TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Joosep Lepland 222908

DIAGRAMMATIC MODELING OF NEURAL NETWORKS WITH CATEGORY THEORY AND STRING DIAGRAMS

Bachelor's Thesis

Supervisor: Pawel Maria Sobocinski
PhD

Co-supervisor: Niels Frits Willem Voorneveld
PhD

Tallinn 2025

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Joosep Lepland 222908

NÄRVIVÕRKUDE SKEMAATILINE MODELLEERIMINE KATEGOORIATEOORIA JA NÖÖRSKEEMIDE ABIL

Bakalaureusetöö

Juhendaja: Pawel Maria Sobocinski
PhD

Kaasjuhendaja: Niels Frits Willem Voorneveld
PhD

Tallinn 2025

Author's Declaration of Originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Joosep Lepland

19.05.2025

Abstract

The main goal of this bachelor's thesis involves creating an Ivaldi diagramming tool extension that allows users to create neural network schematics and automatically produce code through a category theory and string diagrams based domain-specific language (DSL). The Ivaldi application received new predefined boxes that represent PyTorch layer modules for constructor and layer functions that contain input and output elements, together with parameter settings such as neuron numbers and hidden-state dimensions.

The research began by examining basic neural network architectures, including feedforward networks, convolutional networks, recurrent networks, and transformers, while studying their standard layers and data movement patterns. Ivaldi received a new DSL that enables users to build diagrams by placing layers and setting parameters while supporting both linear and partially non-linear neural network modeling.

The Python script generation process starts with a diagram that produces a hypergraph representation, which then generates code in topological order by substituting template placeholders with user-defined parameters. The solution was demonstrated on several example models: simple feedforward networks, a multi-layer CNN classifier, recurrent networks, and transformer-based models. The diagram-driven construction process produced accurate results, and code generation functionality enabled the models to undergo direct training.

The Ivaldi add-on developed through this thesis enables users to construct different neural networks diagrammatically while automatically generating their corresponding code. Future development of the DSL should focus on enhancing its ability to show the actual data movement within the model.

The thesis is written in English and is 60 pages long, including 4 chapters, 49 figures, and 4 tables.

Annotatsioon

Närvivõrkude skemaatiline modelleerimine kategooriateooria ja nöörskeemide abil

Käesoleva bakalaureusetöö peamine eesmärk on välja töötada diagrammitööriista Ivaldi lisamoodul, mis võimaldab närvivõrkude skeemilist modelleerimist ja automaatset koodi genereerimist, kasutades kategooriateoorial ja nöörskeemidel (string diagrams) põhinevat domeenispetsiifilist keelt (DSL). Selleks loodi Ivaldi diagrammi rakendusse DSL-i täiendavad konstruktor- ja kiht kastid, mis vastavad PyTorchi kihimoodulitele ning kannavad endas sisendeid, väljundeid ja parameetreid (nt neuronite arv, sisemine varjatud dimensioon).

Töö esimeses etapis käsitleti närvivõrkude põhiarhitektuure, pärilevivõrke, konvolutsioonilisi võrke, rekurentseid võrke ja transformereid ning selgitati iga arhitektuuri tüüpilisi kihte ja omavahelisi andmevooge. Seejärel integreeriti Ivaldi rakendusse uus DSL, mis võimaldab kasutajal visuaalselt paigutada kihte diagrammile, määrata neile parameetreid ning modelleerida nii lineaarseid kui ka osaliselt mitte lineaarseid närvivõrke.

Diagrammilt genereeritakse hüpergraafi alusel topoloogilises järjekorras Pythoni skript, kus asendatakse koodimallide kohatäited vastavalt kasutaja määratud parameetritega. Töös demonstreeriti lahendust mitmete näidismudelite peal: lihtsad pärilevivõrgud, mitmekihiline CNN-klassifikaator, rekurentsed võrgud ning transformer-baasil mudelid. Kõigi näidismudelite puhul saavutati korrektsed diagrammipõhised konstruktsioonid ja töötab koodi genereerimine, mis võimaldab mudelit otse treenida.

Selle lõputöö tulemuseks on loodud Ivaldi rakendusele lisamoodul, mis võimaldab diagrammil luua erinevaid närvivõrke ja sealt otse ka vastav kood genereerida. Edasises arengus tuleks DSL-i rikastada nii, et oleks võimalik paremini illustreerida ka seda kuidas andmed läbi mudeli voolavad.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 60 leheküljel, 4 peatükki, 49 joonist, 4 tabelit.

List of Abbreviations and Terms

API	Application Programming Interface
DSL	Domain-Specific Language
FFN	Feedforward Neural Network
MLP	Multi-Layer Perceptron
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
LSTM	Long Short-Term Memory
GRU	Gated Recurrent Unit
UI	User Interface
BERT	Bidirectional Encoder Representations from Transformers

Table of Contents

1 Introduction **12**
1.1 Problem Statement 12

2 Background **15**
2.1 Category Theory 15
2.2 String Diagrams 16
2.3 Neural Networks 16
2.3.1 Transformer Neural Networks 17
2.4 String Diagrams and Category Theory in Neural Network Modeling . . . 18
2.5 Project Context: The Ivaldi Application 19

3 Methodology **21**
3.1 Ivaldi 21
3.1.1 Overview 21
3.1.2 Backend 21
3.1.3 Frontend 22
3.1.4 Code Generation 23
3.1.5 Predefined Boxes 24
3.2 Classical Neural Network Architecture Modeling 26
3.2.1 Feedforward Neural Networks 26
3.2.2 Convolutional Neural Networks 27
3.2.3 Recurrent Neural Networks 28
3.2.4 Splitting 30
3.2.5 Creating Boxes 32
3.3 Transformer Neural Network Modeling 37
3.3.1 Transformer Neural Networks 37
3.3.2 Transformer Architecture 39
3.3.3 Splitting 43
3.3.4 Creating Boxes 47
3.4 Validating 50
3.4.1 Creating Models On Diagram 50
3.4.2 Generating Code From Diagram 54
3.4.3 Training Generated Models 67

4 Summary **72**

References **73**

Appendix 1 – Non-Exclusive License for Reproduction and Publication of a Graduation Thesis **77**

Appendix 2 – Info And Edit Parameters Pop-up Examples **78**

Appendix 3 – Transformer Fourth Split Sub-Diagrams **79**

Appendix 4 – Transformer Fifth Split Sub-Diagrams **82**

Appendix 5 – Transformer Fifth Split Encoder Only Architecture **86**

Appendix 6 – Transformer Fifth Split Decoder Only Architecture **87**

Appendix 7 – Further Information About Transformer Boxes **89**

Appendix 8 – Encoder-Decoder Transformer Code Validation 1 . . . . . **97**

Appendix 9 – Encoder-Decoder Transformer Code Validation 2 . . . . . **101**

Appendix 10 – Encoder-Only Transformer Code Validation 1 . . . . . **103**

Appendix 11 – Encoder-Only Transformer Code Validation 2 . . . . . **105**

Appendix 12 – Decoder-Only Transformer Code Validation 1 . . . . . **106**

Appendix 13 – Decoder-Only Transformer Code Validation 2 . . . . . **108**

Appendix 14 – Encoder-Decoder Transformer Better Flow Representation . . **109**

List of Figures

1 *The Transformer Model Architecture [9]* 39
2 *The First Split Of The Transformer Model* 43
3 *The Second Split Of The Transformer Model* 44
4 *The Third Split Of The Transformer Model* 45
5 *The Fourth Split Of The Transformer Model* 45
6 *The Fifth Split Of The Transformer Model* 46
7 *Example One Feedforward Model* 50
8 *Example Two Feedforward Model* 51
9 *Example Convolutional Model* 51
10 *Example One Recurrent Model* 52
11 *Example Two Recurrent Model* 52
12 *Example Three Recurrent Model* 53
13 *Script Two Output For First FFN Model* 56
14 *Script Two Output For Second FFN Model* 57
15 *Script Two Output For CNN Model* 59
16 *Script Two Output For First RNN Model* 61
17 *Script Two Output For Second RNN Model* 62
18 *Script Two Output For Third RNN Model* 63

