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.

Dr. Jason Sauppe
Examination Committee Chairperson

Date

Dr. Rig Das
Examination Committee Member

Date

Dr. Dipankar Mitra
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 process, design, implementation, deployment, testing, and security 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 and Computer Engineering 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 thank I can possibly express in words.

Table of Contents

List of Tables

List of Figures

Glossary

Artificial Neural Network

A computational model that is inspired by the way biological neural networks in the human brain process information. It is composed of layers of nodes, each of which is connected to the next layer via links. Each connection has a weight associated with it, and the network learns by adjusting these weights based on the input data. The network can be trained to recognize patterns in data, make predictions, and perform other tasks.

Activation Function

A function that determines the output of a node in a neural network based on the weighted sum of its inputs. Common activation functions include the sigmoid function, the hyperbolic tangent function, and the rectified linear unit (ReLU) function. The activation function introduces non-linearity into the network, allowing it to learn complex patterns in the data.

LaTeX

LaTeX is a document markup language for the TeX typesetting program.

TikZ

TikZ is a graph drawing package for LaTeX that allows users to create high-quality diagrams and illustrations.

Scalable Vector Graphics (SVG)

SVG is an XML-based vector image format for two-dimensional graphics with support for interactivity and animation. It is commonly used in web applications.

Raster Graphics

Raster graphics are digital images composed of a grid of pixels. Each pixel has a specific color value, and the image is displayed by rendering the pixels on a screen. Common raster graphics formats include JPEG, PNG, and GIF.

Python

Python is a high-level programming language known for its simplicity and readability. It is widely used in data science, machine learning, web development, and other fields.

PyTorch

PyTorch is a popular open-source machine learning library for building artificial neural networks. It is known for its flexibility and ease of use, making it a popular choice for researchers and developers.

Keras

Keras is an open-source deep learning library written in Python that provides a high-level interface for building and training neural networks. Distributed by the Tensorflow team, it is a high-level API on top of Tensorflow and other popular frameworks.

TypeScript

TypeScript is a superset of JavaScript that adds static typing and other features to the language. It is commonly used in web development to improve code quality and maintainability.

Parsing

Parsing is the process of analyzing a sequence of symbols to determine its grammatical structure. In the context of this project, it refers to the process of extracting information from a machine learning model object to generate a visual representation of the model's architecture.

Sprint

A sprint is a short, time-boxed period during which a development team works to complete a set amount of work. Sprints are a key component of the Scrum agile framework and are typically 1-4 weeks long.

JavaScript Object Notation (JSON)

JSON is a lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate. It is commonly used to transmit data between a server and a web application.

1. Introduction

1.1. Overview

This project aims to develop a software system to visualize the architecture of artificial neural networks, or neural networks for short. Neural networks 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. An example of what one of these networks might look like can be found in Figure 1. 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.

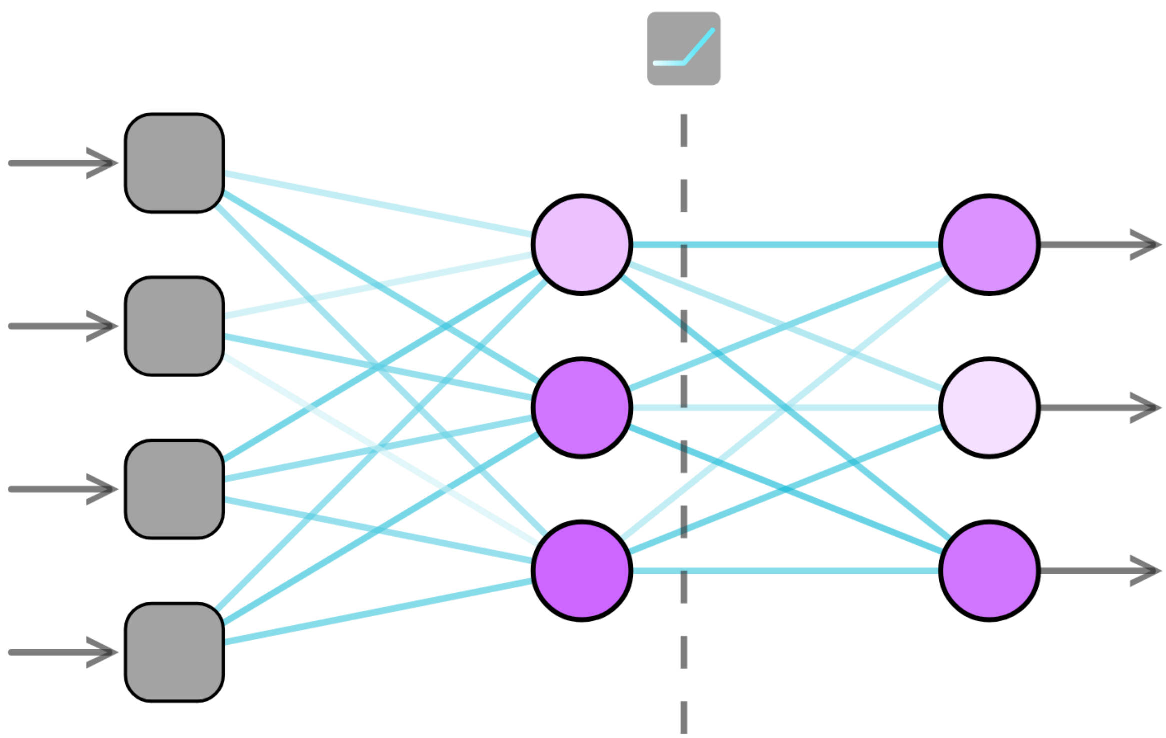

Figure 1. Example of a Neural Network

This project developed a web application that allows users to upload pre-trained neural network models and generate visual representations of their architecture. The application supports models trained using popular machine learning frameworks such as PyTorch [1] and Keras [2]. The resulting graph structure is visualized in a pannable and zoomable Scalable Vector Graphic (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 is designed to be user-friendly and accessible to a wide range of users, including students and researchers. It is implemented using modern web technologies and deployed as a web service that can be accessed from any device with an internet connection. The project was developed according to best practices in software engineering, including requirements analysis, design, implementation, testing, and deployment. The resulting application is a valuable tool for anyone working with neural networks, but in particular it will be beneficial for 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 [3] is a popular tool, but only natively supports TensorFlow and Keras models. In researching existing tools such as TensorBoard, Netron [4], and ENNUI [5], it was found that none of them supported the same range of formats as NeuraViz. This project developed 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 it supports a wider range of model formats than other offerings.

