Applying Automatic Visualisation to Deep Neural Network Development

Mette Preuhsler

Bachelor's Thesis
April 2024

Prof. Dr. Reinhard von Hanxleden
Real-Time and Embedded Systems Group
Department of Computer Science
Kiel University

Advised by
M. Sc. Maximilian Kasperowski
M. Sc. Connor Schönberner

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Weiterhin erkläre ich, dass die digitale Fassung dieser Arbeit, die dem Prüfungsamt per E-Mail zugegangen ist, der vorliegenden schriftlichen Fassung entspricht.

Kiel,

Abstract

In the last two decades, the application of *artificial neural networks* has seen unprecedented increase in adoption rate. Artificial neural networks gave rise to increasingly complex models, called Deep Neural Networks. These are able to learn more complex contexts and relationships and can therefore also solve more demanding tasks. This thesis examines the automatic visualisation of Deep Neural Networks (DNNs), precisely through the use of the PyTorchKGT tool, which was developed in this thesis.

Most other tools focus on different visualisations, and computational graph visualisations are uncommon. While TensorBoard now supports visualisations of computational graphs, it is mainly a multi-purpose live monitoring tool that is used for tracking, monitoring and plotting metrics of Machine Learning experiments. In contrast, PyTorchKGT, an extension to the PyTorch framework, is specialised for automatic neural network visualisation and is also lightweight and extensible. Especially computational graph visualisations can be viewed without restrictions such as no limitations in size. The tool uses the diagramming framework KLighD, which then uses the automatic layout of ELK. It focuses on visualising computational graphs, which serve as the foundation for computing gradients by *backwards propagation*, which is at the heart of how deep neural networks learn in the first place.

The evaluation showed that the visualisation of DNNs is an important part of development and research in this area. The thesis also investigates the potential for further development of the PyTorchKGT tool to create a live monitoring tool for applied development and research.

Acknowledgements

I would like to start by expressing my gratitude to Prof. Dr. Reinhard von Hanxleden, who heads the Real-Time and Embedded Systems Group, for giving me the opportunity to work on this fascinating and cutting-edge topic.

Furthermore, I want to express my thanks to Maximilian Kasperowski and Connor Schönberner for supervising my thesis. They helped and guided me in to the right direction. The weekly meetings and the discussions helped me to decide the next steps.

I would also like to thank Michel Spils, from the Intelligent Systems group, and the whole working group, Real-Time and Embedded Systems, for the help and the friendly and forthcoming office environment.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Contributions . . . 2

2 Foundations **3**
- 2.1 Machine Learning Architectures . . . 3
- 2.2 Deep Neural Networks . . . 4
- 2.3 Training and Computational Graphs . . . 5
- 2.4 Graphs and Automatic Graph Drawing . . . 5
- 2.5 KLighD and ELK from the KIELER Project . . . 6

3 Related Work **7**
- 3.1 Automated Visualisation of DNNs . . . 7
- 3.2 Existing DNN Visualisation Tools . . . 7
- 3.3 GraphViz . . . 8
- 3.4 PyTorchViz . . . 9

4 Deep Neural Network Visualisation Tool PyTorchKGT **11**
- 4.1 Visualisation tool architecture . . . 11
- 4.2 Design Choice for DNN Visualisations . . . 12
- 4.3 Determination of PyTorchViz as Tool of Choice . . . 13
- 4.4 Computational Graph Visualisation . . . 13
- 4.5 Architectural Graph Visualisation . . . 15

5 Implementation **19**
- 5.1 KGraph Format . . . 19
- 5.2 KGraph Specification . . . 19
- 5.3 PyTorchViz Node Definitions . . . 24
- 5.4 KGraph Definition and Implementation . . . 27
- 5.5 PyTorchKGT Implementation . . . 29

6 Evaluation **31**
- 6.1 PyTorchKGT Visualisation Results . . . 31
- 6.2 Survey . . . 32
  - 6.2.1 Results . . . 33
  - 6.2.2 Discussion . . . 34

7 Conclusion **35**

Bibliography **37**

List of Figures

3.1 A GraphViz example of a neural network diagram for a multi-class classification problem, taken from [Zey24]. 8
3.2 A visualisation of a computational graph by PyTorchViz, taken from [Git24]. . 10

4.1 Overview of the tool's general process from input to the representation. 11
4.2 PyTorch neural network module model. 16

5.1 Class diagram of the KGraph structure from the KLighD implementation [Spi14]. 20
5.2 KRendering class diagram from the KLighD implementation [Sch14]. 21
5.3 KIELER Lightweight Diagrams (KLighD) project KStyle class diagram. 22
5.4 The class diagram of the ContainerRendering. 23
5.5 PyTorchKGT visualisation of an LSTM start and middle section. 24
5.6 PyTorchKGT visualisation of an LSTM start and middle section, with results shown. 26
5.7 The class diagram of the KRendering and the KStyle classes. 27

6.1 PyTorchKGT visualisation of an Long Short-Term Memory Network (LSTM) computational graph. 31
6.2 PyTorchKGT visualisation of an LSTM as a compact view and result view. . . . 32
6.3 PyTorchKGT visualisation of an LSTM as a compact view and result view with additional values. 33

Acronyms

KLighD	KIELER Lightweight Diagrams
KGT	KGraph Text
ML	Machine Learning
DNN	Deep Neural Network
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
LSTM	Long Short-Term Memory Network
GAN	Generative Adversarial Network

Chapter 1

Introduction

Machine Learning (ML) is more present in everyday life than ever before. This includes many different areas of daily life, such as finance, the automotive industry and retail. ML supports business management and research in different fields through predictions and optimisation. As described by [Zi24], ML is used in the healthcare sector to evaluate results of MRIs or X-rays. It is also used for development of drugs and the recognition of symptoms in clinical pictures and the diagnosis of the underlying illness. Large machine learning models are mostly used for solving more complex problems such as natural language recognition, strategic decision making and image processing.

ML, especially Deep Neural Networks, make these technologies possible. To develop these DNN models, one needs to understand the underlying processes. Visualisations of these models can compactly describe parts of themselves. Deep neural networks learn by computing gradients through *backwards propagation*. This process can be visualised within a computational graph, providing a clear understanding of how the network learns.

1.1 Motivation

The field of deep learning is constantly evolving. Computational graphs are the technical basis of backwards propagation, which drives the training of DNNs. Visualisation of these foundations, as well as live monitoring visualisations, can help to understand the inner workings of DNNs. Visualisations require manual creation and adaptation for papers, lectures or architectural explanations. This can be difficult for larger DNNs and can be simplified and made more efficient by an automatic visualisation of computational graphs. Better visualisation provides benefits not only for general understanding, but also for solving specific problems with its help [Hee19]. It provides a variety of benefits that help researchers, developers, and users to better understand the inner workings and performance of these complex computational graph structures. It can also be used for educational materials to provide a better understanding of the computations used in a DNN.

PyTorchViz is an existing tool that allows for the generation of visualisations from DNNs. The DNN model is specified in Python code, using the deep learning library PyTorch. The created visualisations are exported as PNG image files. PyTorch is a widely used deep learning framework due to its flexibility, ease of use, and high performance. It has also become increasingly popular with deep learning practitioners because it enables an imperative style of programming, making modelling and debugging more intuitive [SAV+20].

1.2 Contributions

This thesis examines the automatic visualisation of DNNs, in particular through the use of the PyTorchKGT tool developed in this thesis. The tool provides a computational graph visualisation that is displayed using KLighD, part of the KIELER project at the University of Kiel. This allows PyTorchKGT to use any visualisation options that are provided by KLighD.

According to [Cho22], Python is one of the most widely used programming languages in the field. This fact combined with PyTorchViz being implemented in Python are the reasons that Python was used for this project. To use Python with KLighD, the python datastructure that stores the DNN model, needs to be converted to be used in KLighD. To achieve that a converter into the KGraph Text format was developed. The converter or parts of the converter can be reused to implement other Python programs that shell be exported to the KGraph Text (KGT) format.

It is possible to visualise different aspects of the DNN model. One of the aspects is the computational graph. This thesis includes the implementation of KGT export of the computational graph for a given DNN model, as well as the concept of an architecture of DNN structures visualisation. Further research or implementation effort is required for additional different visualisations.

Additionally, the following research questions were answered in this thesis. How to visualise computational graphs of DNNs? What advantages does PyTorchKGT bring to the KIELER project? How does PyTorchKGT be integrated as a use case for authors of academic papers and developers of DNNs?

The rest of this thesis is organized as follows. Chapter 2 lays the foundations by covering the basics of machine learning, graph drawing and KLighD. The related works of different visualisation tools are elucidated in Chapter 3. Chapter 4 provides a detailed explanation of the implementation concepts of the PyTorchKGT tool, followed by the definition and implementation of its various components in Chapter 5. In Chapter 6 the visualisation results are evaluated, and in Chapter 7 the general results are summarised and future extensions are discussed.

Chapter 2

Foundations

In this chapter the fundamentals of machine learning are explained. First ML and its model architectures are defined, as well as the specific functions of different ML models. Afterwards DNNs, their applications, and the training processes are discussed. Then the computational graph, which visualises the computations and the *forward propagation* and backwards propagation is described. Lastly, the structure of the ML and their interrelationships are reviewed and these concepts are outlined. Also the importance of graph drawing and KLighD is explained.

2.1 Machine Learning Architectures

The following principles of artificial neural networks align with Goodfellow, Bengio, and Courville [GBC16]. ML is the field of algorithms that learn from examples and experience, rather than relying on hard-coded rules, to be able to operate on new data. They employ mathematical computations which can be represented as directed graphs, also known as *computational graphs*. These graphs visually convey the structure and relationships within the data. ML models are mathematical structures that can learn from data and make accurate predictions or decisions. The landscape of ML models is diverse and includes supervised learning models such as linear regression and support vector machines, unsupervised learning models, and neural network models such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) [RR96]. In the context of DNNs, *model architecture* is the arrangement of layers, neurons, and connections that enables networks to learn patterns and relationships from data. *Deep learning* is a sub-field of machine learning that focuses on artificial neural networks and the processing of large amounts of data using DNN models [SA23]. This encompasses various types of ML models, including feedforward Neural Networks, CNNs, RNNs, and transformers. Supervised learning models are trained on labeled datasets, while unsupervised learning models identify patterns in unlabeled data. *Reinforcement Learning models* learn through trial and error. Specialised architectures, such as Generative Adversarial Networks (GANs) for synthetic data generation and autoencoders for unsupervised feature learning further contribute to the diversity of ML model types. Integrating computational graphs and automatic visualisation tools enhances understanding and interpretability of complex ML models.

The range of deep learning models is vast, encompassing various types tailored to different data types and tasks. Each model type has its strengths and application areas. The fundamental deep learning model is a *deep feedforward network* that consists of fully connected

layers and *non-linear activation functions*. CNNs are composed of convolution layers, non-linear activation functions and *pooling layers*. CNNs are commonly used for processing visual data, particularly in image recognition, due to their ability to extract features from images. RNNs, on the other hand, are well-suited for sequential data, making them ideal for language processing tasks where the order of information is crucial. RNNs have an internal state representation that enables them to store and utilise information about previous data points. RNNs are created by concatenating the intermediate results as a new input for the previous state. Long Short-Term Memory Networks (LSTMs), a specialised type of RNN, are particularly effective at capturing long-term dependencies in sequential data. This makes them well-suited for tasks such as natural language understanding and speech recognition. They go beyond having just short-term memory such as an RNN, by also having long-term memory.

ML models exhibit diverse functionalities, with decision trees adept at classification and regression tasks. DNNs play a key role in feature extraction, enabling the identification of intricate patterns in complex datasets [KNP+]. The range of ML models extends to supervised learning, where models are trained on labeled datasets for prediction or classification. Unsupervised learning models uncover patterns and structures in unlabeled data, providing insights into the inherent organisation of data. Reinforcement Learning models learn decision strategies through trial and error interactions with the environment. Specialised architectures such as GANs generate synthetic data, while autoencoders contribute to unsupervised feature learning, demonstrating the adaptability and versatility of ML to address a wide range of tasks and challenges.