19 *Info And Edit Parameters Pop-up Example* 78
20 *Info Pop-up Example* 78
21 *Fourth Split Sub-Diagram Structure* 79
22 *Sub-Diagram Inside The Custom Transformer Box* 79
23 *Sub-Diagram Inside The Encoders Box* 80
24 *Sub-Diagram Inside The Custom Encoder Box* 80
25 *Sub-Diagram Inside The Encoder Layers Box* 80
26 *Sub-Diagram Inside The Decoders Box* 80
27 *Sub-Diagram Inside The Custom Decoder Box* 81
28 *Sub-Diagram Inside The Decoder Layers Box* 81
29 *Fifth Split Sub-Diagram Structure* 82
30 *Sub-Diagram Inside The Encoders Block Box* 83
31 *Sub-Diagram Inside The Basic Encoder Box* 83
32 *Sub-Diagram Inside The Enc (Encoder) Self Attention Block Box* 83
33 *Sub-Diagram Inside The Feedforward Block Box* 83
34 *Sub-Diagram Inside The Decoders Block Box* 84

35 *Sub-Diagram Inside The Basic Decoder Box* 84
36 *Sub-Diagram Inside The Dec (Decoder) Self Attention Block Box* 84
37 *Sub-Diagram Inside The Cross Attention Block Box* 84
38 *Sub-Diagram Inside The Feedforward Block Box* 85
39 *Fifth Split Encoder Only Model* 86
40 *Fifth Split Encoder Only Architecture Sub-Diagram Structure* 86
41 *Fifth Split Decoder Only Model* 87
42 *Fifth Split Decoder Only Architecture Sub-Diagram Structure* 87
43 *Sub-Diagram Inside The Basic Decoder Box* 88
44 *Validation Script Output Part 2.1* 101
45 *Validation Script Output Part 2.2* 102
46 *Validation Script Output* 105
47 *Validation Script Output* 108
48 *Transformer Better Data Flow Representation* 109
49 *One Basic Decoder Block Sub-Diagram Unfolded* 109

List of Tables

1 *Transformer Boxes Table* 48
2 *Transformer Translation Table* 67
3 *Transformer Generation Table* 69
4 *Transformer Prediction Table* 70

1. Introduction

An extension for a string-diagramming application is developed in this bachelor's thesis to simplify the visualization and compilation of neural networks. To achieve this, a string-diagrammatic domain-specific language (DSL) is embedded directly into the Ivaldi diagramming tool, leveraging the mathematical rigor of category theory and its graphical boxes and wires notation. Ivaldi is augmented with a palette of layer primitives, dense, convolutional, recurrent, and transformer blocks, whose inputs, outputs, and parameters are mapped one-to-one to PyTorch modules. Layers can be dragged onto a canvas, hyperparameters can be configured via a property panel, both linear and partially non-linear architectures can be composed, and ready-to-run Python code can be generated instantly.

By unifying high-level diagrammatic modeling with executable code generation, neural-network design is made more transparent, interactive, and accessible. In the following sections, the motivating problem statement is outlined, the theoretical foundations of category theory, string diagrams, and various neural network architectures are reviewed, the DSL's design and implementation in Ivaldi are described, and the tool is demonstrated on representative models ranging from feedforward classifiers to transformer encoder–decoder architectures.

1.1 Problem Statement

Though various tools and techniques exist to visualize the structure and different layers of neural networks in constructing large artificial intelligence models, especially transformers, they are yet to be sophisticated enough to ensure maximum ease and agility in developing large neural networks. Though some techniques, such as saliency maps, activation networks, and other visualization techniques, are employed, they often require specialized expertise and entail extra development efforts. Existing solutions make it possible to visualize the neural network structure and layers to some degree, though they lack accuracy, ease of use, and dynamic visualization.

Developers typically must imagine in their heads what the neural network that they create might be or utilize other tools to observe how its architecture and structure might be. Otherwise, they might read the code and attempt to understand what the neural network might be like; this one is a labor-intensive and complex process, particularly for larger models. Furthermore, hand-drawn diagrams of networks are not terribly informative

regarding the connections and neurons. Whenever one attempts to make alterations in the neural network, the visualization has to be redrawn, which is a time-consuming process.

While there are computer programs that can graph the structure of a neural network, these programs are not yet mature enough to be more effective and to provide a more detailed and dynamic view. This is an area where one can see that there is a clear need for new solutions and innovations to simplify the development of large models and make it quicker and easier to do [1].

The purpose of this bachelor's thesis is to create an extension for the diagramming tool Ivaldi that provides a clearer visual representation of the neural network being developed, along with more advanced information regarding neurons, connections, and layers. In addition, the add-on will allow quick modification and compilation of the AI model's architecture. Visualization will be provided in the shape of a string diagram within the software.

The goal of the bachelor's thesis is to develop a diagramming add-on that enables the compilation and visualization of neural network models. The diagram will include essential information about the network's layers and connections. Therefore, the goals of this thesis are to:

- Visualize neural networks
- Modify neural networks on the diagram
- Generate code from the diagram
- See information about the neural network on the diagram

Neural network visualization will be done in the Ivaldi application (see Section 2.5) using string diagrams. The application must be able to visualize a large AI transformer neural network as a diagram, showing layers and the connections between them.

Modifying neural networks on the diagram must be possible, so it must be possible to change the model's architecture conveniently. Mainly to change the model's layer structure by adding or removing layers.

Generating code from the diagram is done by using predefined boxes that have domain-specific language (DSL) code snippets in them. Code generation must work with different neural network architectures.

Information about the neural network layers must be accessible in the application

user interface (UI). The diagram must provide the necessary information about the layers to offer a better understanding of how the specific neural network operates.

2. Background

This chapter presents a summary of fundamental concepts that connect the Ivaldi application to the thesis work. The text starts by explaining category theory before moving to string diagrams and major neural network architectures, before concluding with an overview of the Ivaldi application. The background information establishes both theoretical and practical bases for diagrammatic modeling extensions that will be developed later.

2.1 Category Theory

Category theory provides an abstract framework for comparing mathematical structures based on their relationships, concentrating on the process of composition instead of their inner workings. Category theory is based on three concepts: categories, functors, and natural transformations [2].

