A New Visual Notation For The Design Of Deep Neural Networks

Master's Thesis submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics

presented by
Adam Kasperski

under the supervision of
Prof. Paolo Tonella
co-supervised by
Prof. Michele Lanza

January 2026

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Adam Kasperski
Lugano, 19 January 2026

Abstract

For the past several years, Artificial Intelligence, particularly neural networks, has been one of the most popular topics across multiple domains. As those various models grow in complexity and size, the capability to clearly visualize their structure is vital for understanding and design. In addition, the increasing usage of Artificial Intelligence in real-world applications further necessitates the need for clear and standardized visual representation, giving experts and non-experts a view to reason about those models. Despite this widespread adoption, there is a notable absence of a standardized or proposed visual notation for neural networks. Researchers and practitioners typically depict new concepts or architectures in varied and inconsistent visual formats. Additionally, the current tools for designing neural networks more often than not do not meet the criteria of effective information visualization. This thesis aims to address these issues by reviewing current tools for neural network visualization and proposing new and improved notation for the design neural network architectures.

Contents

Figures

Tables

Chapter 1

Introduction

Neural networks, particularly deep neural networks, have experienced a surge in popularity over the past decade, becoming a go-to method of modern artificial intelligence and machine learning applications for different tasks [1]. Despite this rapid advancement and widespread adoption across various industries, there remains a lack of development and proposals for visual notations to depict their architectures, in contrast to fields such as software visualization, where researchers propose various techniques for visualizing architectures [2][3][4].

The visual representations of neural networks architecture remain largely ad-hoc and non-standardized. Researchers often rely on boxes and arrows diagrams created in general-purpose drawing tools, which vary significantly between publications. However, some common conventions do emerge. This lack of standardization can lead to semiotic ambiguity, where multiple concepts are represented by the same visual symbol. Furthermore, the continuously evolving nature of machine learning research introduces novel components and configurations, complicating the establishment of a standardized notation system.

While the lack of a unified standard persists, there have been notable attempts to address the fragmentation of neural network diagrams. These efforts generally fall into two categories: tools for publication aesthetics and tools for model explainability.

Tools such as Net2Vis [5] and NN-SVG [6] automate the generation of diagrams to streamline the creation of figures for academic papers. However, these tools are primarily generative. They parse existing code to produce a static image. Their focus is on the documentation of a finished model rather than providing a notation for the initial design of new architectures.

In parallel, a significant body of literature has explored the visualization of neural network components to explain their decision-making processes. Interactive tools such as CNN Explainer [7] and Transformer Explainer [8], alongside studies by Yosinski et al. [9] and Qin et al. [10], focus on visualizing internal computations, such as feature map activations and attention weights.

While these efforts are invaluable for explainable AI, they do not address the problem of Architectural Notation. They visualize the dynamics of data flow and learned features during inference, rather than the structural definition of the system's topology. Consequently, the field lacks a standardized visual language specifically engineered for the architectural design phase, leaving engineers to rely on ad-hoc diagrams that lack semantic integrity.

To bridge this gap, this thesis proposes a formalized visual notation based on principles defined in Physics of Notation [11] for the design of deep neural network architectures. Subse-

quently, we introduce the interactive web-based tool developed to instantiate the proposed notation. Unlike existing solutions that mainly focus on post-training visualization, the proposed artifact enables the initial construction of valid models, utilizing a visual vocabulary designed to support various topologies.

The effectiveness of the proposed notation and tool is then later assessed through a pilot study, evaluating the tool's impact on architectural comprehension and design usability.

1.1 Structure of the Thesis

This thesis is composed of the following chapters:

- Chapter 2 - Theoretical Foundations and State of the Art. This chapter explores the current state of neural network visualization, the limitations of existing tools, and the theoretical foundations of visual notation as well as principles guiding effective design of such notation.
- Chapter 3 - Solution. This chapter introduces the solution of the thesis, the visual notation. It also provides a theoretical evaluation of the notation based on established principles.
- Chapter 4 - Implementation. This chapter describes the technical architecture of the interactive web tool developed to instantiate the notation.
- Chapter 5 - Pilot Study. This chapter presents the design and results of a pilot study conducted to evaluate the usability and cognitive effectiveness of the DeepSketch tool and its underlying notation.
- Chapter 6 - Conclusion. This chapter concludes on the thesis findings and discusses future work and limitations.

Chapter 2

Theoretical Foundations and State of the Art

This chapter establishes the theoretical and practical foundations necessary for the development of a cognitively effective neural network notation and subsequent tool. It begins by defining the domain of deep learning, followed by a distinction drawn between structure visualization (the focus of this work) and behavior visualization.

Following the domain background, the chapter explores the theoretical principles of visual notation. It adopts Moody's Physics of Notations as the primary evaluative framework, supplemented by software visualization taxonomies by Gračanin and Maletic. These theoretical models are then utilized to conduct an analysis of the current state of the art, evaluating existing tools. By benchmarking these tools against cognitive principles, the chapter identifies significant gaps. The chapter concludes by defining a set of design requirements that the proposed solution must address.

2.1 Domain Background

2.1.1 Structure of Neural Networks

To understand the visualization of neural networks, their architecture has to be described first. Fundamentally, an Artificial Neural Network is a computational model inspired by the biological structure of the brain, designed to approximate complex functions that depend on a large number of inputs. As defined by LeCun et al. [1] in their review of deep learning, deep learning allows computational models composed of multiple processing layers to learn representations of data with multiple levels of abstraction.

The structure of these systems relies on three primary concepts:

- Layers and Neurons: The fundamental building blocks are neurons , which are organized into distinct layers. A standard architecture consists of an input layer, one or more hidden layers, and an output layer [12]. An example of a specialized layer is the Convolutional Layer, that uses a mathematical operation called convolution to extract features from input data. These are typically categorized by dimensionality:
  - Conv1D: Used for 1D signals or sequences.

  - Conv2D: Used for spatial data.
  - Conv3D: Used for volumetric data.

  Each neuron performs a weighted sum of its inputs followed by a non-linear activation function. Common examples include:

  - ReLU(Rectified Linear Unit): Defined as $f(x) = \max(0, x)$, it outputs the input directly if it is positive, otherwise, it outputs zero.
  - Sigmoid: Defined as $\sigma(x) = \frac{1}{1+e^{-x}}$, it squashes input values into a range between 0 and 1, often used to represent probabilities.

  By applying these non-linear functions, the network can learn decision boundaries which are conceptual lines or surfaces that separates different classes of data in a high-dimensional space.

- Tensors: While neurons provide the processing logic, tensors represent the data flowing through the system. A tensor is a generalization of scalars, vectors, and matrices to higher dimensions. This generalization is based on rank, which defines the number of indices required to access a specific element:

  - Rank 0(Scalar): A single value, $x \in \mathbb{R}$.
  - Rank 1(Vector): A 1D array of values, x_i.
  - Rank 2(Matrix): A 2D grid of values, $x_{i,j}$.
  - Rank n(n-Tensor): An n-dimensional array, $x_{i,j,k\ldots n}$.

- Computation Graphs: To execute the sequence of mathematical transformations(weighted sums, activation functions) neural networks are often represented as Directed Acyclic Graphs(with exceptions such as Recurrent Neural Networks). In this graph structure, nodes represent mathematical operations, and edges represent the tensors flowing between these operations [13] [12]. This graph perspective is vital for visualization, as it transforms the abstract mathematical formulas into a topological structure that can be mapped to a visual space, where relationships provide meaning.

2.1.2 Scope: Structure vs Behaviour

The one aspect of neural network visualization that can divide them into sub-domains is the scope of the visualization. The distinction is between the static definition of the model, including its network architecture and connectivity, and the dynamic execution of the model, covering its internal states, activations, and decision boundaries as it processes the data.

Structure Structure visualization is concerned about showing the topology of the neural network. We treat the model like a graph where the nodes are the computational parts(like individual neurons, layers, or operations), and the lines show how the data flows between them. The goal of structure visualization is to represent the model's design. This approach allows researchers to easily share their network architectures in papers or helps engineers verify that a complex model architecture is structurally valid.

However, Bäuerle et al. [5] argue that achieving effective architectural communication is often hindered by the time expenditure, lack of common visual grammar and ambiguity introduced through the reliance on manually drawn diagrams and addressed this problem by proposing a generative approach that abstracts the network architecture directly from code using a visual grammar, ensuring the resulting visualization remains faithful to the code implementation. This approach is also present in automated visualization tool Netron [14].

While these tools ensure technical accuracy, they possess limitations. Because they are retrospective, they require a valid code implementation to exist before a visualization can be generated. Furthermore, these tools reveal a semantic gap. By prioritizing low-level operational completeness, they often produce cluttered diagrams, especially for more complex architectures, where the intent of the model is obscured by representing every operation performed in the code.

Behaviour This category focuses on the dynamic states of the model during either training or inference. This scope encompasses visualizations of training metrics, such as loss curves, alongside methods that reveal internal processing, such as activation heatmaps [15], [16], [17], [18] and feature projections [19]. Furthermore, it includes interactive systems [20] which visualize neuron activations and the computation graph to provide insight into the model's internal decision-making. Specialized tools allow researchers to observe the live activations produced at each layer of a trained model as it processes input streams, such as image or video data [9]. The majority of scientific work is dedicated to this category [21],[22]. While behavior visualization helps explain what a model has learned, architecture visualization is required to explain how the model is constructed.

2.2 Theoretical Foundations

2.2.1 Visualization

Card et al. [23] define Information visualization as "the use of computer-supported, interactive, visual representations of abstract data in order to amplify cognition". In this context, abstract data refers to information that does not have a physical form, such as activation functions of a neural network. Because this data is not tied to a physical object, it requires a visual mapping to become interpretable. Cognition refers to the acquisition or use of knowledge, for instance, when evaluating or designing neural network architectures.

Claire Knight et al. [24] define software visualization as "a discipline that makes use of various forms of imagery to provide insight and understanding and to reduce complexity of the existing software system under consideration". Software visualization is a broad research field [25], with numerous publications tackling problems such as visualizing software changes throughout time [26] or even presenting its structure using the city metaphor in 3D [27].

Neural networks are close to software visualization, as they are primarily implemented through programming languages and software frameworks. Similar to how UML diagrams are used to design and visualize software systems before actual implementation, neural network architectures can be conceptualized and visualized prior to coding.

2.2.2 Definition of Visual Notation

Before analyzing neural network diagrams, we need a definition of visual notation. It's important to note that in software engineering, a notation is more than just sketches, it's a structured language. Moody [11] defines this in The Physics of Notations as having a set of symbols(visual vocabulary), a set of rules(grammar) that govern how they are combined and definitions of the meaning of each symbol(visual semantics).

2.2.3 Theoretical Foundations of Notation

To create a visual notation that serves as a practical engineering tool rather than just a pleasant picture, it is essential to base the design on the cognitive science of diagramming. The representation of complex systems, such as deep neural networks, relies on the cognitive effectiveness of the representation, defined as the speed, ease, and accuracy with which a representation can be processed by the human mind [28]. This section, which mostly draws from the Physics of Notations, lays out the theoretical framework that will be used to assess current tools and guide the creation of the suggested notation and its implementation.

To ensure the proposed visual language is cognitively effective, this work adopts the Physics of Notations(PoN) framework [11]. Recognized as a systematic approach for evaluating visual syntax, this theory provides nine evidence-based principles for designing cognitively effective notations. The following principles are described, and parallels are drawn with neural network architecture design.

- Semiotic Clarity: According to this principle, semantic constructs(the concepts) and visual symbols(the representation) must exactly correspond. Current AI diagrams often fail to meet this standard, suffering from either symbol deficit, when there are semantic constructs that are not represented by any graphical symbol, or symbol overload, where generic shapes like rectangles are used freely for layers, operations, and tensors. A standardized notation must enforce a strict correspondence, where specific icons map uniquely to specific concepts.

- Perceptual Discriminability: Based on their visual characteristics, symbols should be simple to differentiate from one another. According to Moody [11], the main visual factor used to differentiate semantic groups is shape. Notations that only use text labels inside identical boxes, which is a typical pattern in automatically generated graphical representations of DNN models, have poor discriminability and need unnecessarily long, serial scanning as opposed to quick, pre-attentive processing.

