

NeuraViz: A Web Application For Visualizing Artificial Neural Network Structures

A Manuscript

Submitted to

the Department of Computer Science

and the Faculty of the

University of Wisconsin–La Crosse

La Crosse, Wisconsin

by

Bennett Wendorf

in Partial Fulfillment of the

Requirements for the Degree of

Master of Software Engineering

May, 2024

NeuraViz: A Web Application For Visualizing Artificial Neural Network Structures

By Bennett Wendorf

We recommend acceptance of this manuscript in partial fulfillment of this candidate's requirements for the degree of Master of Software Engineering in Computer Science. The candidate has completed the oral examination requirement of the capstone project for the degree.

Prof. Albert Einstein
Examination Committee Chairperson

Date

Prof. Isaac Newton
Examination Committee Member

Date

Prof. Marie Curie
Examination Committee Member

Date

Abstract

Wendorf, Bennett, “NeuraViz: A Web Application For Visualizing Artificial Neural Network Structures,” Master of Software Engineering, May 2024, (Jason Sauppe, Ph.D.).

This manuscript describes the software engineering processes and principles adhered to during the development of Neuraviz, a web application for visualizing artificial neural network structures. Users upload pre-trained machine learning models from popular frameworks including Pytorch and Keras, and Neuraviz generates a visual representation of the model’s architecture. The following manuscript focuses on the design, implementation, testing, and deployment of NeuraViz in an effort to comprehensively encapsulate the entire development process.

Acknowledgements

I would like to extend my sincerest thanks to my project advisor, Dr. Jason Sauppe, for his guidance and support throughout the development of NeuraViz. His feedback was always crucial in pointing me in the right direction, especially when I was overwhelmed with possibilities.

Thank you also to the entire computer science department at the University of Wisconsin-La Crosse for tirelessly helping me through all my coursework and projects throughout my tenure at the university. My ability to complete this monumental task would not have been possible without them.

I would also like to thank the open source community for providing the tools and resources used in this project. Open source software is an integral part of the modern software space and none of our lives would be the same without it.

Finally, I would like to thank my parents, family, friends, and all the teachers, mentors, and colleagues who have helped, supported, and encouraged me throughout my life. I am more grateful than I can possibly express in words.

Table of Contents

Abstract	i
Acknowledgments	ii
List of Tables	v
List of Figures	vi
Glossary	vii
1. Introduction	1
1.1. Overview	1
1.2. Similar Projects	1
1.3. Goals	2
1.4. Neural Networks	2
1.4.1. Neurons	2
1.4.2. Edges	2
1.4.3. Activation Functions	3
2. Software Development Process	4
2.1. Overview	4
2.2. Life Cycle Model	4
2.2.1. Waterfall Model	4
2.2.2. Iterative Model	5
2.2.3. Agile (Scrum) Model	6
2.3. Development Process Technologies	6
2.4. Functional Requirements	7
2.5. Non-Functional Requirements	8
3. Design	10
3.1. Overview	10
3.2. UML Class Diagram	10
3.2.1. Frontend/Client	10
3.2.2. Backend/Server	11
3.3. Database	11
3.4. User Interface	11
3.4.1. Final Interface	12
4. Implementation	14
4.1. Technologies Used	14
4.1.1. Client	14
4.1.2. Server	15
4.1.3. Data Layer	16
4.2. Development	17
4.3. Deployment	18
5. Testing	20
5.1. Overview	20
5.2. Unit Testing	20
5.2.1. Frontend Testing	20
5.2.2. Backend Testing	20

5.3.	Regression and Integration Testing	22
6.	Security	23
6.1.	Overview	23
6.2.	Threat Model	23
6.3.	Session Management	23
6.4.	Web Application Security	23
7.	Conclusion	24
7.1.	Overview	24
7.2.	Challenges	24
7.3.	Future Work	24
8.	References	25
9.	Appendices	26

List of Tables

1	Server API Endpoints	16
---	--------------------------------	----

List of Figures

1	Waterfall Model Diagram	5
2	Iterative Model Diagram	6
3	Scrum Model Diagram	7
4	Jira Scrum Board	8
5	UML Class Diagram	10
6	User Interface Mockup	12
7	Final Interface	13

Glossary

1. Introduction

1.1. Overview

This project aims to develop a software system to visualize the architecture of artificial neural networks. Neural networks (NNs) are a class of machine learning models that are inspired by the structure and function of the human brain. They are composed of a large number of interconnected processing elements, called neurons, which work together to solve complex problems. The architecture of a neural network refers to the arrangement of neurons and the connections between them. In addition to the general structure of a network, the weights and biases that control its function can provide useful insight into the inner workings of the model. The final integral parts of the neural network architecture are the activation functions that govern how data propagates through the network. Visualizing the architecture of a neural network can help students and researchers understand the structure of the model, identify potential issues, and communicate the model to others. More information on neural networks in general can be found in section 1.4.

This project will develop a web application that allows users to upload pre-trained neural network models and generate visual representations of their architecture. The application will support models trained using popular machine learning frameworks such as PyTorch and Keras. The resulting graph structure will be visualized in a pannable and zoomable svg format that shows the ordering of neurons, biases on those neurons, the weights of the connections between them and activation functions on each layer.

The application will be designed to be user-friendly and accessible to a wide range of users, including students and researchers. It will be implemented using modern web technologies and will be deployed as a web service that can be accessed from any device with an internet connection. The project will follow best practices in software engineering, including requirements analysis, design, implementation, testing, and deployment. The resulting application will be a valuable tool for anyone working with neural networks, but will be primarily targeted toward post-secondary educational students learning about machine learning and neural networks for the first time.

1.2. Similar Projects

Neural networks are a notoriously difficult concept to understand, especially for those who are newer to the field of machine learning. The architecture of a neural network is a key component of understanding how the model functions, but it can be difficult to visualize and comprehend. There are a handful of tools available for visualizing neural network architectures, but they are often limited in their scope. For example, Google's TensorBoard is a popular tool, but only natively supports TensorFlow and Keras models. In researching existing tools such as TensorBoard, Netron, and ENNUI, it was found that none of them supported the same range of formats as NeuraViz. This project aims to develop a user-friendly tool that is accessible to a wide range of users, including students and researchers who are new to the field of machine learning, and to support a wider range of model formats than other offerings.