A category has objects and arrows (morphisms) between objects. Every object has an identity arrow, and arrows can be composed. Composition always associates, meaning that the order you place them in can be rearranged. Objects in a category can be sets, groups, rings, or vector spaces, and the morphisms capture structure-preserving mappings, like functions, homomorphisms, or linear transformations. Categories are mathematical structures, and there is a concept of maps between categories called functors, which preserve identities and composition. Functors may be related by natural transformations, which are morphisms between functors. The hierarchy in category theory, therefore, is objects, arrows, and arrows between arrows [2].

Category theory's key concept is universal properties describing constructions, which map uniquely to any other object, such as products, coproducts, limits, and colimits. The origins of category theory emerged from algebraic topology, yet it now shapes the development of logic and type theory, and programming-language semantics. Category theory reveals concealed relationships between different subjects through relation-based descriptions, which create a common mathematical and computational framework for pure mathematics and computer science applications [2][1].

2.2 String Diagrams

String diagrams function as a visual language that enables both representation and reasoning about morphisms (abstract processes) within monoidal categories. The mathematical tool has found applications across multiple domains, starting from traditional circuit theory and probability through to contemporary uses in quantum mechanics and machine learning, and natural language processing. The fundamental basis of string diagrams exists in category theory because any set of sequential or parallel processes forms a monoidal category structure. The visual calculus of string diagrams utilizes this structure to provide an intuitive yet rigorous method for working with composed morphisms. The vertical connection of two diagrams through their wires represents the standard composition of their morphisms, and placing diagrams next to each other represents their monoidal tensor product, which enables parallel process execution [3][4].

2.3 Neural Networks

Neural networks are a class of machine learning models that are somewhat inspired by the brain and how it works. Neural networks consist of many neurons, which are simple computing units. Neurons are organized in layers and connected by weighted links. Neural networks can be trained, and through this training, these weights are adjusted so that the network can transform the inputs into useful outputs. Due to their layered structure and the huge number of parameters, modern neural networks have achieved human-level performance on tasks ranging from image recognition to language understanding [5].

A neural network is a function that converts inputs to outputs defined by a directed acyclic graph, where nodes are organized in layers and correspond to neurons. Edges carry the output of a neuron to another, associated with weights. Each neuron takes in one or more inputs, computes a weighted sum, applies a non-linear activation function, and passes the result onward to neurons in the next layer. By composing many of these nonlinear transformations, neural nets can approximate very intricate relationships in data.

Researchers have developed many neural network architectures tailored to different tasks and data. For instance:

- Feedforward Neural Networks (FNNs): Also called multilayer perceptrons, these are the most basic neural nets where information moves in a single direction from an input layer to one or more hidden layers and finally to an output layer. They have no loops or internal state and are suited for static input-output mapping problems [6].

- Convolutional Neural Networks (CNNs): A variant of the feedforward network for data with a grid-like structure (e.g., images). CNNs introduce convolutional layers with local receptive fields and weight sharing, as well as pooling layers to downsample. This architecture is especially suited for automatic hierarchical feature representation learning from raw image pixels [7].
- Recurrent Neural Networks (RNNs): Sequence-specific networks. RNNs differ from feedforward nets because they contain recurrent (feedback) connections, which enable them to store previous input information. RNNs use their output to depend on both current and past inputs, which makes them suitable for applications like language modeling and time-series prediction. The vanishing gradient problem is addressed through LSTMs (Long Short-Term Memory) and GRUs (Gated Recurrent Unit), which are specialized RNN architectures [8].
- Transformer Networks: Another, more recent architecture that has revolutionized the process of sequences. Transformers eliminate any recurrent structure and instead only use an attention mechanism to weigh relationships between every input sequence element. Transformers process data in parallel and can effectively use long-range dependencies. Transformers were initially released for translation tasks, using stacked encoder–decoder architecture and self-attention, and have now come to play the central role in NLP (Natural Language Processing) and beyond [9].

2.3.1 Transformer Neural Networks

The transformer is a deep learning model originally designed for sequence-to-sequence tasks in NLP (Natural Language Processing), but its architecture differs significantly from the earlier recurrent networks. A transformer model dispenses with recurrence altogether and uses self-attention mechanisms to allow each element of an input sequence to directly interact with all other elements. Practically, a transformer is built by stacking multiple layers, and each layer (often called a transformer block) contains two main sub-layers: a multi-head self-attention module and a position-wise feedforward module. Each block often also includes residual (skip) connections and layer normalization to make training easier, but essentially, the attention and feedforward sub-layers are the most crucial parts [9][10].

In a single transformer layer, the multi-head self-attention takes a sequence of input vectors and, for each position, computes a weighted sum over all position vectors (including itself). It's called "multi-head" because it's done in parallel many times (with different learned weights) in an attempt to catch different types of relationships, and the results are concatenated. This permits the model to pay attention, for instance, to words which are relevant to a certain word in a sentence, irrespective of how separated from

each other they may be. Since it's not confined to looking only at nearby timesteps (as is the case with RNNs), attention can detect long-distance context very effectively. After the attention sub-layer, the feedforward sub-layer simply applies a small fully-connected neural network to every position's vector (separately) to further project the representation. This is usually a two-layer MLP (Multi-Layer Perceptron) applied over each position, identical for all positions. The transformer architecture will typically handle input with a number of these stacked layers (the original Transformer had 6 layers both in the encoder and the decoder). Positional encodings are also added as inputs to give the model an idea of the sequence order, since raw self-attention is order-insensitive [9][10].

Overall, the distinguishing feature of transformers is that they perform sequence modeling through attention-based interactions rather than sequential recurrence. This structure does a great deal more parallelization at training (since each layer can observe the whole sequence at once) and has worked out amazingly well. Transformers now power most state-of-the-art language models (e.g. GPT) and even computer vision models (Vision Transformers), showing that the self-attention mechanism is a general way to encode complex dependencies in data [9][10].

2.4 String Diagrams and Category Theory in Neural Network Modeling

While neural networks are normally described in layers and weights, they may be described in more abstract terms, mathematically too. At a very general level, a neural network is a function composition: a function that has as an argument the output of the previous function, where the function is defined for each layer. Category theory, a mathematical investigation of abstract structures and how they compose, provides a formal vocabulary to talk about such composed systems. Category theory has in fact been referred to as a "lingua franca" of science across many disciplines precisely because it is able to speak the common compositional structure across different domains. For neural networks, category theory as a framework means we can think of each layer (or even each neuron) as being a morphism (an arrow) in a category, and the network as being the composite of these morphisms [11].

One of the category theory tools that proves especially handy in this case is the string diagram. A string diagram is a form of graphical calculus for composing morphisms within monoidal categories – basically, it's a picture with boxes (functions/operations) and wires (the flow of data) that satisfy specific mathematical rules. We can draw a neural network as a string diagram: a box for each neuron or layer, and the lines joining the layers are wires

joining the boxes (with wires combining into and branching out of boxes to indicate how outputs of one layer feed into multiple inputs of the next, etc.). This diagrammatic notation coincidentally fits with how neural networks operate. Applied category theory studies have already shown that yes, we actually can build string diagrams of neural networks with the use of neurons or layers as composable things [12][13].