2.2 Deep Neural Networks

A Deep Neural Network (DNN) is an *artificial neural network* with additional hidden layers, that is an interconnected group of artificial neurons organised in layers as mentioned in Section 2.1, and they are arranged hierarchically [AOM17]. This network structure is used to model relationships and solve various tasks. Due to the simple structure of an artificial neural network, consisting of only one or a few layers, they are mainly used for binary classification, simple pattern recognition or simple regression tasks [YWC20]. Hornik, Stinchcombe, and White [HSW89] also refer to them as *universal approximators*, which is yet another indication of their fundamental purpose. DNNs, however are used for classification, regression, pattern recognition, and feature extraction [SSR20]. Their ability to process large volumes of data and extract high-level features from raw input is one of their key strengths. DNNs have applications in various fields, including image recognition, natural language processing, and speech recognition [SSR20]. The depth of a DNN is determined by the number of hidden layers it contains. Each layer applies mathematical transformations to input data using weights and biases. By stacking multiple layers, DNNs can learn more abstract representations of input data, resulting in better performance in complex tasks such as image recognition [SCW+16]. This differs from an artificial neural network, which generally has only one hidden layer, while DNNs have a high count of hidden layers. The training of DNNs is accomplished through

backwards propagation and *stochastic gradient descent*, which are used to optimize the DNNs weights and biases [CXS+20]. Further explanations to the training of DNNs and computational graphs follow in the next section.

2.3 Training and Computational Graphs

DNNs are trained to minimise the loss. The loss measures how close the predicted outcome is to the true outcome. A smaller loss value indicates higher accuracy. To achieve this, a form of gradient descent is used, and the gradients of the weights of the neural network are calculated using backwards propagation. All trainable parameters, such as weights and biases, are adjusted by stochastic gradient descent in the direction of their gradients, which is towards the minimum [CXS+20]. In the context of DNNs, a computational graph plays an important role in training. The computational graph visualisation is used to automatically calculate gradients that can take on various values during training. This visualisation presents data on the loss. It provides a systematic representation of the calculations from the input data. This allows for a clear and concise representation of the model's structure and facilitates analysis and optimisation. By visualising a computational graph, one can navigate the complexities of ML models. It consists of nodes that represent mathematical operations or variables, and edges that indicate the direction of data flow. This graphical representation enables researchers to comprehend the dependencies between operations, facilitating the debugging and optimisation of ML models [CM10].

The forward propagation calculation predicts the results of the calculations, which are then compared with the expected results. The backwards propagation algorithm calculates the gradients for all learnable parameters by going backwards through the computational graph step by step. The training of DNNs therefore utilises gradients computed based on the loss to learn from the data and makes adjustments as it progressively reduces errors during training. This process is essential for training the model's result and ensuring it can correctly process unseen data. Visualising computational graphs helps to identify problematic patterns and analyse the training process at a higher level. A comprehensive understanding of the computational graph is important for various ML tasks, such as developing new DNNs and for general understanding when developing and teaching.

2.4 Graphs and Automatic Graph Drawing

Graphs are versatile structures used to model relationships between objects. In data visualisation, they are used to explore complex relationships, with vertices representing entities and edges representing connections or correlations between them. *Graph drawing algorithms* create planar embeddings of graphs with nodes represented by boxes and edges represented by arcs that present information in an informative pleasing manner [BBD+19].

These graphs can be used to visualise different aspects of the model, such as computational graphs illustrating the flow of data through the network, or *dependency graphs* highlighting

the relationships between different model components. Layout algorithms are often used to optimise the arrangement of nodes and edges, providing insight into the relationships within the DNN architecture. These graphical representations of complex structures make it easier to understand the nuanced connections within DNNs. Prioritising factors such as minimising edge crossing and improving overall readability are crucial in the visualisation process [DRS+15; KW01].

In the dynamic field of ML, a wide variety of model architectures are driven by computational graphs. Automatic visualisation plays a key role in revealing the relationships within these complex structures. As ML continues to evolve, the interplay between computational complexity and visualisation remains a paramount area of research and development, as highlighted by ongoing studies [BL10; BL09].

2.5 KLighD and ELK from the KIELER Project

KLighD is a framework which enables the translation of diagram data into the KLighD model in Java that can then easily be rendered. KLighD was developed to enable the creation of interactive diagrams with minimal time expenditure. It provides mechanisms for the simple definition of diagrams and their visual representation [SSH13]. KLighD provides different style options for changing the resulting graph model view. It also handles actions associated with interacting with graph elements. It facilitates the efficient rendering of diagrams with automatic layout and interactive functions. A KLighD visualisation is based on the KGraph structure, which provides an interface for defining models as a graph structure with rendering information and data for positioning the graph elements. KLighD is based on the use of the KGraph structure, which makes it possible to define models as graphs with specific rendering information. This structure provides a flexible way to represent different types of diagrams by using elements such as nodes, edges and labels. KLighD calculates the size of each element and then calls the automatic layout of *Eclipse Layout Kernel* (ELK).

ELK is a open-source framework for automatically generating the layout of graph structures [DHS+23]. By using ELK, graphs can be automatically laid out in a way that is clear and aesthetically pleasing. The ELK graph representation connects diagram viewers and editors [Bor19]. It offers features suitable for computational graph visualisations, including hierarchical nodes and layout algorithms [Pet19; Kas21; Sch16]. The software supports different types of diagrams, including hierarchical data flow diagrams, textual and visual representations, and edge-label diagrams placed within a layered graph drawing context [SWH18; Ren18]. It also emulates zooming during graph layout by scaling [KH23].

Chapter 3

Related Work

In the dynamic realm of ML, visualising the intricacies of DNNs and computational graphs is of importance. This section examines different existing visualisation tools, methods, and solutions that make a significant contribution to this area. It emphasises the importance of understanding DNNs and explores the need to move from manual to automated visualisation approaches. The PyTorchViz[1] and GraphViz[2] libraries are tools that provide visualisations.

3.1 Automated Visualisation of DNNs

Manual visualisation methods are traditionally used to understand the complexities of DNN architectures and to provide a quick overview on them. Hand-drawn diagrams and manually crafted graphs are foundational tools for conveying the intricate details of neural network structures [YCN+15]. It is important to acknowledge the historical significance of these methods and their contextual relevance in the evolving landscape of DNN visualisation. As the visualisations become larger, manual visualisation becomes a difficult task. Automated visualisation is then required to efficiently visualise complex or considerably large DNNs.

Automated visualisation tools brought a paradigm shift in understanding DNNs [FR21]. These tools are scalable and efficient, using sophisticated algorithms to create visual representations. By using automated visualisation tools, the architectures of DNNs are better comprehensible. This section explores the algorithms and methodologies of automated tools, showcasing the technological advancements that have propelled the field forward [ZTS+23].

The analysis considers both historical and current DNN visualisation techniques. The combination of manual and automated approaches contributes to a comprehensive understanding of the potentially large and complex structures within DNNs.

3.2 Existing DNN Visualisation Tools

The landscape of DNN visualisation tools is diverse, with a myriad of options available for researchers and practitioners. This variety leads to a lack of standardization and uniformity in visual representations across different tools. Each tool comes with its unique features, advantages, and potential limitations, contributing to the heterogeneous nature of visualisations in the field.

[1] https://github.com/szagoruyko/pytorchviz

[2] https://graphviz.org/Gallery/directed/neural-network.html

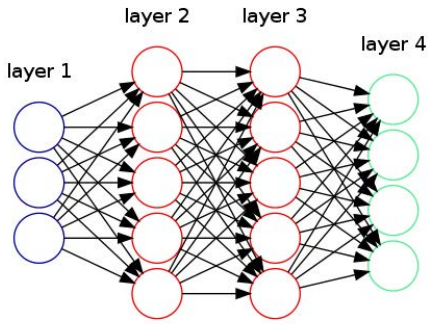


Figure 3.1. A GraphViz example of a neural network diagram for a multi-class classification problem, taken from [Zey24].

Tools like TensorBoard[3], hiddenlayer[4], and PyTorchViz[5] offer specific functionalities tailored to the frameworks they support. TensorBoard is primarily associated with TensorFlow but is also usable with PyTorch. PyTorchViz and hiddenlayer are designed for PyTorch[6]. Additionally, there are more general-purpose tools like Netron[7], GraphViz[8], which can be used to visualise computational graphs and neural network architectures across various frameworks [NHP+18]. However, the flexibility of these tools may come at the cost of specialised features, potentially impacting the depth and specificity of the visualised information. While these tools excel in providing detailed insights into the respective frameworks, their specificity can result in inconsistencies when comparing visualisations across different deep learning ecosystems.

The lack of a standardised visualisation format poses challenges for researchers and practitioners aiming to collaborate or transition between different tools. It emphasises the importance of understanding the capabilities and limitations of each tool in the context of specific use cases. As the field continues to evolve, efforts towards establishing conventions and interoperability standards could contribute to a more cohesive and unified visual representation of DNN architectures [Hay20].

3.3 GraphViz

GraphViz [EGK+02] is a generic tool designed to visualise graphs in a wide range of applications [GN00]. The generic nature makes it versatile, but it is not specifically designed for the structure and requirements of DNNs. However, there may be some practical considerations that

[3] https://www.tensorflow.org/tensorboard/graphs, https://www.youtube.com/watch?v=qEQ-_EId-D0

[4] https://github.com/waleedka/hiddenlayer

[5] https://github.com/szagoruyko/pytorchviz

[6] https://github.com/ashishpatel26/Tools-to-Design-or-Visualize-Architecture-of-Neural-Network

[7] https://github.com/lutzroeder/netron

[8] https://graphviz.org/Gallery/directed/neural-network.html

make it challenging to integrate with more advanced tools in some specific fields such as deep learning, where specific-purpose visualisation tools like TensorBoard are more commonly used due to their seamless integration with the deep learning frameworks. Nonetheless, GraphViz remains a valuable tool in many graph visualisation contexts [AAO20; Gan11]. The simple example in Figure 3.1 shows a neural network diagram that has three input units (purple) and 4 output units (green). The hidden layer units are seen in red in the middle. It displays the connections between the different layers and the high number of units that are used in a simple and quite small example. For a bigger visualisation, manual input of a visualisation is not favourable. A key issue is the manual configuration and the resulting complexity when applied to DNNs, which leads to GraphViz being a less preferred tool for visualising DNNs in Python. Visualising these complex network architectures often requires an automated and customisable solution [SBM+17].

Another crucial point is the availability of specialised alternatives in Python, which are better suited to the needs of DNNs. Tools such as PyTorchViz or TensorBoard are designed to interact with deep learning frameworks in Python and offer specific features for visualising DNNs. These specialised alternatives allow for more efficient integration and ensure that the visualisation meets the specific requirements of DNNs. The ML Python community prefers specialised tools that are better adapted to the needs of DNNs and allow a more efficient integration [VSS+19].

3.4 PyTorchViz

PyTorchViz is a dedicated tool for visualizsing PyTorch models that produces accurate and meaningful visualisations by using the general structures of PyTorch models [PWW+24; LLT23]. The precise coordination with the framework is undoubtedly a strength when it comes to precise visualisation of PyTorch models.

PyTorchViz focuses on the visualisation of computational graphs, which makes it a useful resource for users who need exactly this feature. However, this focus has its limitations. In particular, with more advanced analysis, metrics tracking or real-time updates PyTorchViz is limited. Another drawback is its incompatibility with other deep learning frameworks. This limitation may pose a challenge for users working in different frameworks, such as Jax which is a relatively new deep learning framework that might gain traction in the future[9] [KBH+24]. Another limitation is that you may want to switch between frameworks, which may affect the applicability of PyTorchViz.

Figure 3.2 illustrates a simple example of a LSTM visualisation from the PyTorchViz tool. The grey boxes contain the operators collected during forward-propagation, such as *AccumulateGrad*, *TBackward* or *AddmmBackward* and can be recognised and distinguished. The inputs are shown as a blue square and the end output as a green square. The specific calculation steps are explained in more detail in Section 5.3.