- Visual Expressiveness: This method emphasises using all of the visual variables(position, size, shape, colour, brightness, and texture) to encode information. In the context of DNNs, spatial dimensions(width/height) are automatically transferred to the size variable, allowing for semantic transparency in which a symbol's visual appearance reveals its meaning for instance, a larger box means a greater tensor). Similarly, the shape of the symbol or the iconic representation of a layer can be directly mapped to its function.

- Complexity Management: A notation must include tools for hierarchical abstraction and modularization in order to manage the scale of modern systems. The importance of this rule is demonstrated by ResNet-101 architecture. In a flat representation, every one of the 101 layers would be rendered as an individual node, resulting in a diagram that is

too large to navigate or understand. An efficient notation must allow these repeating sub-components to be collapsed into single modules. This abstraction enables reasoning about the network's whole structure without being overwhelmed by the repetition of lower-level operations.

- Semantic Transparency: This principle represents the degree to which a symbol's meaning may be inferred from its appearance alone. Symbols should be visually similar to the ideas they stand for. For instance, it is semantically transparent to depict an activation function(like ReLU) using a tiny graphic of its mathematical curve. Similarly, representing a dimensionality reduction layer(Pooling Layer) with a trapezoid that visually narrows captures the downsampling nature of the operation.

- Dual Coding: Humans have two separate cognitive channels, one for visual information like images and one for verbal information like text. When text and graphics are used together to reinforce the same meaning, working memory capacity is improved [29].

- Graphic Economy: There should be a manageable number of distinct graphical symbols. It is challenging to learn a notation with hundreds of distinct symbols. A tiny set of symbols should be the goal of the design. For instance, the notation can utilize a single basic Convolution shape and change it with additional visual variables(such as textual annotation) to distinguish subtypes instead of having a distinct shape for each variation of Convolution(Conv1D, Conv2D, Conv3D).

- Cognitive Fit: The notation should be adapted to the task and the audience. A notation designed for implementation(debugging code) requires high detail and strict adherence to the computational graph. A notation designed for architecture design(conceptualization) requires higher abstraction and emphasis on data flow topology. The latter is the focus of the suggested notation and tool, which favours topological understanding over implementation details.

- Cognitive Integration: This principle addresses the increased mental burden placed on users when a system is split across multiple diagrams, which requires them to mentally piece together fragmented information and track their location. An effective notation should mitigate this burden by providing visual cues to signal relationships or navigational aids to facilitate movement between distinct views.

2.2.4 The Cognitive Dimensions of Notations Framework

While the Physics of Notations serves as the primary theoretical foundation for this work, it is also vital to address the Cognitive Dimensions of Notations(CDs) [30] framework, which has historically been a dominant concept in visual language research. The CDs framework provides a vocabulary for describing the structure of cognitive artifacts and the tradeoffs involved in their use [31], [32]. However, for the objective of designing and evaluating the static visual syntax of a deep neural network notation, the Physics of Notations was selected over the CDs framework for reasons derived from Moody's comparative analysis [33]. First, the primary limitation of the CDs framework in this context is its domain generality. As Moody argues, the CDs framework was derived from research on human-computer interaction and programming research, applying to visual languages only as a particular special case. In their 2006 review,

[34], authors of CDs themselves, state that the original definitions of the dimensions were often vague, leading to inconsistent applications.

In contrast, the Physics of Notations was developed specifically for visual notations, basing its principles in the physical and perceptual properties of graphical representation. Since we are focused on defining exactly how DNN diagrams should look, the specific guidance from PoN is much more useful than the broad rules of CDs.

Second, the theories work in fundamentally different ways. CDs is a descriptive(Type I) theory, it gives a vocabulary to talk about design, but it doesn't necessarily determine whether one design is empirically superior to another. Dagit et al. [32] explicitly state that CDs "cannot validate usability; rather, they can only find usability problems." PoN, on the other hand, is prescriptive(Type V). It gives us actual rules to follow, like Semantic Transparency, that predict that representations satisfying these principles will be more cognitively effective. This prescriptive nature provides the necessary guidance for constructing a new notation from the ground up.

Finally, there needs to be a distinction between the notation and the tool applying this notation. When users are actually editing, navigating, refactoring, or making changes, CDs is relevant for analysis by providing a conceptual framework through its concepts like viscosity (the effort required to make changes to a created artifact) and premature commitment(the constraint of being forced to make a decision before the necessary information is available). So, as PoN was chosen to design the visual syntax, CDs framework remains valuable for analyzing the interactive qualities of the software tool itself.

2.2.5 Frameworks for Software Visualization

While Moody's Physics of Notations provides principles for the design of static graphical elements, the design of a comprehensive visualization system requires a broader architectural framework. To this end, this research draws upon the established frameworks of Gracanin et al. [25] and Maletic et al. [35], which define the essential dimensions required for a visualization tool to be effective and meaningful in a software engineering context.

Gracanin's framework identifies eight key areas that determine the efficacy of a software visualization system. Adapting these to the domain of neural networks provides the following design requirements:

- Scope of Representation: The visualization must isolate specific characteristics of the system. For Neural Networks, this implies separating the topology(architecture) from the training dynamics, as attempting to visualize both simultaneously leads to clutter.

- Medium of Representation: This refers to where the visualization is rendered. Given the collaborative nature of modern AI research, a web-based medium is prioritized to facilitate accessibility and sharing compared to desktop tools.

- Visual Metaphor: This is the use of symbols to represent abstract components. A consistent metaphor is required to map components of neural network architecture to visual objects.

- Consistency of the Metaphor: This part state that "multiple software artifacts cannot be mapped to the same metaphor" [25]. In this context, it necessitates that distinct neural network layers must not share identical visual representations, as this creates ambiguity.

- Semantic Richness: The metaphor must be "rich enough to provide mappings for all aspects of the software." This supports the need for enough expressiveness, where visual elements like color and size are used to encode layer properties.
- Abstractedness: Deep learning requires the capacity to "focus away from certain parts... and focus in detail on others". By enabling users to treat a group of operations as a single unit or expand it into its individual components.
- Ease of Navigation: Users must be able to effortlessly move between views. This implies standard interaction patterns(pan, zoom, clicking to expand view).
- Level of Automation: Automation specifies the degree to which the visualization is constructed without user intervention. While generative tools([14]) are fully automated, a design tool requires a balance where the layout is automated, but the topology is user-defined.

Looking at the work of Moody [11] and Gračanin [25], both strongly emphasize reducing confusion by using strict rules for how concepts are visually represented. Moody's idea of Semiotic Clarity and Gračanin's concept of Consistency of Metaphor both boil down to the same requirement, that there must be a one-to-one correspondence between a software concept and its graphical representation.

Furthermore, Moody's principle of Visual Expressiveness, which encourages using different visual elements like color or texture to convey data, directly supports Gračanin's need for Semantic Richness, the ability to pack dense data qualities into a visualization.

Finally, both frameworks agree that scale is also a challenge. Moody's Complexity Management and Gračanin's Abstractedness both argue that users need the ability to hide low-level details so they can focus on and understand the overall high-level system design.

Complementing the structural requirements, Maletic et al. [35] define five dimensions of software visualization. While their dimensions of Representation, Medium, and Granularity overlap with the frameworks provided by Gračanin, the dimensions of Task and Audience offer unique and necessary perspectives for this research:

Task(Why is the visualization needed?): Machine learning researchers frequently rely on general-purpose diagramming tools to construct and illustrate network architectures. This is often prone to errors. Furthermore, it often results in semantically inconsistent diagrams where the same operation is represented by different shapes. This occurs because general-purpose tools are not guided by visualization expertise, not enforcing visualization principles

Audience(Who will use the visualization?): The target audience ranges from novice AI students, who benefit from high levels of abstraction and semantic transparency, to expert engineers, who require more detailed view of the model, like parameter specifications.

2.3 Analysis of Existing Tools

2.3.1 The Standardization Gap

Recent research [36] [37] identifies a disconnect between the definitions of visual notation and current practices in Machine Learning scholars. Despite the presence of diagrams in the field, appearing in 82% of ACL papers and serving as the preferred method for summarizing architectures, the landscape is defined by high heterogeneity. Marshall et al. [36] observe

that "no two diagrams in major conferences have the same visual representation," a lack of consistency that renders the comparison of different architectures "cognitively challenging."

This fragmentation stands in contrast to other technical disciplines. Unlike software engineering, which utilizes the Unified Modeling Language, the machine learning community currently lacks a standardized visual language. Consequently, neural network diagrams rarely satisfy the criteria of a formal notation. Marshall et al. [37] find that without a shared grammar, authors frequently utilize graphical attributes, such as color coding or arrow styles, for aesthetic purposes rather than functional encoding. This practice leads to failures in "precision meaningfulness", creating ambiguity, reducing reproducibility, clarity, comparability and accessibility. Ultimately, this lack of standardization results in a "high risk of miscommunication". Instead of serving as clear summary that support reproducibility, these diagrams often act more like ambiguous drawings, requiring readers to rely heavily on the text to understand the system's architecture.

2.3.2 Comparison

Identifying the specific design requirements for the proposed notation necessitates a critical evaluation of existing neural network visualization tools and their notations. The current landscape is largely segregated, with tools prioritizing either direct implementation fidelity or high-level publication aesthetics, meaning few effectively provide the necessary cognitive support for architectural design. This section therefore analyzes representative tools from these categories, introduced in Table 2.1, against the functional requirements and principles presented in Section 2.2.3. By analyzing the typical tasks of a neural network design we identified the core behaviors the tool should support. Functional requirements address the technical aspects of a tool, design principles address its cognitive effectiveness.

A simple Convolutional Neural Network(CNN) model was created to analyze each of the following tools.

Model

The architecture consists of a 32-filter convolutional layer using a 3×3 kernel and ReLU activation, followed by a 2×2 max-pooling layer, then adds a second convolutional layer with 64 filters and ReLU activation followed by another 2×2 max-pooling layer, proceeds to flatten the feature maps, includes a fully connected dense layer with 128 neurons and ReLU activation, and concludes with a 10-neuron softmax output layer for multi-class classification.

2.3.3 Comparison Metrics

The following features were identified to evaluate and compare different visualization tools (Table 2.1) based primarily on their functional attributes. While functional attributes assess the tool's capacity to process and display neural network architectures, Moody's Principles evaluate the cognitive effectiveness of how that information is visualized. The results are presented in Table 2.2 and Table 2.3.

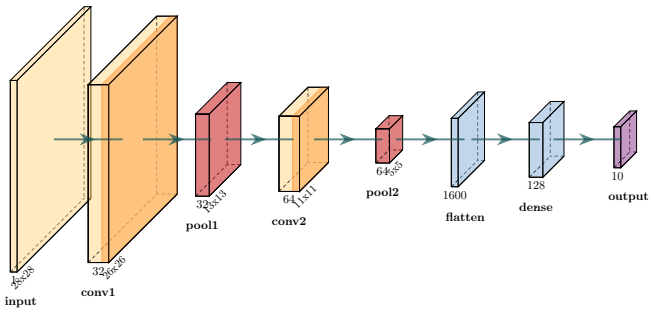
- *Model Format Support* enables reading and visualizing models from various formats, as well as from frameworks including TensorFlow, Keras, and PyTorch.
- *Zooming* allows users to magnify specific parts of the neural network.