The benefit of defining neural networks in such abstract terms is simplicity of concept and the hope for generalizability. When we view a network as a morphism within the right category (e.g., a vector space category or another algebraic structure category), it is easier to conceptualize its properties on an abstract plane. Category theory focuses on compositionally - the idea that complicated systems are built out of simpler components, exactly as neural networks are built (complicated functions out of simpler layer functions). String diagrams make compositional structure both concrete and formal. Briefly, diagrammatic modeling using string diagrams allows to model and analyze neural networks in a mathematically rigorous but intuitive graphical way. This category-theory-inspired approach has a unifying vision: different network topologies (feedforward, transformer, etc.) can each be embodied as particular compositions of morphisms in a category. Thus, category theory facilitates an interrealm transfer from the domain of neural network engineering and ad hoc mathematical computation to a more formal kind of algebraic reasoning on which the thesis "Diagrammatic Modeling of Neural Networks with Category Theory and String Diagrams" is based [1].

2.5 Project Context: The Ivaldi Application

Ivaldi is a string diagram construction and manipulation tool, which was originally created by Peeter Maran and Anton Osvald Kuusk at Taltech. Two student groups, one working on the frontend and the other on the backend, took Ivaldi to the next level as part of the Team Project course. The groups didn't stop after completing that course and continued to enhance the application in their respective thesis works. I came on board the project in fall 2024, attending team meetings to learn the codebase and design thinking. By the end of the Team Project timeframe, Ivaldi already supported:

- defining diagram elements (boxes) and connecting wires among them
- diagram inputs and outputs
- the development of domain-specific diagrammatic languages
- the generation of executable Python code directly from the string diagrams constructed.

For this reason, my thesis work will extend Ivaldi to allow users to graphically construct

neural-network modules into a diagram and subsequently export an entire trainable model as code.

3. Methodology

3.1 Ivaldi

The thesis work is built on the Ivaldi application (Ivaldi application background is discussed in Section 2.5). This section further discusses the Ivaldi application and what the author has been working on.

3.1.1 Overview

Ivaldi has been implemented as a string-diagrammatic modeling tool, allowing the construction of diagrams from boxes, wires, spiders, and connections. Both the frontend and the backend of the application are written in Python. At the outset of this thesis work, the codebase had already been extended by two teams (as mentioned in the Section 2.5), one responsible for the frontend and one for the backend, during their Team Project course. Because custom boxes could already be defined and code could already be generated from those predefined boxes, a foundation was in place for the development of a domain-specific language (DSL) tailored to neural network modeling (further discussed in Sections 3.1.5 and 3.1.4). Furthermore, the existing modular structure of the frontend and backend eased the task of layering new functionality on top of the application's core. However, both the backend and frontend were found to be imperfect and had to be modified before the implementation could function correctly (further discussed in Sections 3.1.2 and 3.1.3).

3.1.2 Backend

Key features that the Ivaldi application already had in the backend were the hypergraph construction functionality, initial code generation functionality, and the possibility to create predefined boxes. Boxes could also have a specified code snippet in them that is used in the code generation.

The backend functions that were mentioned work something like this. First, the drawn diagram is used to build a hypergraph whose nodes correspond to every box and spider on the diagram. Each node contains in itself exactly which wire is fed into it and out of it, and these are identified with IDs. Once the hypergraph is created and validated, the code generation engine looks up each box's associated snippet from the predefined boxes (see Section 3.1.5), and guided by the hypergraph structure, renames and puts those snippets

together into a single Python script. This ensures that in the generated code, the sequence of imports, function definitions, and calls is in the original diagram's topological order.

A domain-specific language was established in the backend through the implementation of a catalog of predefined boxes (see Section 3.1.5). In parallel, the existing code-generation pipeline, harvesting snippets from the predefined boxes, was improved. Although an infrastructure for extracting code from predefined boxes had already been in place, it lacked the support of some of the neural network layers whose parameters can be configured in the frontend (further discussed in Section 3.1.3). To fix this, functionality was added that replaces placeholder variables in all snippets, using default values whenever a parameter was not provided.

A more severe issue occurred with the introduction of nested subdiagrams: the generated script no longer had the correct dependency order. While code was properly generated for a flat diagram, subdiagram inclusion resulted in improperly ordered calls within the final main function. As a remedy, the routine that constructs the main function was re-implemented to enforce the proper topological ordering of imports, function definitions, and call sequences (see Section 3.1.4).

3.1.3 Frontend

On the frontend side, the Ivaldi application already had a user interface (UI), where you could access boxes and all the functionalities to create string diagrams. The key features were that you could use boxes and create custom boxes and also add inputs and outputs, and use wires to connect the boxes as you would like. There were also a possibility to create subdiagrams on the diagram so that you could add boxes inside of boxes for a visually more appealing view. The subdiagram structure is also visible in the UI and it also has a functionality to move easily between subdiagrams. So overall, you could create different string diagrams and manage boxes and connections as freely as you would like.

In the frontend, a possibility was created for some boxes to set specific variables like neurons in a layer, activation function, optimizer, or loss function. This was added because it gave a better overview of what the neural network looks like. This implementation was used in classical neural networks. Also, there was added information for every box that can be seen through the UI by clicking on the box. This information was specific to each box to show overall information about the box and specific parameters for each box as well. For better understanding there are examples in the Appendix 2.

3.1.4 Code Generation

In the back-end, the generation of code is managed by the CodeGenerator, which directs three main steps:

1. Construction of a hypergraph. The user's diagram is first converted into a Hypergraph. Each node of the resulting hypergraph represents a box (or spider) and holds its input/output wiring and frontend-configured parameters.
2. Retrieval of snippets and parameter substitution. For each hypergraph node, topologically ordered by a topological sort to maintain data dependencies, CodeGenerator retrieves the corresponding code template from the predefined boxes directory. For some custom boxes, placeholder variables in snippets are then substituted either with values set interactively in the frontend or with reasonable defaults if no value was given.
3. Assembly of the final script. Once all snippets have been customized, they are concatenated into one Python source string. The generator ensures that import statements, function definitions, and layer-invocation calls occur in the correct order, mirroring the diagram structure, and wraps them in a top-level function. This compiled script is then emitted for execution or download by the frontend.

This pipeline cleanly separates diagram interpretation (via the hypergraph), snippet management, and final code templating, allowing new layer types or parameter conventions to be easily added. It should also be noted that an invoke function must be included in every code snippet derived from a predefined box (further discussed in Section 3.1.5), since the code generator is designed such that these invoke functions correspond to the box's inputs and outputs.

When the diagram was flat the code generation worked, but when the sub diagrams were added, the code generation broke (see Section 3.1.2). Sub diagrams in the Ivaldi application work like that you could select some boxes that you want to create a sub diagram, and than they are put inside a new subdiagram box. When you click on the sub diagram box it opens up the inside diagram of that box, and it consist of the boxes that you selected and created a sub diagram. Inside the subdiagram box you can add boxes and create string diagrams like you would normally create them. Also, you could create subdiagrams inside subdiagrams, so you could organize your string diagram, and the structure of the subdiagrams and main diagram can also be seen in the frontend, so it is easy to navigate between subdiagrams. But when generating code from the diagram, it did not create the correct main function, specifically, the invoke functions in the final main function were in the wrong order, and also had the wrong inputs given to them.