1.3. Goals

With an aim to solve the issues with existing tools, NeuraViz will be developed with a few key goals in mind. One main goal is to support a wide range of formats and to visualize each format in a standard way. This will allow users to upload models from a variety of frameworks and see a consistent and comparable visualization of their model.

NeuraViz is also designed specifically with user-friendliness and simplicity in mind. The minimal interface focuses on the visualizes of the neural networks themselves, with minimal distractions. The use of color and shape in the visualizations will help to make the network architecture more understandable at a glance, quickly identifying neurons and connections that are the most important. Where needed, users can also zoom in or out an click on elements to see more detailed information.

Portability is another key goal of the development. Modern web technologies ensure the application is accessible from a range of devices, though it is most optimized for a desktop or laptop experience. Since the application is deployed as a web application, users don't need to install software on their own machines, or even sign in to be able to use the tool. To enhance this ability further, graph representations can also be exported as raw SVG or in tikz format for use within LaTeX documents.

1.4. Neural Networks

Neural networks are a type of machine learning model that are inspired by the human brain. They are made up of layers of neurons, which are connected to each other. Each layer of neurons takes in a number of inputs, processes them, and then outputs a value. The value that is output is then passed to the next layer of neurons, and so on, until the final layer of neurons outputs the final result. [7]

1.4.1. Neurons

In an artificial neural network, neurons are the primary pieces of the network that perform the computation necessary. They are organized into groups called layers, typically represented in a graph structure organized vertically so the neurons in a layer are in a sort of column. Typically, the first layer is called the input layer, and behaves differently than other layers. For this reason, input neurons are represented as grey squares in NeuraViz as opposed to the typical neurons' circles. Neurons run the computations needed for the network to function with complex algorithms that I'll skip over here for simplicity. In general, neurons in a layer are connected to all the neurons in the previous layer, and all the neurons in the next layer. These connections are represented as lines between the neurons in NeuraViz.

1.4.2. Edges

Edges are the connections between neurons in a neural network. They are the primary way that information is passed between neurons. In NeuraViz, edges are represented as lines between neurons. Edges also have weights, which are used in the computation to determine how important that connection is. For small enough networks, the weights can be seen by hovering over edges in NeuraViz.

1.4.3. Activation Functions

Activation functions are a key part of how neural networks work. They are used to determine the output of a neuron based on the inputs it receives. There are many different activation functions, but one of the most common ones is the sigmoid function. The sigmoid function takes in a number and returns a number between 0 and 1. This is useful because it allows the network to output a probability. In NeuraViz, activation functions are shown at a layer-level, and represented as icons toward the top of each layer. Hovering over these icons will show the activation function used in that layer.

2. Software Development Process

2.1. Overview

Developing software is an extensive and complex process that requires a lot of planning, both in relation to the methodologies used during the development process and the requirements, both functional and non-functional of the software. This section details life cycle models considered for NeuraViz’s development, the model that was eventually chosen, and modifications to the model necessary for the development of this particular system. It also outlines functional and non-functional requirements for NeuraViz.

2.2. Life Cycle Model

Prior to beginning development of NeuraViz, a number of software life cycle models were considered to govern the pace and structure of development. In all, the waterfall model, iterative model, and agile model were considered. More specifically with agile, a variation of scrum, modified for a single developer, was considered. Ultimately, the modified scrum model was chosen for its flexibility and ability to adapt quickly to changing requirements.

2.2.1. Waterfall Model

The waterfall model is one of the oldest software development lifecycle (SDLC) models, originally proposed by Winston Royce in 1970 [4]. The model is a linear, sequential approach to software development, with each phase of the development process directly following the previous phase. Each subsequent phase relies on the previous phase, and as such the model does not allow going back to previous phases once they are completed. The first phase of the waterfall model is the requirement analysis phase, in which project requirements, both functional and non-functional, are gathered and documented. At this phase, requirements are also often analyzed for traits like consistency and feasibility. The second phase is the system design phase, in which the architecture of the software system is designed in full. All details of what needs to be done and how it will be completed are considered and documented during this phase. Third, the implementation phase is where the code for the software is written and the design from the previous step is implemented in full, exactly as specified during the design phase. During the fourth phase, the software is tested for bugs and errors, and issues are resolved as needed. Fifth, the software is deployed to the client in its entirety. In the waterfall model, this is the first time the client has seen the software. Finally, the software is maintained and updated as needed for as long as the client needs it. Figure 1 shows a diagram of the waterfall model and its constituent parts.

Because of its rigid structure, the waterfall model excels at being very easy to understand and pick up quickly for new developers, which was initially intriguing during model selection for NeuraViz. In addition, it is easy to manage with relatively little overhead in management. When project requirements are well understood up front and unlikely to change, the waterfall model also serves the benefit of ensuring design is completed before implementation begins, which leads to fewer mistakes and less necessity to change the code once it has been written. However, for projects where requirements are less well understood or are likely to change,

such as in this project, the waterfall model struggles to adapt and may lead to a design that was flawed in the first place with no way to fix it. In addition, the waterfall model does not allow for client feedback until the software is fully completed, which can lead to a lot of wasted time and effort if the client is not satisfied with the final product. In the case of this project where adaptability was crucial, the waterfall model would have been a poor choice.



Figure 1. Waterfall Model Diagram [16]

2.2.2. Iterative Model

Like the waterfall model, the iterative model is mostly linear and sequential with a relatively rigid structure where each step directly follows the previous step. The phases in the iterative model, as shown in Figure 2, roughly match those in the waterfall model, including a requirements analysis phase, a design phase, and implementation phase, a testing phase, and a deployment phase. Unlike the waterfall model, however, the iterative model runs these phases, with the exception of requirements analysis, multiple times, restarting the sequence of phases after each deployment. This serves the major benefit of allowing the ability for the client to give feedback on the project sooner and more often.