1.3. Goals

With an aim to solve the issues with existing tools, NeuraViz was 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 visualization of the neural networks themselves, with minimal distractions. The use of color and shape in the visualizations helps to make the network architecture more understandable at a glance, allowing the user to quickly identify neurons and connections that are the most important. Where needed, users can also zoom in or out and click on elements to see more detailed information.

Portability was another key goal of the development. Modern web technologies ensure the application is accessible from a range of devices, though it is optimized for a desktop or laptop experience. Because 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 [6] 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 that 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 elements of the network that perform computations. 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 process input (details can be found in "Neural Networks and Deep Learning" [7]). In general, neurons in a layer are directionally linked to all the neurons in the previous layer, and all the neurons in the next layer, with data traveling on these links from one layer to the next. These connections are represented as directed 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 directed 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 value that can be interpreted as a probability. In NeuraViz, activation functions are shown at the layer-level and represented as icons near 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 functional and non-functional requirements of the software. This section discusses life cycle models considered for NeuraViz's development, the model that was eventually chosen, and modifications to the model that were 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 [8]. 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 2 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,

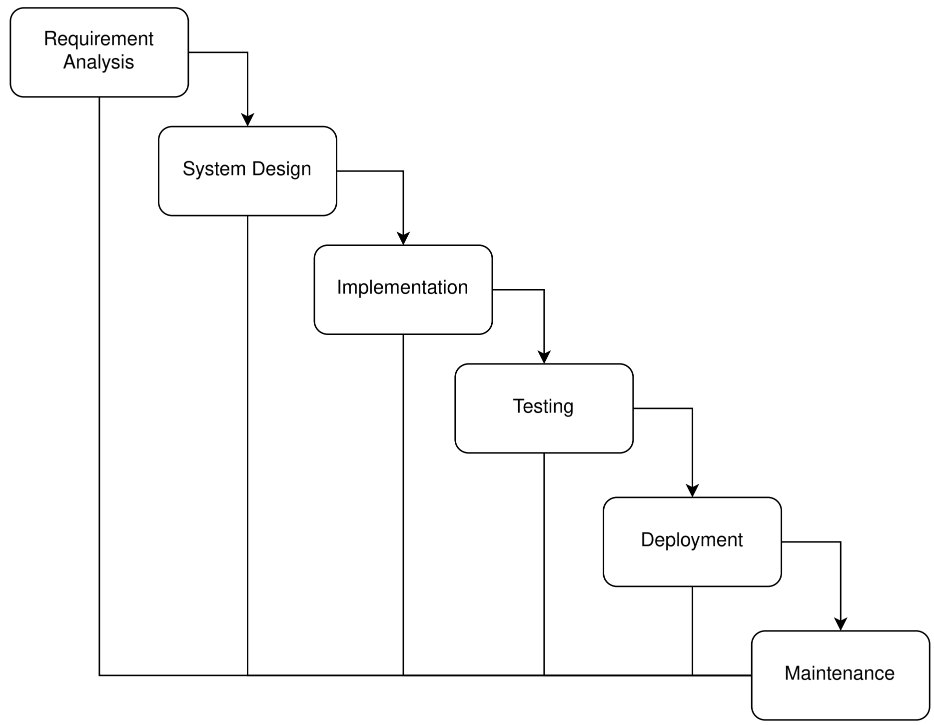

Figure 2. Waterfall Model Diagram (recreated from [9])

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 was determined to be a poor choice.

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, shown in Figure 3, roughly match those in the waterfall model, including a requirements analysis phase, a design phase, an 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 for the client to give feedback on the project sooner and more frequently.

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.

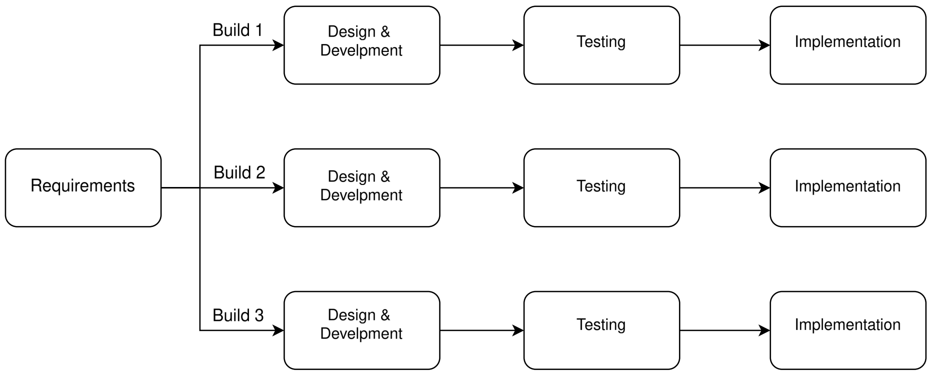

Figure 3. Iterative Model Diagram (recreated from [9])

2.2.3. Agile (Scrum) Model

In contrast to the other models, agile models, and specifically scrum, are designed with the express intent to adapt to changing requirements quickly. Because the exact requirements and design for NeuraViz were not well-known 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 focuses 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 4. While the diagram shows a two to three month timeline per iteration, scrum more typically follows a shorter sprint length.

Because NeuraViz only had one developer, the scrum model did not fit perfectly. However, many aspects of the model did 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 stayed 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 did not match exactly with any part of the scrum model, they served 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.

