of the power factor of the three phases shows significant differences between the phases. This reflects the different characteristics of the appliances driven by the phases. This is promising regarding the expected results of the proposed learning system. The histogram shows the amount of values per class on the x-axis and value of the class on the y-axis.

The power factors, as shown in figure 4.5 *Dataset histogram of power factor* on page 27, show that phase 1 has a much better power factor than the other two phases. This may be because the appliances connected to this phase are of high power and have a good power factor. A kitchen with an oven, cooking plate, toaster and kettle could be a good candidate.

The visual analysis shows that the dataset seems to be good enough for the learning process and stays within the expected requirements described in the chapters and sections above.

Chapter 5

Implementation

The implementation of the learning system is based on Python, TensorFlow and Keras as this is the most common setup in the field. All work is done using Jupyter (in Jupyter Notebooks) running in Docker (section B.1 *Docker container* on page 73). It should be possible to reproduce the setup and results easily.

To get a good overview and visualisation of the training process, the Weights & Biases cloud service (wandb.ai) is used. It provides summaries and graphs comparable to other plotting solutions. As it is an online service, it is easier to use and observe the learning process if the training is computed remotely, as external GPUs are used. This fact and the good visual qualities of the service are the main reasons for using the tool.

The data is processed in pandas (framework), the usual frameworks (scipy, numpy, etc.) and the data is stored with pickle. The tools and frameworks used are state of the art and common in the field and are not described here.

The setup of data collection has been changed to reduce the amount of errors and data loss to the approach described in section 3.1 *Data collection* on page 16 with simple CSV files written to disk.

The training was done locally during an exploration phase and moved to the university's GPU cluster as soon as the resources of a local setup were exceeded.

The system is implemented to be modular, flexible and as reproducible and state of the art as possible.

5.1 Stage 1

Stage 1 is implemented like planned in section 3.2.1 *Stage 1 - single phase* on page 18.

Different auto-encoders are trained to create a latent space of size 3 or 4. As the input vector consists of $[W, VA, var, pf, V, A]$ (see figure 5.2 *Structure dataset for Stage 1* on page 30) and has a size of 6, and the latent space needs to be smaller than that to be reduced, the value has to stay below 6. The values of the input vector could be reduced to 3 if chosen correctly, as some values can be calculated from each other ($VA = V \cdot A, VA = W + var, pf = W/VA$).

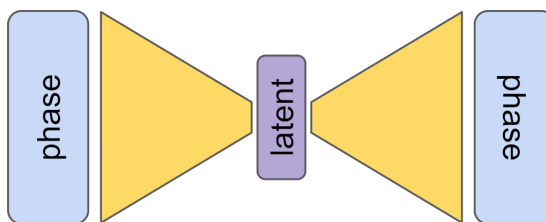


Figure 5.1: In Stage 1, the auto-encoder receives the (scaled) phase data to learn a good representation of a single data point of a single phase. The aim is to learn a good embedding, resistant to noise and other factors for the next stages.

Therefore, it seems likely that a latent space of size 3 would be sufficient to encode the input data.

The data of the phases like shown in figure 5.2 *Structure dataset for Stage 1* on page 30 are concatenated to form the new dataset consisting of all data of phase 1, followed by phase 2 and phase 3. The timestamps are discarded. The dataset is duplicated and random noise between 0 and 1 multiplied by $factor_{noise} = 0.002$ is added.

The final input dataset (X) is created by concatenating the noiseless and noisy data. The output dataset (Y) is created by concatenating the noiseless data

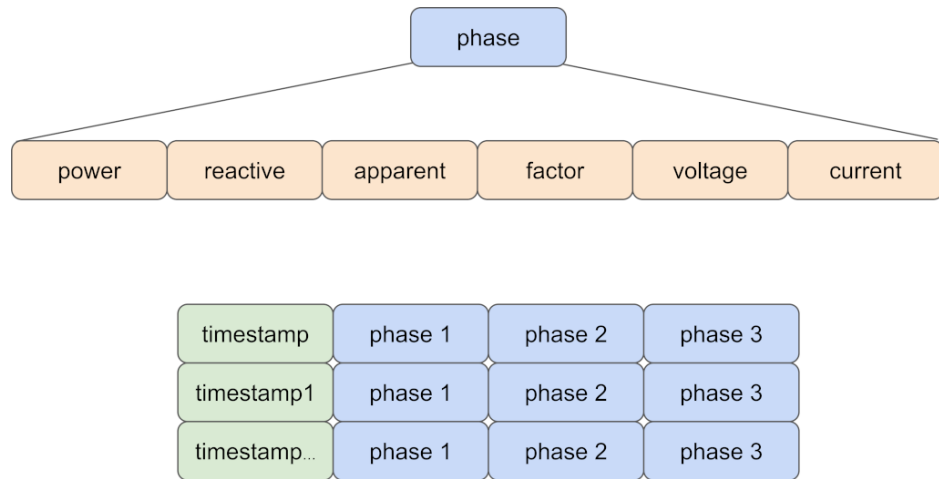


Figure 5.2: The data of each phase consists of power, reactive power, apparent power, voltage, current and power factor. The dataset for Stage 1 consists of timestamped rows and the data of the 3 phases.

twice. This way the model is trained to handle soft noise.

The test and training dataset are created by shuffling the data and split it 30% for testing and 70% for training.

The training is done on auto-encoders like shown in figure 5.1 *Auto-encoder Stage 1* on page 29. The auto-encoder is trained with the vectors (size 6) from the input dataset (some noisy) and the corresponding output is checked against the vector from the output dataset (free of noise). The error is calculated by the error function. The training aims to minimise the error.

An error significantly below 1% is accepted as a good result, especially as the noise will contribute to some margin of error. An error of less than 0.1% would be considered a problem, as this could indicate some kind of undetected overfitting, even if checked against the test data. If a training would not achieve a certain minimum error, it should show at least a constant decreasing error rate. This case could be acceptable as it could simply indicate a lack of computational power as the GPU resources granted to the project are limited. If the error rate oscillates or remains constant after a certain point, it is unlikely that the training would benefit from additional computational resources.

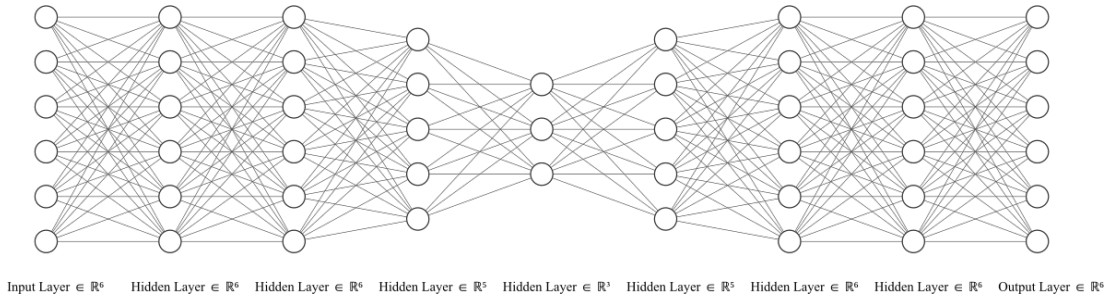


Figure 5.3: Illustration of an auto-encoder ('bigger') for Stage 1 with 5 dense layers, a latent space of 3 (middle), 6 input and output neurons. All layers are simple fully connected layers.

Two example network of latent space size 3 and latent space size 4 are shown in figure 5.3 *Auto-encoder Stage 1 latent space 3 example* on page 31 and figure 5.4 *Auto-encoder Stage 1 latent space 4 example* on page 31. These auto-encoders consist of varying amounts of simple dense layers of varying sizes, with the middle layers having a size equal to the targeted latent space (3 and 4).

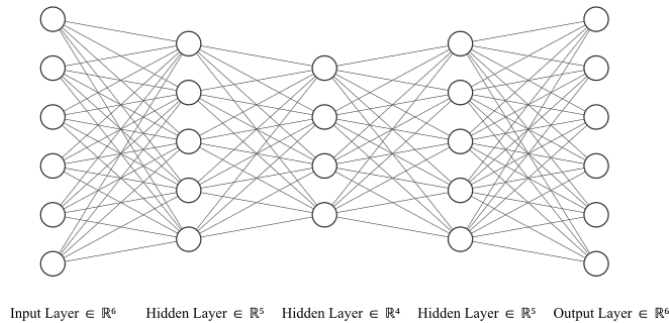


Figure 5.4: Illustration of an auto-encoder ('smaller') for Stage 1 with 3 dense layers, a latent space of 4 (middle), 6 input and output neurons. All layers are simple fully connected layers.

Due to limited computational resources or time (resources per time), training was done step by step and only the best models were modified and trained further. All models were trained using the default parameters of TensorFlow to keep the complexity as low as possible. This involved 10 epochs and the 'adam' optimiser, with 'mae' (mean absolute error) used for the loss function and 'accuracy' for the metric. These parameters are recommended as they are the most robust for data

with unknown properties (e.g. MAE vs. SGD, see below).

The training of Stage 1 was started with two simple auto-encoders with 3 layers (encoder) and latent space 3 and 4 (simple-latend3, simple-latend4).

Model: "simple-latend3"

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 6)]	0
dense (Dense)	(None, 6)	42
dense_1 (Dense)	(None, 5)	35
dense_2 (Dense)	(None, 3)	18
dense_3 (Dense)	(None, 5)	20
dense_4 (Dense)	(None, 6)	36

Model: "simple-latend4"

Layer (type)	Output Shape	Param #
input_5 (InputLayer)	[(None, 6)]	0
dense_10 (Dense)	(None, 6)	42
dense_11 (Dense)	(None, 5)	35
dense_12 (Dense)	(None, 4)	24
dense_13 (Dense)	(None, 5)	25
dense_14 (Dense)	(None, 6)	36

The goal was to check if a size 3 latent space is good enough or if a size 4 latent space leads to better results.

The error for latent space 3 was 0.346%, and for latent space 4, it was 4.507%. As the results indicated a preference for a size of 3, further parameters were modified based on a size of latent space 3. For the next training the optimiser was changed from 'mae' to stochastic gradient descent (SGD) to determine if it would perform better. Although it could only find local minima and performed poorly on the test data, which would lead to poor results, it was worth a try. The changes

resulted in worse outcomes, with an error of 0.394%.

Model: "simple-latend3-optimizer_sgd"

Layer (type)	Output Shape	Param #
input_3 (InputLayer)	[(None, 6)]	0
dense_5 (Dense)	(None, 6)	42
dense_6 (Dense)	(None, 5)	35
dense_7 (Dense)	(None, 3)	18
dense_8 (Dense)	(None, 5)	20
dense_9 (Dense)	(None, 6)	36

The activation function was changed from relu to sigmoid. The first idea was that relu would be good enough as the input data was scaled (between 0 and 1) and no negative data could appear. Relu with all negative values set to 0 would have been sufficient. This reduced the internal freedom of the decoder's activation function, which could indeed get all kinds of values from the encoder. With the activation function changed to sigmoid, two other auto-encoders were trained (simple-latend3-all_sigmoid, simple-latend4-all_sigmoid). The training showed much better results after this change with an error of 0.326% and 0.240%.

Model: "simple-latend3-all_sigmoid"

Layer (type)	Output Shape	Param #
input_7 (InputLayer)	[(None, 6)]	0
dense_15 (Dense)	(None, 6)	42
dense_16 (Dense)	(None, 5)	35
dense_17 (Dense)	(None, 3)	18
dense_18 (Dense)	(None, 5)	20
dense_19 (Dense)	(None, 6)	36

=====
=====

Model: "simple-latend4-all_sigmoid"

Layer (type)	Output Shape	Param #
--------------	--------------	---------

```

=====
input_9 (InputLayer)      [(None, 6)]           0
dense_20 (Dense)          (None, 6)             42
dense_21 (Dense)          (None, 5)             35
dense_22 (Dense)          (None, 4)             24
dense_23 (Dense)          (None, 5)             25
dense_24 (Dense)          (None, 6)             36

```

Two final trainings with a dropout layer and one with many more layers were tried. Both showed that neither more layers (4.506%) nor a dropout layer (2.484%) could improve the training.

Model: "layer5-latend3"

```

Layer (type)              Output Shape          Param #
=====
input_11 (InputLayer)    [(None, 6)]          0
dense_25 (Dense)         (None, 6)            42
dense_26 (Dense)         (None, 6)            42
dense_27 (Dense)         (None, 6)            42
dense_28 (Dense)         (None, 5)            35
dense_29 (Dense)         (None, 3)            18
dense_30 (Dense)         (None, 5)            20
dense_31 (Dense)         (None, 6)            36
dense_32 (Dense)         (None, 6)            42
dense_33 (Dense)         (None, 6)            42

```

```

=====
=====

```

Model: "layer5-latend3-dropout0_2"

```

Layer (type)              Output Shape          Param #
=====
input_13 (InputLayer)    [(None, 6)]          0
dense_34 (Dense)         (None, 6)            42
dense_35 (Dense)         (None, 6)            42
dropout (Dropout)        (None, 6)            0

```

dense_36 (Dense)	(None, 6)	42
dense_37 (Dense)	(None, 5)	35
dense_38 (Dense)	(None, 3)	18
dense_39 (Dense)	(None, 5)	20
dense_40 (Dense)	(None, 6)	36
dense_41 (Dense)	(None, 6)	42
dense_42 (Dense)	(None, 6)	42

The training graphs, as shown in figure 5.5 *Result of the training of Stage 1* on page 35, show that all the results are too close together to be visually distinguishable. The only distinctive graphs are those of layer5-latend3, simple-latend4 and layer5-latend3-dropout0_2 drawn on top of each other and the start of simple-latend3-optimizer_sgd where the slope is different (worse at first).