Due to its similarity to the waterfall model, the iterative model exhibits many of the same benefits of waterfall in that it is easy to understand with a relatively linear structure and minimal overhead. Like the waterfall model, when requirements are understood at the project outset, the design is likely to be almost fully complete before development begins, so the code is also less likely to require changes later in the process. While the iterative model does improve on the waterfall model's lack of ability for client feedback, it still struggles with changing requirements as each iteration of the project is still long and expected to be a relatively complete implementation of the software. The lack of adaptability made the iterative model a poor choice for NeuraViz's development where changing requirements were expected from the beginning.

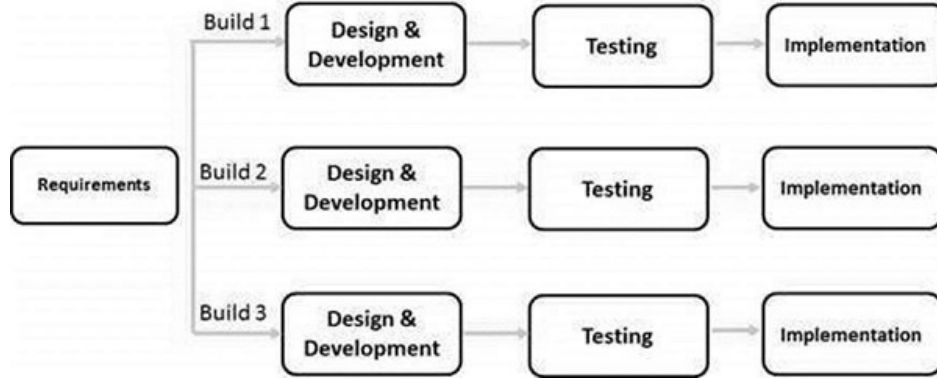


Figure 2. Iterative Model Diagram [15]

2.2.3. Agile (Scrum) Model

In contrast to the other models, agile models, and specifically scrum, was designed with the express intent to adapt to changing requirements quickly. Since the exact requirements and design for NeuraViz were not known well ahead of time, scrum was chosen for development due to its ability to pivot quickly when new design details were discovered. While agile is a category of software development models that focus on adaptability, scrum is a specific type of agile model that places emphasis on small, self-organized teams working in short, iterative cycles called sprints. Each sprint is typically two to four weeks long and ends with a review of the work completed during the sprint and a planning session for the next sprint. The scrum model is shown in Figure 3. While the diagram shows a two to three month timeline per iteration, scrum more typically follows a shorter sprint length.

Since NeuraViz only has one developer, the scrum model doesn't fit perfectly. However, many aspects of the model do fit relatively well, with some slight modifications. Scrum typically emphasizes daily stand-up meetings with each team of developers. Due to the longer timeline of NeuraViz's development and the single developer, these meetings were partially dealt with through daily review of the project board to help ensure that projects stay on track. In addition, the developer met every week with the project advisor, Dr. Jason Sauppe, who in some sense served as a stakeholder on the project and a product owner. While these meetings don't match exactly with any part of the scrum model, they serve as a combination of stand-up meetings and sprint retrospectives. For the purposes of this project, sprints were completed every week. This allowed for very quick turnaround on features, and enabled constant feedback and reflection on project requirements.

2.3. Development Process Technologies

An important part of the scrum model is managing small, independent projects that can be completed in a short amount of time. To help manage these projects, Jira's scrum template was used. This template allows for a main project backlog where user stories can be converted to sprint tasks. Each sprint can then be created and tasks can be scheduled into one. Once each sprint begins, projects/tasks move through columns including backlog, programming, testing required, and done. This allows for a clear view of exactly what is being worked on



Figure 3. Scrum Model Diagram [14]

at any given time. The scrum board for sprint 21 (March 5th, 2024 - March 19, 2024) can be seen in Figure 4. Jira provides exceptional features for managing projects in a scrum format, allowing the tracking of not only sprint tasks through the stages of development on the project board, but additional features such as time tracking on projects as well.

In addition to Jira, Github and specifically branches were used to help keep track of individual sprint tasks. A new branch was created for each task, with the name of the branch including the task key from Jira and a brief description. Jira’s integration with GitHub provided a link to see what phase the code was actually in directly from the Jira task, including whether a pull request had been created or merged.

2.4. Functional Requirements

Since agile methodologies were chosen for the development of this project, a set of functional requirements in the form of user stories were collected prior to the start of development. These user stories served as a guide for features to implement and helped ensure that no major functionality was missed during development.

NeuraViz is a relatively simple application with only one type of user. As such, the functional requirements are relatively straightforward, including the ability to upload a pre-trained machine learning model trained in either Pytorch or Keras. In addition, users should be able to see either the full graph of the uploaded model, or a collapsed version depending on the scale of the uploaded model. Additional functionality is also documented such as the ability to navigate the page via pan and zoom functionality on the graph, and clicking or hovering on various network components.

In addition to the functionality of viewing the network itself, user stories were also docu-



Figure 4. Jira Scrum Board

mented for export functionality, allowing the graph of the model to be exported both as an svg and as a LaTeX document in the format of the tikz library.

The full set of user stories is documented in the user stories document. This document also includes user stories for functionality that was not implemented in the current version of NeuraViz, but may be implemented in future versions.

2.5. Non-Functional Requirements

In addition to the functionality documented as user stories, non-functional requirements were also documented to ensure that the user experience of NeuraViz was as smooth as possible. Identified non-functional requirements are as follows:

- Large network layers are collapsed if they are too big to reasonably render.
- If I am unable to view the model visually, labels exist for screen readers as much as possible.
- If the site takes a long time to load, skeletonized components are shown to indicate that the site is still loading.
- As a user, my data is reasonably secure, both during transmission and processing.
- Themes are sufficiently differentiable for colorblind users.

- Invalid models are rejected and not stored unnecessarily.