[9] https://github.com/google/jax

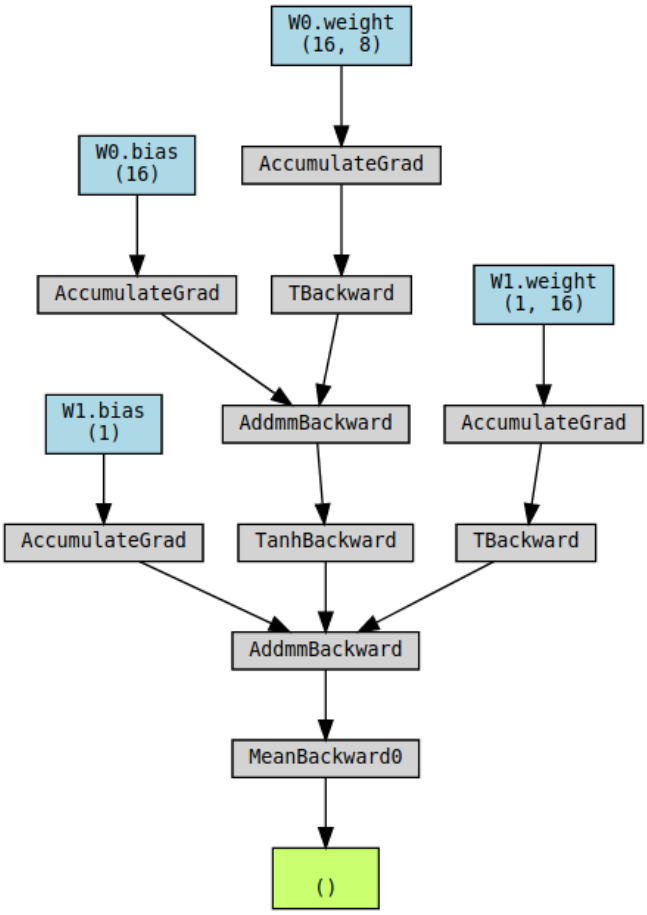


Figure 3.2. A visualisation of a computational graph by PyTorchViz, taken from [Git24].

The integration of PyTorchViz with PyTorch provides a useful resource for users who need to understand how data flows through the network and how computations are performed [SBM+17]. Powered by the PyTorch framework, PyTorchViz interacts with the internal structures of PyTorch models to produce accurate and meaningful visualisations. Developers and researchers using PyTorch as their preferred framework benefit from the specialised nature of PyTorchViz, as it provides optimal support for their specific requirements.

Chapter 4

Deep Neural Network Visualisation Tool PyTorchKGT

This chapter presents the concepts, ideas and discussions about how the implementation and its parts should work in different levels of detail before looking at the implementation itself. Since the implementation is divided into a KGraph component and a tool implementation part, it shows the concepts of each part and the communication between them. The concepts on how to display the visualisation and how to form the different design choices are described. This chapter starts by looking at the general structure of the concept of graph generation and the general steps needed to achieve a visualisation.

4.1 Visualisation tool architecture

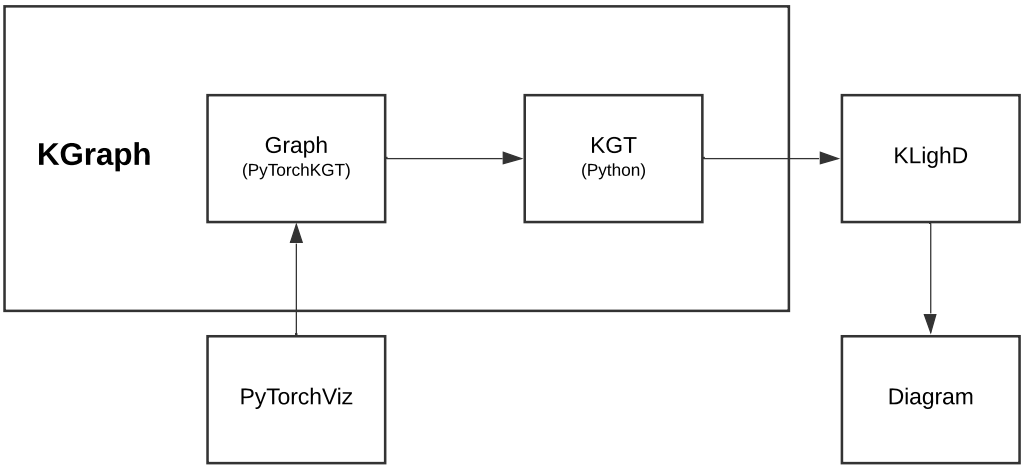


Figure 4.1. Overview of the tool's general process from input to the representation.

Figure 4.1 provides an overview of the concept of this thesis. The PyTorchViz tool is adapted to interact with the developed graph structure and can be used as an input. Other tools can be connected to the graph structure, and further applications occur automatically. The graph structure is at the center and outputs a KGT structure, which can be displayed in

the KLighD application. The main task involves designing a Python tool that can be used to generate visualisations.

4.2 Design Choice for DNN Visualisations

The design of a visualisation is crucial in drawing the user's attention to the most important and essential areas, as the human eye can only focus on small areas and perceive a rough form of the overall visualisation. Various aspects of visualisation optimization can individually enhance a visualisation, including the *Gestalt laws* and the *data ink ratio* [McG15]. The following section explains the different Gestalt laws.

Conciseness Perceiving and interpreting complex images as the simplest forms as possible.
Proximity Elements that are visually close to each other are related.
Similarity Elements that look like each other in size, colour or shape are related.
Connection Elements that are visually connected are related.
Enclosure Elements that are separated together are related.
Symmetry Elements that are symmetric can be perceived as forming a visual connection.
Figure / Ground Elements are perceived as either figures or background.
Common Fate Elements with the same moving direction are perceived as a unit.

The Gestalt laws are used to comprehend how users perceive and interpret visualisations, allowing to determine which visualisation is most effective. To prevent visual overload, we aim to use minimal ink in our visualisations. The data ink ratio can be used to determine the amount of ink required to represent data without sacrificing information. This ensures that the user is not overwhelmed. The visualisations in DNN contain significant information that should be presented clearly and objectively. To adapt the visualisations to their purpose, several visualisation options should prioritise relevant information. This requires a graph visualisation that standardises the basic structures of DNNs to reflect the different visualisation priorities. The data is visualised in a left-to-right flow to make it more legible for the user, as it represents a process and structure that is common for them. To ensure clear and concise visualisations, it is recommended to use basic structures with straight edges [BVB+11]. This allows for easier comprehension by the user while maintaining symmetry.

Nodes should be represented as squares containing information. Nodes with similar tasks should be placed close to each other to fulfil the principles of enclosure and similarity. The nodes should then be connected by edges using an asymmetric relationship, as it creates a flow in one direction. The principle of connection is highlighted here, and one-sided arrows are used for all connections that also receive similarity. These principles can be applied to distinguish inputs and outputs, providing nodes with conciseness and a sense of common fate. To highlight important content, additional colours can be used, such as blue and orange, taking into account colour-blindness. Another way to achieve this is by using different shapes for the nodes. However, visualisations that contain too many shapes and colours can become confusing and lose their purpose, as explained by the data ink ratio. Therefore, it makes sense to use only one variant.

The same specifications apply to a visualisation of DNNs, and therefore also to a visualisation of a computational graph. Due to the potentially enormous size of DNNs, it is more practical to use colours rather than different shapes for specification purposes. This helps to avoid unnecessary enlargement of the visualisation. Additionally, since DNN and computational graph visualisations include inputs, intermediate results, and an output result, an additional colour, such as green, is needed to represent the input result. Colours can also be used to distinguish different areas of the visualisation at greater distances.

4.3 Determination of PyTorchViz as Tool of Choice

As stated in the overview, the Python code should use the realised function from a DNN model and output an automated visualisation as KGT. To achieve this, the DNN data model must be extracted. Since there are existing tools available to visualise DNNs and extract important information from the code, these can be used as a base for importing data into existing software. As the KIELER project currently lacks an application to convert DNN Python code into a KGT visualisation, a transition must be constructed manually.

Existing tools have been surveyed and a promising candidate has been selected, to extract the computational graph elements and their connections.

Several authors [NHP+18; PWW+24; LLT23] have discussed the usefulness of PyTorchViz in visualising PyTorch models. However, PyTorchViz's specificity to PyTorch may limit its functionality and hinders its compatibility with other deep learning frameworks as mentioned before. Nevertheless, PyTorchViz is a promising starting point for the development of new tools for visualising the complex structures of DNNs. Python is a widely used language for DNN development, and PyTorch is one of the most popular deep learning frameworks in Python. PyTorchViz takes advantage of these two popular approaches and combines them. In addition, PyTorchViz uses the DOT visualisation representation, which is commonly used to visualise neural networks in PyTorch. The tool visualises the graphs and outputs them as a PDF[1]. The tool allows the extraction of the computational graph from a given model. The model data must be prepared for transformation into the KGT format. This involves ensuring that the data is structured correctly and follows the necessary conventions.

4.4 Computational Graph Visualisation

This section describes the steps of extracting DNN models using PyTorchViz and visualising their computational graphs. It considers properties such as input and output representation, as well as visualisation readability that is necessary for effective visualisation.

The first step is to extract the DNN models. As further explained in the Section 4.5, this was done by using parts of an existing tool PyTorchViz. This extraction and use of the obtained DNN model data leads to the generation of the graph. This thesis uses Python as the main

[1] https://github.com/szagoruyko/pytorchviz/blob/master/examples.ipynb

language and outputs a KGT file that can be automatically visualised by KLighD, for example with the *Visual Studio Code* extension[2]. This can display the KGT file and the visualisation at the same time.

To take a closer look at the development of the overall structure of the process, it is important to understand what requirements passed to the final development structure. During the research, it was clarified that there were many different possibilities with different priorities. However, each of these visualisations also has similarities. In order to obtain the most appropriate visualisation, the most important requirements for this thesis are as follows.

Requirement 1 Compatibility with the structure of a KGT format should be ensured. Which does not rely on other modules from other frameworks.

Requirement 2 The input, output and hidden layers should be visualised. This should ideally be done in different nodes.

Requirement 3 Since the DNN development proceeds mainly in Python, the implementation should be written in Python too. Furthermore, the widely used Python PyTorch neural network models should be a possible input, as this is used in many areas.

Requirement 4 The different layers must be clearly organised. The individual hidden layers as well as the labelling of inputs and outputs. The data used should be presented efficiently and visually while informative. It should be possible not to display sub-areas in order not to see unnecessary or extra information.

Requirement 5 Due to the flowing structure, it should be possible to display this visualisation horizontally, as this intuitively supports the reading direction from left to right.

A visualisation of a computational graph should include input, output and the hidden layers, which can display some data as well as different information. Displaying anything other than the input and output tensors is not essential for the hidden layers. However, it is important to display a representation of the backwards propagation calculations as this is a significant and interesting area.

Variants that only display a basic structure and show further data through hovering over them, fulfil requirements 3 and 4, but not the other requirements. Especially the requirement 1 that is essential for the project.

Another possibility is to visualise the basic structure in nodes and display the values saved in the nodes next to them. However, this often leads to a confusing graph, because of the extra information than is not necessary. Therefore, the requirements 1, 2, 3, and 5 are satisfied, but not 4.

PyTorchViz is a visualisation tool that displays data only when results are given and can show additional data when needed. It displays the basic structure as a graph drawing in a visualisation of a vertical orientation and meets the requirements 1, 2, 3 and 4, but not 5. Despite the vertical orientation being hard to read, PyTorchViz provides a solid basic structure. However, it should be noted that the color scheme used is not suitable for those with color blindness, deuteranopia and tritanopia, which should be taken into consideration

[2] https://marketplace.visualstudio.com/items?itemName=kieler.klighd-vscode

when creating effective visualisations. Furthermore, large visualisations cannot be displayed on a single page and are cut off. The tool uses a customized structure to explore the required data from the model and generate a graph. The graph structure follows that of KLighD, which is used for visualising the KGT. To convert the Python KGraph to a KGT representation, the Python KGraph structure must be synthesised. The components of the Python KGraph structure are exported text in the order of the tree structure. The main KGraph element contains all nodes of the graph. The nodes can contain edges, labels, and any other rendering or style of all elements inside them, as well as other nodes. This code generates a recursive structure of calls, creating new elements for each run. The structure of the resulting elements resembles a tree, allowing for iteration over individual elements using depth-first search. The KGT structure contains the data that builds the visualisation. It includes nodes surrounded by a rectangle, which can hold varying amounts of information. The nodes also connect to the next nodes in the process and other connections. The visualisation displays both the inputs and outputs, as well as the backwards propagation results as shape representations of the calculations, as expected from requirement 4. The edges are arranged straight and evenly to clearly show connections and make it easier to recognise assignments. A generally uniform visualisation of the nodes and edges is maintained to ensure a clear structure and sequence of events. The use of only a few colours ensures clarity and distinguishes the individual different areas of the layers. If a graph is overloaded with visualisations of colours, shapes, and alignments, it can become confusing and difficult to read. For this reason, the colour scheme is divided into the main colours black and white, blue and orange appropriate for the colour-blind. The inputs and outputs are coloured because they are given more relevance. A solution against a crowded visualisation should be available.