Tool	Description	References
Netron	A tool for visualizing deep learning models supporting ONNX, TensorFlow, PyTorch and other. It provides an interactive interface for exploring architectures, layers, and parameters.(See Figure 2.2)	[14]
VisualKeras	A Python library for generating diagrams of Keras models, allowing customizations in layer coloring, spacing, and legends. (See Figure 2.4)	[38]
NN-SVG	Generates SVG diagrams for fully connected and convolutional networks, inspired by AlexNet-style architectures. (See Figure 2.7)	[6]
CNN Analyzer	A web-based visualization tool supporting Caffe framework configurations.(See Figure 2.2b)	[39]
PlotNeuralNet	A Python library that generates LaTeX/TikZ-based diagrams for visualizations of neural networks.(See Figure 2.1a)	[40]
TensorBoard	A TensorFlow toolkit for visualizing model graphs, metrics, and data distributions. (See Figure 2.6a)	[41]
plot_model	A Keras utility function to generate diagrams showing layer connections, names, and shapes. (See Figure 2.6)	[42]
ENNUI	A browser-based tool for building and visualizing neural network architectures with drag-and-drop functionality.2.1b	[43]
Moniel	A declarative notation system that uses dataflow graphs to define neural network architectures. (See Figure 2.6b)	[44]
nnviz	A Python package for visualizing PyTorch neural networks using Graphviz. (See Figure 2.5a)	[45]

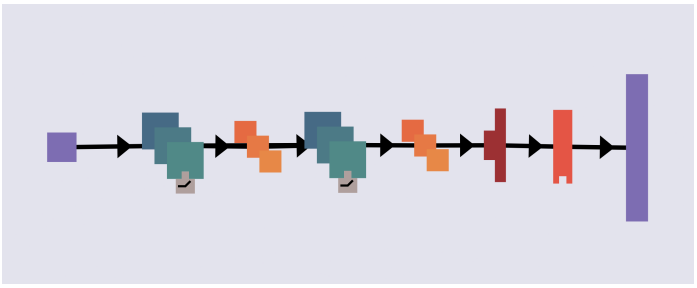
Table 2.1. Neural Network Visualization Tools

- *Panning* enables users to navigate horizontally or vertically across the visualization.
- *Input/Output Dimensions* specifies the shape and size of the data entering and leaving a layer.
- *Layer Specific Parameters* displays unique attributes for each layer.
- *Visualization Flow Direction* indicates the direction of the data flow in the network.
- *Predefined Layers* consists of standard, pre-implemented layers provided by the tool.
- *Support for Custom Layers* flexibility to visualize user-defined layer types.
- *Support for Various Network Architectures* refers to compatibility with different neural network architectures, such as Convolutional, Recurrent, and Feedforward neural networks.
- *Layer Type Distinction* uniquely identifies various types of layers in a model.

- *Input/Output Size Distinction* emphasizes differences in dimensions as data transitions between layers.
- *Modularization* organizes related layers into cohesive groups.
- *Script-based vs GUI-based* differentiates whether the tool is operated through a script or through a graphical user interface(GUI).
- *Customizability* evaluates the extent to which users can modify visual aspects of the network diagram, such as changing colors, labels, or layout settings.
- *Graphical Export* indicates the tool's capability to export the generated visualizations in various formats.
- *Accessibility(Desktop/Web)* describes whether the tool is available as a desktop application, a web-based interface, or both.



(a) PlotNeuralNet.



(b) ENNUI.

Figure 2.1. Comparison of model architectures visualized using different tools:(a) PlotNeuralNet, (b) ENNUI

2.3.4 Comparison of Features

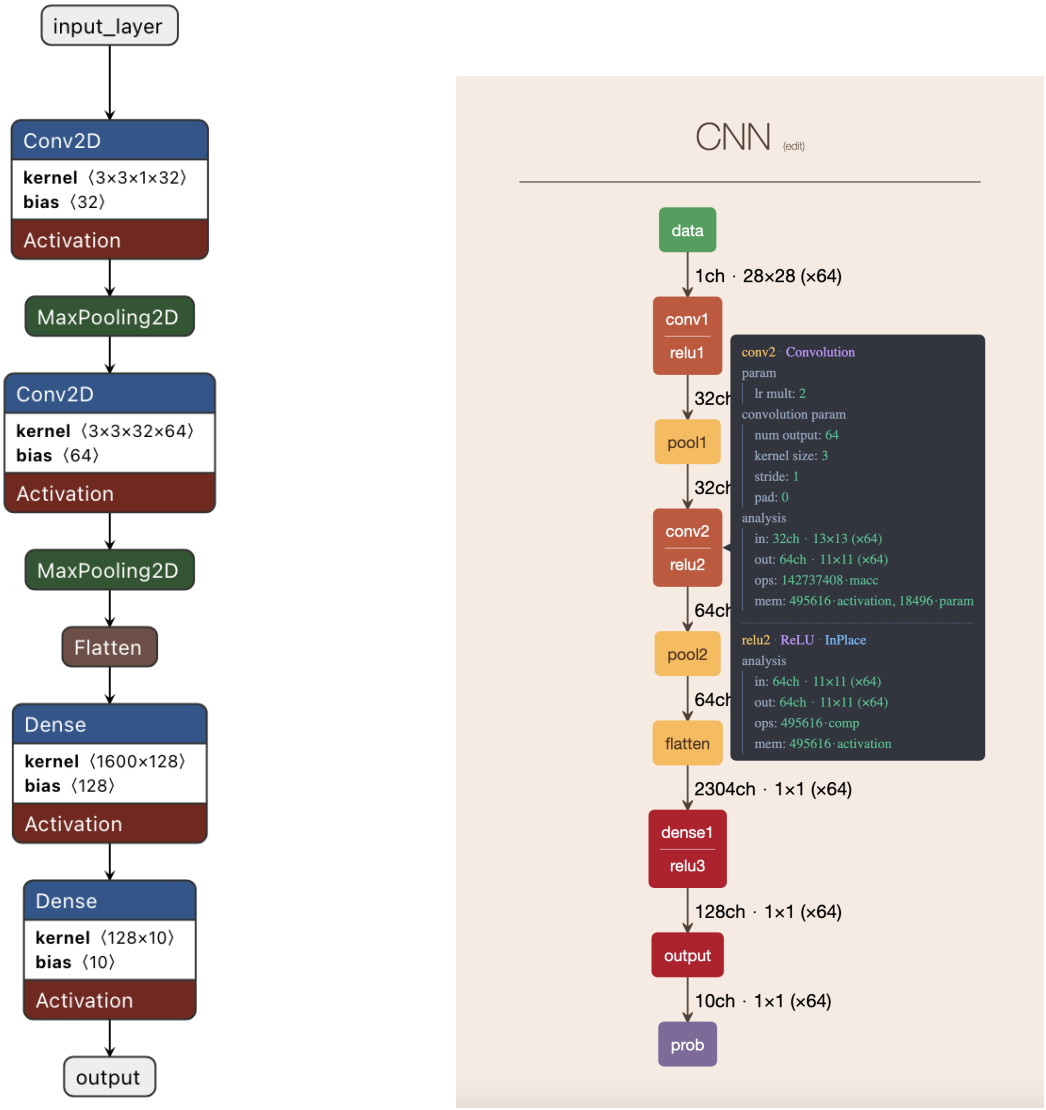
S - Supported, **P** - Partially Supported, **N** - Not Supported

Feature	**Netron**	**VisualKeras**	**NN-SVG**	**Netscope**	**PNN**
Model Format Support	S	P	N	P	N
Zooming	S	N	N	N	N
Panning	S	N	S	N	N
Input/Output Dimensions	N	S	S	S	S
Layer Specific Parameters	S	N	P	S	N
Vis Flow Direction	Up-Down	Left-Right	Left-Right	Up-Down	Left-Right
Predefined Layers	S	S	N	S	N
Support for Custom Layers	N	N	N	N	S
Architecture Support	S	S	P	P	S
Layer Type Distinction	S	S	P	S	S
Visual I/O Size Distinction	N	S	S	N	S
Modularization	N	N	N	N	N
Script-based vs GUI-based	GUI	Script	Script	GUI	Script
Customizability	N	P	P	N	S
Graphical Export Options	S	S	S	N	S
Accessibility	Both	Desktop	Web	Web	Both

Table 2.2. Comparison Features for Neural Network Visualization Tools(Part 1)

Feature	**TensorBoard**	**plot_model**	**ENNUI**	**Moniel**	**nnviz**
Model Format Support	S	S	N	N	S
Zooming	S	N	N	N	N
Panning	S	N	N	N	N
Input/Output Dimensions	N	S	S	N	N
Layer Specific Parameters	N	N	S	P	N
Vis Flow Direction	Down-Up	Up-Down	Left-Right	Down-Up	S
Predefined Layers	N	P	S	N	S
Support for Custom Layers	N	N	N	S	N
Architecture Support	S	S	P	S	S
Layer Type Distinction	N	N	S	S	S
I/O Size Distinction	N	N	N	N	N
Modularization	N	N	N	S	N
Script-based vs GUI-based	GUI	Script	GUI	Script	Script
Customizability	N	P	N	N	S
Graphical Export Options	N	S	S	N	S
Accessibility	Web	Desktop	Web	Desktop	Desktop

Table 2.3. Comparison Features for Neural Network Visualization Tools(Part 2)



(a) Netron. (b) Netscope CNN Analyzer.

Figure 2.2. Comparison of model architectures visualized using different tools: (a) Netron, (b) Netscope CNN Analyzer.

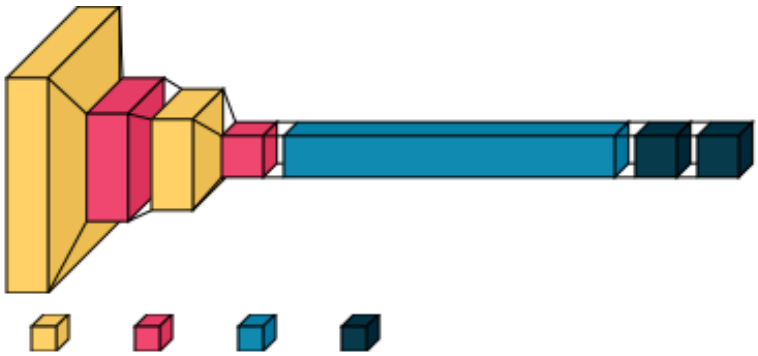


Figure 2.3. VisualKeras

Figure 2.4. Model architectures visualized using VisualKeras

2.3.5 Discussion

All the tools present neural network architecture information using their own visual conventions, although common patterns appear.

The visualization is performed by means of directed, connected graph, where nodes represent layers and edges the directed connections between them, indicating the data flow. This method is a natural way of representing a neural network, as it mirrors the directed flow of data within it.

As shown in Table 2.2 and Table 2.3, Model Format Support varies. While tools like Netron(see Figure 2.2a) and NNviz(see Figure 2.5a) support a wide range of existing frameworks(TensorFlow, Keras, PyTorch), others like NN-SVG or PNN(see Figure 2.1a) require manual definition. Netscope CNN Analyzer(see Figure 2.2b) partially supports Model Format Support by specifically accomodating Caffe's prototxt format.

The ease of interaction varies among the tools. Netron and TensorBoard(see Figure 2.6a) support both Zooming and Panning, which are important for navigating large models.

Tools that prioritize high-fidelity to the code implementation, such as Netron and CNN Analyzer, support the display of layer-specific parameters. In contrast, other tools focus exclusively on the visualization of topology, omitting low-level layer details. The Visualization Flow Direction is split between Up-Down flows and Left-Right. The only exception is TensorBoard, which is directed upwards.

The availability and implementation of Predefined Layers vary across the analyzed tools. Generative tools, such as Netron and VisualKeras, dynamically reflect the layers library of the underlying framework, any layer defined in the code is automatically rendered in the visualization. While these tools offer superior ease of use for existing models, they remain dependent on a completed code.

In contrast, tools designed for manual architectural construction offer varying degrees of predefined components. Tools like NN-SVG provide a restricted set of distinct layer types, whereas Moniel and PNN do not include predefined layers at all. Instead, these tools require the user to manually define every component, offering flexibility at the cost of higher manual effort. Consequently, those two tools are the only ones that support Custom Layers, as users have to define everything themselves. ENNUI allows users to select from a standardized set

(a) NNviz visualization. (b) plot_model visualization.

Figure 2.5. Comparison of model architectures visualized using different tools. (a) NNViz, (b) plot_model

of predefined layers to design a network directly on an canvas, supporting the design process without requiring pre-existing code.

In the similar fashion, the tools that are based on frameworks do support various network architectures. ENNUI and NN-SVG are limited by the layer types thay incorporate. As earlier, PlotNeuralNet and Moniel support whatever the user wishes to represent.