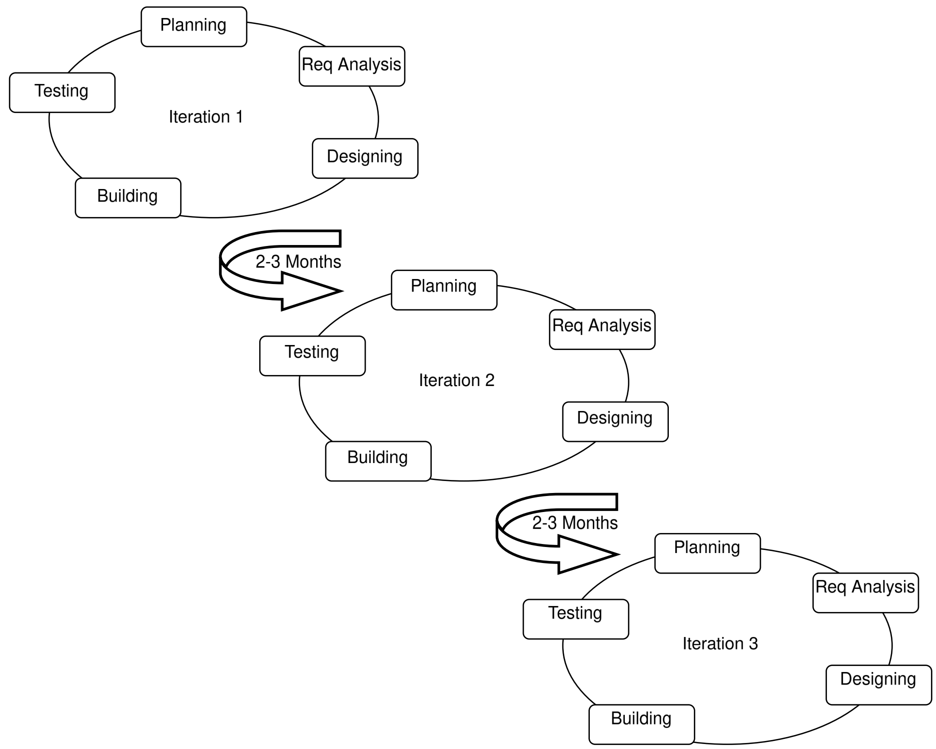

Figure 4. Scrum Model Diagram (recreated from [9])

2.3. Development Process Technologies

An important part of the scrum model is managing small, independent sprint tasks that can be completed in a short amount of time. To help manage these sprint tasks, Jira's [10] 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 a sprint. 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 at any given time. The scrum board for sprint 25 (April 9th, 2024 - April 16th, 2024) can be seen in Figure 5. 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 sprint tasks as well.

For this project, Jira's time estimation and tracking features were used to help keep development on track. Before a sprint task was scheduled into a sprint, it was estimated how long the task would take to complete. Most commonly, these estimations were in integer hour values due to the complexity of estimating development time. When a task was worked on, a time tracking action was placed on the project to record how long was spent on the task. This helped to ensure that tasks were not taking significantly longer than expected and that the project was on track to be completed on time.

In addition to Jira, Github was used to help keep track of individual sprint tasks using separate feature branches. A new branch was created for each task, with the name of the

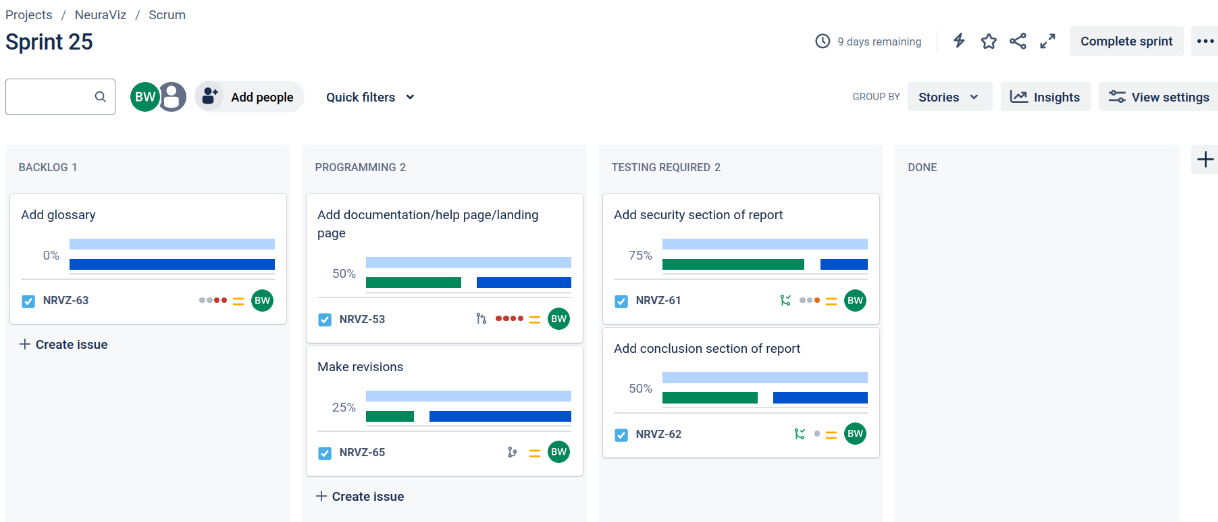

Figure 5. Jira Scrum Board

branch including the task key from Jira and a brief description. Jira's integration with GitHub provided a link to see what development phase the code was in directly from the Jira task, including whether a pull request had been created or merged.

2.4. Functional Requirements

Because 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 being able to click or hover on various network components.

In addition to the functionality of viewing the network itself, user stories were also documented 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 a TikZ picture.

The full set of user stories is documented in the user stories document found in the "docs" directory of the Github repository. 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.
- For visually impaired users, aria labels exist for screen readers where possible.
- If a page takes a long time to load, skeletonized components are shown to indicate that the page 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 on the server unnecessarily.

3. Design

3.1. Overview

NeuraViz follows a fairly standard client-server 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 diagrams in Figures 6 and 7 show 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.