3. Design

3.1. Overview

NeuraViz follows a fairly standard server-client web application architecture. The client is responsible for rendering the user interface and allowing the user to interact with the application. The server handles the actual computationally intensive processes such as parsing the uploaded model and generating the structure of the visual representation. The server also handles the storage of the uploaded models during user sessions and the translation of the visualization into various formats.

3.2. UML Class Diagram

The UML class diagram in Figure 5 shows the classes and their relationships in the NeuraViz application. The diagram is divided into two main sections: the frontend and the backend, which are also commonly referred to as the client and server respectively.

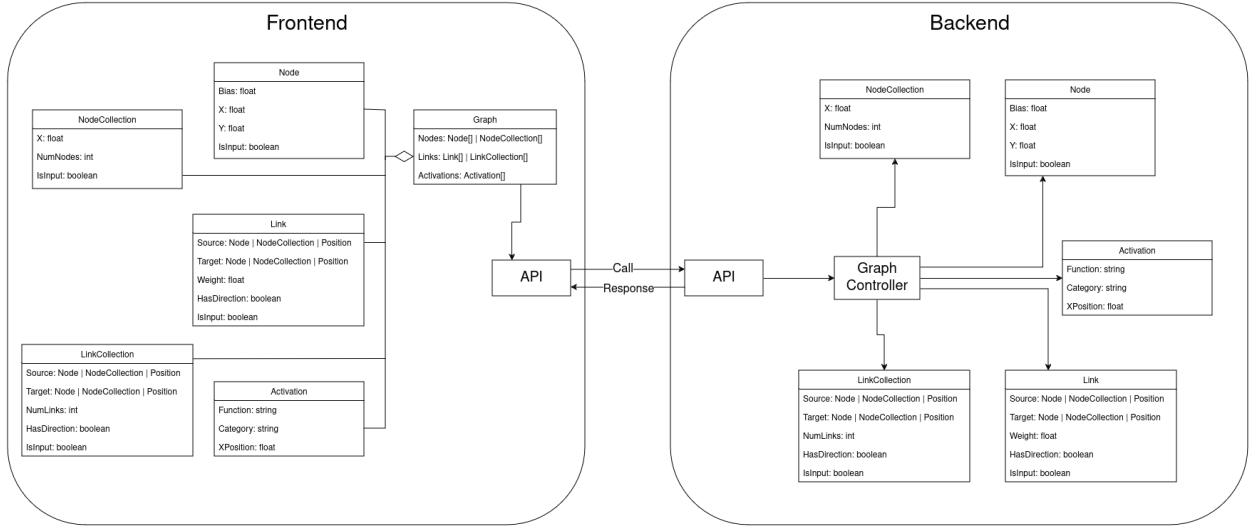


Figure 5. UML Class Diagram

3.2.1. Frontend/Client

The frontend primarily relies on the Graph object, which is comprised of a number of Nodes and/or Node Collections, Links and/or Link Collections, and Activation Functions. Nodes represent individual nodes as represented in the graph, and these are used for nodes in graph layers that are smaller than 10 nodes by default. For layers that are too large, the graph representation instead contains a Node Collection that represents the layer as a whole. Links and Link Collections operate a similar way. Activation Function objects represent the activation functions that can be seen as small icons at the top of each layer in the NeuraViz interface. The graph object houses the representation of the neural network model as ready for rendering. As shown in the UML diagram, the frontend also houses an API component

that is responsible for communicating with the backend architecture via standard HTTP requests.

3.2.2. Backend/Server

The backend is responsible for handling the computationally intensive processes of parsing the uploaded model and generating the structure of the visual representation. As seen in Figure 5, the backend houses objects that almost perfectly mirror the frontend components. However, on the server, these components are all related to the graph controller: the component responsible for the actual graph parsing. In addition to parsing the actual graph, the controller also handles additional requests for retrieving a stored model and converting the representation into various formats. Like with the frontend portion of the application, the backend houses an API component that is responsible for receiving the HTTP requests from the client and routing them to the correct controller endpoint for processing, as well as sending the response back to the client.

3.3. Database

At the outset of NeuraViz’s development, no database was planned to be used. The nature of the application is such that the primary functionality of the application should not require a user to log in, and NeuraViz itself does not need to store information of any kind. Initially, the users’ uploaded models got saved to disk during processing, but were then deleted immediately after for security and space efficiency. However, once the LaTeX export feature was introduced, it became necessary to maintain the graph’s representation for longer, or to send it back and forth between the client more. Since the graph representation can be quite large, it was decided to use a NoSQL database, namely MongoDB, to store the parsed graph information as a session.

When a user makes their first request to the NeuraViz application, a session is created and the client is given an identifier. Upon graph parsing, the graph representation is stored in the database under the session identifier and the uploaded file is deleted. Further requests can then retrieve the stored graph representation from the database, rather than having to re-upload the model and re-parse it. In addition to the LaTeX export feature, this also allows for the possible future features of saving the graph representation of a user’s account for future reference, providing further granularity on larger networks, and more.

3.4. User Interface

A major step in the design process was developing the look and feel of the interface that users would be interacting with. A user interface mockup was drawn in Gimp to give a visual representation of what the application would look like. The mockup served as a guide in developing the actual user interface, though some changes were made to the final product. Since NeuraViz operates on a single page with one main piece of functionality, the required mockup was fairly simple. Figure 6 shows a number of components that were included in the final product. The file upload button can be seen in the side panel on the left, along with its model validation text. Below that can be seen a section for settings with an example

of what a setting with a slider might look like. While no final components use a slider so far, future development may include more complex settings. In addition to the sidebar, the primary visualization window can be seen with a sample model visualization. In the bottom right corner, navigation buttons can be seen in the mockup, mirroring the final interface.