Regarding the layer type distinction, most of the evaluated tools distinguish different components primarily through the use of labels. Additionally, variations in color and node size are present to represent different layers within the network(Netron, NNviz). However, when we exclude color and labels, only ENNUI(see Figure 2.1b) is capable of discerning different components based solely on the shape of the nodes.

Only three of the selected tools(PlotNeuralNet, VisualKeras and NN-SVG) offer input-output visualizations that illustrate the scale of data changes across subsequent stages. Although the high fidelity of the last tool obscures the visualization with the number of elements presented.

(a) tensor_board (b) moniel

Figure 2.6. Model architecture visualized using Moniel and TensorBoard.

Another important aspect is the tool's ability to modularize layers by grouping. This modularization reduces visualization clutter by consolidating repetitive components, avoiding the redundant display of identical elements and allowing for viewing the models at different levels of abstraction. Only Moniel supports th Tools that provide a graphical user interface are generally more convenient to work with. Writing scripts to define layers and connections can be tedious, whereas an interactive graphical interface streamlines this process, making it significantly easier. Furthermore, automatically generated visualizations are even more convenient, as we just need to import the model, and the tool produces the visual representation without any manual intervention.

As for customization capabilities, the tools differ in that regard. Certain tools, such as ENNUI, incorporate predefined layers and others like VisualKeras or Netron integrate with existing frameworks, which limits the extent of customization available to users due to predefined structures and layer configurations. In contrast, tools like PlotNeuralNet and Moniel offer complete customization. Most of the tools have some option for exporting created visualization and as for accessibility, tools like Netron and PNN offer high accessibility by providing both desktop and web versions, whereas others are restricted to a single environment.

2.3.6 Principles Analysis

While the feature comparison highlights the functional capabilities of existing tools, a deeper analysis using Moody's Physics of Notations reveals semiotic flaws that impede their use for architectural design. This subsection assesses the existing tools according to Moody's Physics of Notations principles, providing an assessment of the underlying representational deficiencies. All but Cognitive Integration principles were evaluated, as all of the tools employ single-view

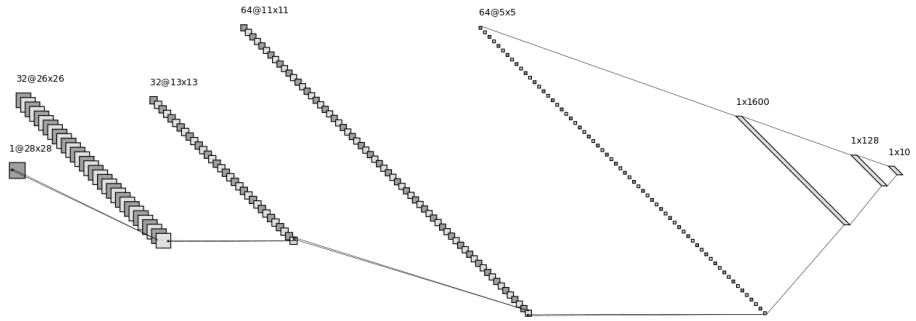


Figure 2.7. Model architecture visualized using NN-SVG tool.

visualization strategy, consolidating all aspects of the architecture into one view.

To evaluate the representational shortcoming in current tools, quantitative assessment was applied using Likert scale ranging from -2 to +2. An emphasis is placed on the distinction between neutral(0) and negative(-1/-2) scores. A neutral(0) score indicates a basic implementation that lacks explicit cognitive optimization, such as relying on text for layer distinction or displaying a standard linear topology without modular abstraction. In contrast, negative scores highlight instances where the visualization hinders the architecture undestanding, for example through severe symbol overload or visual clutter that exceeds working memory limits. Positive scores(+1/+2) are used for features that reduce cognitive effort, such as hierarchical modularization or distinct visual symbols.

Principle	**Netron**	**VisualKeras**	**NN-SVG**	**Netscope**	**PNN**
Semiotic Clarity	0	1	0	0	1
Perceptual Discriminability	0	1	1	-1	1
Semantic Transparency	-1	-1	1	-1	1
Complexity Management	0	0	0	0	0
Cognitive Integration	N/A	N/A	N/A	N/A	N/A
Visual Expressiveness	0	1	0	0	1
Dual Coding	2	2	-2	2	2
Graphic Economy	0	1	-1	0	1
Cognitive Fit	0	1	0	0	1

Table 2.4. Moody's Physics of Notation Assessment(Part 1)

- Semiotic Clarity: Most tools suffer from Symbol Overload. In these tools, a generic rectangle is often used to represent fundamentally different constructs. This forces the user to rely mostly on text labels to distinguish between different layer types as well as color in some of the tools. Conversely, tools like ENNUI show better clarity by assigning distinct visuals to specific layer types, reducing ambiguity. TensorBoard(see Figure 2.6a) gets -2, as it uses only textual labels attached to circular nodes to represent layers.
- Perceptual Discriminability: The analysis shows that most tools like plot_model or Netron distinguish layer types primarily through text labels. According to Moody, text is a secondary notation. In a visual language, the primary notation(shape, color) should carry

Principle	TensorBoard	plot_model	ENNUI	Moniel	nnviz
Semiotic Clarity	-1	0	2	0	0
Perceptual Discriminability	-1	-1	2	0	0
Semantic Transparency	-2	-1	1	-1	-1
Complexity Management	0	0	0	1	0
Cognitive Integration	N/A	N/A	N/A	N/A	N/A
Visual Expressiveness	-1	-1	1	0	0
Dual Coding	2	2	-2	2	2
Graphic Economy	0	0	1	0	0
Cognitive Fit	-1	0	1	0	0

Table 2.5. Moody's Physics of Notation Assessment(Part 2)

the semantic weight. ENNUI and VisualKeras perform better here by using color coding and different shapes, but Netron suffers from Symbol Overload, where identical rectangular glyphs represent different operations, forcing the user to read text to distinguish a Conv2D from a Dense layer. However, it does also employ color, which results in neutral score.

- Semantic Transparency: There is a major lack of this principle in analyzed tools. In Netron, a Conv2D node(a rectangle) bears no visual relationship to the concept of a "volumetric filter sliding over an input." ENNUI, as well as NN-SVG, differentiate NN concepts through different shapes.

- Complexity Management: Tools like Netron and CNN Analyzer excel in fidelity. They represent the computational order of the architecture as it executes. However, they fail the principle of Complexity Management. By displaying every operation as an individual node(see Figure 2.2b), they exceed the limits of human working memory. As observed in the comparison, these tools lack Modularization capabilities. They require the user to integrate mentally on their own higher-level architectural concepts from the low-level network components. As a matter of fact, only one of the tools support modularization as a technique for satisfying Complexity Mangamenet principle, namely Moniel(see Figure 2.8), though it lacks the mechanism to hide complexity.

- Visual Expressiveness: As most of the analyzed tools represent layers by labeled rectangle, the visual expressiveness is limited. However some tools like PlotNeuralNet and NN-SVG use Spatial Encoding, mapping the visualization dimensions of the layers ($H \times W$) to the screen dimensions of the glyph. This adds a higher amount of information. In contrast, Netron(see Figure 2.2a) and Moniel(see Figure 2.6b) use fixed-size nodes regardless of the layer's actual size, not utilizing the visual expressiveness enough and forcing the user to read textual information to understand the scale of the model. They do however employ color to distinguish between layers.

- Dual Coding: All tools, except ENNUI, use text labels to identify and describe the visual objects they represent.

- Graphic Economy: All of tools analyzed maintain graphic economy. However it is not achieved by representing different layers with different symbols(except ENNUI). Existing

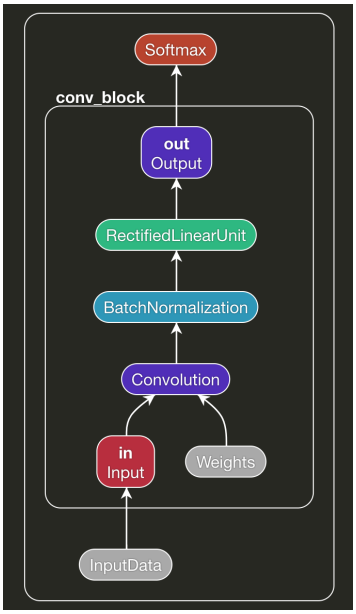


Figure 2.8. Moniel Complexity Management example.

tools achieve Graphic Economy artificially by sacrificing Semiotic Clarity. They use a single symbol(rectangle) for everything, which is economical but semantically empty. NN-SVG is an example of notation that exceed graphic economy, by visually cluttering the visualization(see Figure 2.7). Though it shows the scale of the model, it comes short at clearly representing the architecture. The notation needs enough symbols to be expressive but few enough to be memorizable.

- Cognitive Fit: The analyzed tools do align with the mental model of users who are familiar with neural networks. Netron and CNN Analyzer provide exhaustive parameter descriptions, whereas PlotNeuralNet focus more on topology.

2.4 Requirements

To ensure the cognitive effectiveness of the proposed tool, we derived following features to be implemented in our notation and tool to address specific principles from Moody's framework and gaps found in analysed notations and tools.

- Custom Layer Implementation: This feature allows users to define new components by specifying their parameters and visual representation. This feature addreses the principle of Semiotic Clarity by mitigating Symbol Deficit. By allowing users to create new layers as the field evolves, the notation should maintain a one to one correspondence between concepts and symbols. Furthermore, it should support Complexity Management by allowing users to encapsulate blocks of layers into single units, and should enhance Semantic Transparency by enabling users to assign custom icons to new constructs.

- Predefined Architecture Layers: The tool provides a set of predefined layers with its own

unique graphical symbol. This feature is important for maintaining Semiotic Clarity. By providing a standard set of symbols where every layer type has a unique representation, the notation should avoid Symbol Overload. Additionally, distinct visual shapes for different layer types enforce Perceptual Discriminability, ensuring users can distinguish elements with ease based on the iconic representation alone.

- Layer Modularization: The modularization mechanism enables users to group related layers into a single, high-level container that can be toggled between an expanded view of its internal components and a collapsed abstract representation. This mechanism primarily targets Complexity Management principle. By collapsing sub-networks into single abstract unit, the tool prevents information overload, keeping the visual complexity within human working memory. It further supports Cognitive Integration and Dual Coding by replacing complex mathematical operations with meaningful semantic labels.
- Abstraction Control: While modularization creates the structure, abstraction control facilitates the interaction. This satisfies Complexity Management through hiding of information, allowing users to focus away from lower-level implementation. The dynamic adaptability of the diagram promotes Cognitive Fit, allowing it to represent the structure at a higher-level and more detailed lower-lower. Finally, it aids Cognitive Integration by helping users maintain context as they navigate between abstraction levels.

Chapter 3

Solution

3.1 Solution Design

3.1.1 Layers

Different layers can be represented by the operation they perform, by their unique parameters or by their purpose. Examined tools lacked this distinction(except ENNUI). Take for instance Convolutional Layers. They could be described by the scanning operation, sliding a kernel across matrix data. Similarly, residual layers can be described by their skip connections, where each layer feeds into the next layer and directly into layers further away.

Convolutional Layer

The design for the Convolutional Layer(see Figure 3.1a) emphasizes the concept of local connectivity and feature extraction. Unlike a standard matrix icon, this design features a nested square to visually mimic the "sliding window" or kernel operation scanning a feature map. Additionally, dynamic kernel size is added. As user changes kernel size parameter of the layer, the inner square's size reflects that change, with the addition of text label on the sides of the square, indicating the provided parameter value.

Dense Layer

For the Dense layer(see Figure 3.1c), the notation retains the traditional node-link diagram. This choice was made to leverage existing user mental models, the intersecting lines communicate the all-to-all connectivity.

Flattening Layer

The Flattening Layer(see Figure 3.1d) uses dashed projection lines to show a 2D matrix transforming into a 1D vertical vector. This visual design illustrates the structural change, showing how spatial dimensions(height and width) are collapsed into a single linear feature vector.