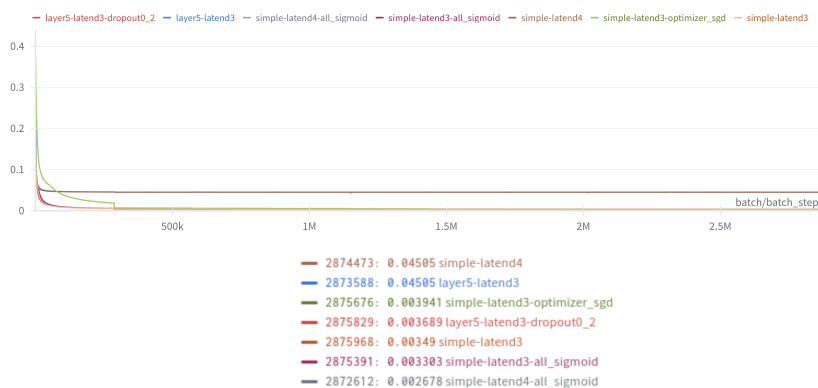


Figure 5.5: The charts shows the graph of the error value during training. The steepest drop of the error occurs in the first epoch. Due to the scale of the error graph it is used to control the training process mainly or to indicate problems.

All trainings showed very little improvement after 10 epochs and were expected to be good enough to prove the feasibility of the premise.

The simple small networks performed much better than the complicated or larger networks. A simple optimisation with MAE was able to outperform SGD. The more computationally intensive sigmoid outperformed relu by far (simple-latend4 vs. simple-latend4-all_sigmoid). As expected, the latent space of 3 seems to be sufficient. Nevertheless, an extra neuron (latent space 4) seems to be important

enough to give better results. It might bring the extra space to express information to handle the noise error.

name	error
simple-latend4-all_sigmoid	0.00240
simple-latend3-all_sigmoid	0.00326
simple-latend3	0.00346
simple-latend3-optimizer_sgd	0.00394
layer5-latend3-dropout0_2	0.02484
layer5-latend3	0.04506
simple-latend4	0.04507

The final results shown in the table above show a good result of simple-latend4-all_sigmoid (0.240%) followed by simple-latend3-all_sigmoid (0.326%).

5.2 Stage 2

The goal of Stage 2 is to find an embedding for behaviour over time, such as events. The approach is inspired by several papers:

The paper 'At the flick of a switch: Detecting and classifying unique electrical events on the residential power line (nominated for the best paper award)' [PRK⁺07] highlights the importance of identifying devices. Although the approach is based on high frequency events, it shows the information gain when switching events are taken into account.

The early work in 'Nonintrusive appliance load monitoring' [Har92] suggested the change in the ratio of active to reactive power as a variable source of information. Graphs in the paper show different load scenarios. According to the authors of [PRK⁺07] it is heavily based on the ideas of this paper.

Similar to this, the papers 'An efficient and inexpensive method for activity recognition within a smart home based on load signatures of appliances' [BGBB14] and 'Applying power meters for appliance recognition on the electric panel' [LLHJ10] focus on the change over time to gain the respective information.

All these papers show the importance of taking data over time into account to be able to identify events (points in time) and behaviour (change over time) among other factors.

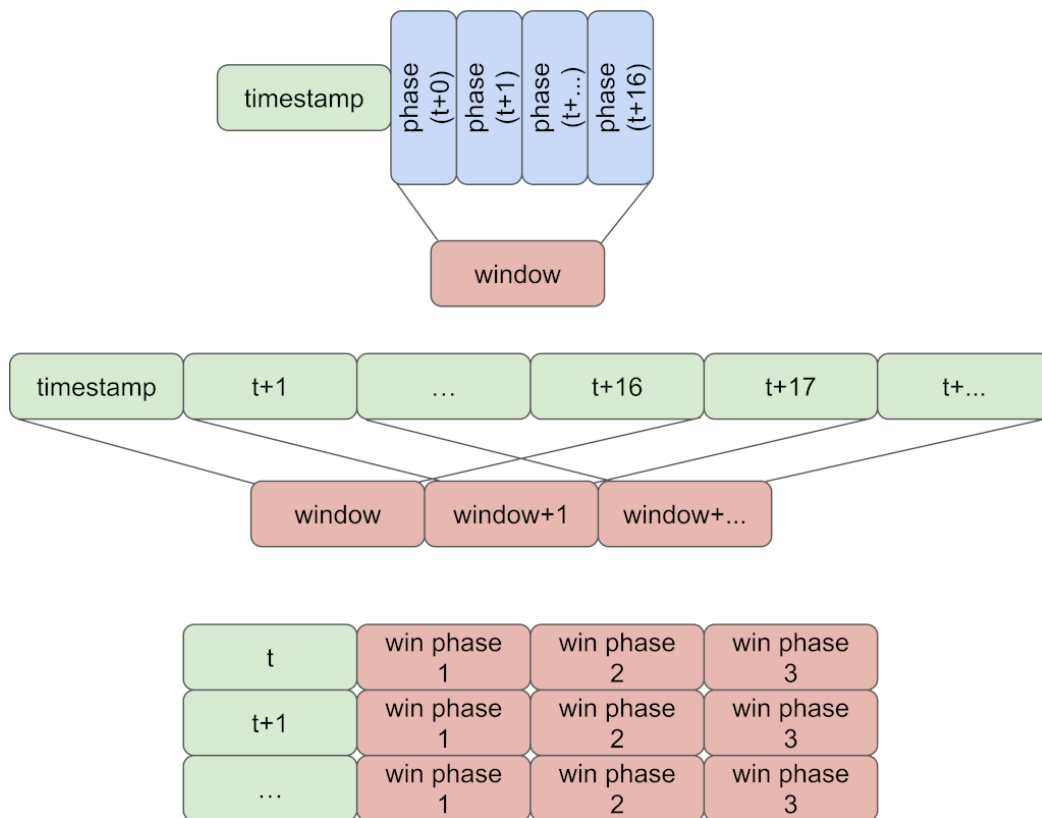


Figure 5.6: The data for Stage 2 is based on the encoded data from Stage 1, aggregated into a window of 16 consecutive values (top). The window is shifted through the Stage 1 dataset to cover all the data (middle). Each row represents a timestamp with the windows of the 3 phases (1 timestamp + 48 embeddings) (bottom). The windowing function takes care of time gaps in the original data.

Before the data can be used for Stage 2, it needs to be checked for gaps. Every time there is a power outage, reboot, software update or network outage, the data for that time frame is lost. If a time series is used for training, there would be an abrupt jump in the data. This could lead to unexpected training results. All data was checked for gaps and the dataset was cut into pieces of uninterrupted, consecutive time series of a minimum size ($>$ window size) to be usable. All fur-

ther processing was performed on these chunks and all chunks were combined into the final dataset. This way each isolated window of data contains a valid portion of the data independent of the other windows, making the final concatenation unproblematic.

The training dataset for this stage is created by using the encoder of the best models from Stage 1, namely 'simple-latend4-all_sigmoid' and 'simple-latend3-all_sigmoid'. Each data point from the raw data (size 6) is transformed to size 3 or 4.

After this, a window of size 16 is run over the data of each single phase, as shown in figure 5.6 *Structure dataset for Stage 2* on page 37. As the data was sampled every 2 seconds, the window covers 30 seconds. Each window has a size of 48 ($16 \cdot 3$) or 64 ($16 \cdot 4$).

The test and training datasets are created by shuffling the data and dividing it 30% for test and 70% for training.

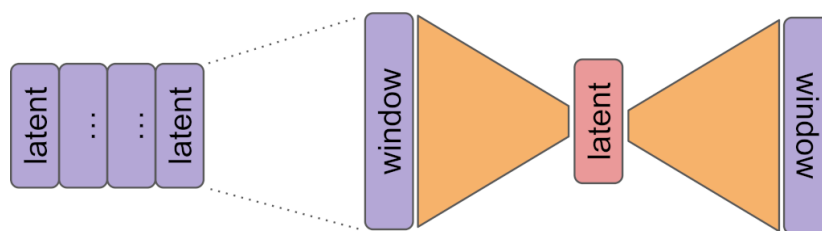


Figure 5.7: The Stage 2 auto-encoder is trained on a window of 16 data points. The aim is to learn a good representation of the 'last 16 data points'. The embedding should represent switching events (on, off) and behaviour of device operation (e.g. characteristics over time).

The auto-encoder of this stage, as shown in figure 5.7 *Auto-encoder Stage 2* on page 38, is trained on 48 and 64 inputs and outputs. The initial idea of adding noise to the data as shown in section 3.2.2 *Stage 2 - phase window* on page 18 was discarded as no additional gain was expected. The data was generated from the Stage 1 encoders and would be used in the same way in the final system. The dataset was already large enough to be a burden on the training process at this point, and a larger dataset was expected to be a possible cause of poorer results.

The training in Stage 1 was implemented iteratively and the parameters were spread a bit over different sets of variables.

```
1     modelConfig = {}
2     modelConfig = {
3         "project": "stage2",
4         "name": "window-in3-lat8-4layers",
5         "data": dataset3,
6         "wandb": {
7             "encoder": {
8                 "type": "sequential",
9                 "layers": [
10                    {
11                        "type": "dense",
12                        "size": 48,
13                        "activation": "relu"
14                    },
15                    ...
16                ]
17            },
18            "decoder": {
19                "type": "sequential",
20                "layers": [
21                    {
22                        "type": "dense",
23                        "size": 18,
24                        "activation": "sigmoid"
25                    },
26                    ...
27                ]
28            },
29            "optimizer": "adam",
30            "loss": "mae",
31            "metric": "accuracy",
32            "epochs": 10,
33            "shuffle": True
34        }
35     }
36
37     addModelToConfig(modelConfig)
38     trainingConfig["models"].append(modelConfig)
```

Figure 5.8: The models of Stage 2 are defined by a single JSON object.

In Stage 2, the training system has been rewritten and each model is defined by a single object in JSON notation, see figure 5.8 *Stage 2 model definition* on page 39

for details. The model is defined with its name, dataset, structure, optimiser, loss function, number of epochs and other parameters.

After definition, the configuration is augmented with the compiled model by 'addModelToConfig' and added to the configuration of the training set.

The newly implemented system led to much more organised training and opened up the possibility of defining a set of models and running them over the next few days. This allowed a much wider range of models to be tested in order to explore the best combination of parameters. The naming scheme was changed to reflect more of the parameters in an ordered fashion.

Due to an off-by-one-error (OBOE) [Wik24] introduced during implementation, all decoders miss their first layer, so the layer after the layer representing the latent space. This was discovered after the Stage 2 training had been completed. As the results of the training were good and the cost of re-training is very high, this bug was ignored and fixed in Stage 3.

Model: window-in3-lat8-4layers

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 48)]	0
dense (Dense)	(None, 48)	2352
dense_1 (Dense)	(None, 32)	1568
dense_2 (Dense)	(None, 18)	594
dense_3 (Dense)	(None, 8)	152
dense_4 (Dense)	(None, 32)	288
dense_5 (Dense)	(None, 48)	1584

The model in the table above shows this effect. The layer between 'dense_3' and 'dense_4' of size 18 is missing.

The interesting aspect of this is that the trainings must have benefited from the smaller decoders and finished a little faster. The fact that simpler, smaller networks outperform larger, more complex ones, as found in Stage 1, may have contributed as a positive factor, even if unintended.

A first series of networks was trained to find a reasonable range of parameters for the Stage 2 data. The parameters for the latent space (8, 12, 16) and the number of layers (6, 4) for the encoder and decoder were explored.

Model: window-in4-lat16-6layers

Layer (type)	Output Shape	Param #
input_23 (InputLayer)	[(None, 64)]	0
dense_86 (Dense)	(None, 64)	4160
dense_87 (Dense)	(None, 32)	2080
dense_88 (Dense)	(None, 32)	1056
dense_89 (Dense)	(None, 18)	594
dense_90 (Dense)	(None, 18)	342
dense_91 (Dense)	(None, 16)	304
dense_92 (Dense)	(None, 18)	306
dense_93 (Dense)	(None, 32)	608
dense_94 (Dense)	(None, 32)	1056
dense_95 (Dense)	(None, 64)	2112

The table above shows a larger auto-encoder (6 layers) with a larger latent space (size 16) and an input of latent space 4 from Stage 1.

The 12 models 'window-in3-lat8-4layers', 'window-in3-lat12-4layers', 'window-in3-lat16-4layers', 'window-in3-lat8-6layers', 'window-in3-lat12-6layers', 'window-in3-lat16-6layers', 'window-in4-lat8-4layers', 'window-in4-lat12-4layers', 'window-in4-lat16-4layers', 'window-in4-lat8-6layers', 'window-in4-lat12-6layers', 'window-in4-lat16-6layers' are constructed based on the above parameters and assembled for the first training (see details in Appendix section B.2.2 *Stage 2* on page 76).

The results shown in the table below indicate comparable results to those seen in Stage 1. A network of 4 layers in the encoder seems to be complex enough to cover the task, and added complexity in the form of more layers degrades the quality of the results. The larger ones show better results with smaller latent spaces. As the error settled after a few epochs during training, shown in figure 5.9 *Result of the training of Stage 2* on page 42, longer training for the deeper models does not

seem to bring any advantage.



Figure 5.9: The graphs of Stage 2 show no unexpected behaviour. The convolutional networks show a slightly different learning characteristic in the first epoch. The largest convolutional network spikes with its worst performance.

name	error
window-in4-lat12-4layers	0.003294
window-in4-lat16-4layers	0.003315
window-in4-lat8-4layers	0.003819
window-in3-lat16-4layers	0.004037
window-in3-lat12-4layers	0.004153
window-in3-lat8-4layers	0.004283
window-in4-lat8-6layers	0.004344
window-in4-lat16-6layers	0.004619
window-in4-lat12-6layers	0.004766
window-in3-lat8-6layers	0.005658
window-in3-lat16-6layers	0.006478
window-in3-lat12-6layers	0.006686

The less complex models with fewer layers perform better with larger latent spaces, and a Stage 1 input of 4 outperforms Stage 1 latent space 3.