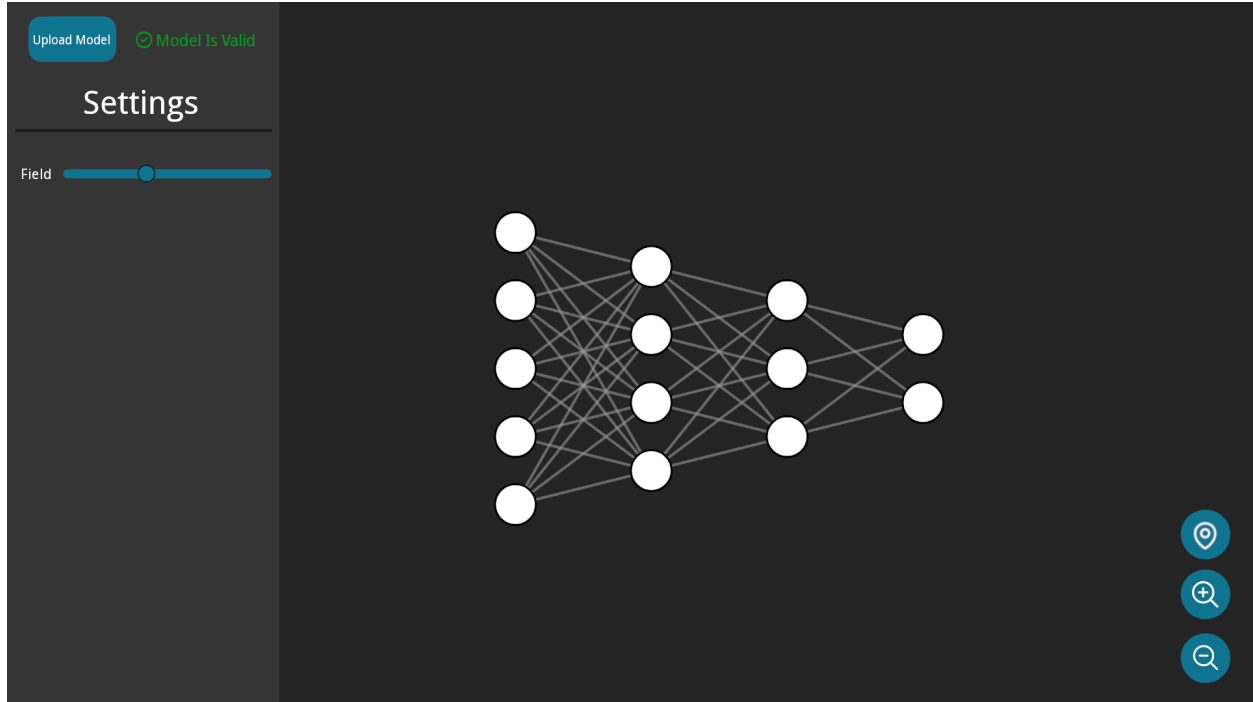


Figure 6. User Interface Mockup

3.4.1. Final Interface

The final interface of NeuraViz is shown in Figure 7. The interface is divided into two main sections: the sidebar and the main visualization window. At the top of the sidebar, the file upload section can be seen, including a file picker, upload button, and model validation text. Below that, the options for visualization export can be seen, with buttons for both exporting the visualization to LaTeX and to SVG. Next the settings panel can be seen, with a mode toggle for the color scheme of the application. At the bottom of the sidebar there is a color reference key for the colors used in the main visualization window.

The main visualization window contains the visualization of the model, or an indication that the user should upload a model. The colors of nodes and edges correspond to the color key found on the sidebar. Navigation buttons can be found at the bottom right of the page.

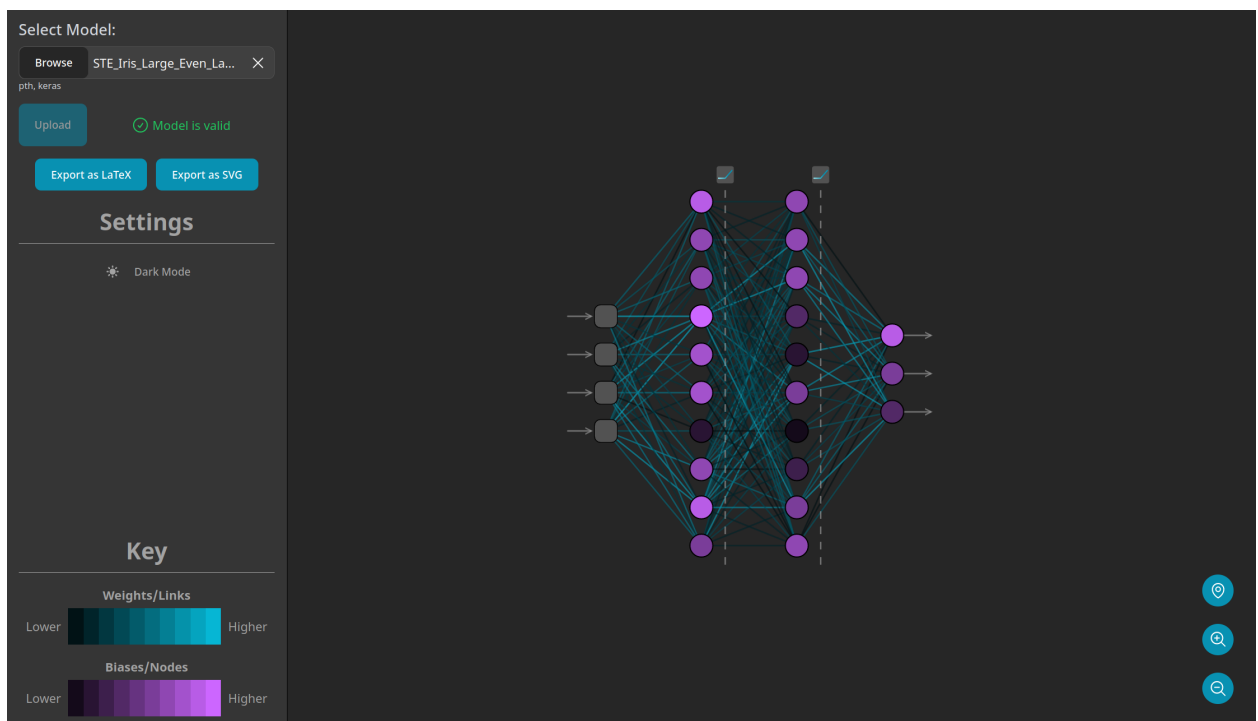


Figure 7. Final Interface

4. Implementation

4.1. Technologies Used

NeuraViz was developed with a multitude of standard web technologies used in conjunction to build the classic client-server architecture that was settled on. The chosen technologies were used for their stable and well-documented nature, as well as their ability to interact with libraries and frameworks specific to neural networks. This section describes what technologies were used for each of the three constituent components of NeuraViz.