This issue was resolved through the application of recursion and an auxiliary function to create each subdiagram's list of layer calls before constructing the top-level main function. The function generating the main function now looks over all hypergraph nodes, and upon encountering a subdiagram, it recursively calls itself to create that subdiagram's code and makes a suitable invoke call for the main function before proceeding to the next node. Invoke methods are also properly renamed, and all the inputs are correctly passed.

This recursive, helper-based approach is particularly suited to the code-generation problem because it mirrors the very structure of nested diagrams: each subdiagram is treated as a standalone unit whose code must be defined before invoking it. The use of recursion on each subdiagram has the effect of automatically enforcing a definition before use discipline: Helper procedures for subdiagrams always appear before any code that invokes them, so there cannot be any circular dependencies. This decomposition also makes the implementation modular: the same logic that generates code for a flat diagram is reused to process each nested layer, without duplicated logic, and it's simple to add support for deeper or more complex nesting. Finally, since each recursive call is on a smaller hypergraph, the overall algorithm is as simple to understand conceptually as possible, but it guarantees that the final main function gets the original diagram's topology correctly.

3.1.5 Predefined Boxes

The domain-specific language (DSL) relies on predefined boxes as its fundamental building blocks. A DSL represents a lightweight programming language that specializes in expressing solutions and concepts for specific problem domains while giving up general-purpose language capabilities for more domain-specific notation and abstractions [14]. The framework includes diagrammatic elements that users use to visually construct neural-network architectures through boxes, wires, spiders, and connections. Each predefined box in this DSL must therefore include:

1. A well-defined interface, explicitly declared input and output ports.
2. An invoke function, a callable whose signature aligns with those ports and encapsulates the box's computation.

And some boxes also have configurable parameters and placeholders that the frontend can override or fall back on defaults when unset.

By enforcing this template, ports plus a dedicated invoke method, new box types can be introduced simply by supplying a corresponding code snippet. During generation,

the backend uniformly retrieves each template, substitutes any configured parameters, and emits a call to its invoke function in dependency-correct order. This design not only makes the DSL highly expressive for end users but also keeps the code-generation logic clean, modular, and extendable.

The neural network layer boxes were implemented in PyTorch because it provides excellent, fast neural network prototyping capabilities while maintaining high performance. The Python-native imperative programming style of PyTorch enables model construction that reads and debugs like regular Python code, thus reducing the cognitive gap between specification and execution [15][16]. The deployment of PyTorch models is supported by TorchScript, which converts eager-mode code into optimized static graphs to achieve both prototyping agility and deployment efficiency. The two-phase workflow enables developers to achieve the best of both worlds by using eager mode for interactive prototyping, followed by selective compilation of performance-critical paths. Dynamic tensor rematerialization in PyTorch extends memory flexibility by dynamically recomputing intermediate activations, which enables training of larger or deeper models on limited GPU memory. The rapid succession of new features and optimizations in PyTorch is ensured by its tight connection to the Python scientific ecosystem (NumPy, SciPy, matplotlib) and its research-focused community [15][16][17][18][19][20].

Predefined boxes were first created for feedforward neural networks, as that architecture represented the simplest case (further discussed in Section 3.2.1). Layer boxes were defined and filled with their corresponding code snippets so that code generation would function correctly. For these feedforward boxes, a straightforward, linear composition of layer snippets was sufficient. The code for each layer was generated sequentially, yielding an entire dense feedforward neural network model. Configurable parameters were also added to these feedforward boxes via the frontend to make constructing feedforward networks as straightforward as possible.

After the feedforward network boxes were completed, the task of specifying transformer network boxes was then taken up. The transformer model was decomposed into layer-specific boxes and corresponding code snippets, a delicate and non-trivial task (further discussed in Section 3.3). Several decomposition methods were attempted to ensure that the resulting diagram remained comprehensible and well-structured and not a web of wires. Adding each layer's forward method introduced more complexity, and considerable time was spent on encoding the DSL representation for such units (see Section 3.3.3).

3.2 Classical Neural Network Architecture Modeling

In this section, three main neural network architectures are examined- feedforward networks, convolutional networks, and recurrent networks- and discuss how their structures can be represented in a modeling context. Describe the architecture of each type, then outline how these models can be split into components and represented as visual boxes in a domain-specific diagramming language. This approach highlights the key building blocks of each architecture and sets the stage for a visual DSL (Domain-Specific Language) that can represent complex neural networks in a modular way.

3.2.1 Feedforward Neural Networks

Feedforward neural networks (or deep feedforward networks or multi-layer perceptrons) are the simplest and "classic" neural network structure. In a feedforward network, information travels in one direction only: from the input layer, through any hidden layers, to the output layer, without cycles or feedback loops. Because feedback links are absent, the output from a layer does not influence the same layer or previous layers. If they were added, technically, the network would be a recurrent network. Feedforward networks are the core of deep learning and give the foundation for more complex designs (e.g., convolutional networks are a specialized form of feedforward networks) [21][22][23].

Architecture

A feedforward neural network is organized in distinct layers of nodes (neurons) ordered sequentially: an input layer, one or more hidden layers, and an output layer. A layer contains a specific number of neurons, and typically each neuron in a layer is connected to all neurons in the next layer, which is a fully connected layer, also known as a dense layer [21][24]. The layers and their operations can be characterized as follows:

1. Input Layer: Accepts the input raw features. The input neurons are as many as the dimension of the input data (e.g., number of features). It performs no computation, it just passes the input values to the next layer [21][24].
2. Hidden Layer(s): These layers transform the input data into higher-level abstract representations. One or more hidden layers can exist. Each neuron in a hidden layer computes a weighted sum of the previous layer's outputs, adds a bias, and applies a nonlinear activation function. Fully connected hidden layers (dense hidden layers) allow the network to learn complex functions by repeatedly extracting and combining features of the data [21][24].
3. Output Layer: The final layer that produces the network output. Output neurons, like

hidden neurons, calculate a weighted sum of the activations of the previous layer (or the input if there are no hidden layers) and apply an activation function. The number of output neurons is determined by the task, for example, one output for a regression prediction, or one output per class for classification [21][24].

In a forward pass, information travels from the input layer to the hidden layer(s) and then to the output layer. The weights and biases of the individual layers make up the learned parameters during training (through mechanisms like backpropagation). The depth of a feedforward network (number of layers) is flexible, those with multiple hidden layers are referred to as deep networks, capable of learning increasingly abstract representations at each subsequent layer. The network size for each task or application is also fairly important, and choosing the appropriate size determines a lot in how the network will perform and behave [25]. Importantly, the feedforward structure may be considered to be a cascade of functional mapping, and that way of thought will come to mind when determining how to separate the model into components for diagrammatic modeling [21][24].

3.2.2 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a special type of feedforward network that is designed to process data with a grid-like topology, e.g., image or audio spectrogram, or natural language processing. In a CNN, the neurons are also arranged in multiple layers, but the layers are designed in certain ways to exploit spatial locality and hierarchical feature extraction. Rather than complete connections between layers as in a standard MLP, convolutional networks introduce two ideas: local receptive fields and weight sharing, typically combined with a form of downsampling called pooling. These architectural ideas allow CNNs to achieve more shift and distortion invariance for image classification problems with far fewer parameters than fully connected networks [26].