Interestingly, latent space 12 from Stage 2 outperforms the larger latent space of 16 with an input of latent space 4 from Stage 1.

As the windowed data resembles a timeline of values, there is some kind of regional meaning in this data. The n -th, n -th + 1, n -th + 2 and so on vector with its

components (Stage 1 latent space) encodes a semantic meaning of a sequence of events or values. If a neural network receives inputs as a flat vector, neighbouring elements in the data may not end up in neighbouring neurons in the network.

Convolutional Neural Networks (CNN), as explained in more detail in section 2.3 *Convolutional neural networks* on page 14, provide a method of learning this structural aspect of the data during training.

Driven by this idea, a new set of networks was defined to test whether a more complex CNN could add quality to the existing results. Previous results show that increased complexity usually degrades the quality of the results.

Based on the results, the latent space of 8 was chosen to explore whether the approach would add value. The smaller latent space was chosen to explore how resource intensive the training would be. Local test trainings with 1000 data points showed a much longer training time beforehand.

The dataset was reshaped to (16, 4, 1) '16 consecutive embeddings of latent space size 4 with 1 value in the vector component' to reflect the original window configuration. Two versions with few layers ('window-in4-lat8-conv-mini') and more layers ('window-in4-lat8-conv-big') were tried. The final saving command did not work and crashed with an error. So the results were saved to the cloud, but the models were lost. There would have been a chance to recover some part of the last training in the list ('window-in4-lat8-conv-mini'), but this was delayed at this point and later dropped altogether. The setup has been fixed to avoid such problems.

name	error
window-in4-lat8-conv-mini	0.003096
window-in4-lat8-conv-big	0.110749

The results of the larger model show a much worse performance compared to the simple dense layer models. The 'mini' version outperformed all previous attempts by some margin, despite the fact that the latent space of 8 performed worse compared to 12 or 16 in the earlier training.

Model: window-in4-lat12-conv-mini

Layer (type)	Output Shape	Param #
input_33 (InputLayer)	[(None, 16, 4, 1)]	0
conv2d_10 (Conv2D)	(None, 16, 4, 64)	640
max_pooling2d_5 (MaxPooling 2D)	(None, 8, 2, 64)	0
flatten_5 (Flatten)	(None, 1024)	0
dense_140 (Dense)	(None, 128)	131200
dense_141 (Dense)	(None, 12)	1548
dense_142 (Dense)	(None, 128)	1664
dense_143 (Dense)	(None, 1024)	132096
reshape_5 (Reshape)	(None, 16, 4, 16)	0
conv2d_transpose_5 (Conv2DT ranspose)	(None, 16, 4, 64)	9280
conv2d_11 (Conv2D)	(None, 16, 4, 1)	577

Model: window-in4-lat16-conv-mini

Layer (type)	Output Shape	Param #
input_35 (InputLayer)	[(None, 16, 4, 1)]	0
conv2d_12 (Conv2D)	(None, 16, 4, 64)	640
max_pooling2d_6 (MaxPooling 2D)	(None, 8, 2, 64)	0
flatten_6 (Flatten)	(None, 1024)	0
dense_144 (Dense)	(None, 128)	131200
dense_145 (Dense)	(None, 16)	2064
dense_146 (Dense)	(None, 128)	2176
dense_147 (Dense)	(None, 1024)	132096
reshape_6 (Reshape)	(None, 16, 4, 16)	0
conv2d_transpose_6 (Conv2DT ranspose)	(None, 16, 4, 64)	9280

ranspose)

conv2d_13 (Conv2D) (None, 16, 4, 1) 577

As the advantage of the new approach became clear, a new set of networks was defined, all based on 'window-in4-lat8-conv-mini', but with the original latent spaces of 12 and 16, as they gave better results in the earlier setup.

The results outperform all previous training. The larger latent space of 16 seems to give a slight advantage, although the difference is small. The results of 0.2779% seemed good enough to start the training of Stage 3.

name	error
window-in4-lat16-conv-mini	0.002779
window-in4-lat12-conv-mini	0.002893

Based on these results, a final setup with a latent space of 9 was trained to try a 3D convolution at Stage 3. The vector of size 9 could be reshaped to (3, 3) 'a square of 3 rows and 3 columns' and would represent a single phase. As Stage 3 combines all three phases, the input could be shaped as (3, 3, 3) 'a cube of 3 planes with 3 rows and 3 columns'. This could have some advantage as all the phases and all the values of the phases could have some regional aspect or be 'neighbours' as described above.

Model: window-in4-lat9-conv-mini

Layer (type)	Output Shape	Param #
=====		
input_37 (InputLayer)	[(None, 16, 4, 1)]	0
conv2d_14 (Conv2D)	(None, 16, 4, 64)	640
max_pooling2d_7 (MaxPooling 2D)	(None, 8, 2, 64)	0
flatten_7 (Flatten)	(None, 1024)	0
dense_148 (Dense)	(None, 128)	131200
dense_149 (Dense)	(None, 9)	1161
dense_150 (Dense)	(None, 128)	1280
dense_151 (Dense)	(None, 1024)	132096
reshape_7 (Reshape)	(None, 16, 4, 16)	0


```
conv2d_transpose_7 (Conv2DT (None, 16, 4, 64)          9280
ranspose)
conv2d_15 (Conv2D)          (None, 16, 4, 1)          577
```

The network is identical except for the latent space size of 9, and the results show an almost identical error to latent space 8, which was expected to some extent.

name	error
window-in4-lat16-conv-mini	0.002779
window-in4-lat12-conv-mini	0.002893
window-in4-lat8-conv-mini	0.003096
window-in4-lat9-conv-mini	0.003099
window-in4-lat12-4layers	0.003294
window-in4-lat16-4layers	0.003315
window-in4-lat8-4layers	0.003819
window-in3-lat16-4layers	0.004037
window-in3-lat12-4layers	0.004153
window-in3-lat8-4layers	0.004283
window-in4-lat8-6layers	0.004344
window-in4-lat16-6layers	0.004619
window-in4-lat12-6layers	0.004766
window-in3-lat8-6layers	0.005658
window-in3-lat16-6layers	0.006478
window-in3-lat12-6layers	0.006686
window-in4-lat8-conv-big	0.110749

The final result as shown above illustrates the clear advantage of simple networks of low complexity and the advantage of convolutional layers. A latent space of 16 seems to give a small advantage compared to 12.

The graphs of the training (see figure 5.9 *Result of the training of Stage 2* on page 42) do not show any surprising aspects. The only interesting fact might be that the convolutional networks show a different error graph in the first epoch. The single graph is the one with its bad performance and no sign of significant improvement over the course of training after the first epochs. It seems that the

error correction after the first epoch brings the important change for the CNNs to perform much better.

5.3 Stage 3

Stage 1 targets the encoding of a single data point. Stage 2 targets the encoding of time series aspects such as events or changes in behaviour. Stage 3 aims at a general encoding of states of a household and its devices.

The main idea is to abstract devices from their specific or current setup in a household by making the phases commutative.

A good example could be an electric kettle that is used in the kitchen and then moved to the office. The device remains the same with all its built-in properties, but could appear at a different phase with different parameters. This could be a lower power consumption due to longer wires (seen from the junction box) due to higher resistance and therefore higher voltage drop. This could result in a slightly different power factor due to slightly different electrical properties or interaction with other nearby equipment.

The aim of embedding is to make the representation of an appliance as similar as possible, despite the phase configuration in a household.

High power appliances such as boilers, ovens or cookers connected to the three phases would be displayed similarly regardless of their connection configuration.

The dataset for Stage 3 was created from scratch. The datasets from Stage 1 and Stage 2 were optimised to focus on a single phase and the computation was saved whenever possible. In this way, the data from the three phases and the timestamps were not present in this dataset, but most of the data was concatenated into one long stream of single phase data.

The best models from Stage 1 were used to create a dataset with all three phases including the timestamp and the time gaps were addressed as before (chunks).

The window function of Stage 2 was then used to create the windows as before. With the changed shape of the data and the different memory layout there was no

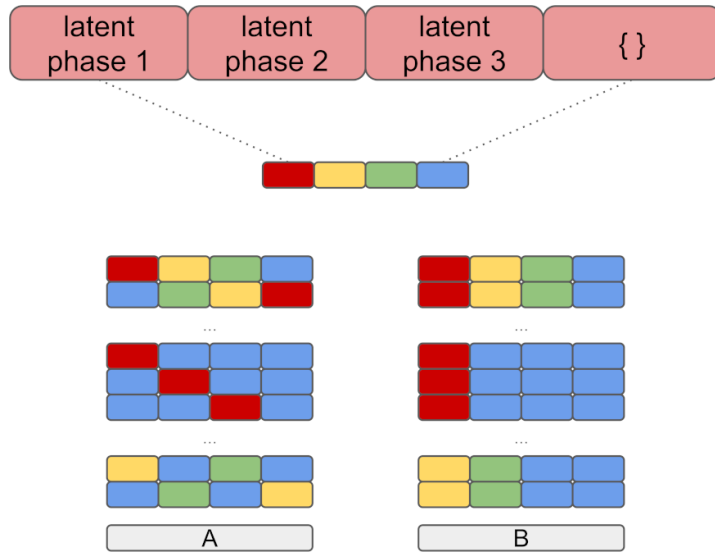


Figure 5.10: The input data (A) of the dataset for Stage 3 is created by permuting the data of the phases and the empty set to create all possible configurations. The output data (B) of the dataset is created by sorting the rows of the input set in the order phase 1, 2, 3 and empty set. This way the phase data is trained to be commutative.

easy way to use the framework’s built-in functions like numpy as before without modifications to target the data in its non-linear memory.

It turned out that simple for-loops and computations without the built-in functions, which seem to make heavy use of hardware acceleration, lead to computations with a runtime and memory consumption far beyond the scope of this project.

With the help of the community, a solution was found to get the values of a windowing function and reshape the data to create a windowed dataset of all phases including the timestamp around $10^3 - 10^4$ times faster and with a much smaller memory footprint print [Sta24]. This allowed data in fragmented memory to be computed in hours to days. The windowed dataset was used and encoded with the best models from Stage 2.

To create a dataset with all possible variants of the phase configuration, including single phase and two phase households, the data was permuted, including the empty set to represent 'no data', as shown in figure 3.2.3 *Stage 3 - all phases*

invariant on page 19. The input data is permuted while the output (expected) data remains in a reliable and fixed defined order. The encoding of different phase configurations should tend to be represented similarly if the input data remains the same but is shifted to different input neurons.

The output dataset became very large (around 100GB) due to the nature of the permutation. As the resources granted to this project are not sufficient to target such immense training setups, a subset was used for training.

The dataset for Stage 3 is based only on the latent space of size 9 from Stage 2, as it allows to test the idea of the reshaped input of shape (3, 3, 3) to be used with convolutional layers. The amount of data is reduced to 10% and 1% by randomly sampling from the large dataset. Each test and training dataset is created by shuffling the data and dividing it 30% for test and 70% for training.

The training system was modified, refactored and improved again to be able to define parameters such as the amount of input data (percentage) or the size of the latent space of Stage 1, Stage 2 and Stage 3 and to select the input dataset accordingly. The name of the model and the size of the latent space were derived from this and additional checks were applied. The input data is checked beforehand to see if it fits the parameters.

All of this was implemented to target the longer runtime of the training. In this way the exploration and research was done during the day and the training sessions were shifted to the nights or ran reliably over days for very long runs.

As in the previous stages, an auto-encoder as shown in figure 5.11 *Auto-encoder Stage 3* on page 50 is used. This time the input and output data are not the same as in the previous stages, but semantically the same in the form of information on a different phase.

The training of this stage mainly explores the parameters instead of creating sufficient artefacts in the form of models as in the previous stages. The potentially complex nature of the data combined with the heavily reduced set of training data is expected to strongly influence the results in the form of high error values.

Starting from a latent space of 32, the training of the first three networks

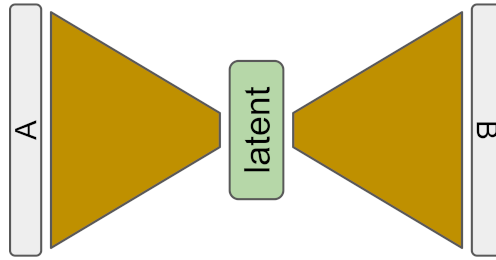


Figure 5.11: The Stage 3 auto-encoder is trained to find a representation of devices independent of the phase. The colours represent the data of the 3 phases and a placeholder 'none'. The training dataset is created by permuted phase data to represent all variations of possible phase configurations in a real household. The aim is to have a representation of devices independent of phase, so that e.g. a toaster connected to phase 1 would be represented very similarly if connected to phase 2 or 3.

'...lat32-0_01_percent-conv-bigger', '...lat32-0_1_percent-conv' and '...lat32-0_01_percent-conv' explores the effect of the smallest (1%) to the largest (10%) dataset and whether the training could benefit from more layers ('...-bigger'). The networks use a 3D convolution of the shape (3, 3, 3) discussed earlier.

name	error
...lat32-0_1_percent-conv	0.03274
...lat32-0_01_percent-conv	0.04007
...lat32-0_01_percent-conv-bigger	0.17401

The table above shows that there is no improvement with more layers and that training with more data gives much better results.

With the next network 'lat64-0_01_percent-conv-bigger' a larger latent space of 64 was explored to see if the original size of 32 might have been too sharp. The much worse result of 0.27319 (error of about 27%) shows that the larger latent size did not improve the training results.