4.1.1. Client

The client component is the primary interface that the user acts with, and therefore must be visually appealing and responsive. In researching available web frameworks for building frontend web applications, a number of possibilities were considered including ReactJS, Angular, and Svelte. While ReactJS would have been a good choice for its popularity and the developer’s experience, Svelte was eventually chosen for its simplicity and performance.

Svelte [12] is a relatively new web framework that is similar to ReactJS in that it is component-based, but it is different in that it compiles the components into vanilla JavaScript at build time. This means that the final product is much smaller and faster than a ReactJS application, which must include the React library in the final product. Svelte also has a simpler syntax than ReactJS, which makes it easier to learn and use. In particular, Svelte’s feature set across reactivity of its components, ease of implementing special features such as animations, and built-in global store system turned out to be especially helpful during the development of NeuraViz. Svelte also natively supports TypeScript, a superset of JavaScript that introduces a strong static type system and a powerful type inference system that helped keep the codebase clean and maintainable, while still providing the freedom to work in a web environment seamlessly.

Though Svelte is a powerful and featureful framework on its own, two shortcomings were identified during NeuraViz’s development that had to be overcome with additional libraries. Svelte’s component system does a fantastic job of abstracting components into small, reusable pieces, but it lacks prebuilt components that would keep the application’s look and feel consistent and the development rapid. To remedy this problem the Flowbite Svelte library [13] was used. Flowbite provides prebuilt Svelte components that fulfill many of the necessary functionalities of a web application. For example, it contains components such as buttons, text boxes, dialogs, and more. Being built on top of TailwindCSS [2], Flowbite also provides excellent support for light and dark application themes. As its second shortcoming, Svelte does not include functionality for sending or receiving HTTP requests. While JavaScript’s native fetch function would work for this, Axios [1] provides a much cleaner experience for handling HTTP calls and responses. Axios also provides support for handling the asynchronous nature of HTTP requests via JavaScript’s promise API and callback functions.

Graph visualization, being the most critical feature of NeuraViz, was a major consideration when determining client technologies. To aid in the construction of the graph visualization, the D3.js library [3] was used. D3 is a highly customizable JavaScript library for

creating visualizations of data in all different formats. It works by providing abstractions on the DOM for building SVG elements easily and dynamically. In addition to providing methods for creating large numbers of static SVG elements, D3 also contains abstractions for dealing with animations and movement such as the functionality in NeuraViz for dragging the graph around and zooming in and out. The modular nature of the library also makes it perfect for dealing with the programmatic generation of complex visualizations such as those necessary in this project.

4.1.2. Server

When deciding what programming language to use for the server side of NeuraViz, a major consideration was the integration with the uploaded neural network models and the complexities of parsing them. The eventual decision was that it would not just be easier, but also more maintainable to include the libraries themselves and then extract the necessary information from the objects rather than parsing the model files manually. Since most popular machine learning frameworks are written for Python, it made sense to write NeuraViz’s server in Python as well.

To accommodate this language choice fully, Quart [11] was selected for managing the server side of the application. Quart is a partial rewrite of the popular Python web framework Flask, with support for asynchronous operations baked in. In NeuraViz, Quart is simply used as an API manager and to serve the static frontend compiled with Svelte, though it does have support for compiling and serving the HTML, CSS, and JS necessary to run a web application itself. Managing an API in Quart is as simple as defining async handler functions and adding Quart’s decorator to specify what route should call that function. While the parsing operations are complex, the number of endpoints that this API requires is relatively small. Table 1 shows the full list of routes that the server provides.

Endpoint	POST /graph
Description	Upload the graph model and retrieve the intermittent representation.
Inputs	The graph model as a file in the request
Responses	200 OK - Graph representation is returned 400 Bad Request - The file was not valid 501 Not Implemented - The file type is not supported
Endpoint	GET /tikz
Description	Retrieve the session's graph as a LaTeX file using Tikz (in light mode)
Inputs	None
Responses	200 OK - LaTeX is returned as a string 400 Bad Request - No graph was found in the session
Endpoint	GET /tikz/dark
Description	Get the dark mode version of the Tikz representation
Inputs	None
Responses	200 OK - LaTeX is returned as a string 400 Bad Request - No graph was found in the session

Table 1. Server API Endpoints

Currently, NeuraViz supports visualizing models creating with Keras and Pytorch. As such, both of those frameworks are also included in the server of the application. When the graph endpoint is hit, the correct framework is determined based on the file extension of the uploaded file. From there, the framework is used to decode the model into a Python object and that Python object is then parsed to retrieve the information needed for NeuraViz's visualization.

4.1.3. Data Layer

NeuraViz has a relatively simple data layer as the only persistent data is session information. Due to this simple nature, MongoDB [6] was decided on for storing the data. Mongo is a NoSQL database implementation that Python has good support for through the PyMongo package [9]. Models were written for each of the constituent parts of the stored graph as well as for the session itself. A custom module was also written for managing the sessions to encourage session management to happen for each API endpoint.

4.2. Development

The development of NeuraViz was completed over the course of 22 one-week long sprints. The first two sprints comprised primarily of designing the application. Sprints 3-10 focused on building the main functionality of the application for models uploaded with Pytorch. The remaining sprints added Keras support and additional features.