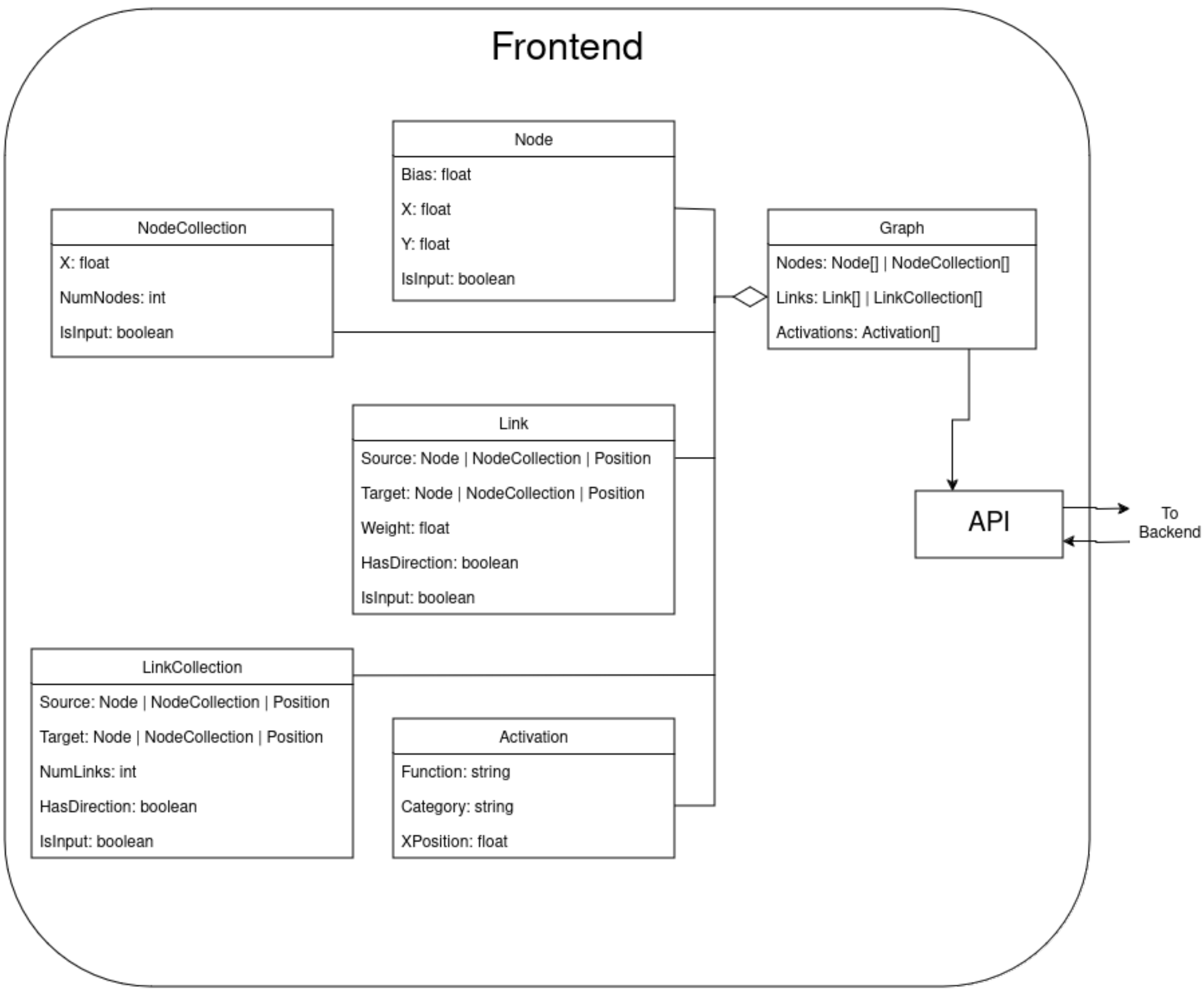

Figure 6. Frontend UML Class Diagram

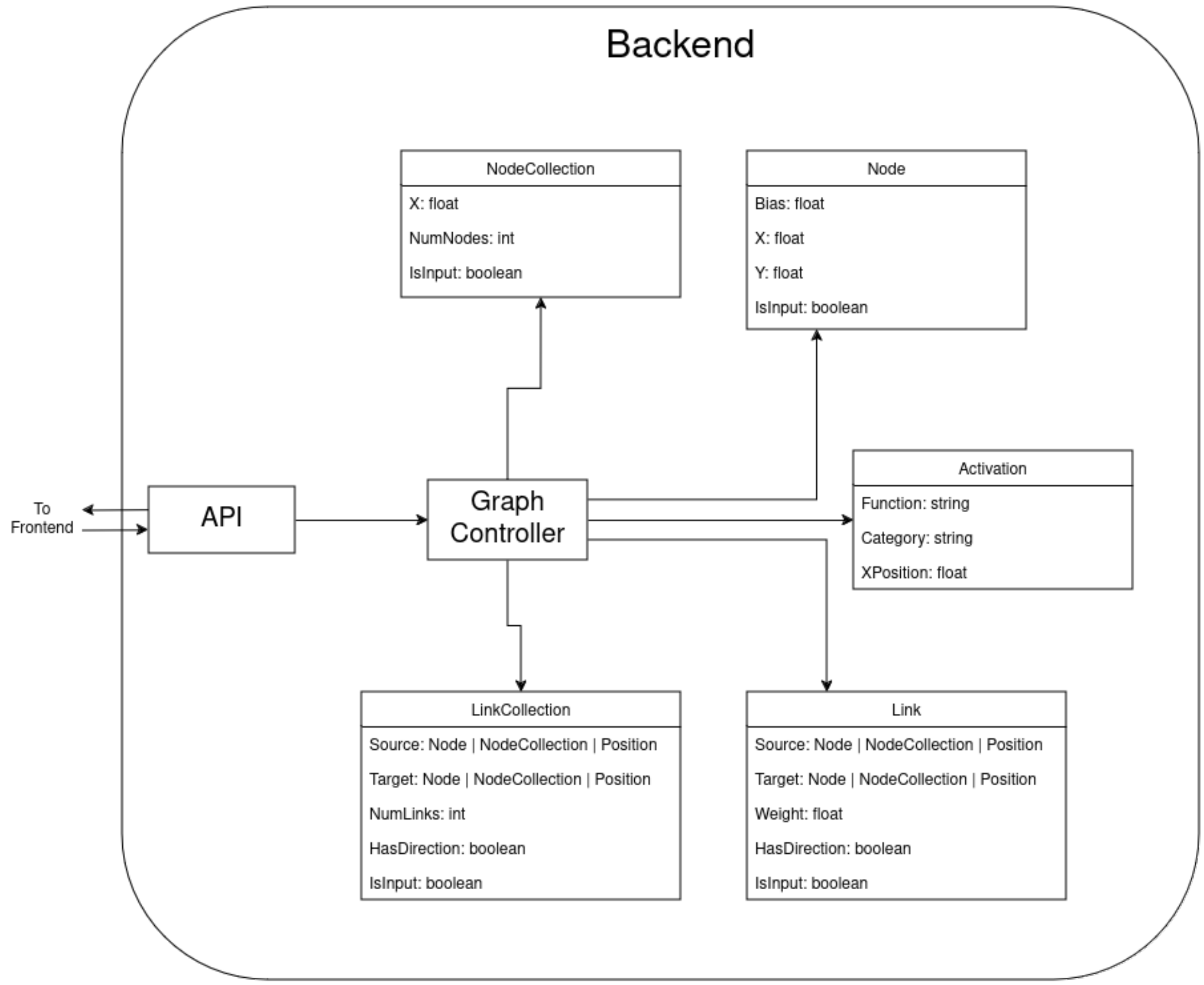

Figure 7. Backend UML Class Diagram

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. Node objects represent individual neurons in the graph representation of the network, and are used for graph layers that are smaller than 10 nodes. For layers that are too large, the graph representation instead contains a Node Collection object that represents the layer as a whole. Link objects and Link Collection objects operate in a similar way, representing edges or groups of edges in the network. 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 contains all of the constituent components in the format that the frontend uses to create the SVG rendering of the network. As shown in the UML diagram, the frontend also includes 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 7, the backend includes objects that almost perfectly mirror the frontend components. However, on the server, these components are all related to the graph controller which is the component responsible for the actual graph parsing. The controller also handles additional requests for retrieving a stored model and converting the representation into various formats. As