Architecture

A CNN architecture typically includes a sequence of convolution and pooling layers that transform an input volume (e.g., an image) to an output volume (e.g., class scores). The architectures of CNNs have a significant influence on the design of neural network designs, as a more sensible network design can improve the fitting effect between layers or remove duplicate computation within the network, which in most cases suggests that it can bring more superior performance [27]. The standard layers in a convolutional network include:

1. Convolutional Layers: These layers are the fundamental components of a CNN. Every convolutional layer applies multiple learnable filters (kernels) in a sliding

window manner over the height and width of the input volume, computing feature maps. A neuron in a convolutional layer, as opposed to a fully connected layer, is connected only to a local area of the input (its local receptive field) rather than the entire inputs. The same filter (set of weights) is applied at every position of the input, i.e., weights are shared across spatial positions. This weight sharing produces feature maps—each map highlights where in the input a particular feature (pattern) is available. Local connectivity and weight sharing reduce the number of parameters dramatically and force the learned features to be shift-invariant, i.e., the network learns to detect a feature anywhere in the input. The linear convolutional operation is followed by a non-linear activation (such as ReLU) for every feature map, introducing the necessary non-linearity [27][28].

2. Pooling Layers: A pooling (subsampling) layer, or the down-sampling layer, follows some of the convolutional layers to reduce the spatial resolution of feature maps. Pooling is applied to small regions of the feature map and reduces values. Common operations are max pooling (taking the maximum) or average pooling (taking the mean) in a region. With downsampling, pooling layers accomplish two things: they reduce the number of computations and parameters in subsequent layers, and prevent overfitting of the network. This holds the network focused on the most important features. Pooling itself has no learnable parameters, it is a fixed function [27][28].
3. Fully Connected Layers: Often, the high-level feature maps are flattened to a vector and fed into one or multiple fully connected layers (like a feedforward network). These fully connected layers tie the features together to produce the final predictions. Pooling and convolutional layers are a set of automatic feature extractors, and the final fully connected layers are a classifier based on those features [27][28].

A convolutional neural network works by transforming input through alternating cycles of convolution (feature extraction) and pooling (feature consolidation) until it eventually maps onto an output layer. Lower layers of a CNN might learn to recognize simple features like edges or textures, and higher layers combine those together into higher-order abstractions (object parts, then objects). This multi-layered, hierarchical structure is what provides CNNs with their power for vision and other uses, each layer thereafter takes the output from the previous layer and picks up increasingly complicated patterns in the data. As with other neural networks, a CNN can be viewed as a directed acyclic graph of computations [26][27][28].

3.2.3 Recurrent Neural Networks

Recurrent neural networks (RNNs) are a form of neural network designed to handle sequential data. Unlike feedforward networks, recurrent networks have feedback loops,

allowing information to be passed from one step of the sequence to the next. In an RNN, the units (or neurons) form directed cycles over time: the network's output at a particular time step is fed back into the network and added to the next input. This gives RNNs a form of memory, enabling them to encode temporal dynamics and long-distance dependencies in sequences. RNNs are extensively used in applications such as language modeling, where word order matters, or time-series prediction, where future values depend on past context [29].

Architecture

At the heart of an RNN is the notion of a hidden state, updated at each time step within the sequence. The hidden state is thus a compaction of past inputs, feeding information down. Each time step applies the same weights (the network is recurrent, employing the same parameters at each step). The recurrent layout can be expressed recursively. But to gain a better feel for it, one can unwind the recurrence: for a sequence, the RNN can be imagined as layers of a network in a chain, with each layer passing its hidden state on to the next. What makes an RNN different from a plain feed-forward network is its recurrent connections. If you unroll it over time without any truncation, you end up with a network as deep as your sequence is long, potentially infinitely deep for an unbounded sequence. In practice, it is trained using backpropagation through time, which treats the unrolled network as a deep network with layers [29][30]. The primary features of the RNN architecture are the following:

1. Hidden State (Memory): RNN retains a vector to store information of past inputs. The hidden state is obtained in every step. This enables the network to remember earlier context in the sequence [29].
2. Recurrent Connections: The previous hidden state output at time "t-1" is presented as input to the network at time "t". These feedback connections allow sequential context to influence the present output of the network [29].
3. Input and Output Layers: Aside from the repetitive hidden units, an RNN has input and output links. For example, a text-processed RNN would get one word (or character) per time step as input and produce an output each step (like the probability of the next word) or just at the final step (for tasks like sequence classification). The individual configuration (one-to-one, one-to-many, many-to-many, etc.) will vary based on the application, but the overall cyclic architecture remains the same [29][30].

In general, the structure of an RNN is defined by its repeating module (the update of the hidden state and the generation of the outputs). Simpler RNNs contain a single tanh

or ReLU layer as the recurrent function (usually known as "Simple RNN"), while more advanced ones like LSTM and GRU have internal gating mechanisms to further control information flow. Regardless of the variant, all RNNs have the feature of loops in their computation graph, allowing for some form of memory across time steps. This is why they are applied to sequential modeling [29][30].

3.2.4 Splitting

Splitting in this chapter refers to breaking down the model's architectures into a linear sequence of reusable building blocks. Because feedforward and convolutional neural networks are inherently sequential and their computations go from layer to layer in one chain, so that each layer's output becomes the next layer's input. It was decided to split these networks by layers so that they could be added sequentially, and after all the layers were added, the model could be compiled sequentially as well, exactly as depicted in the diagram.

Recurrent networks in their structure introduce feedback loops across time, each time step's hidden state feeds back into the same layer at the next step. But most deep learning frameworks encapsulate this recurrence inside a single RNN-layer abstraction, and PyTorch is used in this thesis project, which also uses abstractions that wrap the entire time-step recursion inside one layer class. So RNN layers can also be added sequentially, and layers take one input tensor and produce one output tensor. This means that in the diagram, the layers can be added sequentially, and the model can also be compiled in linear order.

Feedforward Neural Network

So, because the feedforward neural network could be created sequentially by adding layers next to each other, firstly, to support arbitrary straight-through fully connected architectures, the implementation defines three core layers. These layers are the input layer, the hidden layer, and the output layer, and all are fully connected layers (dense layers). With these layers it is possible to create essentially any classic feedforward topology of fully connected layers with arbitrary activation functions. So with this split up, it is possible to chain any number of dense linear layers and activation functions, and these layers should be between one input and one output layer.

But real-world feedforward neural networks often include more than just Linear layers with activation functions. In practice, these networks frequently have some regularization and normalization layers alongside the dense layers. To account for this, the design should also include dropout and batch-normalization layers, which can be used after hidden layers.

With these additions, the split stack that I have, includes input layers, hidden layers, output layers, dropout, and batch-normalization. With all these can be created a wide variety of real-world FFNs suitable for regression, classification, or other tasks or applications.

Despite this added flexibility, it still can't create certain feedforward networks that are not purely sequential, like there could be added skip or residual connections, but adding these requires a non-linear graph structure rather than a simple chain of layers. Also, there could be added custom weight initialization, parameter tying, or shared layers that introduce dependencies that cannot be expressed with the vanilla Sequential API. So these are things in this application that can be developed further and represent a promising direction for future extensions. These patterns were not used in this thesis to preserve the simplicity and clarity of the sequential model. While parameter tying or custom initialization can be useful in certain FFN designs, they are not essential and therefore stayed out from the current project scope.