To compare the results of the convolutional networks to a much simpler dense layered network approach, a simple setup with three layers in the encoder was trained with the small 1% dataset for a short runtime, namely '...lat32-0_01_percent-dense' (see all networks in appendix figure B.2.3 *Stage 3* on

page 85). It performed better than the convolutional network with an error rate of 0.03359 (3.3%) and reached an error rate comparable to training with 10 times more training data (10%) of '...lat32-0_1_percent-conv' with 3.2%.

To explore whether the dense approach could be tuned to produce better results than the convolutional approach, a set of networks was created for the next training session.

The networks 'lat32-0_01_percent-dense-bigger' and '...lat32-0_01_percent-dense-epoch100-bigger' with more layers were trained on 1% of the data, one trained with 100 epochs. With '...lat32-0_01_percent-dense-epoch100' and '...lat32-0_1_percent-dense-epoch20' a dense network version with 3 layers like earlier was trained on 1% of the data and 100 epochs and 10% and 20 epochs which was the most resource intensive run so far.

name	error
...lat32-0_1_percent-dense-epoch20	0.02868
...lat32-0_01_percent-dense-epoch100	0.02874
...lat32-0_01_percent-dense-epoch100-bigger	0.02999
...lat32-0_01_percent-dense-bigger	0.03646

The results in the table above show again that the networks with fewer layers perform better and that longer training in the form of more epochs improves the results significantly (previous error of 3.3%). The effect of more training data as shown in 'lat32-0_1_percent-dense-epoch20' is an improvement, but surprisingly less than initially expected.

As longer training improved the deeper network ('...-bigger'), wider layers with more neurons were trained with '...lat32-0_01_percent-dense-epoch100-bigger-wider' in 100 epochs to see if the deeper network could produce better results if more neurons were provided and sufficient training would compensate for the more complex structure.

With a result of 0.02972 (2.97%) compared to 0.02999 (2.99%), there seemed to be no significant improvement by investigating deeper.

To test smaller latent spaces, the best performing simple dense network was changed to a latent space of 24, 16, 12 and 8.

name	error
...lat32-0_01_percent-dense	0.03359
...lat16-0_01_percent-dense	0.03385
...lat12-0_01_percent-dense	0.03462
...lat24-0_01_percent-dense	0.03584
...lat8-0_01_percent-dense	0.03743

The results showed no major differences or improvements. Much longer training with much more data would probably show a clearer difference. It is interesting to note that the smaller latent spaces of 16 and 12 seem to outperform the latent space of 24, but not 32. Further training might have been interesting, but this would have been beyond the resources of this work.

The graphs of the training show no significant surprising behaviour and are attached in the appendix section B.2.3.1 *Graphs* on page 85.

name	error
...lat32-0_1_percent-dense-epoch20	0.02868
...lat32-0_01_percent-dense-epoch100	0.02874
...lat32-0_01_percent-dense-epoch100-bigger-wider	0.02972
...lat32-0_01_percent-dense-epoch100-bigger	0.02999
...lat32-0_1_percent-conv	0.03274
...lat32-0_01_percent-dense	0.03359
...lat16-0_01_percent-dense	0.03385
...lat12-0_01_percent-dense	0.03462
...lat24-0_01_percent-dense	0.03584
...lat32-0_01_percent-dense-bigger	0.03646
...lat8-0_01_percent-dense	0.03743
...lat32-0_01_percent-conv	0.04007
...lat32-0_01_percent-conv-bigger	0.17401
...lat64-0_01_percent-conv-bigger	0.27319

The final results clearly show the lack of sufficient resources to train the networks to a point where the error would drop below an acceptable level of 0.5% or much less, as in the previous stages. The top-ranked models were those with the longest training times and the most data.

Interestingly, the convolutional approach was not as effective as in the previous stage and the simplest dense network outperformed it by far. The estimated effect of neighbouring components of the input vector seems to have less impact than expected.

The most promising models are the dense models with less complexity and latent spaces 32, 16, 12 and 24, which if trained on 100% of the dataset should achieve results similar to the previous stages.

Nevertheless, an error of around 2.8% is good enough to convert the data from Stage 2 in order to evaluate the quality of the transformation.

5.4 Final assembly

The big picture of the final system, as shown in figure 5.12 *Encoder final version* on page 54, illustrates the complete setup of Stage 1, 2 and 3. A system takes a sample every 2 seconds. The sample is scaled according to the scale given in section 4.3 *Data ranges* on page 23. Each phase vector of the measurement is transformed with a Stage 1 model and added to the sliding buffer (array) of that phase for Stage 2. Systems with less than three phases are configured to encode only the necessary phases. Each buffer is transformed by a Stage 2 model into a single vector. The vectors are concatenated and transformed by a Stage 3 model to produce the final result for the input sample.

Using a Stage 3 model of latent space 32, each sample of size 18 is transformed into an embedded vector of size 32. The size of the setup is small enough to run on a single board computer (SBC) such as a Raspberry Pi, or on a microcontroller such as an ESP32 or Raspberry Pi Pico if reduced using e.g. TensorFlow Lite (tflite). The additional memory requirements of the last 16 samples are minimal. In this way, a code could be generated inside an embedded device such as the

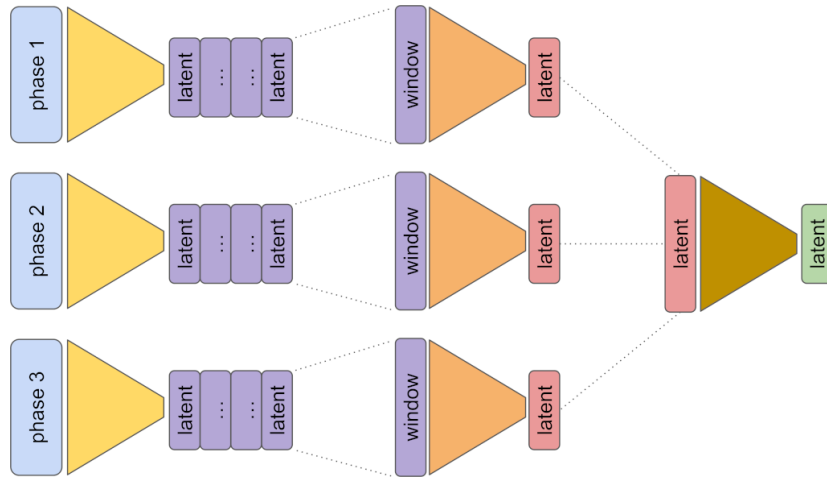


Figure 5.12: The final version of the encoder consists of the Stage 1, Stage 2 and Stage 3 encoders. The individual data from each phase of the power meter is encoded by the Stage 1 encoder (left). It is fed into the 'last 16 values array' window (centre). This window is encoded by the Stage 2 encoder. These encodings are combined to represent all three phases and encoded with the Stage 3 encoder to obtain the final phase invariant encoding (right).

Shelly 3EM without any external system.

5.5 Real world training and use

In order to implement a usable system for finding an embedding trained on data from thousands of households, an approach needs to be found which is open and easy to use for a community willing to contribute data and computing resources. It would be ideal to run the computation for each household locally on a GPU and share the results to collectively improve the embedding.

The new Web Neural Network API (webNN) in its W3C candidate recommendation draft from 22.02.2024 [W3C24a] makes it possible to train and use neural network models in a browser natively and with hardware acceleration. The work 'WebNN: A Distributed Framework for Deep Learning' [GCZ18] describes a similar earlier approach with the same goals. The webNN API seems to be an ideal solution as browsers like Google Chrome already support it in an early stage.

Training can be done locally in the browser on the local GPU.

Earlier APIs like WebCL (computation)[Khr14], WebGL (graphics rendering)[Khr23], WebGPU (GPU access) [W3C24b] and webASM (universal assembler) [Web24] are an example of the possible impact of such an API and the effect of easy access to resources in the browser.

With a framework like tensorflow.js, defining training setups becomes easy and independent of the browser or current state of the APIs. If better support for GPUs, local machine learning accelerators such as Jetson Nano, Google Coral or ML cores of modern CPUs becomes available, the framework would most likely support it at some point without changing the setup of the proposed system.

To make it possible to donate data to the community without risking bad privacy implications, anamorphic cryptography methods could be used to work with neural networks on encrypted data, as described in 'Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy' [GBDL⁺16].

The improvement of an individual neural network would have to be possible collectively, distributed and based on a zero-trust model to protect individuals from harm.

A simple solution would be a blockchain to prove the work and protect the results. Starting with an initial block from a trained neural network, each new block would be created by the work of the community in epochs.

A new epoch would be announced by a central project entity, a coordinator. The epoch consists of a dataset, optionally the network of the last epoch and a threshold. The dataset is divided into a training and a test set by a commonly agreed algorithm. The proof of work is done in form of the next block.

To successfully commit a new block, the network is trained locally with the block's dataset until the error is below the threshold. The local data can be used to improve the training. Once the block is computed, it is announced to other peers.

The peers check the threshold with their local test data and then add the block

to the list of candidates for the next block. Blocks with better error values are ranked higher.

Epochs could start at midnight of each new day. All candidates from the previous epochs are tested against the dataset of the new epoch to select the winning block and its network. The blockchain is defined as the longest chain of valid blocks.

The community may split at some point and split temporally into different chains, but at some point one of the branches will fall behind and the collective work will be consolidated at that point, including the central project entity. The current best block would define the best network to train on, and the central project entity would not need to provide the current best model, just the data and the threshold, if not iteratively calculated by the error gradient of the last few blocks.

Chapter 6

Evaluation

The work started with the idea to find a general encoding of a local power grid with unlabelled data. If such an embedding would be found it should be possible to encode current data from a power meter with such an encoding and decode data encoded this way.

Some information would get lost in the process as the encoding and decoding would be lossy. The better the method the less error is expected in this process.

The auto-encoders target the very aspect to minimise the error while be forced to bring the information from the input side to the output side with the constrained latent space in the middle so to speak.

The stages focusing on specific aspects were able to find such embeddings. The error values of the auto-encoders became low. A low error means a good reconstruction and thus a good representation in the latent space.

The final embedding represents a certain state in time of the household. All devices with their states are encoded in it and it should be possible to use it with labelled data to train a system to disaggregated devices which would be far out of scope of this work.

An early approach to visualise the data was to plot the data as a waterfall plot. Each row represents a single three phase input and the columns represent the

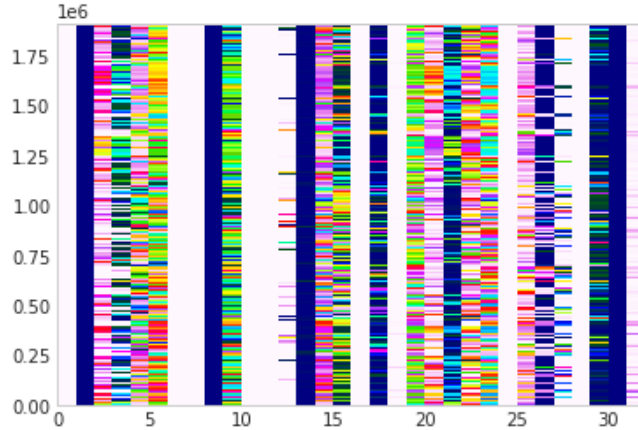


Figure 6.1: The waterfall plot illustrates the projects' data encoded in colours. While some components show minimal variation (plain columns), others show intense changes in value (colourful columns).

component of the latent space vector of size 32. This method provides minimal insight as the meaning of the vector component remains obscure. The only thing we can take away from this is that of the household is encoded using components with a high degree of variation, represented by the colourful columns, and components with a low degree of variation, represented by the plain columns. The plain columns could encode some kind of scale, as most of the input data was low energy consumption data.

As the data has a high dimensionality, making interpretation much more difficult, Principal Component Analysis (PCA) was used to project it into a lower dimensional space.

The idea is similar to a shadow cast by an object. A three-dimensional (3D) object is projected onto a two-dimensional (2D) plane. This projection enables the observer to view the object in 2D, despite the lack of features due to this transformation.

The dataset containing 32-dimensional vectors is projected onto a 3D space in the hope of finding cluster structures. These would indicate some kind of semantic grouping in the data. The recurring patterns in the electrical grid of a household

cloud form clusters and paths throughout the day. If present, these patterns should differ from the raw data.

Three methods were used to project the high dimensional data into a lower dimensional space: Principal Component Analysis (PCA), t-distributed Stochastic Neighbour Embedding (TSNE) and Isometric Mapping (ISO), as they were easy to use and available in the frameworks used. The methods were used as a tool and were not investigated any further.

The calculation of PCA was comparably fast with a runtime of minutes to hours, TSNE acceptable in the order of hours to days, but ISO crashed with an error. The error message indicated that the calculation would have required '26.7 TiB for an array with shape (1915968, 1915968) and data type float64', which gives an idea of the internal data structure. To avoid further crashes, the calculation was limited to the first 10000 values, which still took a few days.