with the frontend portion of the application, the backend includes 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 either maintain the graph's representation for a longer period of time, or to send the graph back and forth between the client and server multiple times. Because the graph representation can be quite large, it was decided to use a NoSQL database, namely MongoDB, to store the parsed graph information as part of 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 possible future features of saving the graph representation to 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 a raster graphics editor called GIMP [11] 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. Because NeuraViz operates on a single page with one main piece of functionality, the required mockup was fairly simple. Figure 8 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 actual components currently use a slider, future development may include more complex settings that would use this. 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.

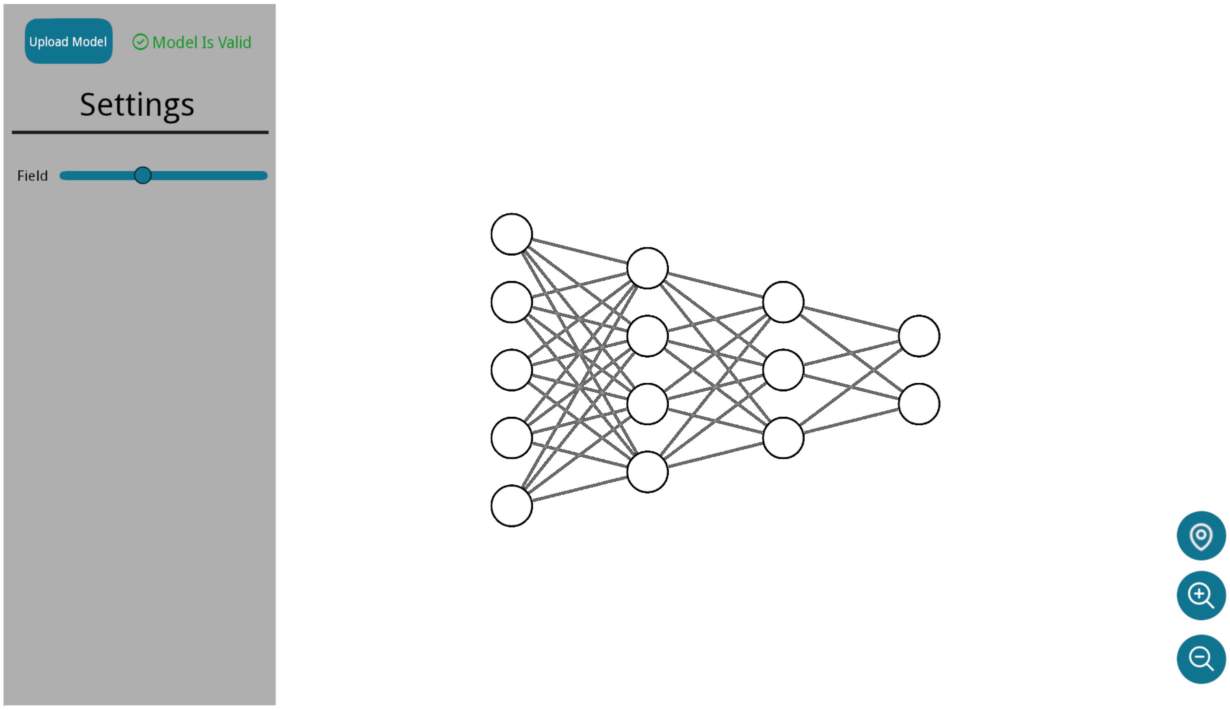

Figure 8. User Interface Mockup

3.4.1. Final Interface

The final interface of NeuraViz is shown in Figure 9. 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.