Design work on the application began by thinking through a set of desired features. Since NeuraViz isn't built for a particular client, this primarily consisted of the developer coming up with ideas and those ideas being vetted by the project advisor to come up with a final concept for the project. Once there was a solid understanding of at least the basic functionality that the application would fulfill, a user interface mockup (see Figure 6) was created to give a visual representation of what the application would look like. This design step also served to iron out some flaws before development on the actual application began. The UI mockup was heavily referenced when creating the actual application UI, and the final product is very similar to what the mockup originally showed.

Once the design was ready, development began. The first step was learning the two new frameworks (namely Svelte and Quart) and working to get them integrated with each other. The developer had previous experience with ReactJS, so Svelte's similar structure made the learning curve relatively small, and the concepts were picked up quickly. With experience building web application backends, Quart was also relatively easy to pick up and get working. The first few sprints after design were spent getting these two frameworks set up, laying out the basic structure of the frontend and following the mockups by following the mockups, and setting up the backend to serve the compiled frontend and handle the necessary API endpoints.

After the basic structure of the application was in place, the next step was figuring out how to use D3.js to create the visualization of the graph. Since the server-side graph parser had not been completed at this point, the graph structure was hard-coded in the application to allow for quick iteration on what the data needed to look like for the visualization to work and be as easy as possible to transmit across the network. An additional major step in this process was getting the pan and zoom functionality of the graph visualization working properly. While D3 provides some of the functionality, getting it working in the way that NeuraViz needed was a significant challenge that required a lot of trial and error.

The next major step was getting the server to parse the uploaded model files. This was a significant challenge as the structure of the model files is complex and not well documented. Because of this complexity, it was decided to let the frameworks parse the models themselves, and to extract the necessary information from the objects that the frameworks produced. While this cut out some of the complexity of trying to parse the files the frameworks produce, it still maintained the complexity of dealing with the object-oriented representation of the objects in Python. Most machine learning frameworks expect developers to build, train, and execute models through their frameworks, but parsing those models out to alternative representations is generally not an intended use-case. Because of this, documentation on how the objects are structured and exactly where the necessary information is stored is sparse. An extensive amount of time was required to investigate the object structure in each framework and determine how that information can be converted into NeuraViz's standardized representation. To speed up this process, Pytorch alone was focused on at first, and then

support for Keras models was added later.

Once server support for parsing Pytorch models and frontend graph visualization were complete, additional features could be added to further enhance the application. Some of the more useful additional features included adding support for activation function representations on the graph, exporting the visualization to LaTeX or SVG, and color-coding model parts to indicate more information at a glance. These features were added toward the second half of the development process.

4.3. Deployment

Being a web application, deployment of NeuraViz is a complex process that involves multiple steps. The first step is to compile the Svelte frontend into static HTML, CSS, and JavaScript files. This is done by running the command `npm run build` in the frontend directory. This command compiles the Svelte code into a set of static files that can be served by a web server. The next step is to copy these files into the server directory so the backend portion of the application can serve up these files. The final step is to start the Quart server, which listens for incoming requests on port 5000 by default. Once the server is running, the application can be accessed by navigating to `http://localhost:5000` in a web browser.

However, running the the application on localhost does not make it available the internet as a whole. To make the application accessible to the public, it was deployed to a private web server and routed from `https://neuraviz.bennettwendorf.dev`. The private server runs Ubuntu and PM2 [8] was used to help manage the process and allow it to automatically restart when the server restarts.

To speed up deployment to the server, Github Actions was used to automatically build and deploy the application whenever a new commit was pushed to the main branch. The action itself works by setting up a vpn connection to the server and using SSH to run a script on the server. The action configuration is shown in Listing 1.

The referenced build script begins by pulling new changes from the remote Github repository. It then cleans all old build files to ensure that everything gets rebuilt fresh. Once the environment is clean, it ensures that all project dependencies are up to date on the server, and installs any new ones if needed. It then recompiles the Svelte frontend and copies it into the necessary location in the backend. Finally, the script uses pm2 to restart the server and serve the new version of the application.

```

1 name: Deploy
2
3 on:
4   push:
5     branches: [ "main" ]
6
7   # Allows the action to be run manually
8   workflow_dispatch:
9
10 jobs:
11   deploy:
12     runs-on: ubuntu-latest
13
14     steps:
15       - name: Install iproute2
16         run: sudo apt-get update && sudo apt-get install -y iproute2
17
18       - name: Set up WireGuard
19         uses: egor-tensin/setup-wireguard@v1.2.0
20         with:
21           endpoint: '${{ secrets.WIREGUARD_ENDPOINT }}'
22           endpoint_public_key: '${{ secrets.
WIREGUARD_ENDPOINT_PUBLIC_KEY }}'
23           ips: '${{ secrets.WIREGUARD_IPS }}'
24           allowed_ips: '${{ secrets.WIREGUARD_ALLOWED_IPS }}'
25           private_key: '${{ secrets.WIREGUARD_PRIVATE_KEY }}'
26
27       - name: Set up SSH
28         id: ssh
29         uses: invi5H/ssh-action@v1
30         with:
31           SSH_HOST: '${{ secrets.SSH_HOST }}'
32           SSH_PORT: '${{ secrets.SSH_PORT }}'
33           SSH_USER: '${{ secrets.SSH_USER }}'
34           SSH_KEY: '${{ secrets.SSH_KEY }}'
35
36       - name: Run Build Script Over SSH
37         run: ssh '${{ steps.ssh.outputs.SERVER }}' "/home/${{ secrets.
SSH_USER }}/NeuraViz/src/build.sh"

```

Listing 1. Github Action Configuration

5. Testing

5.1. Overview

Per the scrum software design process, software verification and validation were performed continuously throughout the development process. To accommodate this and ensure that thorough testing was performed, a *Testing Required* status column was added to the project board. Once projects finished development, they were moved to this column to ensure that testing was completed before the project was considered complete.

5.2. Unit Testing

During development, a large amount of unit testing was defined and performed to support the continuous and thorough testing process that was defined at the outset of the project. Due to the small size of the development team, test cases were both written and executed by the developer. Test processes that were used during the development of NeuraViz can be broken down into two major categories: frontend or client testing and backend or server testing. In total, 31 automated test cases and 10 manual test cases were written and executed for the application. A one hundred percent pass rate was achieved before each project was considered complete.