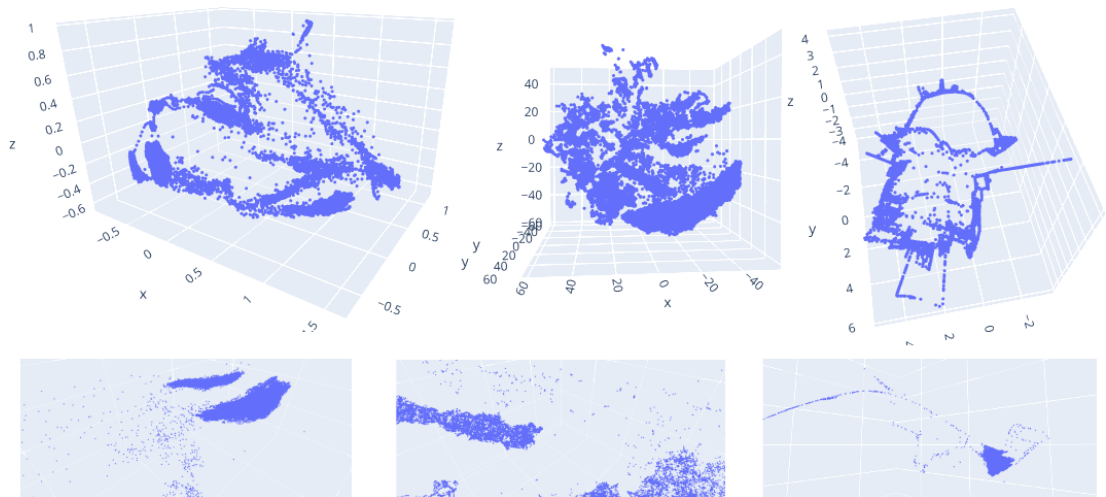


Figure 6.2: The high dimensional data is projected to a lower dimensional space in the same way that the shadow of a 3D object projects its shape to a 2D plane. The projection methods used, from left to right, are Principal Component Analysis (PCA), t-distributed Stochastic Neighbour Embedding (TSNE) and Isometric Mapping (ISO). The bottom row shows the solid structure of the clusters when zoomed in.

The results illustrated in figure 6.2 *Result illustrated with reduced dimensions* on page 59 show the clusters mentioned above with a path-like shape. The bottom

row shows the solid structure of the clusters when zoomed in, emphasising the impression of the dense semantic character of the embedded data.

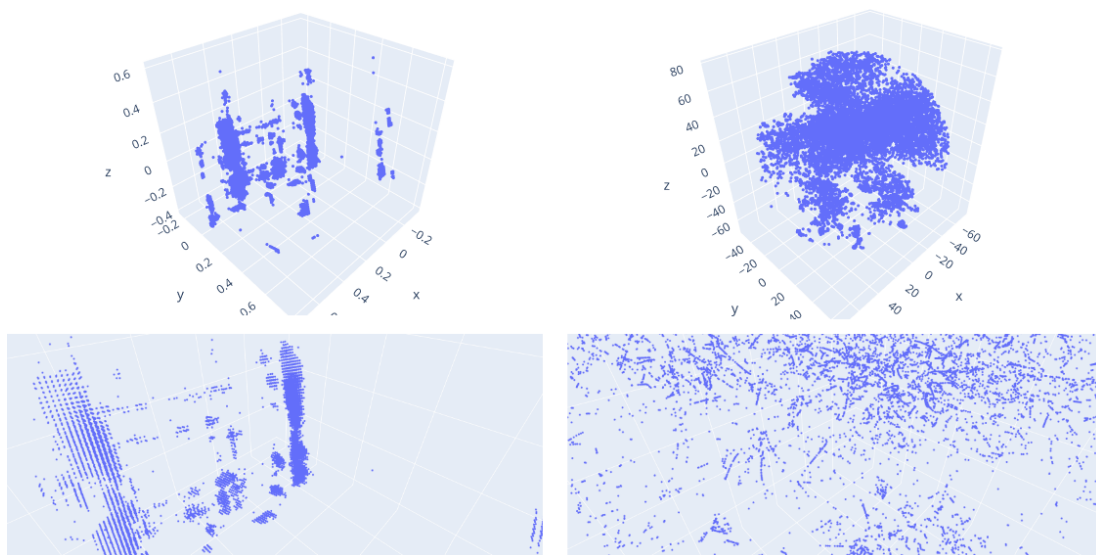


Figure 6.3: The raw (scaled) input data plotted with PCA shows some grid structure. The two clusters appear much less rich compared to the computed results. Zooming in on the bottom row reveals the sparse structure of the clusters.

The scaled raw dataset was transformed using PCA and TSNE. Several attempts to transform the dataset using ISO crashed reliably after a few days, leading to the exclusion of this projection.

The projection in figure 6.3 *Raw data illustrated with reduced dimensions* on page 60 appears to show stripe-like clusters in PCA on the left and cloud-like clusters in TSNE on the right at first. The bottom row shows a sparse point cloud when zoomed in. When the 3D model is moved and rotated, it is easy to see that the clouds are only a visual effect, as the points are far apart and the impression is mainly due to the overlapping of the points.

The projection suggests that the embedding is effective even with the limited training of Stage 3.

The results are good especially for unlabelled data and unsupervised training.

In a final test setup, a kettle was used to make a cup of tea to get a set of

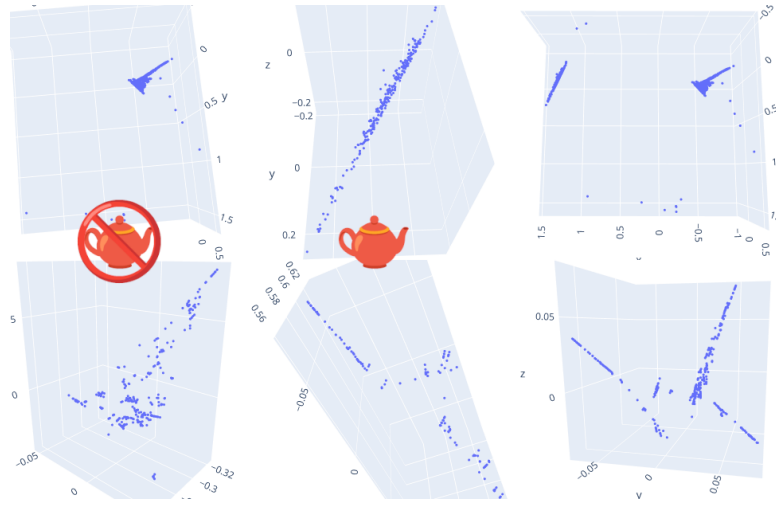


Figure 6.4: The figure displays the projection of the high dimensional data with the embedded data in the top row and the raw data in the bottom row. The first column shows the household before the kettle was switched on, the center row shows the isolated data of the household while the kettle was switched on, and the right row shows both data together. The cluster is distinguishable in the top row but not in the bottom row.

data. Reference data was collected before the kettle was switched on. The data was then encoded with the final setup and projected into the 3D space shown in figure 6.4 *Example to show operation with a kettle* on page 61. The top row displays the embedded data, while the bottom row displays the raw data. The left column shows the data collected shortly before the kettle was switched on, the center column shows the data collected while the kettle was in operation, and the right column shows all the data together.

The dense clusters in the top row are easily identifiable and can be located in the right column of the combined data. The household cluster is on the right (triangle shape) and the kettle appears on the left (stripe).

In contrast, the bottom row exhibits sparse, fragmented, and scattered data points when compared to the embedded version of the data. This result is very satisfying as the training has not yet focused on disaggregation or embedding of isolated devices, but already shows the good quality of the embedding.

Chapter 7

Conclusion and future work

The chapter figure 6 *Evaluation* on page 57 indicates a good embedding of the local power grid. The way the embedding was found provides a good basis for an embedding of individual appliances and opens a good way for disaggregation similar to the work in 'Real-time recognition and profiling of appliances through a single electricity sensor'[RNSO10], 'Activity analysis based on low sample rate smart meters'[CDW⁺11] and 'A practical solution for non-intrusive type II load monitoring based on deep learning and post-processing'[KDW⁺19]. This should be particularly true for Type II devices such as a washing machine, with its changing load patterns over the course of a wash cycle, which should benefit from embedding as Stage 2 focuses on event-like aspects over time.

With more data from different households and distributed computational power as described in section 5.5 *Real world training and use* on page 54 a even better embedding could be found. The existing embedding could be used to create a new embedding for single devices and disaggregated devices at the same time.

Google's Facenet [SKP15] is a good example of a good method for reinforcing the learning of a high-dimensional Euclidean space with the desired properties. In the training the embedding, a 128-dimensional vector of a picture of a single face, is created with the network. Another image of a single face is encoded this way. If the face in the pictures belongs to the same person, the distance between the

two vectors is decreased (moved closer together) and increased (moved apart) by modifying the network accordingly. This way the network is trained to represent the faces in a high-dimensional Euclidean space, with the properties that images of the face of the same person tend to form clusters which are distinct from clusters of other people.

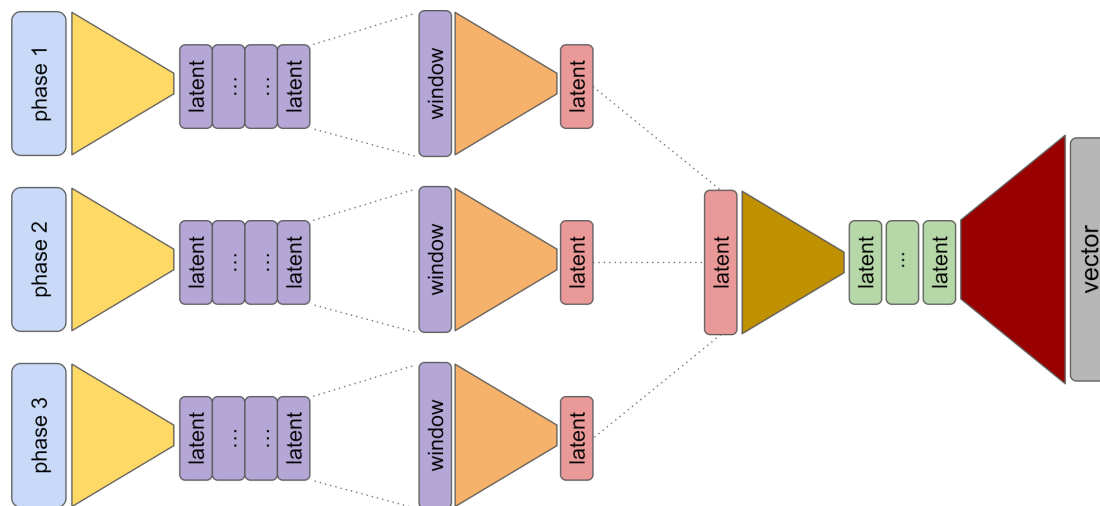


Figure 7.1: A possible future implementation could be to extend the system with a neural network to create a vector. The network could be trained to represent either a single device or a household scenario.

A similar idea is illustrated in figure 7.1 *Future extension of the system* on page 63, where several outputs from Stage 3 are fed into a new network with an output vector of some size. Reinforcement training similar to that described above, as shown in figure 7.2 *Training of future extension* on page 64, would emphasise similarities with positive and differences with negative contributions to the vector similarities.

The main difference to the Facenet approach is that each data point represents a single face, whereas the current state of the household represents multiple appliances together. If the labelled data of the household would be used as shown in figure 7.2 *Training of future extension* on page 64, where the similarity is trained with the lamp (left) and the TV is ignored, but a situation without the lamp (right) is used to give an example of a different device, the embedding should in-

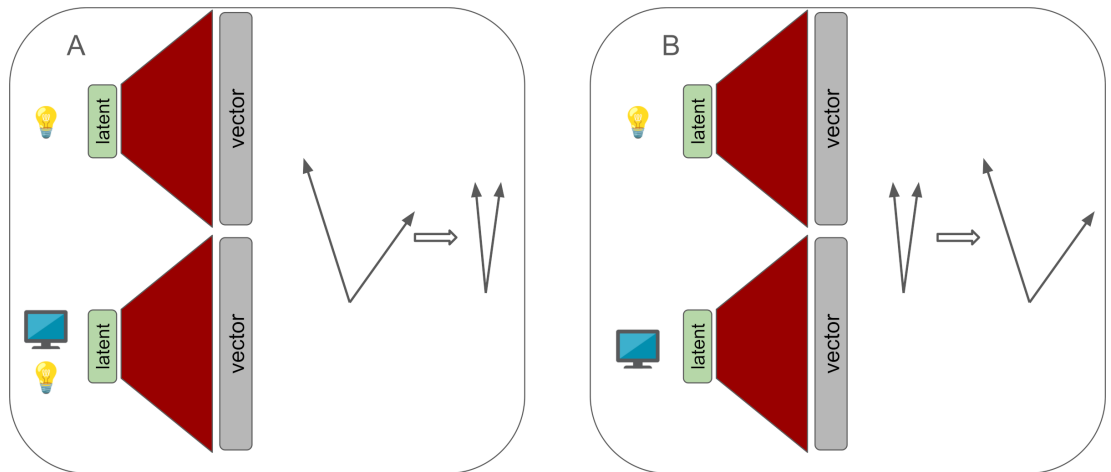


Figure 7.2: The neural network for the vector is trained by correcting the vector. The vectors of similar patterns are 'pushed' together and the opposite if they are not. Over the course of training, the vectors will tend to represent similar patterns with vectors that are 'close' to each other.

clude the disaggregation even if there are no examples of single isolated devices.

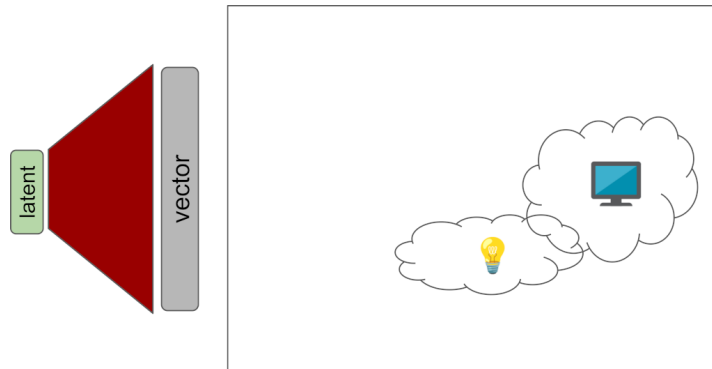


Figure 7.3: After training the neural network, the embedding represents a high-dimensional Euclidean space with the property that similar devices form clusters.