Convolutional Neural Network

The convolutional neural networks can also be created sequentially, the same way as feedforward neural networks, and so the split up of the CNN was similar to the FFN split up, but some other layers were introduced. The convolutional neural network was split up like this that there was an input layer, conv layers, pooling layers, 2D dropout, flatten, dense layers, 1D dropout (regular dropout), and output layer. With all these layers can be created many different convolutional neural networks for different tasks. For most basic example with these can be created a model for image classification that can be trained on the CIFAR-10 dataset.

In addition to these layers that were implemented, the split up can be developed further, for example by adding residual blocks or custom layers. Adding these can also be useful for some CNN projects or tasks, but these are not essential, and to preserve the clarity of a strictly sequential API, these were not implemented in this thesis. But for further development of this application in the future, these additions may be useful for some and are promising avenues for extension.

Recurrent Neural Network

As mentioned earlier, the recurrent neural networks could also be decomposed into a linear sequence of reusable layer types, and the decision was made to split up the RNN into these layers: input layer, simple RNN layer, LSTM layer, GRU layer, dropout, and output layer. So by adding simple RNN layers, LSTM layers, and GRU layers, this split enables to creation of many different RNN architectures for different real-world tasks or applications.

These layers could also be mixed in the model's architecture, enabling the creation of nearly all standard RNN architectures available in PyTorch.

There could also be done some more development so that the RNNs could cover more kinds of RNN architectures. For example there could be added custom RNN cells or residual connections between RNN layers. These concepts were postponed for future development because they require definitions outside of the vanilla Sequential API. Also, these additions are not so important because they are used in more specific RNN architectures. But implementing these in the future may be a useful avenue to extend the RNN architecture features.

One more thing to mention in RNN architecture is that, as mentioned earlier, the layers can be added sequentially and the PyTorch framework encapsulates the recurrents, but in further development of this application, there could be developed some kind of solution to show these recurrents. But for my project, this wasn't essential, and the diagram itself does not yet support loops, and implementing that loops are supported for this string diagram application (Ivaldi) is itself a big project, but certainly a very good extra addition to this project.

3.2.5 Creating Boxes

As mentioned in the previous section (see Section 3.2.4), all the networks are split up this way that the layers could be added sequentially, and the creation of boxes relies on this linearity. Every layer is represented in the diagram as a distinct box that can be placed, configured, moved, and connected in a single chain of layers. Users can use these boxes to visually assemble a model layer at a time, and the underlying DSL will generate the code for the model, mirroring the exact order and parameters chosen on the diagram.

Each box encapsulates a set of parameters to instantiate its corresponding PyTorch module. These parameters may be the number of neurons in a dense layer, the kernel size of a convolution, or a float value of the dropout. These parameters can be set on the diagram by clicking on the box and selecting Edit Properties, which pops up a little window where the specific parameters for this selected box can be set. This implementation of setting parameters in this manner was chosen because the alternative, adding parameters through the main function's arguments, proved unwieldy. As models grow in size, the wiring becomes progressively more confusing and complicated. Each additional wire requires a corresponding input in the main function, and keeping track of their sequence can become unclear. That complexity makes it all too easy to introduce errors. Parameters for each box are also saved when the project is saved, so that when reloading a project, the set parameters are already present and do not need to start over to set the correct parameters for the project.

Because all network types, feedforward, convolutional, and recurrent, follow the same single-input, single-output contract, the box creation workflow does not need to account for branching or loops. The single input that is given to the main function is the input dimension, and the single output is the compiled model. This works the same with all the models mentioned in this section, so from the generated code, the model will be returned by calling the main function with the input of the model dimension. In the following sub sections are specific box implementations for feedforward, convolutional, and recurrent neural networks.

Feedforward Neural Network

For the feedforward network five different boxes were created that can be used to model a wide range of fully connected architectures and generate code from the diagram:

- Input Layer Box:
  - Inputs: Number of input features (an integer) and this is set by calling the main function and giving it one input witch is the input dimension.
  - Purpose: This box defines the dimensionality of the data entering the network. Every model must begin with this input layer box so that other layers know how many features to expect. Also this input layer contains the main builder code inside of it so that the next added layers will be compiled into code correctly.
- Hidden Layer Box:
  - Inputs: Number of neurons (integer) and activation function (e.g., ReLU, Sigmoid), and these are set by using the application's UI by selecting the box and editing properties.
  - Purpose: This box adds a fully connected layer to the model that transforms its inputs into a higher or a lower dimensional space and then applies a nonlinearity. These boxes can be chained to increase the model's depth and capacity to fit a specific task or application in the real world.
- Dropout Box:
  - Inputs: Dropout probability (float between 0.0 and 1.0), and is also added through the application's UI.
  - Purpose: Dropout box randomly zeroes a fraction of the hidden activations at training time to reduce overfitting and improve generalization. This box is usually placed right after a hidden layer.
- Batch Normalization Box:
  - Inputs: There are no inputs for this box because the layer's dimension is inferred from the previous layer.

  - Purpose: This batch normalization box normalizes and rescales the activations on each batch to stabilize and accelerate the training process. This box is also most often applied after a hidden layer.
- Output Layer Box:
  - Inputs: Number of output units (integer) and optional activation (e.g. Softmax for classification or None to emit raw logits).
  - Purpose: This layer produces the final predictions or scores. Every network must end with a single output layer box, witch compiles the previous transformations into the desired output format. Also, the output layer compiles all the layers that were added together and returns the complete model.

Convolutional Neural Network

For the convolutional network, eight different boxes were created, which could be used to build everything from simple feature extractors to deeper image classifiers and generate code from these built models:

- Input Layer Box:
  - Inputs: Number of channels, image height, image width (three integers) in a single tuple in this order. This tuple can be given to the main function as its input.
  - Purpose: This box establishes the shape of the incoming image tensor so that subsequent layers can be configured correctly. Also this input layer box contains the main builder code inside it so that subsequent layers will be added and compiled into code correctly.
- Convolutional Layer Box:
  - Inputs: Number of output channels (integer), kernel size (integer or pair), stride (integer or pair), padding (integer or pair), and activation. All these inputs could be set using the edit properties pop-up window in the UI.
  - Purpose: This layer applies a bank of learnable filters to the input, producing feature maps that highlight local patterns, and activation adds nonlinearity right after. These boxes could be chained up to build deeper models and capture more complex features.
- Pooling Box:
  - Inputs: Kernel size (int or pair), stride (int or pair), and pool type (max or average), and added trough UI as well.
  - Purpose: This box reduces spatial dimensions and elevates translational invariance by pooling each region of feature maps. These boxes are placed between the convolutional layer boxes.