Attention Layer

The Attention Layer(see Figure 3.1b) takes a different approach. Instead of trying to look like the layer's abstract operation, it uses a simple symbol to represent its meaning. Since the math behind an Attention layer is too complex to visualize as a simple shape, the design uses a familiar symbol instead; an exclamation mark in a circle. Because this symbol typically represents importance, it serves as an intuitive reminder of the layer's actual function, which is to focus on the most relevant parts of the input data.

Pooling Layer

The Pooling Layer(see Figure 3.2) is represented by a grid projecting into a smaller unit via converging dashed lines. This design visualizes the operation of dimensionality reduction. This icon uses the visual metaphor of compression to signal that the input feature map is being summarized into a lower-resolution output. By showing a 2x2 region reducing to a single block, the design communicates the aggregation of local information(such as Max or Average pooling) into a compact representation. This is the only layer representation that is not contained in a square shape, as the boundaries of it would limit the visibility of the symbol.

Embedding Layer

The Embedding Layer(see Figure 3.3a) is visualized using the metaphor of a projector. The icon shows a single source at the top projecting information down onto a row of blocks. This visual icon explains the concept of taking a discrete input and expanding it into a vector of features. It moves away from abstract math and instead shows the action of mapping one item to a distributed representation.

Normalization Layer

The Normalization Layer(see Figure 3.3b) uses the universally recognized "bell curve". The choice to represent the Normalization Layer with a Gaussian bell curve is a semiotic decision, not a literal mathematical one. Technically, normalization layers perform linear transformations that preserve the probability density function of the input. They do not transform non-Gaussian data into a Gaussian distribution.

However, representing this with a "bell curve" is effective because the curve is the universal symbol for statistics and standardization. While there is a risk that this icon could suggest a forcing of Gaussian distribution, the trade-off favors the immediate recognition of the layer's intent; to stabilize the learning process by centering features.

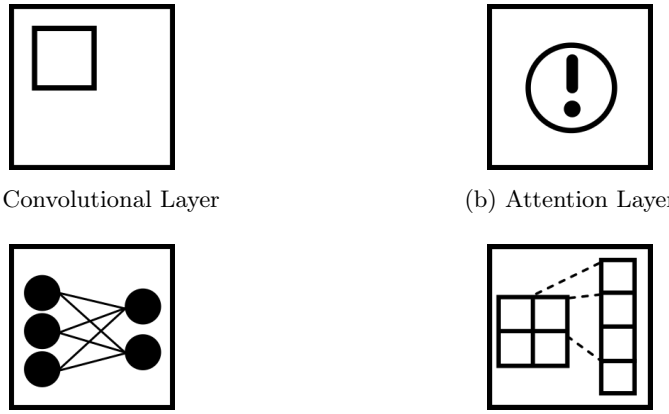
Recurrent Layer

The Recurrent Layer(see Figure 3.3c) depicts a series of stepping blocks connected by curved arrow.

Dropout Layer

The Dropout Layer(see Figure 3.3d) introduces a destructive element, a red X, universal signifier of negation, crossing out a node and erasing connections coming from previous set of nodes.

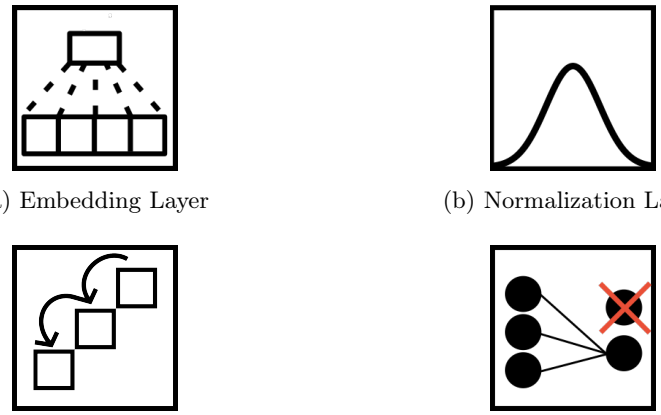
Dropout is a regularization technique that works by randomly turning off neurons. The design makes this literal. By showing a standard connection being forcibly canceled, the icon serves as an immediate visual cue for deactivation.



(a) Convolutional Layer (b) Attention Layer

(c) Dense Layer (d) Flattening Layer

Figure 3.1. Layer Notation



(a) Embedding Layer (b) Normalization Layer

(c) Recurrent Layer (d) Dropout Layer

Figure 3.3. Layer Notation

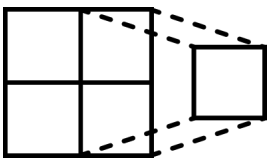
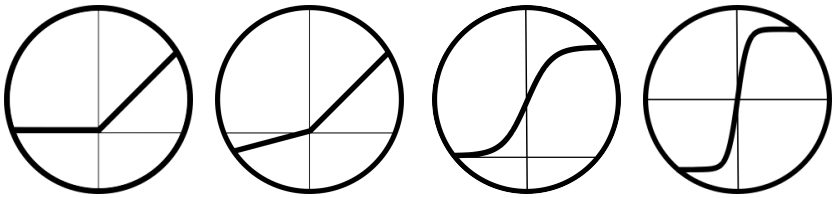


Figure 3.2. Pooling Layer

3.1.2 Activation Functions

Having drawn inspiration from D'Onofrio's work [46], we evaluated that the simplicity of representing activation functions through their graph-based depiction makes this approach suitable for inclusion in the proposed neural network visualization tool. Users familiar with common activation functions should be able to recognize them intuitively based on their graphical form, for instance, the sigmoid function can be identified by its characteristic "S"-shaped curve. However, this method assumes prior familiarity. Users without such background knowledge may find these visualizations challenging to interpret. To address potential ambiguity between similar-looking activation functions, axes were added to the graphs to provide additional visual differentiation as well as textual labeling. The proposed notation is included in Figure 3.4. Regarding the structural connection, the activation function is not displayed as a separate node. Instead, the function's glyph is "glued" to the layer it modifies. The graph is encapsulated within a circular badge and attached to the top-right corner of the primary layer(Figure 3.5).



(a) ReLU (b) Leaky ReLU (c) Softmax (d) Tanh

Figure 3.4. Proposed notation for activation functions.

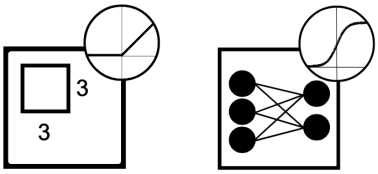


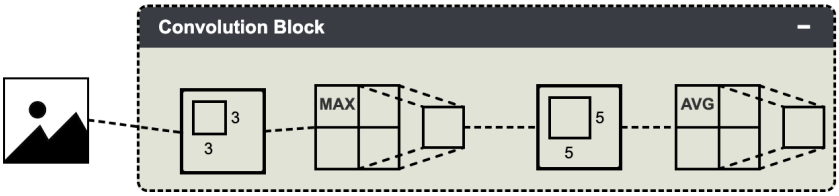
Figure 3.5. Activation functions attached to Convolutional and Dense Layers.

3.1.3 Modularization and Abstraction

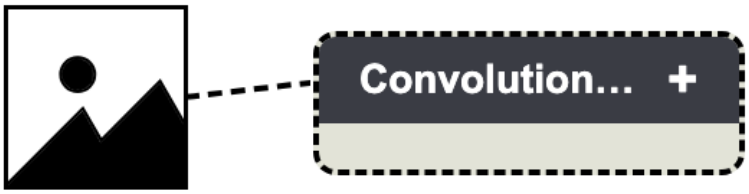
As deep neural networks increase in depth, visualizing every individual operation simultaneously leads to visual clutter and high cognitive load. To address this, the proposed tool implements a modular abstraction mechanism, allowing users to group sequential operations into higher-level units.

Visual Encapsulation, as illustrated in Figure 3.6, a sequence of layers(such as a Convolution, Pooling, and subsequent Convolution) can be encapsulated into a single block. This grouping is visually limited by a container with a dashed border. The choice of a dashed line rather

than a solid box suggests a logical grouping, indicating that the internal flow is part of the larger network context. Each module is assigned a distinct header bar(shown in dark grey) containing the module's name, such as "Convolution Block". The header includes interaction controls(represented by the minimize symbol). This enables progressive disclosure, users can toggle the visibility of the module's internal contents. When expanded, the internal layers are visible. When collapsed, the entire block is abstracted into a single node. This functionality allows the user to oscillate between a high-level and a low-level architectural view, effectively managing the complexity of large-scale model design.



(a) Modularization Expanded



(b) Modularization Toggled

Figure 3.6. Modularization.

3.1.4 Connections and Data Flow

As illustrated in Figure 3.7, connections between layers are visually represented as dashed lines terminating in directional arrowheads. This design was chosen to indicate the flow of data. A dashed line is used to differentiate between the dynamic flow of data and the solid boundaries of the layers, reducing cognitive load in complex graphs.

Neural networks are inherently directed graphs. To enforce this logic visually, every connection terminates with an arrowhead pointing toward the target node. This explicitly denotes the input-output relationship, removing ambiguity regarding the direction of forward propagation.

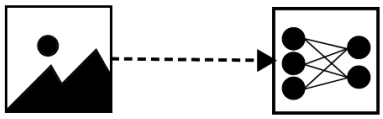


Figure 3.7. Connection between layers

3.2 Evaluation

The same model used to analyze existing tools was chosen to represent the architecture visualization made using the tool we developed to incorporate our notation named DeepSketch [1] (see Figure 3.8).

The visualization demonstrates Semiotic Clarity, establishing a one-to-one mapping between architectural concepts and visual symbols. Unlike frameworks that rely on generic geometrical shapes, DeepSketch employs a unique graphical vocabulary where every operation is assigned a unique geometric symbol. This visual distinctiveness ensures Perceptual Discriminability. The viewer can pre-attentively distinguish a convolutional layer(represented by nested squares) from a fully connected layer(represented by a bipartite graph) solely through shape, eliminating the need to decode textual labels or color to identify layer types.

The symbols are not arbitrary. They correspond to the mathematical operations they represent or to universal symbols. The Flatten layer is visualized as a 2D grid transforming into a 1D vertical vector, providing a visual cue for the reshape operation. Similarly, the Pooling layer uses a spatial contraction metaphor to communicate downsampling. By using these visual metaphors, the notation makes the underlying logic Semantically Transparent to the viewer, significantly reducing the cognitive effort required to learn the visual language compared to notations that use abstract boxes.

The notation utilizes Dual Coding by employing symbols that integrate text and graphics into a single unit. Key hyperparameters, such as the kernel size or the pooling type, are embedded directly within the relevant graphical nodes. This prevents the user to split attention between parameters panel and the visual representation. Additionally, as presented in Figure 3.9, every layer is embedded with textual information about its type. Furthermore, the visualization maintains Graphic Economy. It uses a concise set of distinct symbols, remaining within the limits of human memory while avoiding the artificial Graphical Economy of using a single shape for all functions.

Visual Expressiveness is in some degree limited by our notation's static sizing. Although the notation uses texture and shape, it does not utilize the size of the nodes to represent the actual dimensions of the data tensors. Additionally, the notation is mono-chromatic, with exception of Dropout Layer, not utilizing color to greater degree, which misses an opportunity to help users distinguish between architectural concepts in their minds more effectively.

The notation demonstrates Complexity Management through a modularization mechanism. As illustrated in Figure 3.6, the tool allows users to group multiple operations into a single,

[1] https://github.com/eliczi/deep_sketch

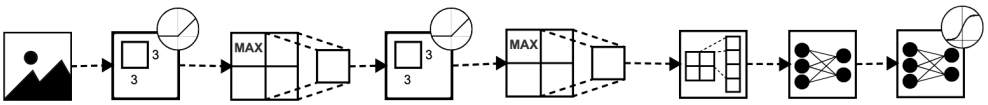


Figure 3.8. Model architecture visualized using DeepSketch.