A visualisation meeting various criteria for a useful visualisation and reflecting the differences in the visualisations is required. The user can generate different visualisations and customise them through input variables. The normal view does not display any additional values, so no further user input is required. The **show_saved** input variable can be used to add results to the visualisation. The representative values of the backwards propagation are displayed as additional KNodes at the KNodes from which they originate. The **show_attrs** input variable can be used to display the values available for calculations in the KNodes, which are represented within the KNodes themselves. To obtain a compressed view of the computational graph visualisation, the **resultView** input variable can be used. This will display only KNodes that are inputs, outputs, or have computation outputs. Each input variable can be linked individually, allowing for a compressed view without the need to view the results.

4.5 Architectural Graph Visualisation

The architecture of a DNN graph can be visualised by converting it to a graph that then can be visualised, identical to the KGT visualisation. Since Hiddenlayer [Fer18] is a tool that can already perform such visualisations and extract data from PyTorch neural network models, it

is a good starting point for a visualisation tool. It uses a combination of PyTorch, TensorFlow and Keras [Fer18]. TensorFlow was used for the extraction from the neural network model data. Other models also visualise these architectures similarly or with other extraction tools like ONNX[3]. They are used to extract the Data from function of the neural network modules, like `forward(self, x)` seen in Figure 4.2 code example.

Hiddenlayer has not been updated in several years so, therefore this tool can no longer be used effectively. TensorFlow is another deep learning framework that also aids with the model data transformation, but it has changed in a way that requires the program to be reworked in order to function properly. Furthermore TensorFlow's transition to version 2.0 made Hiddenlayer incompatible. For this reason, we opted for an alternative approach without the use of a tool and developed a new program.

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

Figure 4.2. PyTorch neural network module model.

From the given neural network module code, the model can be interpreted and the differed layers of the architecture visualisation extracted. The extracted data from the PyTorch neural network model, is structured as follows.

The `__init__()` creates the model including all its layers, such as convolutional layers, and their distribution in a corresponding variable. These can be extracted using the PyTorch framework. The method `forward(self, x)` is the forward-propagation and establishes the connections between the layers. To reduce the amount of data and improve computational efficiency, pooling functions divide the input volumes into smaller regions and select the maximum value for each region as output. For example, the *maxpooling*, which can reduce the dimensionality of imported data without loss, making it more efficient and easier to process[GK20]. Non-linear activation functions enable the neural network to model complex patterns and structures in the data, thereby enhancing the networks performance and effectiveness in solving complex problems. To extract functions not stored in any variable, a TorchDynamo-based ONNX exporter can be used. This exporter transforms the functions

[3] `https://onnx.ai/onnx/intro/concepts.html`

into an ONNX graph. An ONNX graph is a format used to describe the structure and model of a neural network in a machine-readable format. It makes the provided layers and functions from the PyTorch neural network module accessible as variables. That is necessary to extract the pooling and the non-linear activation functions data from the forward function, because these are not stored in a variable. The KGT version should be created from this model. A more unconventional and unstable way is to read this textually from the forward function, which was used in this prototype. This approach may encounter difficulties when dealing with more complex neural networks. Therefore, it is necessary to use the standard specification of the model and convert extractions clearly. It is important to avoid using invented specifications with loops, external variables, and methods as they can make the process more complex and less comprehensible.

Chapter 5

Implementation

In this chapter we discuss various implementation aspects of the PyTorchKGT tool in detail and explain their implementation to illustrate the functionality of the components.

The program executes Python code to generate an automated visualisation in the form of a KGT, based on a given PyTorch neural network model. Figure 4.1 illustrates the structural setup, which is a Python code input that is transformed into a graph structure inside of PyTorchKGT. This graph structure is designed to match the graph structure of KLighD. KLighD can then work with this KGT.

The graph structure includes the specification and implementation of the *KGraph* class, which forms the basic framework for the graphical representation of deep learning models. The modified PyTorchViz implementation and functionality, which had to be adapted for the further integration of their applications, is also discussed. Furthermore, a prototype is presented to demonstrate how architecture graphs can be visualised.

5.1 KGraph Format

This project combines the visualisation of DNNs with KLighD, which is designed to create effective visualisations. The class diagram of the KGraph model is presented in Figure 5.1. The KGraph model consists of different parts, with the nodes *KNode* and the edges *KEdge* being the basic elements of a KGraph. Nodes may contain ports for input and output. These elements belong to *KLabeledGraphElements*, with labels *KLabels* used to attach text. Additional customization to the KGraph can be made with *KGraphElements*. KGraphData and *KRendering* allow for changes to colour, shape, and position. The project implementation involved retaining the existing structure of the Java implementation. This approach allowed to benefit from compatibility with KLighD, resulting in a consistent and well-structured visualisation solution. To implement this structure, the existing class division was maintained and the necessary parameters were divided into the individual classes. Specifically, the edges were modeled as children of the node similar to the KNode. It is important to include this information in the KGraphData class so that the data is consistent with the visualised data.

5.2 KGraph Specification

When implementing the Python data structure to store the DNN model, the same structure as in KLighD was used. Examining their structure reveals similarities and highlights fundamental

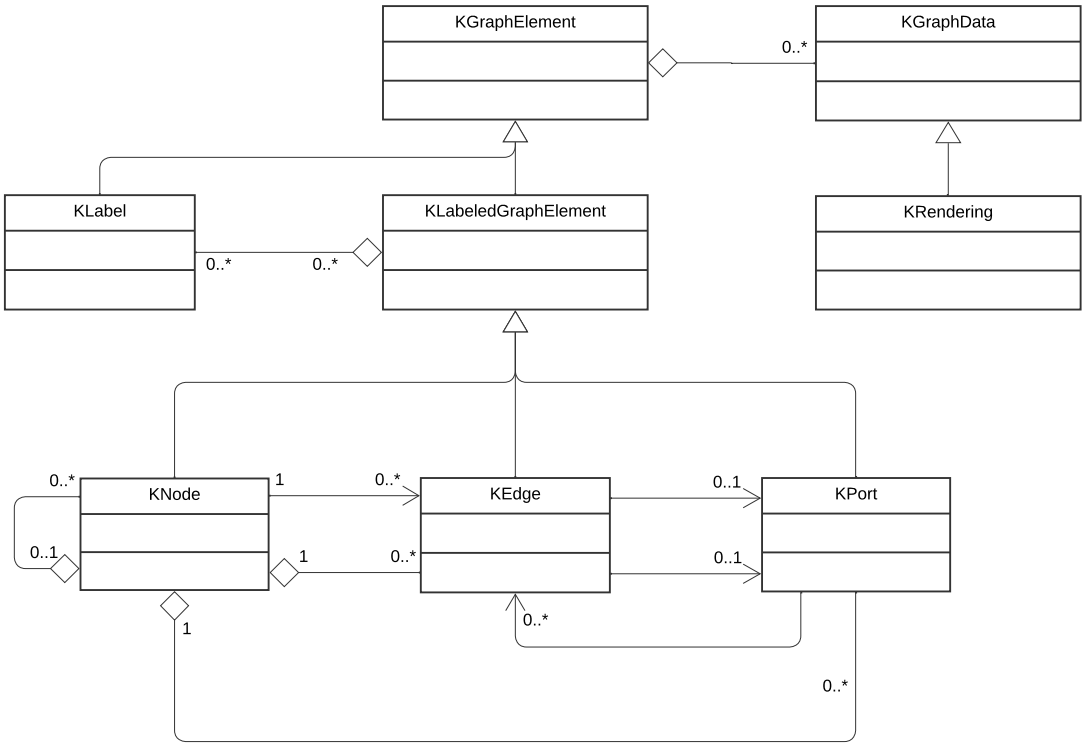


Figure 5.1. Class diagram of the KGraph structure from the KLighD implementation [Spi14].

differences in their design. Figure 5.1 illustrates a class diagram of the KGraph model view. The KGraph is modeled with the shown classes.

KNode A KNode in the KGraph is modeled to contain further KNodes (children), KEdges (target) and KPort connections (ports). Nodes are the core of the graph structure and enable the visualisation of nested graphics and network topologies. It can also have a KLabel (label) that shows names or other information of a KNode.

KEdge A KEdge is a representation of a connection between KNodes. They have a target reference and an optional port reference. It can also have a label similar to the KNode.

KPort A KPort is a connection point on a KNode. It can be used as a source port or a target port for KEdges. This helps to group KEdges at one point or to connect children KNodes through a specific point.

KLabel A KLabel is added to the object to add infomation that is then shown at the visualisation stage. It is always optional to give an object a KLabel but is availabe for children of the KLabeledGraphElement.

Using KGraphData and KRendering more options to improve the KGraph exist. The KNode has mutable options to extend the features used for improvement. The Figure 5.2 gives an overview of the classes.

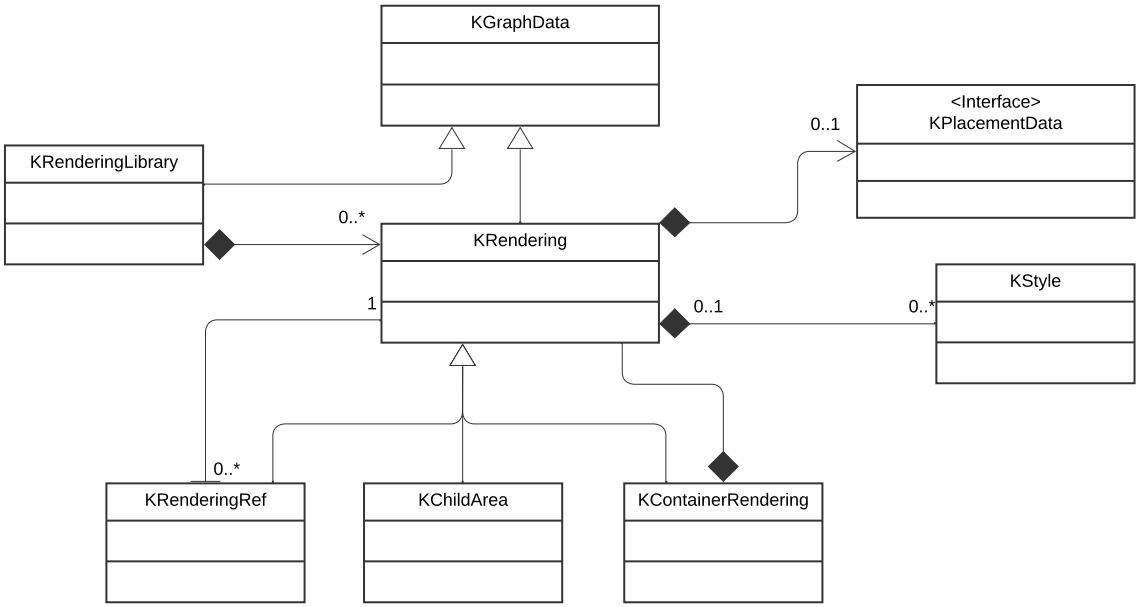


Figure 5.2. KRendering class diagram from the KLighD implementation [Sch14].

Several classes extend KRendering for style and rendering options for modifications. The Figure 5.3 and Figure 5.4 provide a more detailed view of the KContainerRendering and KStyle classes, which contain the most prominent changes between KLighD and PyTorchKGT. Another main difference is the KStyle on KRendering Layer, which is shown in Figure 5.7. Below, the different sections in Figure 5.2, Figure 5.3 and Figure 5.4 are described.

KRenderingRef A KRenderingRef contains a reference to KRenderings in the KRenderingLibrary. However, the KRenderingRef only contains an ID, which is an indirect reference to the corresponding rendering in the KRenderingLibrary.

KChildArea KChildArea rendering is used to specify where the children of the current KGraph element should be placed. This implementation simply calls the function to generate the view for each child element.

KContainerRendering KContainerRendering is never directly created, only child classes of this class can be rendered.

KText KText is a textual element that allows for KRenderings to add descriptions.

KStyle KStyle is extended by classes that allow for the rendering of objects with additional features, such as rotation or colouring. It is possible to style any KRendering.

KRotation KRotation is a subclass of KStyle that rotates the KRendering around an anchor point, with the angle specified as a property.