5.2.1. Frontend Testing

The frontend of NeuraViz was tested using a combination of automated and manual unit testing. Testing a user interface is a challenge due to its primarily visual nature, but tools exist to help automate this process. For this application, the automated testing platform Playwright [5] was used to develop test cases that could be executed on each individual project and at any time. Automated testing in this manner is especially useful to ensure that test cases are performed both consistently and repeatably, with minimal human error.

Playwright test cases work by defining actions that should be taken in the user interface, and asserting various results. These results can include a variety of things, such as the presence of certain elements on the page, that a certain number of elements exist, or text on an element. Test cases are written directly in JavaScript and by default run in an automated headless browser so as not to interfere with other work that is being done. An example of a test case that was written for NeuraViz is shown in Listing 2. This test case ensures that clicking the model upload button with no model file selected leads to the validation text staying red and saying "Model is not valid". On line 3 an example can be seen of an action being taken; in this case, the button with the name "Upload" is clicked. On lines 4-6, assertions are made about the state of the page after the action is taken.

5.2.2. Backend Testing

Like the frontend, NeuraViz's backend code was also tested using a mix of automated and manual unit testing. While most of the functionality could easily be tested automatically,

```

1  test('Upload no model: Sidebar', async ({ page }) => {
2      await page.goto('/');
3      await page.getByRole('button',
4          { name: 'Upload' }).click();
5      await expect(page.locator('#model-upload'))
6          .toHaveValue('');
7      await expect(page.locator('#model-validation'))
8          .toHaveText('Model is not valid');
9      await expect(page.locator('#model-validation'))
10         .toHaveClass(/text-error/);
11  });
12

```

Listing 2. Playwright Test Case Example

features such as svg and tikz export need to be inspected visually, and so were included in manual testing rather than automated testing. As with the frontend, however, as much functionality as possible was tested automatically to reduce the overhead in maintaining a test suite and further enforce a continuous testing process. The backend automated test suite was developed with the Python testing framework Pytest [10]. This framework allows for the easy definition of test cases and the execution of those test cases in a repeatable and consistent manner. An example of a test case that was written for NeuraViz is shown in Listing 3. This test ensures that the json representation of an uploaded model matches what is expected, specifically for the PyTorch Iris model. The test case first uploads the model with Quart’s test client, then asserts that the result exactly matches the expected result.

```

1  @pytest.mark.asyncio
2  async def test_get_graph_nominal_pytorch_iris(self) -> None:
3      test_client = app.test_client()
4      file_name = "STE_Iris.pth"
5      files = {
6          'files[]': FileStorage(stream=open(f"testing/input_files/
7      pytorch/{file_name}", 'rb'), filename=file_name)
8      }
9      response = await test_client.post('/api/graph/', files=files,
10     headers={'Content-Type': 'multipart/form-data'})
11     assert response.status_code == 200
12     data = await response.get_json()
13     with open("src/backend/tests/expected_results/get_graph/pytorch/
14     nominal_STE_Iris.json", 'r') as file:
15         expected_data = load(file)
16         assert data == expected_data

```

Listing 3. Pytest Test Case Example

5.3. Regression and Integration Testing

Regression testing during the development of NeuraViz was handled primarily implicitly during the test phase on each individual project and before that project was merged and considered complete. During this phase, new test cases were written for the project. Once written, these test cases along with all exiting automated test cases were executed to ensure that nothing new or old contained a regression. In addition, manual test cases were investigated to ensure that the new project did not break any existing functionality. To further ensure that the product remained stable, test cases were also run before each version release and publication to the production server.

While unit tests can only really cover functionality on the frontend or backend, testing was also performed during feature development and periodically to ensure that features all worked together seamlessly. This was done by running the application and testing a feature from start to finish, typically referred to as end-to-end testing. This type of testing was also performed extensively before the release of each version to the production server.

6. Security

6.1. Overview

6.2. Threat Model

6.3. Session Management

6.4. Web Application Security

7. Conclusion

7.1. Overview

7.2. Challenges

7.3. Future Work

8. References

- [1] Axios. Axios. <https://axios-http.com/>. accessed: 03.24.2024.
- [2] Tailwind CSS. Tailwind css. <https://tailwindcss.com/>. accessed: 03.24.2024.
- [3] D3.js. D3.js. <https://d3js.org/>. accessed: 03.24.2024.
- [4] Gagan Gurung, Rahul Shah, and Dhiraj Jaiswal. Software development life cycle models-a comparative study. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, pages 30–37, 07 2020.
- [5] Microsoft. Playwright. <https://playwright.dev/>. accessed: 03.26.2024.
- [6] MongoDB. Mongoddb. <https://www.mongodb.com/>. accessed: 03.24.2024.
- [7] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.
- [8] PM2. Pm2. <https://pm2.keymetrics.io/>. accessed: 03.24.2024.
- [9] PyMongo. Pymongo. <https://pymongo.readthedocs.io/en/stable/>. accessed: 03.24.2024.
- [10] Pytest. Pytest. <https://pytest.org/>. accessed: 04.04.2024.
- [11] Quart. Quart. <https://pgjones.gitlab.io/quart/>. accessed: 03.24.2024.
- [12] Svelte. Svelte. <https://svelte.dev/>. accessed: 03.24.2024.
- [13] Flowbite Svelte. Flowbite svelte. <https://flowbite-svelte.com/>. accessed: 03.24.2024.
- [14] Tutorialspoint. Sdlc - agile model. https://www.tutorialspoint.com/sdlc/sdlc_agile_model.htm. accessed: 03.14.2024.
- [15] Tutorialspoint. Sdlc - iterative model. https://www.tutorialspoint.com/sdlc/sdlc_iterative_model.htm. accessed: 03.05.2024.
- [16] Tutorialspoint. Sdlc - waterfall model. https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm. accessed: 03.05.2024.

9. Appendices