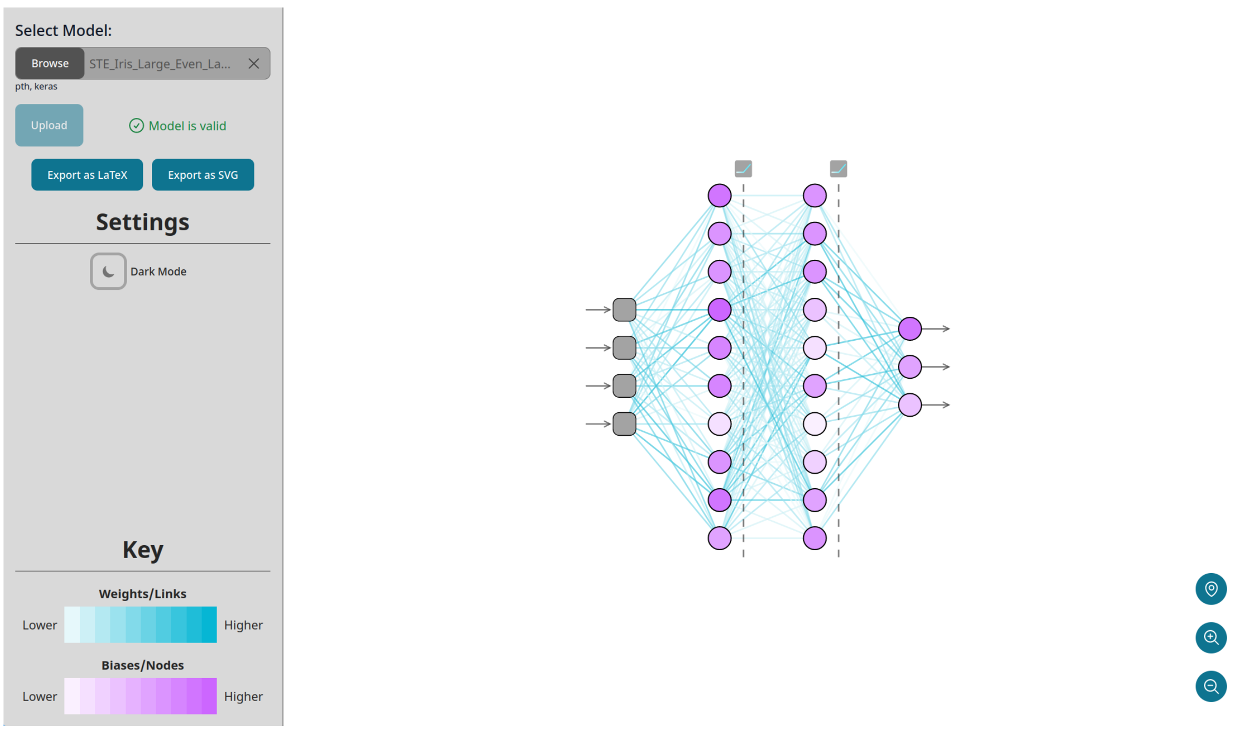

Figure 9. Final Interface

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 used by this project. 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 for user interaction, and therefore it must be visually appealing and responsive. In researching available web frameworks for building frontend web applications, a number of possibilities were considered including ReactJS [12], Angular [13], and Svelte [14]. While ReactJS would have been a good choice because of its popularity and the developer's prior experience with it, Svelte was eventually chosen for its simplicity and performance.

Svelte 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 [15] 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 [16], 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 [17] 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 is being the most critical feature of NeuraViz and it was a major consideration when determining client technologies. To aid in the construction of the graph visualization, the D3.js library [18] was used. D3 is a highly customizable JavaScript library for creating visualizations of data in a variety of 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

Prior to beginning development on NeuraViz, the developer spent a semester doing an independent research project comparing various machine learning frameworks. During this research, PyTorch, Keras, and scikit-learn [19] were explored in depth. Standardized models were also written in each of the frameworks, which were then used to test NeuraViz's parsing algorithm. One of the models was trained to classify data in the famous Iris data set [20], a set of data containing measurements of three different species of iris flowers and their corresponding species. In addition, a much larger model was created for classifying handwritten digits from the MNIST data set [21]. The smaller iris models helped test basic functionality of NeuraViz with layers of nodes that were small enough to visualize in full. When testing the MNIST models, it was also discovered that some models would be too big to visualize effectively, so it was decided to add the functionality of collapsing layers into Node Collections and Link Collections. Variations on these models were also added to test combinations of collapsed and uncollapsed layers. These models can all be found in the test/input_files directory in the NeuraViz repository [22].

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. Because 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 [23] was selected for managing the server side of the application for its simplicity, great documentation, and support for asynchronous request handling. Quart is a partial rewrite of the popular Python web framework Flask [24], with support for asynchronous operations included. 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 JavaScript 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 created with Keras and PyTorch. As such, both of those frameworks are also included in the server portion 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, which is then parsed to retrieve the information needed for NeuraViz's visualization. The object parsing algorithm differs slightly between the frameworks, but the general idea is the same. Both frameworks store the model as a series of layers, each of which contains information about the layer's type, activation function, and input and output shapes. From the input and output shapes, the number of nodes in each layer can be determined. The layer data also contains information about the weights on links and the biases on nodes, which are incorporated into the 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 [25] was decided on for storing the data. Mongo is a NoSQL database implementation that Python has good support for through the PyMongo package [26]. 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 be integrated into each API endpoint during development.

4.2. Development

The development of NeuraViz was completed over the course of 22 one-week long sprints. The first two sprints consisted 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. PyTorch was chosen first because it was identified as one of the simplest to work with during the preliminary research and because it is incredibly popular.

Design work on the application began by thinking through a set of desired features. Because 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 8) was created to give a visual representation of what the application would look like. This design step also served to iron out some key focus points such as the desire for pan and zoom functionality and unobtrusive locations for settings and controls 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 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. Because 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 smoothly and dynamically integrated with the NeuraViz logic was a significant challenge that required a lot of trial and error. In particular, a major challenge early on was ensuring that the panning was smooth. The initial implementation was very choppy and hard to use, and correcting that took multiple hours.

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 could be converted into NeuraViz's standardized representation. To speed up this process, PyTorch alone was focused on initially, 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 to 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 22.04 LTS and PM2 [27] 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 [28] 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 build script referenced in the configuration file 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 web server process and serve the new version of the application. The full build script is shown in Listing 4.

```yaml
name: Deploy

on:
    push:
        branches: [ "main" ]

    # Allows the action to be run manually
    workflow_dispatch:

jobs:
    deploy:
        runs-on: ubuntu-latest

        steps:
            - name: Install iproute2
              run: sudo apt-get update && sudo apt-get install -y iproute2

            - name: Set up WireGuard
              uses: egor-tensin/setup-wireguard@v1.2.0
              with:
                endpoint: '${{ secrets.WIREGUARD_ENDPOINT }}'
                endpoint_public_key: '${{ secrets.WIREGUARD_ENDPOINT_PUBLIC_KEY }}'
                ips: '${{ secrets.WIREGUARD_IPS }}'
                allowed_ips: '${{ secrets.WIREGUARD_ALLOWED_IPS }}'
                private_key: '${{ secrets.WIREGUARD_PRIVATE_KEY }}'

            - name: Set up SSH
              id: ssh
              uses: invi5H/ssh-action@v1
              with:
                SSH_HOST: ${{ secrets.SSH_HOST }}
                SSH_PORT: ${{ secrets.SSH_PORT }}
                SSH_USER: ${{ secrets.SSH_USER }}
                SSH_KEY: ${{ secrets.SSH_KEY }}

            - name: Run Build Script Over SSH
              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 development. To accommodate this and ensure that thorough testing was performed, a *Testing Required* status column was added to the scrum board. Once sprint tasks finished development, they were moved to this column to ensure that testing was completed before the sprint task was considered complete.

5.2. Unit Testing

During development, a large amount of unit testing was created 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 sprint task 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 primarily due to its visual nature, but tools exist to help automate this process. For this application, the automated testing platform Playwright [29] was used to develop test cases that could be executed against each individual sprint task 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,