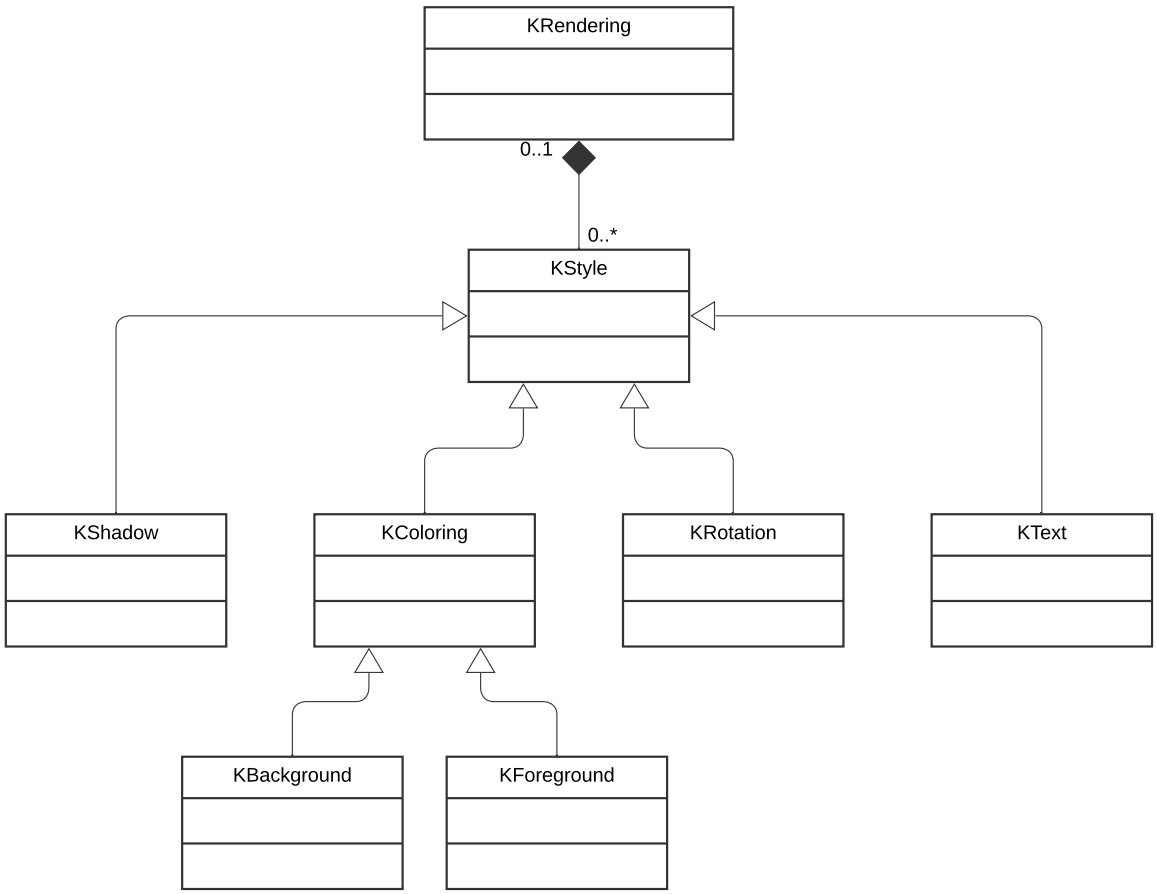


Figure 5.3. KLighD project KStyle class diagram.

KShadow KShadow is a subclass of KStyle that adds a shadow to the KRendering, making it stand out more. The colour defined in the shadow is applied to each copy with decreasing levels of opacity.

KColoring KColoring is a subclass of KStyle and contains the colours in the RGB colour codes.

KBackground KBackground is a subclass of KStyle, it fills the KRendering without the objects itself.

KForeground KForeground is a subclass of KStyle, it sets the colour from the text in a KForeground style.

The subclasses of KStyle provide several options to regulate the amount of information displayed in order to manage visual overload, these options are presented in the KGT. The options can be applied to all objects that have been rendered, because it extends from KRendering. As KLabels are used from KNodes, KEdges und KPorts, an additional KText is typically not necessary. However, in certain circumstances, it may prove beneficial, and is possible for every KRendering individually. Classes that are less relevant for this visualisation

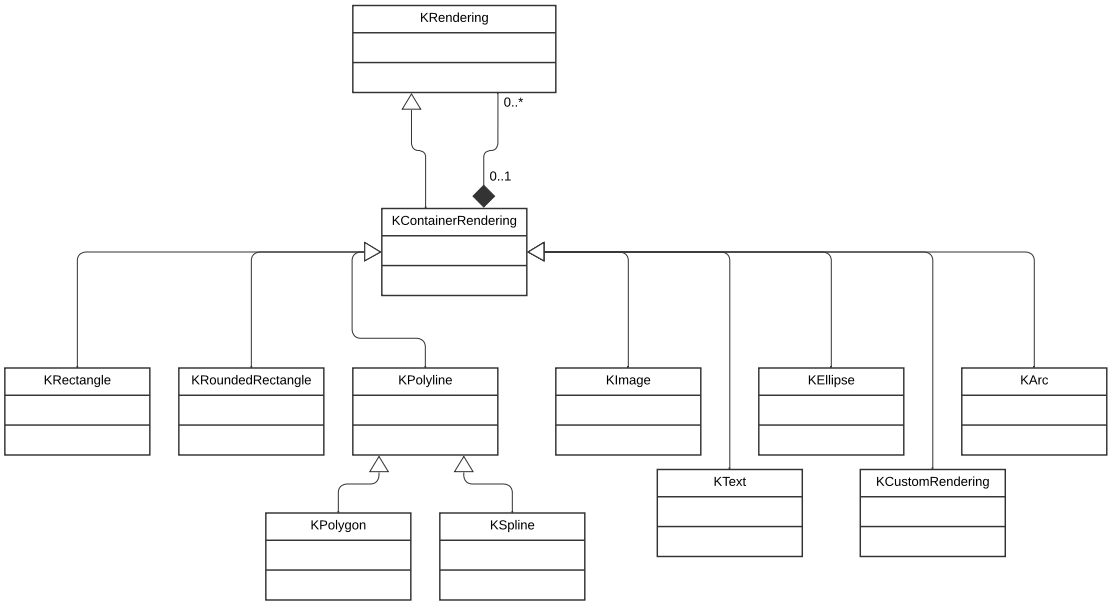


Figure 5.4. The class diagram of the ContainerRendering.

which can also be added but have not yet been implemented, include KImage, KArc and KCustomRendering. They are included in the KGraph implementation for completeness, however, they will not be explained here or used in the PyTorchKGT visualisation.

KEllipse The KEllipse is a subclass of KContainerRendering and is a ellipse element. The radii in x and y direction are calculated from the bounds' width and height.

KRectangle The KRectangle is a subclass of KContainerRendering, it is a rectangle with sharp corners.

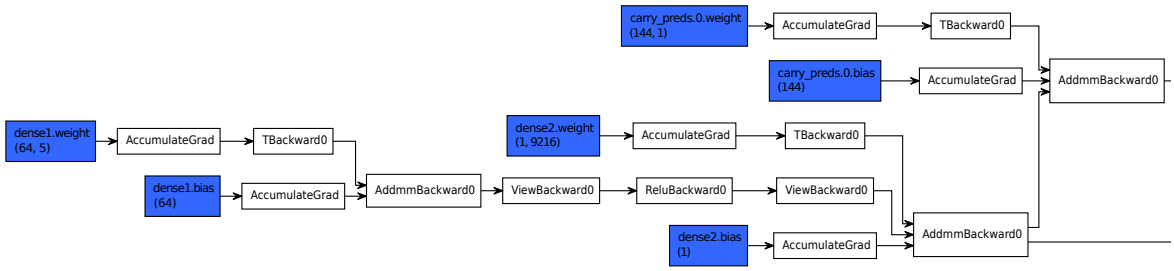
KPolyline The KPolyline is a subclass of KContainerRendering, it is a line that connects objects.

KPolygon The KPolygon is a subclass of KContainerRendering, it is a closed polyline.

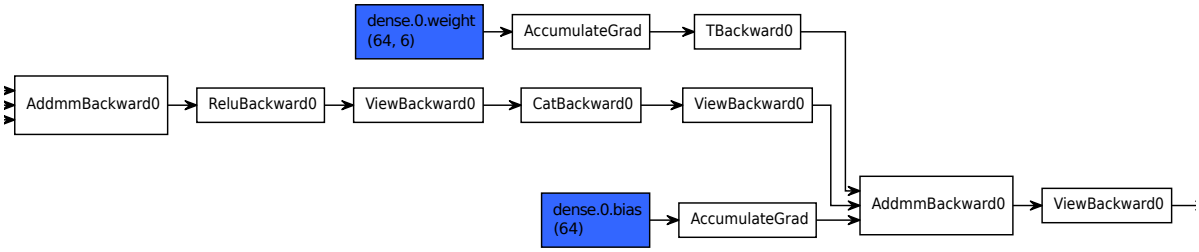
KSpline The KSpline is a subclass of KContainerRendering and is a polyline with curves that are smoothed.

KRoundedRectangle The KRoundedRectangle is a subclass of KContainerRendering, it is a rectangle that has corners that are rounded.

As opinions differ as to whether colour differences or shape differences provide better clarity, both options are offered here. Colours can also be used in different ways to cover KBackground and KForeground.



(a) LSTM start section.



(b) LSTM middle section

Figure 5.5. PyTorchKGT visualisation of an LSTM start and middle section.

5.3 PyTorchViz Node Definitions

Figure 5.5 shows a start and middle section of an LSTM with the different computational steps as nodes with their specific names. These are used and adapted by PyTorchViz from the PyTorch model. They are static and cannot be changed during the visualisation process. The function of each node is explained below. The visualisations are showing a LSTM visualisation form the PyTorchViz tool for the explanations. The Figure 5.6 shows the same visualisation as the Figure 5.5, however with shown results.

The following terms are typically related to the implementation details of a deep neural network framework, particularly when using automatic differentiation to compute gradients during the training process. These are shown at the graph visualisation of a computational graph with PyTorchViz.

AccumulateGrad This refers to the process of accumulating gradients computed during backwards propagation. In deep learning frameworks, gradients are computed for each parameter during backwards propagation and accumulated over multiple iterations before applying them to update the model parameters.

TBackward This represents the backwards propagation through a transposition operation.

Transposition involves flipping the dimensions of a tensor, and during backwards propagation, gradients are computed with respect to the input tensor based on the gradients of the output tensor.

AddmmBackward This corresponds to the backwards propagation through an operation that involves adding a matrix-matrix multiplication to another matrix. During backwards propagation, gradients are computed with respect to the inputs of this operation, which may include gradients with respect to the two input matrices and the bias vector of these.

CatBackward This relates to the backwards propagation through a concatenation operation. During backwards propagation, gradients are computed and propagated back to the input tensors that were concatenated together.

ReluBackward This represents the *Rectified Linear Unit* (ReLU) activation function during the backwards propagation. Throughout the backwards propagation, gradients are computed with respect to the inputs of the *ReLU function* based on the gradients of the outputs. There are other activation functions that are used instead of the ReLU function. This is one of the most commonly used activating features available.

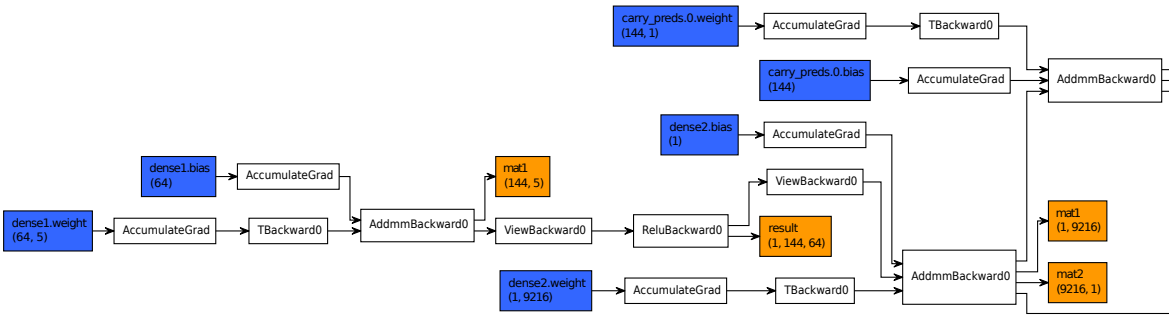
ViewBackward This is an operation that is performed during backwards propagation, which reshapes a tensor and views it differently. In backwards propagation, gradients are calculated based on the gradients of the reshaped tensor and propagated back to the input tensor.

MulBackward This represents backwards propagation through a multiplication operation. In backwards propagation, gradients are computed with respect to the inputs of the multiplication operation, based on the gradients of the output. The use of automatic differentiation to compute gradients during backwards propagation and the updating of model parameters during training.