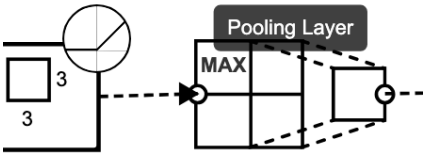


Figure 3.9. Textual annotation in DeepSketch.

high-level abstract unit. This mechanism is interactive, allowing the user to toggle between an expanded view and a collapsed, simplified view. This capability addresses the limitations of human working memory by allowing the viewer to hide detail and view the architecture at a higher level of abstraction, preventing information overload.

We argue that DeepSketch demonstrates an optimal Cognitive Fit for educational and conceptual design tasks. Its semantic transparency and use of familiar neural network metaphors aligns with the mental models of students and researchers explaining architectural logic.

3.3 Summary

This chapter has detailed the design of a new visual notation for deep neural network architectures. The proposed notation utilizes distinct visual metaphors. Each design choice was made to ensure Semantic Transparency, allowing users to infer the mathematical or logical function of a layer through its visual representation.

The evaluation of the proposed notation indicates that it successfully addresses the challenges of architectural visualization and the shortcomings identified in existing tools, as analyzed in Sections 2.3.5 and 2.3.6.

The following chapter describes how this notation was instantiated into an interactive web-based tool. It details the technical architecture of the DeepSketch program, the implementation of translating static visual rules into dynamic user interfaces, and the logic behind the tool's interactive features.

Chapter 4

Implementation

This chapter details the technical realization of the DeepSketch tool. It describes the software architecture, the specific technologies employed, and the implementation details of the core components, that is the Python backend and the interactive frontend.

4.1 System Architecture

DeepSketch is built as a client-server web application. This architecture was chosen to ensure accessibility across different platforms without the need for local installation, fulfilling the requirement for a web-based medium.

The system consists of two primary subsystems:

1. The Backend: A Python-based RESTful API server responsible for defining the neural network ontology and handling authentication.
2. The Frontend: A Single Page Application built with JavaScript, HTML5, and CSS. It handles the visualization, user interaction and state management.

4.1.1 Technology Stack

- Backend: The server is implemented using Flask, a lightweight WSGI web application framework. FlaskCORS is used to handle Cross-Origin Resource Sharing, allowing the frontend to communicate securely with the deployed API.
- Frontend: The user interface is built using vanilla JavaScript without use of any frameworks.
- Rendering Engine: The tool utilizes a hybrid rendering approach where HTML DOM elements serve as containers for layer nodes to handle positioning, selection, and drag-and-drop events, while Scalable Vector Graphics(SVG) are used for the visual content of layers and connections to ensure detailed rendering at any zoom level. Additionally, the HTML5 Canvas is used for the background grid. While most layer icons are loaded as static SVGs, the ConvolutionalLayer is a special case where the SVG representation is generated programmatically using a dedicated SVGGenerator class to dynamically visualize parameters like kernel size.

4.2 Domain Model

The structural logic of the neural network is formalized in the Domain Model presented in Figure 4.1. The architecture is centered around a `Node` abstraction, which can be instantiated either as an individual `Layer` or as a `Group`. To manage complexity, a `Group` acts as a container for two or more `Layer` instances. Individual layers are further defined by their configuration, optionally including a single `ActivationFunction` and any number of `Parameters`.

Connectivity within the network is modeled through the `Connection` class, which requires exactly one `source` and one `target` node to be valid. However, the model does not require that a node must participate in a relationship. `Node` can exist within the `Network` as a standalone entity without any incoming or outgoing connections.

It should be noted that specific layer subtypes and concrete activation functions are omitted from this diagram. This is a deliberate abstraction intended to represent the architectural logic and connectivity rules rather than exhaustive implementation details.

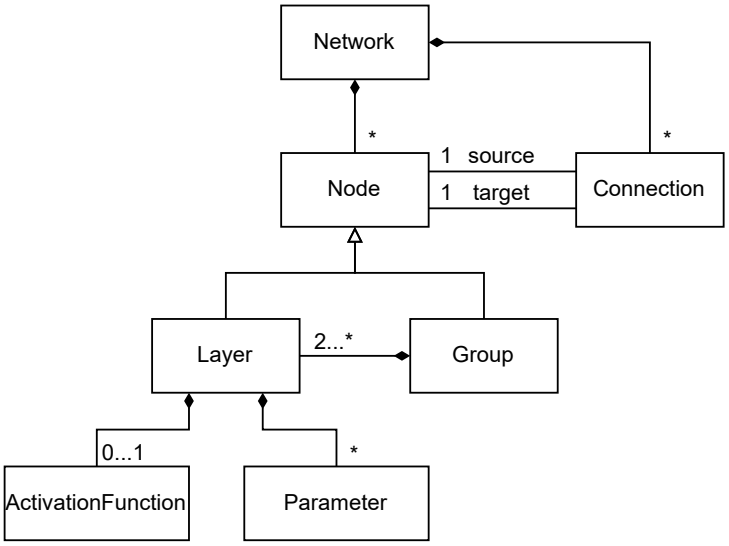


Figure 4.1. Domain Model of the DeepSketch tool.

4.3 Backend Implementation

The backend serves two main purposes: it provides the static definitions of neural network layers and manages the API for features like user logging.

Layer Ontology Rather than hard-coding layer parameters in the frontend, the backend defines a base `Layer` class and specific subclasses. The `app.py` module contains a `get_class_info` function that utilizes Python's `inspect` module to dynamically read the constructor signatures of these layer classes.

When the frontend requests available layer types(`GET /api/layer-types`), the backend performs the following steps:

1. Inspects the `__init__` method of the layer class.
2. Extracts parameter names, data types, and default values.
3. Identifies enumerated types(`InputType` for input layers) and serializes their possible values.
4. Bundles the SVG representation of the layer(defined in the class) with this metadata.

This approach ensures that if a new parameter is added to a layer in the Python backend, the frontend automatically adapts its property panels without requiring changes to the JavaScript code.

API Endpoints The Flask application exposes several endpoints:

- `/api/layer-types`: Returns the definitions and parameters for all supported layers.
- `/api/networks`: Manages the creation of network session IDs.
- `/api/user-logs`: Handles the logging of user actions.

4.4 Frontend Implementation

The frontend is the core of the artifact. It is structured around a central `Canvas` class that coordinates several manager classes.

4.4.1 Interface

The DeepSketch user interface(see Figure 4.2) is organized as a four panel layout. It consists of:

- Component Library: The left sidebar functions as a palette containing the building blocks of the neural network. It is categorized into Standard Layers and Activation Functions. Each component is represented by a distinct icon and a text label, designed for drag-and-drop interaction.
- Main Canvas: The central workspace is an infinite canvas where the network architecture is assembled. Users populate the canvas by dragging and dropping the components form the left panel onto the drawing area.
- Control Panel: The top-right sidebar includes utility controls. It includes buttons for session management(Loading and Saving), canvas manipulation(Clear Canvas) and logical organization of layers(Group Layers).
- Layer Properties Panel: The bottom-right panel includes layer properties. Its purpose is to expose the configuration parameters of the currently selected node, allowing users to change the properties of neural network layers. If no node is selected, it is hidden from the interface.

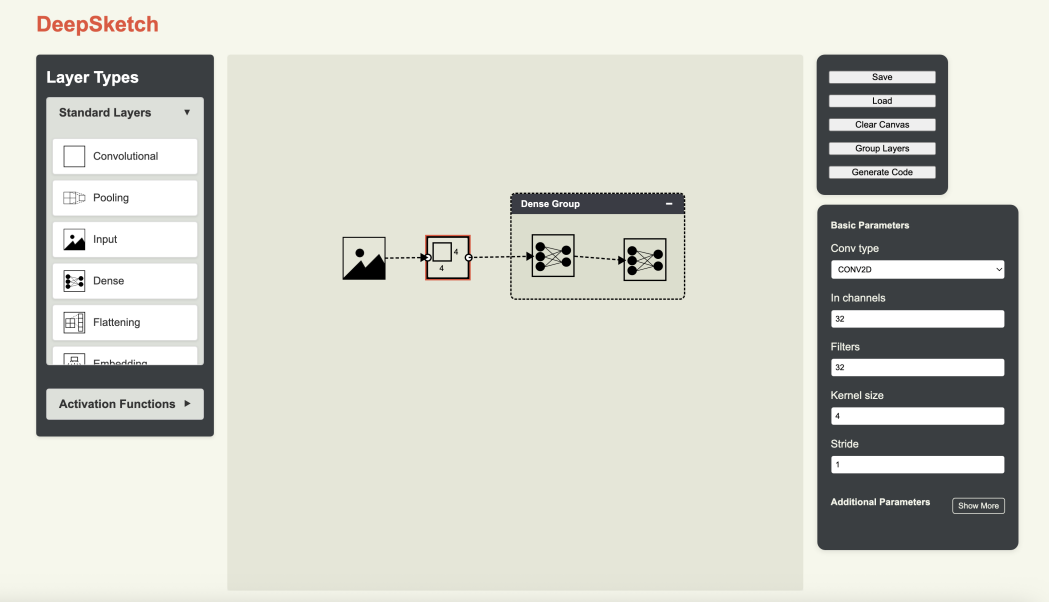


Figure 4.2. DeepSketch Interface

4.4.2 Canvas and Navigation

The `Canvas` class(`Canvas.js`) manages the viewport state, including panning and zooming. It implements a coordinate system transformation to map screen coordinates to canvas space coordinates.

- Zooming: Implemented via the `wheel` event. The system calculates a `zoomPoint` relative to the current mouse position to ensure the canvas zooms towards the cursor, rather than the top-left corner.
- Panning: Handled by tracking mouse movement while the right mouse button is held down. The `updateElementPositions` method recalculates the CSS `transform`, `left`, and `top` properties of all nodes based on the current panning and zooming values.

4.4.3 Layer and Group Management

Layers are managed by the `LayerManager` and `LayerFactory` classes, which handle the DOM creation and content rendering for layers. Additionally, `LayerFactory` defines the functionality for layers movement across the canvas.

Node Rendering Each layer is rendered as a `div` element containing an SVG visualization. The `LayerFactory` distinguishes between rendering strategies based on the layer type:

- Static Rendering: For majority of layers, the `LayerFactory` injects the pre-defined SVG string provided by the backend's type definition.

- Dynamic Rendering: For the `ConvolutionalLayer`, the factory utilizes the `SVGGenerator` class to construct the SVG string programmatically. This allows the visualization to update in real-time, reflecting changes to the layer's configuration(modifying the kernel size visualization).
- Custom Layer: To accommodate user-defined architectures, the system includes a CustomLayer. Property Panel includes a dedicated mechanism for customizing the layer's graphical representation. When a user selects the "Load Image" option for a `CustomLayer`, the `LayerPanelManager` triggers a file input dialog accepting standard image formats. The uploaded image is processed to resize the source image to a standardized 64×64 resolution. This processed image data is then applied to the layer's DOM node by overriding the default CSS `backgroundImage` property and setting the `backgroundColor` to transparent, replacing the default generic icon with the user's custom graphic.

Modularization The `GroupManager` implements the Complexity Management principle. It allows users to select multiple nodes and wrap them in a `layer-group` container. The implementation calculates the bounding box of selected nodes to size the group container appropriately. Furthermore, it provides a toggle functionality that allows users to switch between an expanded and collapsed view, dynamically hiding internal nodes to reduce visual clutter. Additionally, to accommodate layout adjustments, these containers support manual resizing via drag handles, allowing users to define custom boundaries for their architectural blocks as well as to name the created groups.

Property Configuration Parameter management is handled by the `LayerPanelManager`, which dynamically generates an editing interface based on the selected layer's type definition. This panel facilitates updates to layer properties, which are synchronized with the underlying data model, and layer visualization. As for the CustomLayer, the Property Panel allows users to add custom parameters, by specifying their name, value and type.

4.4.4 Connection Visualization

Connections are drawn using an SVG layer(`<svg id="connections">`) that sits below the HTML layer nodes. The `ConnectionVisualizer` draws lines between the connection points of source and target nodes. When nodes are dragged or the canvas is panned and zoomed, the `updateAllConnections` method is triggered to redraw these paths.

4.5 Persistence

To support the persistance of workflow, the system implements serialization via JSON.