```
test('Upload no model: Sidebar', async ({ page }) => {
    await page.goto('/');
    await page.getByRole('button',
        { name: 'Upload' }).click();
    await expect(page.locator('#model-upload'))
        .toHaveValue('');
    await expect(page.locator('#model-validation'))
        .toHaveText('Model is not valid');
    await expect(page.locator('#model-validation'))
        .toHaveClass(/text-error/);
});
```

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, as much functionality as possible was tested automatically to reduce the overhead in maintaining a test suite and to further enforce a continuous testing process. The backend automated test suite was developed with the Python testing framework Pytest [30]. 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.

```
@pytest.mark.asyncio
async def test_get_graph_nominal_pytorch_iris(self) -> None:
    test_client = app.test_client()
    file_name = "STE_Iris.pth"
    files = {
        'files[]': FileStorage(stream=open(f"testing/input_files/
pytorch/{file_name}", 'rb'), filename=file_name)
    }
    response = await test_client.post('/api/graph/', files=files,
headers={'Content-Type': 'multipart/form-data'})
    assert response.status_code == 200
    data = await response.get_json()
    with open("src/backend/tests/expected_results/get_graph/pytorch/
nominal_STE_Iris.json", 'r') as file:
        expected_data = load(file)
        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 implicitly during the test phase on each individual sprint task and before that sprint task was merged and considered complete. During this phase, new test cases were written for the feature. Once written, these test cases along with all existing 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 feature 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 the 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

NeuraViz does not store user information long-term in any capacity, so security of data storage was not a major point of concern during development. However, it is feasible that users might upload proprietary models to the system, and so the security of information during transportation was an important consideration during development. Steps taken to combat security threats are outlined in this section.

6.2. Web Application Security

Web applications are inherently vulnerable due to their constant communication of data across the internet and the fact that the client has access to the client-side source code. For NeuraViz, this is less of an issue than other applications because the user is not modifying any stored data on the server that could potentially be corrupted by malicious data. However, precautions were still taken to ensure that the application is secure.

NeuraViz is hosted over HTTPS, which encrypts all data sent between the client and server via the Transport Layer Security (TLS) protocol. This ensures that data is not intercepted or tampered with during transit. In production, NeuraViz is hosted on a dot dev domain, which enforces HTTPS by default, completely disabling users from connecting over unsecure HTTP. In addition, the production application is hosted on a secure private server with a certificate provided by Let's Encrypt [31], which is a free, automated, and open certificate authority. This certificate is used to verify that the server is who it claims to be, and is used to encrypt the data sent between the client and server, providing the user with additional peace of mind when using the application.

Because this project does not take any user input directly, the risk of cross-site scripting attacks or other similar threats related to input sanitization is minimal. However, there is still some risk in the retrieval and storage of uploaded models from the user. To protect the application server itself, uploaded files are never executed directly. Upon upload, the file is stored temporarily on disk and the corresponding model framework is used to parse the file contents, based on the file extension. Because all supported frameworks are open source and well tested, the security of their file parsing is very likely to be robust and secure. If the framework encounters an issue during parsing such as an invalid portion of the file, the file is deleted from disk and the user is notified of the error. Both PyTorch and Keras store their models in a binary object format which contains no executable code. Once the file is parsed by the framework, NeuraViz uses the respective framework's own Python objects to generate the representation of the model, again with minimal execution of third-party code. Throughout this process, NeuraViz is very particular about the kind of data it expects from the model, and will quickly fail if the data is not as expected. This practice of essentially white-listing valid data also helps prevent any unexpected behavior from the application that could negatively impact user experience or security.

6.3. Session Management

To accommodate an elevated user experience, NeuraViz stores session data in a MongoDB database on the server. The stored session information consists of a session identifier, the full representation of the model in a similar format to what is returned to the user after uploading a file, and a timestamp. The session identifier is a randomly generated string that is unique to each session, and as such is not tied to a particular user in any way, making it nearly impossible to associate a stored model with a particular user other than by the session token itself. The session identifier is stored as a cookie by the client and can be used to retrieve the stored model for the duration of the session. To mitigate possible security risks, the database is only accessible from the server where the application is running. Furthermore, while the server is running, sessions are automatically pruned after 10 minutes of inactivity. This trade-off allows the user to continue making requests without the application needing to parse the model again, while providing a reasonable level of security for the stored model.

7. Conclusion

7.1. Overview

The NeuraViz project has successfully accomplished all of its primary goals in creating a user-friendly web application for visualizing artificial neural network architectures. It provides its users with the capability to upload pre-trained models from both PyTorch and Keras, visualize their structure, and export the visualizations to both SVG and LaTeX.

7.2. Challenges

A number of challenges and roadblocks were encountered throughout the development process, including a large amount of research into each framework and personal time management. Each of the machine learning frameworks used have complex internal data structures that they use to model neural networks and be able to run inferences and training on them. However, these data structures are generally only designed to be used internally, and are not well documented for external use. In addition, because neural network training and inferencing is complex, the objects used by the frameworks often contain a large amount of information that is not relevant to the visualization, and must be filtered out. To develop NeuraViz's parsing algorithm, extensive research was required to understand the internal data structures that are used by PyTorch and Keras, and to determine where necessary common information could be extracted to produce a cohesive experience for the user, regardless of how they chose to create their model in the first place.

As with many projects, time management was a constant struggle. Balancing development of NeuraViz with other obligations, whether school, work, or personal, was a significant hinderance to the rate of development on this project, and ultimately limited the feature set that was able to be completed. However, it was known at the outset that many of the goals of this project were extensive and would likely not be fully completed within the time frame of the project.

7.3. Future Work

Throughout the planning and development of this project, a number of additional potential features were identified that could be added to NeuraViz to improve its functionality and usability. These include additions such as adding support for more model types, visualizing more complex networks like recurrent or convolutional networks, and animating the visualization to show the flow of data through the network.