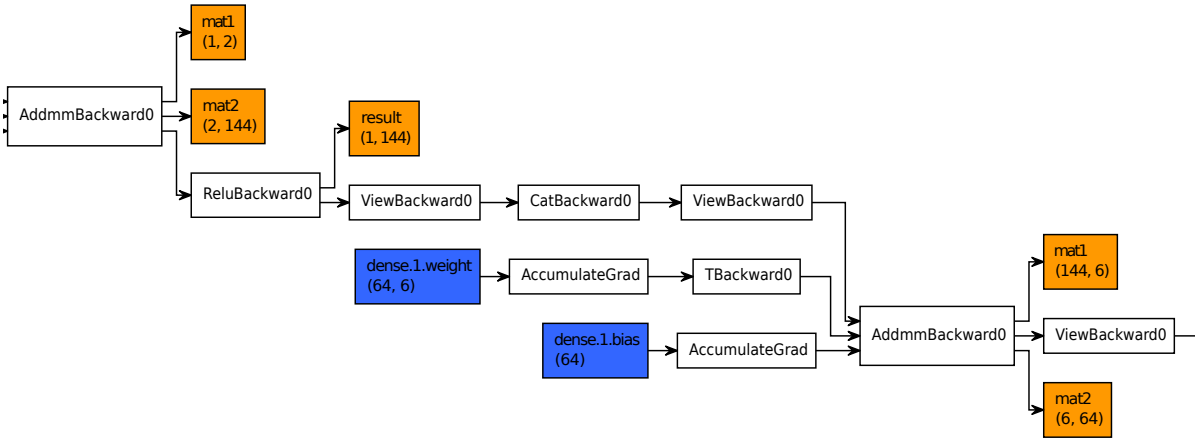
Weight() These are central components in neural networks, representing the strength of the connection between neurons. The dimensions given in parentheses are weight matrices. The first value describes the size of the input feature vectors. The second value indicates how many outputs or targets they are connected to. In the case of an LSTM, it indicates the connection weights between the input data and the internal gates or cell states.

Mat() These are the values of tensor operation, also known as matrix-vector multiplication. The dimensions are given as values in parentheses. They indicate the size of the tensors or input/output. In an LSTM, this indicates the size of a particular input or output vector in the network, where the first value describes the lot size, the number of instances, and the second value represents the features or dimensions of the vector.

Result() This output represents the dimensions of the results (outputs) of certain operations within the network. The parentheses can contain different numbers of values and therefore dimensions. The first value is the batch size of the input instance being computed. The second value describes the number of hidden cells per layer at a given time. The third value represents the level of the data structure, which denotes additional features, output units, or the number of outputs per time step in a sequential process. Three dimensions



(a) LSTM start section.



(b) LSTM middle section

Figure 5.6. PyTorchKGT visualisation of an LSTM start and middle section, with results shown.

indicate a richer or more complex data structure and fewer dimensions indicate already simplified or transformed values.

Bias() These are bias vectors that are added to adjust the output of each neuron before the activation function is applied. The parentheses can contain one or more values. One value indicates a one-dimensional quantity that determines the length of a bias vector. Two values indicate tensors with rows and columns representing input and output dimensions. The larger these values are, the more neurons are used.

5.4 KGraph Definition and Implementation

The KGraph implementation is based on the structure of KLighD, as was explained in Chapter 4. The newly implemented Python datastructures are named after the KLighD classes. In the following paragraphs the Python classes are described.

The KNode and KEdge classes are the main objects, as ports are not necessary in most DNNs and are therefore not included in the implementation of this work. Elsewhere ports can improve the visibility for visualisation with the control of the connection of edges to nodes. As edges require a starting KNode and there are no ports available, the KEdges must also be created within a node. This may vary for certain edges if ports were present. This KNode and KEdge connections results in a tree structure.

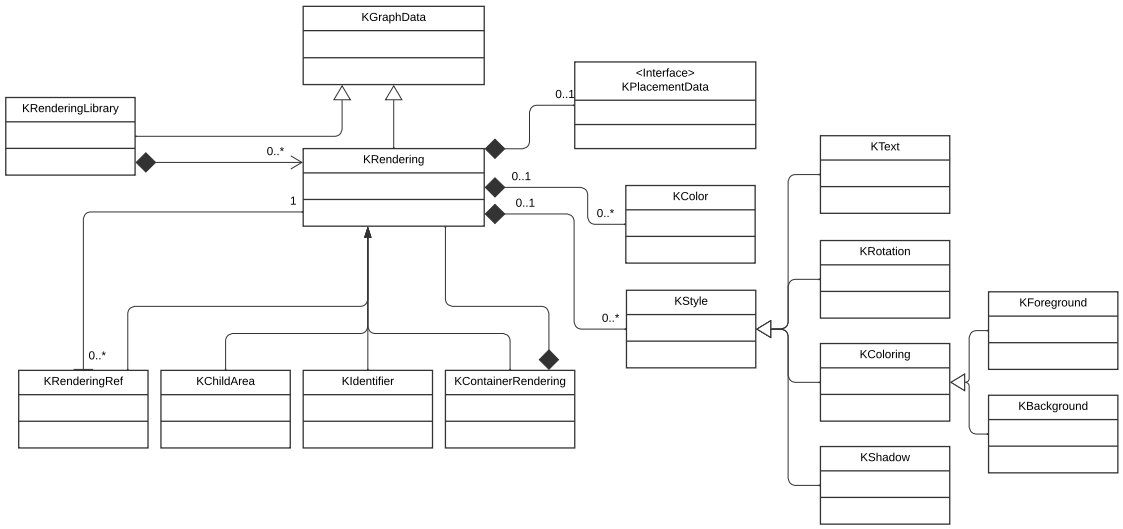


Figure 5.7. The class diagram of the KRendering and the KStyle classes.

The structure of the KRenderings is replicated and can therefore be applied to any KNode. KColor extends this by allowing the use of RGB colour codes, enabling the integration of desired colours in the model. The KStyles are defined in the KGraphData area for a more

compact view and retain their function, however they are not used in the current version of this implementation. Similarly, the various KContainerRenderings are not yet used, with the exception of the KRectangles.

The KStyle not only extends the KGraph, but also the KGraphContainerRendering. Figure 5.4 and 5.7 are showing specific areas that are explained, which are related to the shapes of the borders of the KNodes and the KEdges. As previously mentioned, classes such as KImage, KArc, and KCustomRendering, are less relevant and have not been implemented.

The current implementation is using KRectangles and KColor, but depending on preference a different style could be used. The various extensions of KGraph objects do not include KLighD interactions.

make_KGT The *make_KGT* function, to visualise a neural network, is called with a neural network model as parameter. When using the *make_KGT* function to build a KGT, the KGT file is accessed to create a KGraph element. Upon creating the first KNode, a KRectangle is also generated. The necessary data records are transferred via lists. The *getId* function and *getData* function can be used to identify KNodes that are not the current KNode and to establish connections. This is necessary not only to adapt to the KIELER data structure but also for the exporting functions. During the export functions, the references of multiple KNodes are needed to write the KGT and the export function *export_KGT* is called in the process. To accurately translate particular objects into KGT format, it is essential to stick to the specified sequences. Additionally, due to the nested structure, a depth first search is necessary to retrieve all connected parameters. The following steps will explain this process in detail.

export_KGT The function exports a description of the KGraph type by taking an outer node as input. The KGraph type is represented as a string in the result, containing the complete structure, including nodes, edges, and their respective properties and data. The function iterates over the children of the outer node and then calls the *export_node* function for each of these children to enable the entire structure to be exported.

export_node This function exports a detailed description of an outer KNode to the KGraph. The KNodes comprehensive description includes, its different data and labels, as well as its outgoing edges and child nodes. For each child node, the function recursively calls itself to export the entire hierarchical structure.

export_data This function exports the description of the data to be added to the KGraph, such as KSplines or KRectangles and their additional data.

export_styles This function exports styles in the KGraph. It iterates through the various types of styles, including background colours, and records their properties, such as colour and transparency. The export function is called by other functions in order to integrate the style properties into the specific part of the overall export.

export_label This function exports the label description to the KGraph as a string. The function only includes the label text as a string and is used by other functions to insert label information.

export_edge This function exports the description of a KEdge in the KGraph. Both the source and target nodes of the edge, as well as the data and labels associated with the edge

are taken into account. The nodes can be identified and connected by their IDs. Like other functions, a recursive export function call is used to export all edge information.

These functions are linked recursively and continue to call each other until the result is returned. The KGraph is created by the outer KNode with the necessary properties and is then exported. These properties include the default KGraph synthesis, node size, and node label placement, providing the foundational configuration for the KGraph. The KNode begins its recursive run by executing *export_data*, *export_styles*, *export_label*, and *export_edge*. Then, *export_data*, *export_styles*, and *export_label* are called again for the KEdges before moving on to the next KNode and repeating the process from the beginning until the entire structure has been processed. Additional export functions can be added to the required position and are inserted into the chain of calls with another function. Indentation is used to improve the readability of the KGT file.

5.5 PyTorchKGT Implementation

To use the *make_KGT* method a PyTorch neural network module is necessary. The required shapes are specified there, and if needed, changes can be made to the structure used in the forward function. If a neural network module already exists, it can be used as the first input. Additionally a dictionary with label translations can be passed in to rename certain information contained inside the model. The attributes *show_attrs*, *show_saved*, and *resultView* are Booleans that specify which visualisation to create. If none of the values are set to True, a calculated graph is created without any outputs or values. Only the input instances will be visible.

show_attrs displays the values contained in the nodes, including the mathematical values, tensors, and whether a result has been generated for this KNode.

show_saved attaches the calculation results to the KNodes and outputs the corresponding values. The different variants were explained in more detail in Section 5.3.

resultView provides a compressed view that only displays the nodes and their connections, if they have a calculation result, hiding all other intermediate steps. These specifications can be combined to meet individual requirements.

If the parameters are not passed, they will not be visualised. To create a graph, first, use `add_base_tensor(v)` to add the parameters of the PyTorch neural network to the new KGraph. Each KNode is represented by a KRectangle that outlines the information it contains. To maintain the hierarchy, the graph is used as the parent for the start node. The other data in the tensor is then appended if further information is available. Furthermore, the following nodes are called using the function `add_nodes(var.grad_fn, node, node)`. Before adding this function to the graph, several operations must be considered. If a node already exists, an edge is added to it.

The implementation starts for adding a node, with appending the node and giving a unique identifier for naming purposes. To ensure comprehensibility, a KLabel containing the tensors *var_name* is added. Additionally, an edge is included that points to the preceding node.

This is essential because the tensors are listed in reverse order during backwards propagation, and thus the direction of the links must be established from back to front. The links serve as children and parents of the nodes, ensuring proper nesting. To avoid running the tensors multiple times, they are introduced in `seen = set(tuple())`. Furthermore, the other instances of the node are also checked and any existing results or other nodes are appended. The nested instances are selected one by one, this is a depth first search, and added to their respective positions. Additionally, certain nodes that are identifiable as input or output are highlighted with corresponding colours to distinguish them from the rest and provide a clearer overview.

Chapter 6

Evaluation

This chapter evaluates the visualisation results of the PyTorchKGT tool. Section 6.1 presents the tool's evaluation and provides examples of computational graph visualisations. Section 6.2 discusses the survey approach taken and considers the systematic approach of the survey. The evaluation of the survey is discussed in Section 6.2.1. The responses to the evaluation survey from potential users of the tool were analysed. Section 6.2.2 describes the analysis of the specific results and their implications for future extensions. The results are presented objectively, and the success of the visualisation of computational graphs is evaluated. A conclusion is drawn based on the findings.

6.1 PyTorchKGT Visualisation Results

The tool's visualisation options are implemented in different areas, making them easier to identify and modify. The tool consists of several cases that describe and cover the different implementation areas, enhancing the structure and use of the PyTorchKGT tool. The use of KLighD enables a range of visualisation possibilities, which can be viewed through the Visual Studio Code extension. It is important to note that this improved text adheres to the desired characteristics of objectivity, comprehensibility, logical structure and conventional structure. As well as clear and objective language, format, formal register, structure, balance, precise word choice, and grammatical correctness. These criteria are the basis for the creation of the various visualisation display options. As explained in Chapter 5, the visualisation can be changed by three parameters.

Figure 6.1. PyTorchKGT visualisation of an LSTM computational graph.