- Saving: The `getNetworkState` method in `Canvas.js` iterates through all DOM nodes and connections, extracting their coordinates, IDs, types, and properties into a JSON object.
- Loading: The `loadNetworkState` method accepts a JSON file, clears the current canvas, and reconstructs the DOM elements and connections. It handles the re-mapping of IDs to ensure connections are correctly re-established.

Following user evaluation, undo/redo functionality was implemented, which was based on existing saving mechanism. This was implemented through `HistoryManager` class, which implements a stack to record application states. It facilitates navigation through these recorded states using an index pointer, allowing users to retrieve previous or subsequent snapshots to support standard undo and redo functionality.

4.6 User Interaction and Event Handling

4.6.1 Centralized Event Architecture

To prevent tight coupling between the rendering logic and user input, the application utilizes a centralized event delegation strategy. The `CanvasEventHandler` class acts as the primary controller, capturing DOM events from the canvas element and dispatching them to specialized managers based on the application state.

This separation allows for distinct handling modes. For instance, a `mousedown` event might trigger:

- Panning: If the right mouse button is held(handled by the `Canvas` class).
- Selection: If the left mouse button clicks the empty canvas(handled by `SelectionManager`).
- Dragging: If the click originates on a node(handled by `LayerFactory`).

4.6.2 Coordinate Space Transformation

An important implementation detail from the interactive point of view of the design is the mapping of screen coordinates to the virtual canvas coordinates. Since the canvas supports zooming and panning, raw mouse coordinates cannot be used directly.

All event handlers utilize a transformation function that applies the inverse of the viewport matrix to mouse events. The world coordinate(x_w, y_w) is derived from the client coordinate(x_c, y_c) using the formula:

$$x_w = \frac{x_c - \text{offset}_x - pan_x}{scale}, \quad y_w = \frac{y_c - \text{offset}_y - pan_y}{scale} \tag{4.1}$$

This logic is encapsulated in the `CanvasUtils.getCanvasPosition` method, ensuring that whether a user drops a layer at 200% zoom or selects a node while panned far to the right, the interaction occurs at the correct logical position.

4.6.3 Drag-and-Drop Functionality

DeepSketch implements drag-and-drop functionality that is based on context, modifying its behavior based on the type of the object being dragged. The `handleDragOver` and `handleDrop` methods in `CanvasEventHandler` implement this logic.

1. Instantiation Logic: When a user drags a layer from the sidebar, the system treats the canvas as a valid drop target. Upon release, a new node is instantiated at the transformed cursor coordinates.

2. Attachment Logic: When an activation function is dragged, the system switches to an attachment mode. The handler performs a hit test to detect if the cursor is hovering over an existing layer node.

To provide immediate feedback, valid targets are highlighted by dynamically toggling the `function-drop-target` CSS class. If the drop occurs over a valid node, the activation function is not created as an independent node but is instead linked to the parent layer and visually glued to its top-right corner.

4.6.4 Box Selection

To facilitate the manipulation of large architectures, on top of selection of individual nodes by mouse-clicking, the tool implements a box selection mechanism presented in Figure 4.3. This is handled by the `SelectionManager`, which tracks the start and end coordinates of a drag operation to render a visual selection overlay.

The core of this feature is the collision detection used to determine which nodes fall within the user's selection box. The implementation uses intersection test, checking which nodes are fully encapsulated in the selection rectangle. Because DOM elements exist in the browser's layout space while the selection box exists in the canvas overlay space, the system first normalizes all node rectangles to a common coordinate system before applying the intersection logic:

This check runs on every mouse move during the selection phase, allowing the UI to update the "selected" state of nodes as the user drags the box.

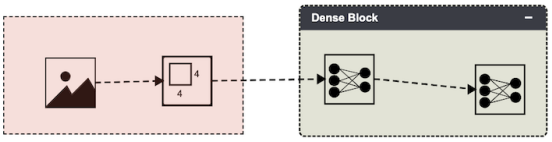


Figure 4.3. Selection mechanism in DeepSketch.

4.6.5 Node Dragging and Connectivity

Direct manipulation of nodes is implemented via the `LayerFactory.makeDraggable` method. This component handles the logic of moving nodes while maintaining their connectivity and group relationships.

When a node is dragged, the program must not only update the node's position but also recalculate the path of any incoming or outgoing connections. The `ConnectionVisualizer` is triggered on every `mousemove` event to redraw the connections attached to the moving node. Furthermore, if the node belongs to a `Group`, the drag logic switches to a relative coordinate system, ensuring the node moves correctly relative to its parent container rather than the global canvas. Additionally, the dragged node inside the group is bounded by the group box dimensions, not allowing the node to be dragged outside of the box.

4.7 Summary

This chapter detailed the technical realization of DeepSketch, describing the development of the system from its Python backend to its interactive JavaScript frontend. By automating the layer ontology through backend inspection and providing specialized frontend features for architectural design, the system allows for the developed notation to be used in a functional tool. With the technical infrastructure of DeepSketch established and the artifact implemented, the focus shifts to validation. The following chapter details a pilot study conducted with users to evaluate the tool's usability, its effectiveness in visualizing neural network architectures and its potential improvements.

Chapter 5

Pilot Study

To validate the design decisions and implementation details discussed in the previous chapters, a pilot study was conducted. This study aimed to evaluate the initial usability of the DeepSketch tool, identify usability issues, and gather qualitative feedback from participants. Given the that the tool was in early phase of development, the study was explicitly designed as a developmental evaluation, aiming to identify design flaws

5.1 Study Design

The pilot study was designed as a task-based usability evaluation followed by a structured questionnaire. The primary goal was to assess whether users with basic knowledge of neural networks could successfully construct a specific Convolutional Neural Network(CNN) architecture using the tool without prior training.

The study involved two participants recruited from the academic field. Both participants possessed a background in Computer Science or Informatics and self-reported a moderate level of experience with neural networks.

- *Participant 1:* A PhD Candidate in Software Engineering and Deep Learning Testing.
- *Participant 2:* A Postdoctoral Researcher in Informatics.

Although the sample size is small, this is acceptable for a pilot usability study, whose purpose is to discover usability issues and implementation flaws. All participants voluntarily took part in the study and were informed about its purpose.

5.1.1 Procedure

The study followed a three step procedure:

1. Tutorial: Participants were given a brief introduction to the DeepSketch interface, demonstrating the core functionalities: adding layers, connecting nodes, and editing layer properties.
2. Task Execution: Participants were asked to complete a specific modeling task within a 15-minute time limit.

3. Questionnaire: After the task, participants filled out a feedback form comprising Likert scale(5-point scale) questions and open-ended text fields.

5.1.2 Task Description

Participants were instructed to design a simple CNN for classifying images of handwritten digits (MNIST). The architecture requirements were defined to test specific features of the tool; layer instantiation, parameter configuration, connections, and grouping.

Architecture Specification:

- Input: Image Input Layer.
- Feature Extraction Block:
  - Convolutional Layer(Filters: 32, Kernel: 3, Activation: ReLU).
  - Pooling Layer(Type: MaxPooling, Kernel: 2).
  - Convolutional Layer(Filters: 64, Kernel: 3, Activation: ReLU).
- Flattening Layer
- Classification Head:
  - Dense Layer(Units: 128, Activation: ReLU).
  - Dropout Layer(Probability: 0.5).
  - Dense Layer(Units: 10, Activation: Softmax).

Organizational Requirement: Upon construction, participants were required to use the grouping functionality to modularize the network:

- Group 1: Layers 1–4 named "FeatureExtractor".
- Group 2: Layers 5–8 named "DecisionMaking".

5.1.3 Questionnaire Design

The questionnaire was designed to collect both quantitative usability metrics and qualitative opinions. Responses were recorded using a 5-point Likert scale(1 = Very Difficult, 5 = Very Easy), followed by open-ended questions.

Demographics and Background

These questions aimed to validate that the participants represented the target audience and had sufficient domain knowledge to evaluate the tool effectively.

- Education level / Field of study: Confirms the academic background of the participant.
- Current Role: Identifies whether the user is a researcher, student, or practitioner.
- Experience with Neural Networks: Assesses domain expertise to ensure a proper “Cognitive Fit” evaluation, as the tool is intended for users familiar with deep learning concepts.

General Usability and Error Tolerance

Questions in this category evaluated standard usability functionalities such as navigation and error recovery:

- How easy was it to find all the required features? This question identifies whether the tool's interface is intuitive enough for users to locate core architectural design functions. Required features include, for example, the layer selection menu for adding new components, the parameter editor for modifying layer attributes, and the save function for serializing the final visualization.
- How easy was it to fix mistakes if any? This evaluates error tolerance by measuring how easily a user can recover from errors. Examples of such mistakes include topological errors, such as connecting wrong layer types, or parameter errors, such as entering an invalid parameter value It also assesses the ease of correcting accidental actions, such as unintentionally deleting a sub-network or misplacing a layer within a dense sequence.

Interaction Assessment

To identify usability issues across the complete model creation process, participants were asked to rate the effectiveness of the following functionalities:

- Drag and drop layers
- Editing layer properties
- Drawing connections
- Moving layers and groups
- Grouping layers
- Saving the architecture

Cognitive Fit and Visual Notation

These questions examined how well the visual notation conveys information.

- How clearly does the visual design express the structure and components of neural network? Assesses representational fidelity and mapping to the user's mental model.
- How intuitive were the icons and visual elements? Evaluates the symbolic syntax of the layer icons.
- If you showed your final architecture to someone else, how helpful would the visual output be in explaining the model? Measures the communicative value of the notation for collaboration.
- Was the way layer properties are displayed clear and useful? Assesses the information hierarchy and visibility of parameters.
- How satisfied are you with the final design produced by the tool? Provides a measure of user satisfaction with the final output.

Qualitative Feedback

Open-ended questions provided context to the numeric ratings and were designed to indicate usability flaws, distinguish implementation faults from design flaws, gather future requirements, and measure future adoption likelihood:

- What aspects of the tool were confusing or frustrating?
- Did you encounter any bugs or technical issues?
- What features or improvements would you like to see?
- Would you use this tool again for designing neural networks?

5.2 Results

5.2.1 Quantitative Analysis

Table 5.1 shows the quantitative questionnaire results of both participants.

Feature / Qualitative Question	**P1 Rating**	**P2 Rating**
Usability & Error Tolerance		
Ease of finding required features	4	5
Ease of fixing mistakes	3	2
Interaction Assessment		
Drag and Drop Layers	4	5
Editing Layer Properties	4	5
Drawing Connections	4	5
Moving Layers/Groups	3	5
Grouping Layers	4	4
Saving Architecture	5	5
Cognitive Fit and Visual Notation		
Clarity of structural expression	4	5
Intuitiveness of icons and visual elements	5	5
Helpfulness for explaining models to others	5	5
Clarity and usefulness of property display	5	5
Satisfaction with the final design	4	4

Table 5.1. User ratings for tool's functionalities.

Participants reported high discoverability for core functionalities, providing ratings of 4/5 and 5/5 for the ease of finding required features. The ease of fixing mistakes received the lowest ratings in the study, recorded at 3/5 and 2/5. As noted in the qualitative feedback, the absence of an automated undo function as well as errors in the code negatively impacted the efficiency of error recovery.

The interaction assessment evaluated the mechanical operations required to construct and modify architectures. The primary interaction method, drag and drop layers, received ratings

of 4/5 and 5/5. Participants rated the process of editing layer properties with scores of 4/5 and 5/5, while the mechanism for drawing connections between nodes was evaluated at 4/5 and 5/5. Spatial manipulation via moving layers/groups received an acceptable rating of 3/5 from P1 and a 5/5 from P2. The grouping layers feature, designed for modularization, received consistent ratings of 4/5 from both participants. Finally, the saving architecture function received a unanimous rating of 5/5.

Cognitive Fit and Visual Notation category evaluated the effectiveness of the visual language and its alignment with the user's mental model. The clarity of structural expression, representing the degree to which the visual design accurately reflects the neural network, was rated at 4/5 and 5/5. The intuitiveness of icons and visual elements received a unanimous rating of 5/5. Participants evaluated the helpfulness for explaining models to others at 5/5, indicating the notation's usefulness for communication. The clarity and usefulness of the property display was rated at 5/5 by both participants. Overall satisfaction with the final design received consistent scores of 4/5.