With such an embedding a high dimensional Euclidean space with the properties that similar devices build clusters is formed like illustrated in figure 7.3 *High dimensional Euclidean space with clusters of devices* on page 64. The grouping of vectors to clusters could be done with a algorithm like Chinese Whispers [Bie06] which is often used for face recognition.

Bibliography

- [BGBB14] Corinne Belley, Sebastien Gaboury, Bruno Bouchard, and Abdenour Bouzouane. An efficient and inexpensive method for activity recognition within a smart home based on load signatures of appliances. *Pervasive and Mobile Computing*, 12:58–78, 2014.
- [Bie06] Chris Biemann. Chinese whispers-an efficient graph clustering algorithm and its application to natural language processing problems. In *Proceedings of TextGraphs: the first workshop on graph based methods for natural language processing*, pages 73–80, 2006.
- [BPH⁺24] Tim Brooks, Bill Peebles, Connor Homes, Will DePue, Yufei Guo, Li Jing, David Schnurr, Joe Taylor, Troy Luhman, Eric Luhman, Clarence Wing Yin Ng, Ricky Wang, and Aditya Ramesh. Video generation models as world simulators. 2024.
- [CDW⁺11] Feng Chen, Jing Dai, Bingsheng Wang, Sambit Sahu, Milind Naphade, and Chang-Tien Lu. Activity analysis based on low sample rate smart meters. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 240–248. ACM, 2011.
- [Dee24] DeepAI. CNN Image. <https://deepai.org/machine-learning-model/text2img>, 2024. Accessed: 2024-02-18.
- [Den12] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

- [GBDL⁺16] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International conference on machine learning*, pages 201–210. PMLR, 2016.
- [GCZ18] Aaron Goin, Ronald Cotton, and Xinghui Zhao. Webnn: A distributed framework for deep learning. 2018.
- [Har92] George William Hart. Nonintrusive appliance load monitoring. *Proceedings of the IEEE*, 80(12):1870–1891, 1992.
- [KDW⁺19] Weicong Kong, Zhao Yang Dong, Bo Wang, Junhua Zhao, and Jie Huang. A practical solution for non-intrusive type ii load monitoring based on deep learning and post-processing. *IEEE Transactions on Smart Grid*, 11(1):148–160, 2019.
- [Khr14] Khronos Group. Webcl. <https://registry.khronos.org/webcl/specs/1.0.0/>, 2014. [Online; accessed 26-February-2024].
- [Khr23] Khronos Group. Webgpu. <https://registry.khronos.org/webgl/specs/latest/2.0/>, 2023. [Online; accessed 26-February-2024].
- [LLHJ10] Gu-yuan Lin, Shih-chiang Lee, Jane Yung-jen Hsu, and Wan-rong Jih. Applying power meters for appliance recognition on the electric panel. In *2010 5th IEEE Conference on Industrial Electronics and Applications*, pages 2254–2259. IEEE, 2010.
- [LMB⁺14] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Doll’ar, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014.
- [Ope24] OpenAI ChatGPT. Conversation ChatGPT 3.5. <https://chat.openai.com/c/7406a47b-42e0-4913-b951-49b75bb6daef>, 2024. Accessed: 2024-02-18.

- [PRK⁺07] Shwetak N Patel, Thomas Robertson, Julie A Kientz, Matthew S Reynolds, and Gregory D Abowd. At the flick of a switch: Detecting and classifying unique electrical events on the residential power line (nominated for the best paper award). In *International Conference on Ubiquitous Computing*, pages 271–288. Springer, 2007.
- [RBL⁺22] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10684–10695, 2022.
- [RNSO10] Antonio G Ruzzelli, C Nicolas, Anthony Schoofs, and Gregory MP O’Hare. Real-time recognition and profiling of appliances through a single electricity sensor. In *2010 7th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON)*, pages 1–9. IEEE, 2010.
- [SKP15] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 815–823, 2015.
- [Sta24] Stack Overflow. python - Pandas Dataframe to sliding window - Stack Overflow. <https://stackoverflow.com/questions/77812013/pandas-dataframe-to-sliding-window/77812136>, 2024. Accessed: 2024-02-18.
- [W3C24a] W3C. Web neural network api. <https://www.w3.org/TR/webnn/>, 2024. [Online; accessed 26-February-2024].
- [W3C24b] W3C. Webgpu. <https://www.w3.org/TR/webgpu/>, 2024. [Online; accessed 26-February-2024].
- [Web24] WebAssembly Community Group. Webassembly. <https://webassembly.org/>, 2024. [Online; accessed 26-February-2024].

- [Wik23] Wikipedia contributors. File:Portrait of a smiling girl with wet face at golden hour in Laos.jpg. https://commons.wikimedia.org/w/index.php?title=File:Portrait_of_a_smiling_girl_with_wet_face_at_golden_hour_in_Laos.jpg&oldid=806198153, 2023. Accessed: 2024-02-14.
- [Wik24] Wikipedia contributors. Off-by-one error — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Off-by-one_error&oldid=1210023548, 2024. [Online; accessed 25-February-2024].

Note of thanks

Antonia Müller-Baumgart for the correction and annotations.

Statutory declaration

I declare that I have authored this work independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

date, signature

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig angefertigt und mich keiner fremden Hilfe bedient sowie keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen, die wörtlich oder sinngemäß veröffentlichten oder nicht veröffentlichten Schriften und anderen Quellen entnommen sind, habe ich als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Erklärung zu eingesetzten Hilfsmitteln

1. Korrekturservice der Fachhochschule bzw. des Fachbereichs genutzt: **Nein**
2. Einsatz eines externen (kommerziellen) Korrekturservice: **Nein**
3. Folgende Personen haben die Arbeit zusätzlich Korrektur gelesen:
Antonia Müller-Baumgart
4. Nutzung von Sprachmodellen für die Texterstellung (z.B. ChatGPT), wenn ja, welche und in welchen Abschnitten: **Ja**
ChatGPT, DeepAI, Zitat im Anhang
5. Sprachübersetzungstools (z.B. Google Übersetzer, DeepL), wenn ja, welche und in welchen Abschnitten: **Ja**
Google Übersetzer, DeepL, mit Betreuer vorher besprochen
6. Einsatz von Software zur Sprachkorrektur (z.B. Grammarly), wenn ja, welche und in welchen Abschnitten: **Ja**
TexStudio, DeepL, mit Betreuer vorher besprochen
7. Einsatz anderer Hilfsmittel: **Nein**

Ich nehme zur Kenntnis, dass meine Thesis mittels Software zur Plagiatserkennung überprüft werden kann. Ich bestätige, dass obige Aussagen vollständig und nach bestem Wissen ausgefüllt wurden.

29.02.2024,

Datum, Unterschrift

Appendix A

Attached source code

The source code of this work is embedded in the document. It can be opened with the following link (depends on the pdf viewer).

[link to the embedded archive](#)

Suitable document viewer can open such files.

Tested with:

- Firefox 123.0 (<https://www.mozilla.org/en-US/firefox/new/>)
- Okular 23.08.1 (<https://okular.kde.org/>)
- qpdfview 0.5.0 (<https://launchpad.net/qpdfview>)

Appendix B

Code

B.1 Docker container

```
1 sudo userdocker run -it tensorflow/tensorflow:latest-gpu-jupyter
```

Figure B.1: Command to creation a Docker container on the GPU cluster of the university.

```
1 sudo userdocker run -it -P
2 -v '/usr/bin/sudo:/usr/local/bin/sudo:ro'
3 -v '/usr/lib/sudo:/usr/lib/sudo:ro'
4 -v '/home/diffproml/userdocker/sudoers:/etc/sudoers:ro'
5 -v '/home/diffproml/userdocker/sudoers:/etc/sudoers.d/apt-get-install:ro'
6 tensorflow/tensorflow:latest-gpu-jupyter
```

Figure B.2: Special command to creation a Docker container on the GPU cluster of the university.

B.2 Models

B.2.1 Stage 1

B.2.1.1 simple-latend3

Model: "simple-latend3"

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 6)]	0
dense (Dense)	(None, 6)	42
dense_1 (Dense)	(None, 5)	35
dense_2 (Dense)	(None, 3)	18
dense_3 (Dense)	(None, 5)	20
dense_4 (Dense)	(None, 6)	36

B.2.1.2 simple-latend4

Model: "simple-latend4"

Layer (type)	Output Shape	Param #
input_5 (InputLayer)	[(None, 6)]	0
dense_10 (Dense)	(None, 6)	42
dense_11 (Dense)	(None, 5)	35
dense_12 (Dense)	(None, 4)	24
dense_13 (Dense)	(None, 5)	25
dense_14 (Dense)	(None, 6)	36

B.2.1.3 simple-latend3-optimizer_sgd

Model: "simple-latend3-optimizer_sgd"

Layer (type)	Output Shape	Param #
input_3 (InputLayer)	[(None, 6)]	0
dense_5 (Dense)	(None, 6)	42

dense_6 (Dense)	(None, 5)	35
dense_7 (Dense)	(None, 3)	18
dense_8 (Dense)	(None, 5)	20
dense_9 (Dense)	(None, 6)	36

B.2.1.4 simple-latend3-all_sigmoid

Model: "simple-latend3-all_sigmoid"

Layer (type)	Output Shape	Param #
input_7 (InputLayer)	[(None, 6)]	0
dense_15 (Dense)	(None, 6)	42
dense_16 (Dense)	(None, 5)	35
dense_17 (Dense)	(None, 3)	18
dense_18 (Dense)	(None, 5)	20
dense_19 (Dense)	(None, 6)	36

B.2.1.5 simple-latend4-all_sigmoid

Model: "simple-latend4-all_sigmoid"

Layer (type)	Output Shape	Param #
input_9 (InputLayer)	[(None, 6)]	0
dense_20 (Dense)	(None, 6)	42
dense_21 (Dense)	(None, 5)	35
dense_22 (Dense)	(None, 4)	24
dense_23 (Dense)	(None, 5)	25
dense_24 (Dense)	(None, 6)	36

B.2.1.6 layer5-latend3

Model: "layer5-latend3"

Layer (type)	Output Shape	Param #
input_11 (InputLayer)	[(None, 6)]	0

dense_25 (Dense)	(None, 6)	42
dense_26 (Dense)	(None, 6)	42
dense_27 (Dense)	(None, 6)	42
dense_28 (Dense)	(None, 5)	35
dense_29 (Dense)	(None, 3)	18
dense_30 (Dense)	(None, 5)	20
dense_31 (Dense)	(None, 6)	36
dense_32 (Dense)	(None, 6)	42
dense_33 (Dense)	(None, 6)	42

B.2.1.7 layer5-latend3-dropout0_2

Model: "layer5-latend3-dropout0_2"

Layer (type)	Output Shape	Param #
input_13 (InputLayer)	[(None, 6)]	0
dense_34 (Dense)	(None, 6)	42
dense_35 (Dense)	(None, 6)	42
dropout (Dropout)	(None, 6)	0
dense_36 (Dense)	(None, 6)	42
dense_37 (Dense)	(None, 5)	35
dense_38 (Dense)	(None, 3)	18
dense_39 (Dense)	(None, 5)	20
dense_40 (Dense)	(None, 6)	36
dense_41 (Dense)	(None, 6)	42
dense_42 (Dense)	(None, 6)	42

B.2.2 Stage 2

B.2.2.1 window-in3-lat8-4layers

Model: window-in3-lat8-4layers

Layer (type)	Output Shape	Param #
--------------	--------------	---------

```

=====
input_1 (InputLayer)      [(None, 48)]           0
dense (Dense)             (None, 48)            2352
dense_1 (Dense)          (None, 32)            1568
dense_2 (Dense)          (None, 18)            594
dense_3 (Dense)          (None, 8)             152
dense_4 (Dense)          (None, 32)            288
dense_5 (Dense)          (None, 48)            1584

```

B.2.2.2 window-in3-lat8-6layers

Model: window-in3-lat8-6layers

```

Layer (type)              Output Shape           Param #
=====
input_3 (InputLayer)     [(None, 48)]          0
dense_6 (Dense)          (None, 48)            2352
dense_7 (Dense)          (None, 32)            1568
dense_8 (Dense)          (None, 32)            1056
dense_9 (Dense)          (None, 18)            594
dense_10 (Dense)         (None, 18)            342
dense_11 (Dense)         (None, 8)             152
dense_12 (Dense)         (None, 18)            162
dense_13 (Dense)         (None, 32)            608
dense_14 (Dense)         (None, 32)            1056
dense_15 (Dense)         (None, 48)            1584

```

B.2.2.3 window-in3-lat12-4layers

Model: window-in3-lat12-4layers

```

Layer (type)              Output Shape           Param #
=====
input_5 (InputLayer)     [(None, 48)]          0

```


dense_16 (Dense)	(None, 48)	2352
dense_17 (Dense)	(None, 32)	1568
dense_18 (Dense)	(None, 18)	594
dense_19 (Dense)	(None, 12)	228
dense_20 (Dense)	(None, 32)	416
dense_21 (Dense)	(None, 48)	1584

B.2.2.4 window-in3-lat12-6layers

Model: window-in3-lat12-6layers

Layer (type)	Output Shape	Param #
=====		
input_7 (InputLayer)	[(None, 48)]	0
dense_22 (Dense)	(None, 48)	2352
dense_23 (Dense)	(None, 32)	1568
dense_24 (Dense)	(None, 32)	1056
dense_25 (Dense)	(None, 18)	594
dense_26 (Dense)	(None, 18)	342
dense_27 (Dense)	(None, 12)	228
dense_28 (Dense)	(None, 18)	234
dense_29 (Dense)	(None, 32)	608
dense_30 (Dense)	(None, 32)	1056
dense_31 (Dense)	(None, 48)	1584