While PyTorch and Keras cover a substantial portion of the industry for creating, training, and deploying neural networks, there are a multitude of other frameworks that are used in various industries. To fully realize the potential that NeuraViz provides, ideally every major framework would be supported, allowing users to visualize their models in a consistent manner regardless of their chosen framework. This would require extensive research both into what frameworks are used in industry, and the exact internal structures they use to model neural networks. On top of that, maintaining the tool with the ever-increasing number of products on the market would be a monumental task. Hopefully with NeuraViz being open source and freely available, more developers will contribute and help expand this list.

Visualizing more complex network types was one of the initial goals of this project. However, development tended more toward a use-case in an educational environment, rather than an industry setting. While recurrent and convolutional networks are still beneficial to teach, they are more complex and not taught as often in an undergraduate academic capacity, and therefore were left out of the project. However, adding support for these types of networks would drastically increase the utility of NeuraViz's major function of providing a consistent representation of many different types of neural networks. Before adding support, some additional research would be required into existing visualization techniques for these types of networks. For example, Kwon et al. talk about a variety of approaches to visualizing convolutional networks [32].

With the tendency more toward education during this project's development, the ability to animate data through a smaller network would be a great addition. An instructor could show a visualization to a class, and step through each step of the inferencing process live, instantly seeing how changes in the input data affect the output. Implementing a feature like this would take a significant amount of work, so it was left out of the project scope for the time being.

Bibliography

[1] PyTorch. *PyTorch*. `https://pytorch.org/`. accessed: 04.12.2024.

[2] Keras. *Keras*. `https://keras.io/`. accessed: 04.12.2024.

[3] TensorBoard. *TensorBoard*. `https://www.tensorflow.org/tensorboard`. accessed: 04.12.2024.

[4] Lutz Roeder. *Netron*. `https://netron.app/`. accessed: 04.12.2024.

[5] Jesse Michel. *Ennui*. `math.mit.edu/ennui/`. accessed: 04.12.2024.

[6] Till Tantau. *TikZ*. `https://github.com/pgf-tikz/pgf`. accessed: 04.12.2024.

[7] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. URL: `http://neuralnetworksanddeeplearning.com/`.

[8] Gagan Gurung, Rahul Shah, and Dhiraj Jaiswal. "Software Development Life Cycle Models-A Comparative Study". In: *International Journal of Scientific Research in Computer Science, Engineering and Information Technology* (July 2020), pp. 30–37. DOI: `10.32628/CSEIT206410`.

[9] TutorialsPoint. *SDLC*. `https://www.tutorialspoint.com/sdlc`. accessed: 03.05.2024.

[10] Atlassian. *Jira*. `https://www.atlassian.com/software/jira`. accessed: 03.24.2024.

[11] GIMP. *GIMP*. `https://www.gimp.org/`. accessed: 04.16.2024.

[12] React. *React*. `https://reactjs.org/`. accessed: 04.18.2024.

[13] Angular. *Angular*. `https://angular.io/`. accessed: 04.18.2024.

[14] Svelte. *Svelte*. `https://svelte.dev/`. accessed: 03.24.2024.

[15] Flowbite Svelte. *Flowbite Svelte*. `https://flowbite-svelte.com/`. accessed: 03.24.2024.

[16] Tailwind CSS. *Tailwind CSS*. `https://tailwindcss.com/`. accessed: 03.24.2024.

[17] Axios. *Axios*. `https://axios-http.com/`. accessed: 03.24.2024.

[18] D3.js. *D3.js*. `https://d3js.org/`. accessed: 03.24.2024.

[19] Scikit-learn. *Scikit-learn*. `https://scikit-learn.org/stable/`. accessed: 04.12.2024.

[20] UCI Machine Learning Repository. *Iris Dataset*. `https://archive.ics.uci.edu/ml/datasets/iris`. accessed: 04.18.2024.

[21] Yann LeCun. *MNIST Database*. `http://yann.lecun.com/exdb/mnist/`. accessed: 04.18.2024.

[22] Bennett Wendorf. *NeuraViz Repository*. https://github.com/Bennett-Wendorf/NeuraViz/. accessed: 04.18.2024.

[23] Quart. *Quart*. https://pgjones.gitlab.io/quart/. accessed: 03.24.2024.

[24] Flask. *Flask*. https://flask.palletsprojects.com/. accessed: 04.18.2024.

[25] MongoDB. *MongoDB*. https://www.mongodb.com/. accessed: 03.24.2024.

[26] PyMongo. *PyMongo*. https://pymongo.readthedocs.io/en/stable/. accessed: 03.24.2024.

[27] PM2. *PM2*. https://pm2.keymetrics.io/. accessed: 03.24.2024.

[28] GitHub. *GitHub Actions*. https://github.com/features/actions. accessed: 04.18.2024.

[29] Microsoft. *Playwright*. https://playwright.dev/. accessed: 03.26.2024.

[30] Pytest. *Pytest*. https://pytest.org/. accessed: 04.04.2024.

[31] Let's Encrypt. *Let's Encrypt*. https://letsencrypt.org/. accessed: 04.11.2024.

[32] Hyuk Jin Kwon et al. "Inverse-Based Approach to Explaining and Visualizing Convolutional Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* 33.12 (2022). accessed: 04.18.2024, pp. 7318–7329. DOI: 10.1109/TNNLS.2021.3084757.

8. Appendices

```bash
#!/bin/bash

SCRIPT_DIR=$( cd -- "$( dirname -- "${BASH_SOURCE[0]}" )" &> /dev/null &&
    pwd )

# Get latest changes
echo "Pulling latest changes from git"
cd $SCRIPT_DIR
git pull

# Cleanup old build files
echo "Cleaning up old build files"
rm -rf $SCRIPT_DIR/frontend/public/build
rm -rf $SCRIPT_DIR/frontend/dist

# Ensure frontend dependencies all exist
echo "Installing frontend dependencies"
cd $SCRIPT_DIR/frontend
npm install

# Build frontend
echo "Building frontend"
npm run build

# Ensure backend dependencies all exist
echo "Installing backend dependencies"
cd $SCRIPT_DIR/backend
python3 -m pipenv install

# Create model uploads directory if it doesn't exist
mkdir -p $SCRIPT_DIR/backend/model_uploads

# Restart the application
echo "Restarting application"
pm2 restart NeuraViz
```

Listing 4. Production Build Script