Figure 6.1 displays a view that does not use any of these parameters. In this view, all hidden layer steps are shown in grey, the inputs in blue, and the final result in green. The text flows smoothly from left to right, with the KEdge arrow directions providing clear guidance. The use of an LSTM allows for a more intricate concatenation that spans the entire graph, although this can make it more challenging to display due to its length. Unfortunately, the full version of this graph is not legible due to the limited resolution, therefore the graph is shown again in a compacted view in Figure 6.2 below. Intermediate results that are coloured orange

are displayed. The addition of values, tensors, and biases can be difficult to recognise and read. To make this easier, the third parameter can be used to reduce the hidden layer elements to those that provide a result, resulting in a clearer and easier-to-recognise visualisation. Both the reduced and full views can utilise each of these parameters. The KLighD application's web view allows for scaling, eliminating the problem of visualisation being too large during active use. However, for this type of work, the size might still be a problem.

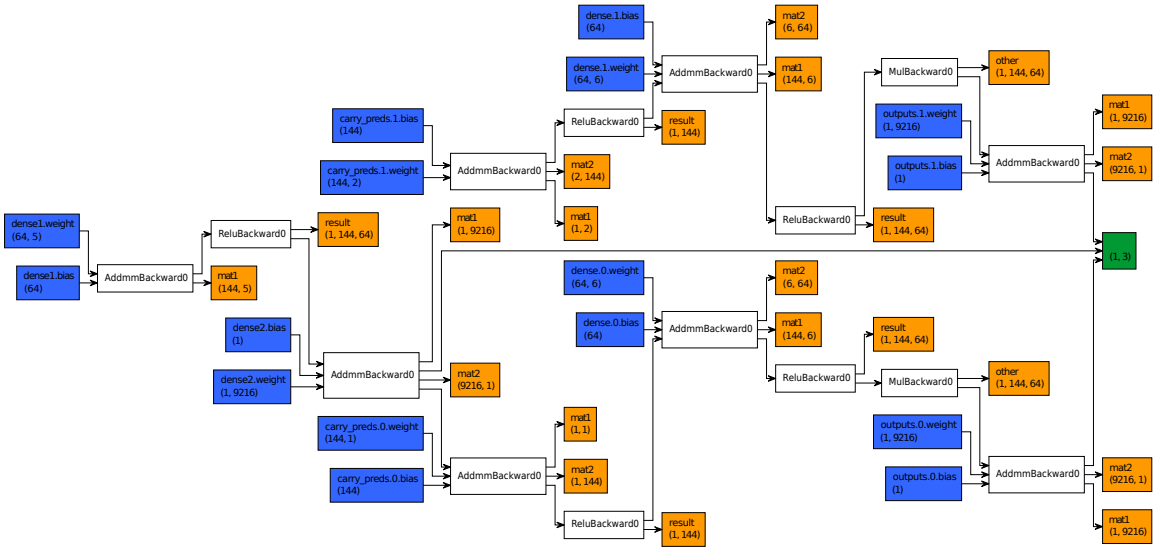


Figure 6.2. PyTorchKGT visualisation of an LSTM as a compact view and result view.

Figure 6.2 displays the compressed view of the results, with orange nodes representing the tensor values from the backwards propagation calculation. This view is also available in the full view, but it is not represented as an image due to its size. The arrow indicates that the node is an output from the direction to the right.

For additional information, all parameters can be utilised, as shown in Figure 6.3. Each result is derived from values at the node and displayed accordingly. This view is also available in the extended format and is not hierarchically displayed. The *AddmmBackward* serves as a connection point where multiple results are calculated, resulting in more vertices.

6.2 Survey

To investigate the relevance and opinions of users in this area, a survey was conducted. The survey was conducted in the Intelligent Systems group of Kiel University to assess the functionality and usability of the PyTorchKGT visualisations. This included questions about the computational graph visualisations and their various results were presented. An overview of the topic and other relevant information were explained and the opinions were gathered. The participants examined the interface between the user and the tool that was

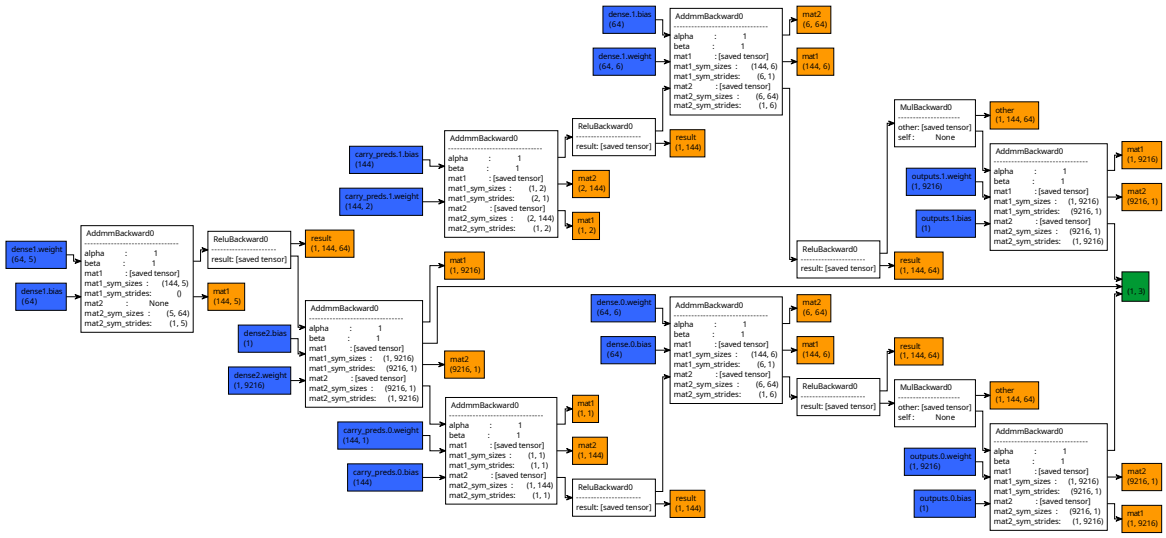


Figure 6.3. PyTorchKGT visualisation of an LSTM as a compact view and result view with additional values.

presented. Relevant questions about the visualisation solutions were asked and the response was provided, and an assessment of their overall impression was conducted, including both the positive and the negative aspects. Other application requests that would extend the range of applications were also included. In the following section, the evaluation of the implementation and the possibility for further development is explained. Additionally, this thesis considers the approach taken and its impact on the final results. The strengths and weaknesses of the approach are analysed and discussed, along with its further impact on the KIELER project.

6.2.1 Results

The negative feedback included both minor comments and major points. Readability is impaired by the dark blue colour and black font used for the entries. There was also a desire for interactive features, such as the ability to change the graph by clicking on it and to collapse and expand data. There is also a need for large visualisations that can be displayed vertically in different ways for publications.

Respondents noted that the colour scheme of blue, white, orange, and green was a good choice as it is also suitable for colour blind people. The clear structure of the graphs and their uniformity contributed to good readability, as calculations were communicated effectively. Additionally, the variety of visualisation options were appreciated, as these were essential for a clear overview. The use of Python models was also highlighted as a positive aspect, leading to discussions about its potential applications.

Further comments on the future development of this visualisation led to development possibilities that these participants would like to use in the future. The interactive Live Monitoring visualisation could allow different ranges of values to be displayed in terms of the time taken to compile the calculation, as well as highlighting borderline result values. The different views could be displayed optionally or continuously if required. This would be useful when searching for problems. Another option could be to visualise the active training with the active values to make the current progress visible. The time aspect and the boundary areas could then also be monitored live, and possible issues during training could be displayed.

Overall, the respondents had a positive view of the visualisation tool and expressed interest in using it if further developed. It was emphasised that the tool offers potential when used for computational graph visualisations in publications and for the monitoring of learning of the DNN models, while training as well as while developing models.

6.2.2 Discussion

Throughout this thesis, the principles of effective visualisation have been applied to enhance visibility and facilitate understanding. This has been achieved through the use of colour coding and common destiny. These techniques have improved the understanding of grouping. The simple and clear presentation of the visualisations was also commended for its structure and readability. The demand for a more interactive application also broadens the potential for smaller visualisations of individual parts, which can utilise distinct forms for differentiation. The positive feedback and quick understanding of the respondents shows that the implementation of the computational graph visualisation and its features are a positive aspect of the PyTorchKGT tool compared to other tools that lack good visualisation options.

The abstract nature of computational graph visualisation can be preserved in the implementation. The necessary features, as described in Section 4.4, have been implemented and are usable for the PyTorchKGT visualisation tool. Further explanations regarding the implementation and the PyTorchKGT tool are provided in the Section 6.1.

Additionally, the visualisation of DNN architectures has not yet been completed. Therefore, a finished visualisation, as explained in Section 4.5, is not available. However, a prototype is already available, which can be used to expand from a more advanced point.

Overall, this project is the first Python implementation to interact with the KIELER project. The ability to continue to develop these structures allows future projects to create additional visualisations using KLighD. This combination with the PyTorchKGT tool and the use of the KLighD framework also allows for further development as it is a framework that is actively used and developed.

The evaluation was mostly positive, while the negative points required further development, which could not be realised due to the scope of this thesis. Despite the lack of higher level applications, the visualisation of the computational graph was visualised well. Therefore, it can be concluded that on the basis of the limited number of interviews and the evaluation of the visualisations, it can be concluded that the visualisation has been successful.

Chapter 7

Conclusion

We have presented a tool that automatically extracts computational graphs from PyTorch models and visualises them by converting them into KGraphs for use with KLighD. The shapes of the different backwards propagation results can be viewed with the different visualisation options, which the graph is providing. Changing an existing tool and connecting it to the KGT structure to improve the visualisation shows an approach to computational graph visualisation and the extension of the KGT structure into the Python area. This Python tool is useful in the field of machine learning as confirmed by the evaluation group. Although the primary use of the tool is for learning purposes, further development could improve the adaptability of the tool for use in other applications as well. In addition, the possibility of further visualisations is provided by the open structure. The KGT structure provides a useful way of visualising DNNs and allows individual customisation through manually adjustable variables. That improves the quality of the visualisation of shown tensors and values. The simplicity of the input makes it user-friendly, and the improvements in seeing optional different values are improving the usability even more. In order to improve the tool, we need to focus on its potential applications. However, further research is needed to standardise and classify specifications for a functional visualisation form in this area.

The project utilised an effective visualisation form, but the main focus is on visualising the hidden layer area by showing data flow of computational calculations and their values.

Future Work

The PyTorchKGT tool could output improved and additional visualisation variants after further improvements and extensions. The tool can visualise not only computational graphs but also architectural visualisation and other visualisation options thanks to the currently unutilised KGT implementation and the help of KLighD. Additional options for customisation are available through the use of colour gradients, shadow rotation, and other features. The computational graph can also be enhanced by incorporating groupings. As the Python KGraph of this thesis shares a similar structure with the KGraph specification of the KIELER project, many of the visualisation options used in that project can be implemented in this tool, providing a wide range of possibilities for customisation and adaptation. The visualisation of the architecture could be improved by extracting ReLU and maxpool differently. This can be achieved via ONNX exporter as mentioned in Section 4.5.

Bibliography

[AAO20] E. O. Aliyu, A. O. Adetunmbi, and B. A. Ojokoh. "Intermediate representation using graph visualization software". In: *Journal of Software Engineering and Applications* 13.05 (2020), pp. 77–90. ISSN: 1945-3116. DOI: 10.4236/jsea.2020.135006.

[AOM17] Ricardo de A. Araújo, Adriano L.I. Oliveira, and Silvio Meira. "A morphological neural network for binary classification problems". In: *Engineering Applications of Artificial Intelligence* 65 (2017), pp. 12–28. ISSN: 0952-1976. DOI: 10.1016/j.engappai.2017.07.014. URL: https://www.sciencedirect.com/science/article/pii/S0952197617301628.

[BBD+19] Carla Binucci, Ulrik Brandes, Tim Dwyer, Martin Gronemann, Reinhard von Hanxleden, Marc J. van Kreveld, Petra Mutzel, Marcus Schaefer, Falk Schreiber, and Bettina Speckmann. "10 reasons to get interested in graph drawing". In: *Computing and Software Science - State of the Art and Perspectives*. Ed. by Bernhard Steffen and Gerhard J. Woeginger. Vol. 10000. Lecture Notes in Computer Science. Springer, 2019, pp. 85–104. DOI: 10.1007/978-3-319-91908-9_6. URL: https://doi.org/10.1007/978-3-319-91908-9%5C_6.