B.2.2.5 window-in3-lat16-4layers

Model: window-in3-lat16-4layers

Layer (type)	Output Shape	Param #
=====		
input_9 (InputLayer)	[(None, 48)]	0
dense_32 (Dense)	(None, 48)	2352
dense_33 (Dense)	(None, 32)	1568

dense_34 (Dense)	(None, 18)	594
dense_35 (Dense)	(None, 16)	304
dense_36 (Dense)	(None, 32)	544
dense_37 (Dense)	(None, 48)	1584

B.2.2.6 window-in3-lat16-6layers

Model: window-in3-lat16-6layers

Layer (type)	Output Shape	Param #
=====		
input_11 (InputLayer)	[(None, 48)]	0
dense_38 (Dense)	(None, 48)	2352
dense_39 (Dense)	(None, 32)	1568
dense_40 (Dense)	(None, 32)	1056
dense_41 (Dense)	(None, 18)	594
dense_42 (Dense)	(None, 18)	342
dense_43 (Dense)	(None, 16)	304
dense_44 (Dense)	(None, 18)	306
dense_45 (Dense)	(None, 32)	608
dense_46 (Dense)	(None, 32)	1056
dense_47 (Dense)	(None, 48)	1584

B.2.2.7 window-in4-lat8-4layers

Model: window-in4-lat8-4layers

Layer (type)	Output Shape	Param #
=====		
input_13 (InputLayer)	[(None, 64)]	0
dense_48 (Dense)	(None, 64)	4160
dense_49 (Dense)	(None, 32)	2080
dense_50 (Dense)	(None, 18)	594
dense_51 (Dense)	(None, 8)	152

dense_52 (Dense)	(None, 32)	288
dense_53 (Dense)	(None, 64)	2112

B.2.2.8 window-in4-lat8-6layers

Model: window-in4-lat8-6layers

Layer (type)	Output Shape	Param #
=====		
input_15 (InputLayer)	[(None, 64)]	0
dense_54 (Dense)	(None, 64)	4160
dense_55 (Dense)	(None, 32)	2080
dense_56 (Dense)	(None, 32)	1056
dense_57 (Dense)	(None, 18)	594
dense_58 (Dense)	(None, 18)	342
dense_59 (Dense)	(None, 8)	152
dense_60 (Dense)	(None, 18)	162
dense_61 (Dense)	(None, 32)	608
dense_62 (Dense)	(None, 32)	1056
dense_63 (Dense)	(None, 64)	2112

B.2.2.9 window-in4-lat12-4layers

Model: window-in4-lat12-4layers

Layer (type)	Output Shape	Param #
=====		
input_17 (InputLayer)	[(None, 64)]	0
dense_64 (Dense)	(None, 64)	4160
dense_65 (Dense)	(None, 32)	2080
dense_66 (Dense)	(None, 18)	594
dense_67 (Dense)	(None, 12)	228
dense_68 (Dense)	(None, 32)	416
dense_69 (Dense)	(None, 64)	2112

B.2.2.10 window-in4-lat12-6layers

Model: window-in4-lat12-6layers

Layer (type)	Output Shape	Param #
=====		
input_19 (InputLayer)	[(None, 64)]	0
dense_70 (Dense)	(None, 64)	4160
dense_71 (Dense)	(None, 32)	2080
dense_72 (Dense)	(None, 32)	1056
dense_73 (Dense)	(None, 18)	594
dense_74 (Dense)	(None, 18)	342
dense_75 (Dense)	(None, 12)	228
dense_76 (Dense)	(None, 18)	234
dense_77 (Dense)	(None, 32)	608
dense_78 (Dense)	(None, 32)	1056
dense_79 (Dense)	(None, 64)	2112

B.2.2.11 window-in4-lat16-4layers

Model: window-in4-lat16-4layers

Layer (type)	Output Shape	Param #
=====		
input_21 (InputLayer)	[(None, 64)]	0
dense_80 (Dense)	(None, 64)	4160
dense_81 (Dense)	(None, 32)	2080
dense_82 (Dense)	(None, 18)	594
dense_83 (Dense)	(None, 16)	304
dense_84 (Dense)	(None, 32)	544
dense_85 (Dense)	(None, 64)	2112

B.2.2.12 window-in4-lat16-6layers

Model: window-in4-lat16-6layers

Layer (type)	Output Shape	Param #
=====		
input_23 (InputLayer)	[(None, 64)]	0
dense_86 (Dense)	(None, 64)	4160
dense_87 (Dense)	(None, 32)	2080
dense_88 (Dense)	(None, 32)	1056
dense_89 (Dense)	(None, 18)	594
dense_90 (Dense)	(None, 18)	342
dense_91 (Dense)	(None, 16)	304
dense_92 (Dense)	(None, 18)	306
dense_93 (Dense)	(None, 32)	608
dense_94 (Dense)	(None, 32)	1056
dense_95 (Dense)	(None, 64)	2112

B.2.2.13 window-in4-lat12-conv-mini

Model: window-in4-lat12-conv-mini

Layer (type)	Output Shape	Param #
=====		
input_33 (InputLayer)	[(None, 16, 4, 1)]	0
conv2d_10 (Conv2D)	(None, 16, 4, 64)	640
max_pooling2d_5 (MaxPooling 2D)	(None, 8, 2, 64)	0
flatten_5 (Flatten)	(None, 1024)	0
dense_140 (Dense)	(None, 128)	131200
dense_141 (Dense)	(None, 12)	1548
dense_142 (Dense)	(None, 128)	1664
dense_143 (Dense)	(None, 1024)	132096
reshape_5 (Reshape)	(None, 16, 4, 16)	0

conv2d_transpose_5 (Conv2DT ranspose)	(None, 16, 4, 64)	9280
conv2d_11 (Conv2D)	(None, 16, 4, 1)	577

B.2.2.14 window-in4-lat16-conv-mini

Model: window-in4-lat16-conv-mini

Layer (type)	Output Shape	Param #
input_35 (InputLayer)	[(None, 16, 4, 1)]	0
conv2d_12 (Conv2D)	(None, 16, 4, 64)	640
max_pooling2d_6 (MaxPooling 2D)	(None, 8, 2, 64)	0
flatten_6 (Flatten)	(None, 1024)	0
dense_144 (Dense)	(None, 128)	131200
dense_145 (Dense)	(None, 16)	2064
dense_146 (Dense)	(None, 128)	2176
dense_147 (Dense)	(None, 1024)	132096
reshape_6 (Reshape)	(None, 16, 4, 16)	0
conv2d_transpose_6 (Conv2DT ranspose)	(None, 16, 4, 64)	9280
conv2d_13 (Conv2D)	(None, 16, 4, 1)	577

B.2.2.15 window-in4-lat9-conv-mini

Model: window-in4-lat9-conv-mini

Layer (type)	Output Shape	Param #
input_37 (InputLayer)	[(None, 16, 4, 1)]	0
conv2d_14 (Conv2D)	(None, 16, 4, 64)	640
max_pooling2d_7 (MaxPooling 2D)	(None, 8, 2, 64)	0

flatten_7 (Flatten)	(None, 1024)	0
dense_148 (Dense)	(None, 128)	131200
dense_149 (Dense)	(None, 9)	1161
dense_150 (Dense)	(None, 128)	1280
dense_151 (Dense)	(None, 1024)	132096
reshape_7 (Reshape)	(None, 16, 4, 16)	0
conv2d_transpose_7 (Conv2DT ranspose)	(None, 16, 4, 64)	9280
conv2d_15 (Conv2D)	(None, 16, 4, 1)	577

B.2.3 Stage 3

B.2.3.1 Graphs

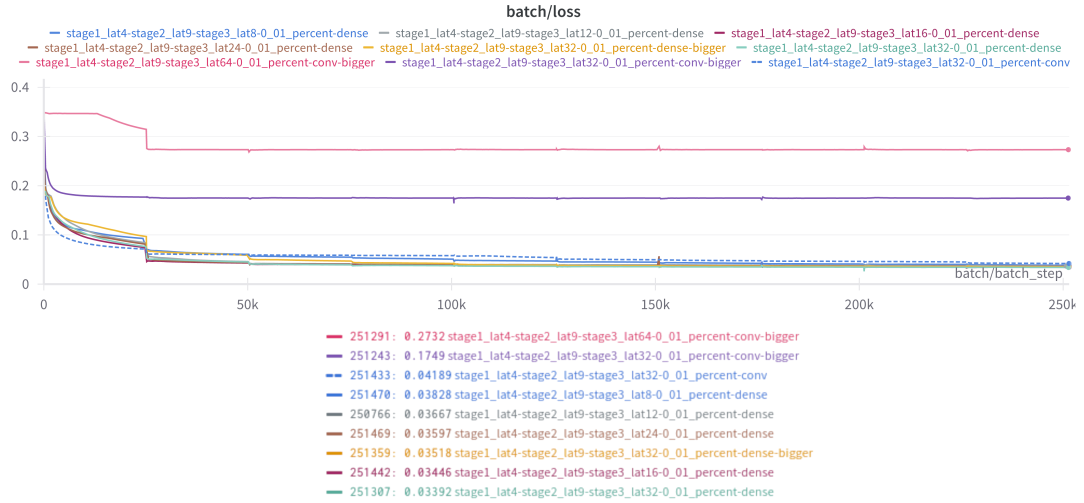


Figure B.3: Graphs of the training of Stage 3 part 1

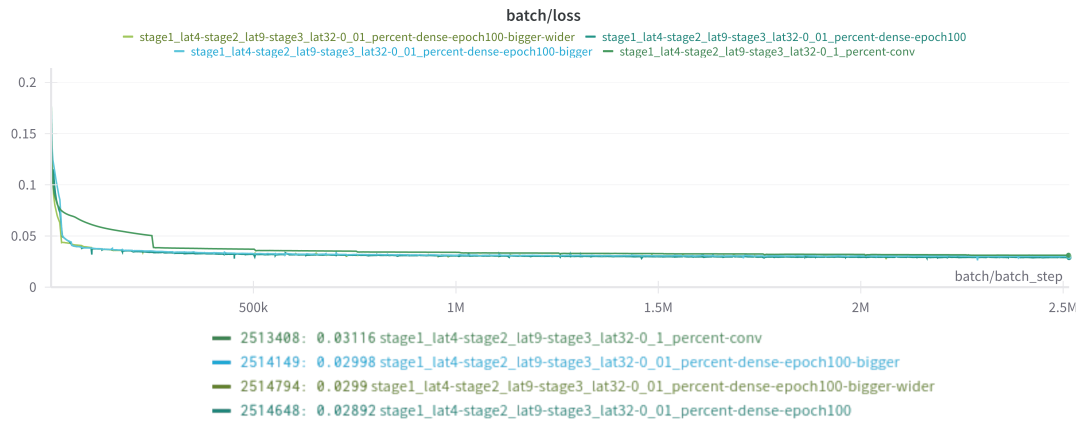


Figure B.4: Graphs of the training of Stage 3 part 2

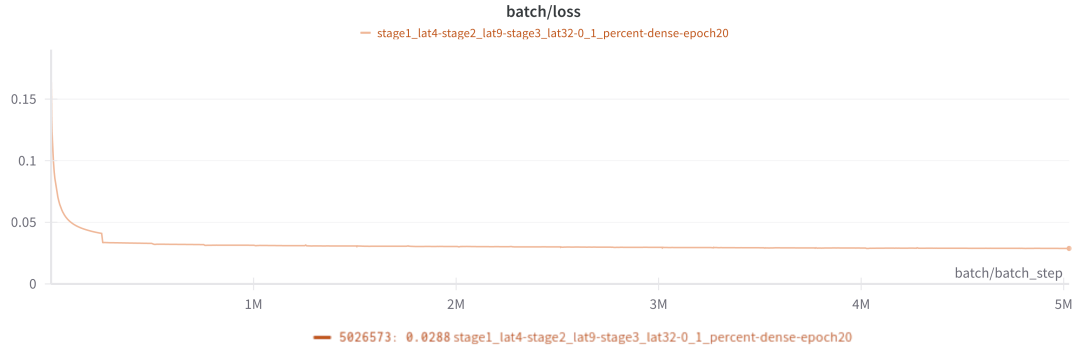


Figure B.5: Graphs of the training of Stage 3 part 3

B.2.3.2 stage1_lat4-stage2_lat9-stage3_lat32-0_01_percent-conv

Model: stage1_lat4-stage2_lat9-stage3_lat32-0_01_percent-conv

Layer (type)	Output Shape	Param #
input_11 (InputLayer)	[(None, 3, 3, 3, 1)]	0
conv3d_5 (Conv3D)	(None, 3, 3, 3, 64)	1792
max_pooling3d_2 (MaxPooling3D)	(None, 1, 1, 1, 64)	0
flatten_5 (Flatten)	(None, 64)	0
dense_6 (Dense)	(None, 128)	8320
dense_7 (Dense)	(None, 32)	4128
dense_8 (Dense)	(None, 128)	4224
dense_9 (Dense)	(None, 1728)	222912
reshape_1 (Reshape)	(None, 3, 3, 3, 64)	0
conv3d_transpose_1 (Conv3DTranspose)	(None, 3, 3, 3, 64)	110656
conv3d_6 (Conv3D)	(None, 3, 3, 3, 1)	1729

B.2.3.3 stage1_lat4-stage2_lat9-stage3_lat32-0_1_percent-conv

Model: stage1_lat4-stage2_lat9-stage3_lat32-0_1_percent-conv

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 3, 3, 3, 1)]	0
conv3d (Conv3D)	(None, 3, 3, 3, 64)	1792
max_pooling3d (MaxPooling3D)	(None, 1, 1, 1, 64)	0
)		
flatten (Flatten)	(None, 64)	0
dense (Dense)	(None, 128)	8320
dense_1 (Dense)	(None, 32)	4128
dense_2 (Dense)	(None, 128)	4224
dense_3 (Dense)	(None, 1728)	222912
reshape (Reshape)	(None, 3, 3, 3, 64)	0
conv3d_transpose (Conv3DTranspose)	(None, 3, 3, 3, 64)	110656
conv3d_1 (Conv3D)	(None, 3, 3, 3, 1)	1729