5.2.2 Qualitative Feedback and Issues

The open-ended responses highlighted areas for improvement.

Issues

- Lack of Undo/Redo: Both participants identified the inability to undo actions as a primary frustration. Participant 1 noted, "It was not possible to Undo a wrong deletion or... it was hard for me to find any button," while Participant 2 listed "Missing Undo" as the first confusing aspect.

- Grouping Behavior: Both participants encountered issues with the grouping logic. Participant 2 noted that "if layers are connected all together, grouping only a subset of them is not feasible (all the layers in the network get included)", suggesting a bug in how the selection logic propagates through connections.

- State Persistence: Participant 1 reported that upon loading a saved network, "ReLU functions were scattered across the canvas", indicating a problem with the serialization/deserialization logic for attached activation functions.

Feature Requests

While participants suggested several enhancements for future iterations, specifically regarding validation, parameter visibility, and aesthetic upgrades, the project maintained its implementation. Although users requested a mechanism to check if parameter values are compatible, the tool is strictly intended as a free design environment. Consequently, no validation is performed, allowing users to create arbitrary connections, such as linking an input directly to another input. This approach also extended to the suggestions for data flow animations. Similarly, the design for parameter visibility remained unchanged, requiring users to click a node to view its values rather than displaying them explicitly on the canvas(except for Convolutional and Pooling Layer), as we argue that it could unnecessarily introduce visual clutter.

5.3 Discussion

The results of the pilot study indicate that the fundamental design of DeepSketch, its direct manipulation interface and visual metaphor, is successful. The high ratings for intuitiveness(5/5) and the willingness of both participants to use the tool again(ratings of 5 and 4) suggest strong Cognitive Fit.

However, the study revealed engineering gaps that hinder the error-tolerancy. The absence of an Undo/Redo functionality forces users to manually correct mistakes, which significantly lowers the usability score for fixing mistakes. Additionally, the identified bugs in the grouping mechanism and file loading logic highlighted some usability problems with the tool.

These findings served as a requirements for the subsequent development phase, where connection deletion, undo functionality were added and visual bugs were eliminated.

Chapter 6

Conclusion

6.1 Contribution

This thesis addressed the lack of a standardized visual notation in the field of deep learning and the currently existing cognitively inefficient tools. In this context, cognitive inefficiency refers to visualizations that impose a high mental workload on the user due to visual clutter, a lack of differentiable symbols, or lack of modularization capabilities. While the complexity and adoption of neural networks have grown, the methods used to visually design their architectures have remained arbitrary. This work introduced a new visual language and the corresponding interactive design tool, DeepSketch.

The primary contribution of this research is a visual notation based on Moody's Physics of Notations. Unlike existing specialized solutions such as Net2Vis or NN-SVG, which are often limited to specific tasks or publication aesthetics, this work proposes a more general approach to architectural design. The current implementation provides a set of 10 distinct layers and 4 activation functions, utilizing semantically transparent symbols to reduce cognitive load. The notation is at an abstraction level, prioritizing topological understanding over lower-level details.

The realized tool, incorporating this notation, enhances the user experience through an interactive design environment that facilitates model construction in a more streamlined and intuitive fashion. These facilitating capabilities allow users to focus on the logical structure of the network rather than the manual efforts of diagramming, like in some of the analyzed tools in section 2.3.5.

6.2 Evaluation

The proposed notation was evaluated through a principle-based analysis against state-of-the-art tools and a pilot study. The results suggest that DeepSketch is more cognitively effective than existing diagramming methods. By adhering to PoN principles such as Semantic Transparency and Complexity Management, the tool allows users to construct and interpret neural network architectures with lower cognitive overhead.

The transition from inconsistent, ad-hoc diagrams toward a standardized visual language is an important step for the efficient communication of deep neural network models. By providing

a notation that respects cognitive limits and adheres to semantic principles, this thesis offers a practical step toward making neural network design visualization more cognitively effective.

6.3 Limitations and Future Work

While this work establishes formalized visual notation, several areas remain for future expansion. To enhance the tool's practical utility, the automatic code generation feature could be fully realized to support some of the most popular deep learning frameworks like PyTorch and TensorFlow. Building upon this, reverse engineering functionality could be implemented, allowing the system to parse existing code and automatically visualize architectures using the proposed notation, similar to the functionality seen in tools like Netron or Net2Vis. Additionally, features suggested by pilot study participants, such as topology validation to prevent invalid connections and data flow animations to visualize tensor dimensions, would further improve the usability of the tool.

Finally, we acknowledge that a more extensive human study is required to fully validate the proposed notation. Specifically, a direct comparison of usability and cognitive effectiveness against existing tools is necessary to assess whether this tool represents a step toward more cognitively effective communication of neural network architecture visualizations.

Bibliography

[1] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[2] Michele Lanza and Stéphane Ducasse. Polymetric views-a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, 2003.

[3] Yaser Ghanam and Sheelagh Carpendale. A survey paper on software architecture visualization. *University of Calgary, Tech. Rep*, page 17, 2008.

[4] Zhengshu Zhou, Qiang Zhi, Shuji Morisaki, and Shuichiro Yamamoto. A systematic literature review on enterprise architecture visualization methodologies. *IEEE Access*, 8:96404–96427, 2020.

[5] Alex Bäuerle, Christian Van Onzenoodt, and Timo Ropinski. Net2vis–a visual grammar for automatically generating publication-tailored cnn architecture visualizations. *IEEE transactions on visualization and computer graphics*, 27(6):2980–2991, 2021.

[6] Alexander LeNail. Nn-svg: Publication-ready neural network architecture schematics. *J. Open Source Softw.*, 4(33):747, 2019.

[7] Zijie J Wang, Robert Turko, Omar Shaikh, Haekyu Park, Nilaksh Das, Fred Hohman, Minsuk Kahng, and Duen Horng Polo Chau. Cnn explainer: learning convolutional neural networks with interactive visualization. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):1396–1406, 2020.

[8] Aeree Cho, Grace C. Kim, Alexander Karpekov, Alec Helbling, Zijie J. Wang, Seongmin Lee, Benjamin Hoover, and Duen Horng Chau. Transformer explainer: Interactive learning of text-generative models. *IEEE VIS*, 2024.

[9] Jason Yosinski, Jeff Clune, Anh Nguyen, Thomas Fuchs, and Hod Lipson. Understanding neural networks through deep visualization. *arXiv preprint arXiv:1506.06579*, 2015.

[10] Zhuwei Qin, Fuxun Yu, Chenchen Liu, and Xiang Chen. How convolutional neural network see the world-a survey of convolutional neural network visualization methods. *arXiv preprint arXiv:1804.11191*, 2018.

[11] Daniel Moody. The "physics" of notations: toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on software engineering*, 35(6):756–779, 2009.

[12] Ian Goodfellow. Deep learning, 2016.

[13] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[14] Lutz Roeder. Netron. `https://github.com/lutzroeder/netron`, 2010.

[15] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034*, 2013.

[16] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[17] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. Learning deep features for discriminative localization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2921–2929, 2016.

[18] Chris Olah, Alexander Mordvintsev, and Ludwig Schubert. Feature visualization. *Distill*, 2(11):e7, 2017.

[19] Laurens Van Der Maaten. Accelerating t-sne using tree-based algorithms. *The journal of machine learning research*, 15(1):3221–3245, 2014.

[20] Minsuk Kahng, Pierre Y Andrews, Aditya Kalro, and Duen Horng Chau. A cti v is: Visual exploration of industry-scale deep neural network models. *IEEE transactions on visualization and computer graphics*, 24(1):88–97, 2017.

[21] Fred Hohman, Minsuk Kahng, Robert Pienta, and Duen Horng Chau. Visual analytics in deep learning: An interrogative survey for the next frontiers. *IEEE transactions on visualization and computer graphics*, 25(8):2674–2693, 2018.

[22] Jun Yuan, Changjian Chen, Weikai Yang, Mengchen Liu, Jiazhi Xia, and Shixia Liu. A survey of visual analytics techniques for machine learning. *Computational Visual Media*, 7(1):3–36, 2021.

[23] Stuart K Card, Jock Mackinlay, and Ben Shneiderman. *Readings in information visualization: using vision to think*. Morgan Kaufmann, 1999.

[24] Claire Knight and Malcolm Munro. Comprehension with [in] virtual environment visualisations. In *Proceedings Seventh International Workshop on Program Comprehension*, pages 4–11. IEEE, 1999.

[25] Denis Gračanin, Krešimir Matković, and Mohamed Eltoweissy. Software visualization. *Innovations in Systems and Software Engineering*, 1(2):221–230, 2005.

[26] Stephen G Eick, Todd L Graves, Alan F Karr, Audris Mockus, and Paul Schuster. Visualizing software changes. *IEEE Transactions on Software Engineering*, 28(4):396–412, 2002.

[27] Richard Wettel and Michele Lanza. Visualizing software systems as cities. In *2007 4th IEEE International workshop on visualizing software for understanding and analysis*, pages 92–99. IEEE, 2007.

[28] Jill H Larkin and Herbert A Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive science*, 11(1):65–100, 1987.

[29] Allan Paivio. *Mental representations: A dual coding approach*. Oxford university press, 1990.

[30] Thomas RG Green. Cognitive dimensions of notations. *People and computers V*, pages 443–460, 1989.

[31] Alan F Blackwell, Carol Britton, Anna Cox, Thomas RG Green, Corin Gurr, Gada Kadoda, Maria S Kutar, Martin Loomes, Chrystopher L Nehaniv, Marian Petre, et al. Cognitive dimensions of notations: Design tools for cognitive technology. In *International conference on cognitive technology*, pages 325–341. Springer, 2001.

[32] Jason Dagit, Joseph Lawrance, Christoph Neumann, Margaret Burnett, Ronald Metoyer, and Sam Adams. Using cognitive dimensions: advice from the trenches. *Journal of Visual Languages & Computing*, 17(4):302–327, 2006.

[33] Daniel Moody. Theory development in visual language research: Beyond the cognitive dimensions of notations. In *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 151–154. IEEE, 2009.

[34] Thomas RG Green, Ann E Blandford, Luke Church, Chris R Roast, and Steven Clarke. Cognitive dimensions: Achievements, new directions, and open questions. *Journal of Visual Languages & Computing*, 17(4):328–365, 2006.

[35] Jonathan I Maletic, Andrian Marcus, and Michael L Collard. A task oriented view of software visualization. In *Proceedings first international workshop on visualizing software for understanding and analysis*, pages 32–40. IEEE, 2002.

[36] Guy Clarke Marshall, Caroline Jay, and André Freitas. Diagrammatic summaries for neural architectures. In *Beyond static papers: Rethinking how we share scientific understanding in ML-ICLR 2021 workshop*, 2021.

[37] Guy Marshall, Andre Freitas, and Caroline Jay. An evidence-based guidance framework for neural network system diagrams. *PloS one*, 20(3):e0318800, 2025.

[38] Paul Gavrikov. Visualkeras. `https://github.com/paulgavrikov/visualkeras`, 2020.

[39] Chaunte W. Lacewell. Netscope cnn analyzer. `https://github.com/cwlacewe/netscope`, 2017.

[40] Haris Iqbal. Plotneuralnet. `https://github.com/HarisIqbal88/PlotNeuralNet`, 2018.

[41] tensorflow. tensorboard. `https://github.com/tensorflow/tensorboard`, 2017.

[42] tensorflow. tensorflow. `https://github.com/tensorflow/tensorflow`, 2015.

[43] Jesse Michel. Ennui. `https://github.com/martinjm97/ENNUI`, 2019.

[44] Milan Lajtoš. Moniel. `https://github.com/mlajtos/moniel`, 2016.

[45] Luca Bonfiglioli. Nnviz. `https://github.com/LucaBonfiglioli/nnviz`, 2016.

[46] A. D'Onofrio. A sim-to-real approach for vision-depth cnn-based obstacle avoidance in challenging environments. Master's thesis, 2024.