[BL09] Enrico Bertini and Denis Lalanne. "Surveying the complementary role of automatic data analysis and visualization in knowledge discovery". In: *Proceedings of the ACM SIGKDD Workshop on Visual Analytics and Knowledge Discovery: Integrating Automated Analysis with Interactive Exploration // VAKD '09*. Ed. by Kai Puolamäki. VAKD '09. New York, NY, USA: Association for Computing Machinery and ACM Press, 2009, pp. 12–20. ISBN: 9781605586700. DOI: 10.1145/1562849.1562851.

[BL10] Enrico Bertini and Denis Lalanne. "Investigating and reflecting on the integration of automatic data analysis and visualization in knowledge discovery". In: *SIGKDD Explor. Newsl.* 11.2 (2010), pp. 9–18. ISSN: 1931-0145. DOI: 10.1145/1809400.1809404.

[Bor19] Yannic Borgfeld. "Tool support for layout algorithm development with elk". In: (2019). URL: https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/yab-bt.pdf?utm_source=textcortex&utm_medium=zenochat.

[BVB+11] Michael Burch, Corinna Vehlow, Fabian Beck, Stephan Diehl, and Daniel Weiskopf. "Parallel edge splatting for scalable dynamic graph visualization". In: *IEEE Transactions on Visualization and Computer Graphics* 17.12 (2011), pp. 2344–2353. DOI: 10.1109/TVCG.2011.226.

[Cho22] Francois Chollet. *Deep learning with python*. Second edition. Shelter Island: Manning Publications, 2022. ISBN: 9781617296864.

[CM10] Michel Chein and Marie-Laure Mugnier. *Graph-based knowledge representation: computational foundations of conceptual graphs*. Advanced Information and Knowledge Processing. London: Springer-Verlag, 2010. ISBN: 978-1-84800286-9.

[CXS+20] Yiran Chen, Yuan Xie, Linghao Song, Fan Chen, and Tianqi Tang. "A survey of accelerator architectures for deep neural networks". In: *Engineering* 6.3 (2020), pp. 264–274. ISSN: 2095-8099. DOI: 10.1016/j.eng.2020.01.007. URL: https://www.sciencedirect.com/science/article/pii/S2095809919306356.

[DHS+23] Sören Domrös, Reinhard von Hanxleden, Miro Spönemann, Ulf Rüegg, and Christoph Daniel Schulze. *The eclipse layout kernel*. 2023.

[DRS+15] C. Dunne, S. I. Ross, B. Shneiderman, and M. Martino. *Readability metric feedback for aiding node-link visualization designers | ibm journals & magazine | ieee xplore // readability metric feedback for aiding node-link visualization designers*. 2015. DOI: 10.1147/JRD.2015.2411412.

[EGK+02] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C. North, and Gordon Woodhull. "Graphviz— open source graph drawing tools". In: *Graph Drawing*. Ed. by Petra Mutzel, Michael Jünger, and Sebastian Leipert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 483–484. ISBN: 978-3-540-45848-7.

[Fer18] Phil Ferriere. *Hiddenlayer/hiddenlayer/transforms.py at master · waleedka/hiddenlayer*. 2018. URL: https://github.com/waleedka/hiddenlayer.

[FR21] Stephen Fitz and Peter Romero. "Neural networks and deep learning: a paradigm shift in information processing, machine learning, and artificial intelligence". In: *The Palgrave Handbook of Technological Finance*. Ed. by Raghavendra. Rau, Robert. Wardrop, and Luigi. Zingales. Cham: Springer International Publishing and Imprint: Palgrave Macmillan, 2021, pp. 589–654. ISBN: 978-3-030-65117-6. DOI: 10.1007/978-3-030-65117-6{\textunderscore}22.

[Gan11] Emden Gansner. "Drawing graphs with graphviz". In: (2011). URL: https://www.ammd.ch/1.pdf?utm_source=textcortex&utm_medium=zenochat.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[Git24] GitHub. *Github - szagoruyko/pytorchviz: a small package to create visualizations of pytorch execution graphs*. 2024. URL: https://github.com/szagoruyko/pytorchviz.

[GK20] Hossein Gholamalinezhad and Hossein Khosravi. *Pooling methods in deep neural networks, a review*. 2020. arXiv: 2009.07485 [cs.CV].

[GN00] Emden R. Gansner and Stephen C. North. "An open graph visualization system and its applications to software engineering". In: *Software: Practice and Experience* 30.11 (2000), pp. 1203–1233. ISSN: 0038-0644. DOI: 10.1002/1097-024X(200009)30:11{\textless}1203::AID-SPE338{\textgreater}3.0.CO;2-N.

[Hay20] Yoichi Hayashi. "New unified insights on deep learning in radiological and pathological images: beyond quantitative performances to qualitative interpretation". In: *Informatics in Medicine Unlocked* 19.6088 (2020), p. 100329. ISSN: 23529148. DOI: 10.1016/j.imu.2020.100329.

[Hee19] Jeffrey Heer. "Agency plus automation: designing artificial intelligence into interactive systems". In: *Proceedings of the National Academy of Sciences* 116.6 (2019), pp. 1844–1850. DOI: 10.1073/pnas.1807184115. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.1807184115. URL: https://www.pnas.org/doi/abs/10.1073/pnas.1807184115.

[HSW89] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8. URL: https://www.sciencedirect.com/science/article/pii/0893608089900208.

[Kas21] Maximilian Kasperowski. "A top-down approach on automatic graph visualization". In: (2021). URL: https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mka-mt.pdf?utm_source=textcortex&utm_medium=zenochat.

[KBH+24] Aymen Rayane Khouas, Mohamed Reda Bouadjenek, Hakim Hacid, and Sunil Aryal. *Training machine learning models at the edge: a survey*. 2024. arXiv: 2403.02619 [cs.LG].

[KH23] Maximilian Kasperowski and Reinhard von Hanxleden. "Top-down drawings of compound graphs". In: *arxiv* (12.2023). URL: https://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/arxiv23b.pdf?utm_source=textcortex&utm_medium=zenochat.

[KNP+] Hartmut Klauck, Danupon Nanongkai, Gopal Pandurangan, and Peter Robinson. "Distributed computation of large-scale graph problems". In: *Proceedings of the 2015 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 391–410. DOI: 10.1137/1.9781611973730.28.

[KW01] Michael Kaufmann and Dorothea Wagner, eds. *Drawing graphs: methods and model*. Vol. 2025. Lecture notes in computer science. Berlin: Springer, 2001. ISBN: 3-540-42062-2.

[LLT23] Shaoxuan Lai, Wanna Luan, and Jun Tao. "Explore your network in minutes: a rapid prototyping toolkit for understanding neural networks with visual analytics". In: *IEEE Transactions on Visualization and Computer Graphics* 14.3 (2023), pp. 1–11. ISSN: 1077-2626. DOI: 10.1109/TVCG.2023.3326575.

[McG15] Kevin McGurgan. "Data-ink ratio and task complexity in graph comprehension." In: *Rochester Institute of Technology* (2015).

[NHP+18] Shaoliang Nie, Christopher Healey, Kalpesh Padia, Samuel Leeman-Munk, Jordan Benson, Dave Caira, Saratendu Sethi, and Ravi Devarajan. "Visualizing deep neural networks for text analytics". In: IEEE, 2018. DOI: 10.1109/pacificvis.2018.00031.

[Pet19] Jette Petzold. "Intentional layout in sprotty diagrams: defining user interaction". In: (2019). URL: https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/jet-bt.pdf?utm_source=textcortex&utm_medium=zenochat.

[PWW+24] Rusheng Pan, Zhiyong Wang, Yating Wei, Han Gao, Gongchang Ou, Caleb Chen Cao, Jingli Xu, Tong Xu, and Wei Chen. "Towards efficient visual simplification of computational graphs in deep neural networks". In: *IEEE Transactions on Visualization and Computer Graphics* (2024), pp. 1–14. ISSN: 1077-2626. DOI: 10.1109/TVCG.2022.3230832.

[Ren18] Niklas Rentz. "Moving transient views from eclipse to web technologies". In: (2018). URL: https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/nir-mt.pdf?utm_source=textcortex&utm_medium=zenochat.

[RR96] G. Ramalingam and Thomas Reps. "On the computational complexity of dynamic graph problems". In: *Theoretical Computer Science* 158.1 (1996), pp. 233–277. ISSN: 0304-3975. DOI: 10.1016/0304-3975(95)00079-8. URL: https://www.sciencedirect.com/science/article/pii/0304397595000798.

[SA23] Thirumurugan Shanmugam and Shweta A. Bansal, eds. *Cutting-edge technologies in innovations in computer science and engineering*. San International Scientific Publications, 2023. ISBN: 9788196384975. DOI: 10.59646/csebook/004.

[SAV+20] Eli Stevens, Luca Antiga, Thomas Viehmann, and Soumith Chintala. *Deep learning with pytorch*. Shelter Island (New York): Manning, 2020. ISBN: 9781617295263.

[SBM+17] Wojciech Samek, Alexander Binder, Gregoire Montavon, Sebastian Lapuschkin, and Klaus-Robert Muller. "Evaluating the visualization of what a deep neural network has learned". In: *IEEE Transactions on Neural Networks and Learning Systems* 28.11 (2017), pp. 2660–2673. DOI: 10.1109/TNNLS.2016.2599820.

[Sch14] Christian Schneider. *Kgraph meta model - kieler project - confluence*. Ed. by Real-Time and Embedded Systems. 2014. URL: https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/The+KRendering+Notation+Model.

[Sch16] Alan Schelten. "Hierarchy-aware layer sweep". In: (2016). URL: https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/alan-mt.pdf?utm_source=textcortex&utm_medium=zenochat.

[SCW+16] Shizhao Sun, Wei Chen, Liwei Wang, Xiaoguang Liu, and Tie-Yan Liu. "On the depth of deep neural networks: a theoretical view". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 30.1 (2016). ISSN: 2374-3468. DOI: 10.1609/aaai.v30i1.10243.

[Spi14] Michel Spils. *Kgraph meta model - kieler project - confluence*. Ed. by Real-Time and Embedded Systems. 2014. URL: https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/KGraph+Meta+Model.

[SSH13] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. "Just model! — putting automatic synthesis of node-link-diagrams into practice". In: *2013 IEEE Symposium on Visual Languages and Human Centric Computing*. Sept. 2013, pp. 75–82. DOI: 10.1109/VLHCC.2013.6645246.

[SSR20] Mohit Sewak, Sanjay K. Sahay, and Hemant Rathore. "An overview of deep learning architecture of deep neural networks and autoencoders". In: *Journal of Computational and Theoretical Nanoscience* 17.1 (2020), pp. 182–188. ISSN: 1546-1955. DOI: 10.1166/jctn.2020.8648.

[SWH18] Christoph Daniel Schulze, Nis Boerge Wechselberg, and Reinhard von Hanxleden. "Edge label placement in layered graph drawing". In: (2018). URL: https://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/report-1802.pdf?utm_source=textcortex&utm_medium=zenochat.

[VSS+19] Ivan Vasilev, Daniel Slater, Gianmario Spacagna, Peter Roelants, and Valentino Zocca. *Python deep learning - second edition*. 2nd edition. Packt Publishing, 2019. ISBN: 978-1-78934-846-0.

[YCN+15] Jason Yosinski, Jeff Clune, Anh Nguyen, Thomas Fuchs, and Hod Lipson. *Understanding neural networks through deep visualization*. 2015.

[YWC20] Shuoheng Yang, Yuxin Wang, and Xiaowen Chu. *A survey of deep learning techniques for neural machine translation*. 2020.

[Zey24] Hu Zeyuan. *Draw a neural network through graphviz*. 2024. URL: https://zhu45.org/posts/2017/May/25/draw-a-neural-network-through-graphviz/.

[Zi24] Zi. *Künstliche intelligenz in der medizin | zentralinstitut für die kassenärztliche versorgung*. Ed. by Zentralinstitut kassenärtzliche Versorgung. 2024. URL: https://www.zi.de/themen/it-und-data-science/kuenstliche-intelligenz-in-der-medizin.

[ZTS+23] Marc-André Zöller, Waldemar Titov, Thomas Schlegel, and Marco F. Huber. "Xautoml: a visual analytics tool for understanding and validating automated machine learning". In: *ACM Transactions on Interactive Intelligent Systems* 13.4 (2023), pp. 1–39. ISSN: 2160-6455. DOI: 10.1145/3625240.