B.2.3.4 stage1_lat4-stage2_lat9-stage3_lat32-0_01_percent-conv-bigger

Model: stage1_lat4-stage2_lat9-stage3_lat32-0_01_percent-conv-bigger

Layer (type)	Output Shape	Param #
input_3 (InputLayer)	[(None, 3, 3, 3, 1)]	0
conv3d_2 (Conv3D)	(None, 3, 3, 3, 64)	1792
dense_4 (Dense)	(None, 3, 3, 3, 1792)	116480
dense_5 (Dense)	(None, 3, 3, 3, 1792)	3213056
max_pooling3d_1 (MaxPooling3D)	(None, 1, 1, 1, 1792)	0
)		
flatten_1 (Flatten)	(None, 1792)	0
dense_6 (Dense)	(None, 128)	229504
dense_7 (Dense)	(None, 32)	4128
dense_8 (Dense)	(None, 128)	4224

dense_9 (Dense)	(None, 1728)	222912
reshape_1 (Reshape)	(None, 3, 3, 3, 64)	0
conv3d_transpose_1 (Conv3DT ranspose)	(None, 3, 3, 3, 64)	110656
dense_10 (Dense)	(None, 3, 3, 3, 1729)	112385
dense_11 (Dense)	(None, 3, 3, 3, 1729)	2991170
conv3d_3 (Conv3D)	(None, 3, 3, 3, 1)	46684

B.2.3.5 stage1_lat4-stage2_lat9-stage3_lat64-0_01_percent-conv-bigger

Model: stage1_lat4-stage2_lat9-stage3_lat64-0_01_percent-conv-bigger

Layer (type)	Output Shape	Param #
=====		
input_7 (InputLayer)	[(None, 3, 3, 3, 1)]	0
conv3d_6 (Conv3D)	(None, 3, 3, 3, 64)	1792
dense_20 (Dense)	(None, 3, 3, 3, 1792)	116480
dense_21 (Dense)	(None, 3, 3, 3, 1792)	3213056
max_pooling3d_3 (MaxPooling 3D)	(None, 1, 1, 1, 1792)	0
flatten_3 (Flatten)	(None, 1792)	0
dense_22 (Dense)	(None, 512)	918016
dense_23 (Dense)	(None, 64)	32832
dense_24 (Dense)	(None, 512)	33280
dense_25 (Dense)	(None, 1728)	886464
reshape_3 (Reshape)	(None, 3, 3, 3, 64)	0
conv3d_transpose_3 (Conv3DT ranspose)	(None, 3, 3, 3, 64)	110656
dense_26 (Dense)	(None, 3, 3, 3, 1729)	112385
dense_27 (Dense)	(None, 3, 3, 3, 1729)	2991170
conv3d_7 (Conv3D)	(None, 3, 3, 3, 1)	46684

B.2.3.6 stage1_lat4-stage2_lat9-stage3_lat32-0_01_percent-dense

Model: stage1_lat4-stage2_lat9-stage3_lat32
-0_01_percent-dense

Layer (type)	Output Shape	Param #
input_11 (InputLayer)	[(None, 27)]	0
dense_35 (Dense)	(None, 27)	756
dense_36 (Dense)	(None, 64)	1792
dense_37 (Dense)	(None, 48)	3120
dense_38 (Dense)	(None, 32)	1568
dense_39 (Dense)	(None, 48)	1584
dense_40 (Dense)	(None, 64)	3136
dense_41 (Dense)	(None, 27)	1755

B.2.3.7 stage1_lat4-stage2_lat9-stage3_lat32-0_01_percent-dense-bigger

Model: stage1_lat4-stage2_lat9-stage3_lat32
-0_01_percent-dense-bigger

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 27)]	0
dense (Dense)	(None, 27)	756
dense_1 (Dense)	(None, 128)	3584
dense_2 (Dense)	(None, 64)	8256
dense_3 (Dense)	(None, 48)	3120
dense_4 (Dense)	(None, 32)	1568
dense_5 (Dense)	(None, 48)	1584
dense_6 (Dense)	(None, 64)	3136
dense_7 (Dense)	(None, 128)	8320
dense_8 (Dense)	(None, 27)	3483

B.2.3.8 stage1_lat4-stage2_lat9-stage3_lat32-0_01_percent-dense-epoch100-bigger

Model: stage1_lat4-stage2_lat9-stage3_lat32-0_01_percent-dense-epoch100-bigger

Layer (type)	Output Shape	Param #
input_3 (InputLayer)	[(None, 27)]	0
dense_9 (Dense)	(None, 27)	756
dense_10 (Dense)	(None, 128)	3584
dense_11 (Dense)	(None, 64)	8256
dense_12 (Dense)	(None, 48)	3120
dense_13 (Dense)	(None, 32)	1568
dense_14 (Dense)	(None, 48)	1584
dense_15 (Dense)	(None, 64)	3136
dense_16 (Dense)	(None, 128)	8320
dense_17 (Dense)	(None, 27)	3483

B.2.3.9 stage1_lat4-stage2_lat9-stage3_lat32-0_01_percent-dense-epoch100

Model: stage1_lat4-stage2_lat9-stage3_lat32-0_01_percent-dense-epoch100

Layer (type)	Output Shape	Param #
input_5 (InputLayer)	[(None, 27)]	0
dense_18 (Dense)	(None, 27)	756
dense_19 (Dense)	(None, 64)	1792
dense_20 (Dense)	(None, 48)	3120
dense_21 (Dense)	(None, 32)	1568
dense_22 (Dense)	(None, 48)	1584
dense_23 (Dense)	(None, 64)	3136
dense_24 (Dense)	(None, 27)	1755

B.2.3.10 stage1_lat4-stage2_lat9-stage3_lat32-0_1_percent-dense-epoch20

Model: stage1_lat4-stage2_lat9-stage3_lat32-0_1_percent-dense-epoch20

Layer (type)	Output Shape	Param #
input_7 (InputLayer)	[(None, 27)]	0
dense_25 (Dense)	(None, 27)	756
dense_26 (Dense)	(None, 64)	1792
dense_27 (Dense)	(None, 48)	3120
dense_28 (Dense)	(None, 32)	1568
dense_29 (Dense)	(None, 48)	1584
dense_30 (Dense)	(None, 64)	3136
dense_31 (Dense)	(None, 27)	1755

B.2.3.11 stage1_lat4-stage2_lat9-stage3_lat32-0_01_percent-dense-epoch100-bigger-wider

Model: stage1_lat4-stage2_lat9-stage3_lat32-0_01_percent-dense-epoch100-bigger-wider

Layer (type)	Output Shape	Param #
input_11 (InputLayer)	[(None, 27)]	0
dense_39 (Dense)	(None, 27)	756
dense_40 (Dense)	(None, 256)	7168
dense_41 (Dense)	(None, 128)	32896
dense_42 (Dense)	(None, 64)	8256
dense_43 (Dense)	(None, 32)	2080
dense_44 (Dense)	(None, 64)	2112
dense_45 (Dense)	(None, 128)	8320
dense_46 (Dense)	(None, 256)	33024
dense_47 (Dense)	(None, 27)	6939

B.2.3.12 stage1_lat4-stage2_lat9-stage3_lat24-0_01_percent-dense

Model: stage1_lat4-stage2_lat9-stage3_lat24-0_01_percent-dense

Layer (type)	Output Shape	Param #
input_13 (InputLayer)	[(None, 27)]	0
dense_48 (Dense)	(None, 27)	756
dense_49 (Dense)	(None, 64)	1792
dense_50 (Dense)	(None, 48)	3120
dense_51 (Dense)	(None, 24)	1176
dense_52 (Dense)	(None, 48)	1200
dense_53 (Dense)	(None, 64)	3136
dense_54 (Dense)	(None, 27)	1755

B.2.3.13 stage1_lat4-stage2_lat9-stage3_lat16-0_01_percent-dense

Model: stage1_lat4-stage2_lat9-stage3_lat16-0_01_percent-dense

Layer (type)	Output Shape	Param #
input_15 (InputLayer)	[(None, 27)]	0
dense_55 (Dense)	(None, 27)	756
dense_56 (Dense)	(None, 64)	1792
dense_57 (Dense)	(None, 48)	3120
dense_58 (Dense)	(None, 16)	784
dense_59 (Dense)	(None, 48)	816
dense_60 (Dense)	(None, 64)	3136
dense_61 (Dense)	(None, 27)	1755

B.2.3.14 stage1_lat4-stage2_lat9-stage3_lat12-0_01_percent-dense

Model: stage1_lat4-stage2_lat9-stage3_lat12-0_01_percent-dense

Layer (type)	Output Shape	Param #
input_17 (InputLayer)	[(None, 27)]	0
dense_62 (Dense)	(None, 27)	756
dense_63 (Dense)	(None, 64)	1792
dense_64 (Dense)	(None, 48)	3120
dense_65 (Dense)	(None, 12)	588
dense_66 (Dense)	(None, 48)	624
dense_67 (Dense)	(None, 64)	3136
dense_68 (Dense)	(None, 27)	1755

B.2.3.15 stage1_lat4-stage2_lat9-stage3_lat8-0_01_percent-dense

Model: stage1_lat4-stage2_lat9-stage3_lat8-0_01_percent-dense

Layer (type)	Output Shape	Param #
input_19 (InputLayer)	[(None, 27)]	0
dense_69 (Dense)	(None, 27)	756
dense_70 (Dense)	(None, 64)	1792
dense_71 (Dense)	(None, 48)	3120
dense_72 (Dense)	(None, 8)	392
dense_73 (Dense)	(None, 48)	432
dense_74 (Dense)	(None, 64)	3136
dense_75 (Dense)	(None, 27)	1755

Appendix C

GAN prompts

C.1 ChatGPT3.5

ME: could you create me an image which illustrates the concept of convolutional neural networks?

ChatGPT: I'm sorry, but as a text-based AI model, I don't have the capability to create images. However, I can certainly help you describe the concept of convolutional neural networks (CNNs) and guide you on how to create an illustration.

A Convolutional Neural Network is a type of artificial neural network designed for processing structured grid data, such as images. The key components of a CNN include convolutional layers, pooling layers, and fully connected layers. Here's a textual description that you can use to guide the creation of an illustration:

Input Layer: Represent an image as a grid of pixels.

Convolutional Layer: Illustrate a convolutional operation by showing a small filter/kernel (a matrix) sliding over the input image, extracting features through element-wise multiplication and summation. Each filter produces a feature map, emphasising certain patterns or features.

Activation Function: Apply an activation function (e.g., ReLU) to introduce non-linearity and enhance the network's ability to learn complex patterns.

Pooling Layer: Depict a pooling operation (e.g., max pooling) that reduces the spatial dimensions of the feature maps, focusing on the most important information.

Flattening: Flatten the output to prepare for the fully connected layers.

Fully Connected Layer: Represent fully connected neurons that take the flattened features as input. Illustrate weights connecting every neuron from the previous layer to each neuron in this layer.

Output Layer: The final layer representing the predicted class probabilities.

You can use visual elements like grids, arrows, and colour variations to highlight the flow of information through the layers. You may also consider incorporating filters, activation functions, and pooling operations in a step-by-step fashion.

For creating an actual image, you can use graphic design tools like Adobe Illustrator, Photoshop, or online platforms like Canva. Alternatively, you can sketch the concept on paper and scan or photograph it for a simple representation.

ME: could you create me a good prompt for dalle2 for this?

ChatGPT: Certainly! To generate an image using DALL·E 2 that illustrates the concept of Convolutional Neural Networks (CNNs), you can use the following prompt:

"Create an image that visually represents the architecture and functionality of a Convolutional Neural Network (CNN). Showcase the key components such as input layer with pixel grid, convolutional layers with filters sliding over the image, activation functions introducing non-linearity, pooling layers reducing spatial dimensions, flattening, fully connected layers with weighted connections, and an output layer predicting class probabilities. Use visual elements like grids, arrows, and color variations to emphasize the flow of information through the layers. Ensure the image is clear, informative, and visually appealing."

Feel free to adjust or add specific details to the prompt based on your preferences.

ME: the prompt needs to be much shorter

ChatGPT: Certainly, here's a shorter prompt for DALL·E 2:

"Generate a clear and concise visual representation of a Convolutional Neural Network (CNN). Depict key components like input layer, convolutional layers with sliding filters, activation functions, pooling layers, flattening, fully connected layers, and output layer. Emphasize the flow of information and use visual elements like grids, arrows, and color variations for clarity." How would I cite this conversation in a scientific paper? ChatGPT

If you need to cite this conversation in a scientific paper, you should follow the citation style recommended by the journal or institution you are submitting to. However, as of my last update in January 2022, there isn't a standardized citation format for AI models like mine. Still, you can follow a general format. Here's an example in APA style:

Author. (Year, Month Day). Title of the conversation. ChatGPT. URL

For instance:

OpenAI. (2022, February 18). Conversations on various topics. ChatGPT. [URL]

Make sure to replace "[URL]" with the specific URL or link to the page where this conversation took place.

Keep in mind that scientific paper guidelines may vary, so it's advisable to check the specific citation style guide of the journal or institution you are submitting to for any additional requirements or variations.

C.2 deepai.org

<https://deepai.org/machine-learning-model/text2img> Date: 18.02.2024 15.40

Generate a clear and concise visual representation of a Convolutional Neural Network (CNN). Depict key components like input layer, convolutional layers with sliding filters, activation functions, pooling layers, flattening, fully connected layers, and output layer. Emphasize the flow of information and use visual elements like grids, arrows, and color variations for clarity