- 2D Dropout Box:
  - Inputs: Dropout probability (float), this can also be set using edit properties.
  - Purpose: This box randomly zeros whole feature maps (channels) during training to avoid coadaptations and increase robustness. This box is typically added after convolutional or pooling layers.
- Flattening Box:
  - Inputs: None, this box automatically takes the total feature dimension from the previous layer.
  - Purpose: Flattening box converts the multidimensional tensor of feature maps into a single vector. This enables to go form convolutional layers to fully connected (dense) layers.
- Dense Layer Box:
  - Inputs: Number of neurons (integer) and activation, witch can be set in the UI the same was as other boxes inputs.
  - Purpose: This box acts as a standard fully connected layer after the flattening box. This aggregates high level features into final representations.
- Dropout Box:
  - Inputs: Dropout probability (float), this can also be set using edit properties.
  - Purpose: This 1D dropout box applies regular dropout in the fully connected portion fo the network, reducing overfitting even more.
- Output Layer Box:
  - Inputs: Number of output units and optional activation, which can be set using the UI functionality.
  - Purpose: This box outputs the final class scores or regression outputs and concludes the convolutional pipeline. This box should be the last box in the model, and this also compiles the model together with all the layers that were added on the diagram.

Recurrent Neural Network

For the recurrent network, six boxes were created, which could be used to model almost all standard RNN architectures and generate code from the diagram:

- Input Layer Box:
  - Inputs: Feature dimension (integer), batch first (boolean), sequence to sequence (boolean). The feature dimension is set like in previous networks by gaining the integer to input for the main function, but the batch first and sequence to sequence booleans are set using the application UI edit properties pop-up window.

  - Purpose: This box establishes how the data will flow into the recurrent stack. Input box defines the input vector size per step, the tensor layout convention, and whether to treat the network as a sequence-to-one or sequence-to-sequence. This box also has the main builder inside it, so that the following recurrent stack will be added in the correct order and compiled into code correctly.
- Simple RNN Layer Box:
  - Inputs: Hidden dimension (integer), number of layers (integer), nonlinearity ('tanh' or 'relu'), bidirectionality (boolean), internal dropout (float), all of these can be set by using the UI.
  - Purpose: Simple RNN layer box adds vanilla RNN layer to the model. This layer box is useful for basic sequence-to-sequence tasks or lightweight time series models. These boxes could be stacked to create a bigger recurrent layers stack to increase the depth of the model.
- LSTM Layer Box:
  - Inputs: Hidden dimension (integer), number of layers (integer), bidirectionality (boolean), internal dropout (float), like others, these inputs could also be added through UI.
  - Purpose: This box can be used to add Long Short-Term Memory layers to the model, which maintains separate cell and hidden states to capture longer temporal dependencies. Like simple RNN boxes LSTM boxes could also be stacked and used to create bigger models.
- GRU Layer Box:
  - Inputs: Hidden dimension (integer), number of layers (integer), bidirectionality (boolean), internal dropout (float), also could be added by using UI functionality.
  - Purpose: The GRU layer box can be used to add Gated Recurrent Unit layers, which are a simpler alternative to LSTM layers that often perform similarly with fewer parameters. These layers could also be stacked, and all the RNN layers, like simple RNN layer, LSTM layer, and GRU layer, can be mixed in the recurrent stack.
- Dropout Box:
  - Inputs: Dropout probability (float), this can also be set using edit properties in the application UI.
  - Purpose: This dropout box applies dropout directly to the RNN output activations, after any internal recurrence, so that units are disabled randomly during training. This dropout can be used even when in the model there is a single RNN layer, where the built-in internal dropout argument would not work, and ensures consistent regularization before any other projection or time-step extraction.

- Output Layer Box:
  - Inputs: Number of output units and optional activation, which could be added via edit properties in the UI.
  - Purpose: This output box projects the final hidden representation, or the last time-step in sequence to sequence mode, to the desired output space. This box also compiles the model with all the recurrent and dropout layers that were added to the diagram.

3.3 Transformer Neural Network Modeling

In this section, the transformer architecture is examined, and it is discussed how its structure can be represented in a modeling context. Describe the architecture of transformer model and outline how could this model can be split up into components or layers and how they can be represented as visual boxes in a domain-specific diagramming language. This approach brings out the key building blocks of transformer architecture and explains how these components can be used to model transformers on a diagram. Also unfold how the DSL (Domain-Specific Language) is used to generate the code directly from the diagram representation of the model.

3.3.1 Transformer Neural Networks

Transformers are a class of deep neural networks introduced first in the paper "Attention Is All You Need" in 2017. The Transformer models have transformed sequence modeling tasks in natural language processing and beyond. The transformer model introduced attention mechanisms as its core innovation instead of using recurrence or convolution. The authors presented an attention-based model architecture in their paper, which eliminated both recurrence and convolution. The model produced better translation results and required less training time than recurrent models. The processing mechanism of transformers differs from RNNs because transformers handle all positions simultaneously instead of sequential token processing. The parallel processing method enhances training efficiency on modern hardware [9].

Origins

Before the transformer architecture were introduced recurrent neural networks (especially LSTMs and GRUs) were the state-of-the-art for sequence tasks, like translation and language modeling tasks. RNN-based sequence-to-sequence models read input tokens sequentially and maintain a hidden state, and this limits their ability to compute steps in parallel. The sequential dependency made RNNs slow to train on long sequences, and also,

there were difficulties in capturing long-range dependencies, because issues arose, like the vanishing gradient. The transformer model architecture idea came from the insight that attention could be used to directly model dependencies between any two tokens in a sequence. This allowed the dependencies to be modeled regardless of the tokens distance, and it did not rely on a chain of hidden states anymore. The attention mechanism itself had already been introduced for some sequence-to-sequence RNN models to help the decoder attend to relevant input words, but the transformer was the first model architecture to use attention alone as the core part of the network, which eliminated the recurrence [9].

Innovations

Transformers introduce self-attention as the core operation; self-attention lets each element of a sequence attend to every other element. This allows the model to capture long-range relations like RNN, but the difference is that this can be done in one layer in the transformer opposed to step-by-step propagation of information in the RNN. The transformer also uses multi-head attention, witch means that the attention mechanism is replicated in parallel multiple times (usually referred to as heads) with independent weight matrices. Multi-head attention allows the transformer model to attend to different types of relationships or features simultaneously, and after this, it can combine the information together. Another very important feature that transformers use is that they process sequences with positional encodings, which allows the inputs to retain order information, unlike an RNN witch has no inherent notion of sequence order. By combining all of these ideas -self-attention, multi-headed parallel attention, and positional encoding- transformers are able to model sequences holistically, each layer processes the whole sequence context simultaneously, and therefore they are very powerful at capturing global dependencies [9].

Parallelism and Performance

Because transformers have parallelization, they do not have to operate sequentially and can make computations much more effectively. All elements or tokens in a sequence can be processed in parallel during training and inference. But inference still generates outputs one step at a time, each step can use parallel matrix operations internally. This design had a massive advantage in terms of scalability. In the "Attention Is All You Need" paper, it was mentioned that their transformer reached state-of-the-art translation quality in a fraction of the training time compared to the recurrent models. The transformers multi-head attention is also very well suited to GPU (Graphics Processing Unit) acceleration, witch is an important factor in its widespread use in large neural networks. But in the transformer model the self-attention has quadratic complexity in sequence length, which means that it compares every pair of positions, but in practice, the ability to parallelize and the improved modeling of long-range patterns have proved useful nonetheless and

have made transformers far more effective on big data. Since 2018, transformers have become the most dominant architecture for natural language processing (NLP), and have created breakthroughs in tasks like translation, question answering, text generation, and more. Parallelization came out to be so powerful that recurrence was not necessary for high-quality sequence modeling, but it could be achieved using only attention mechanisms [9][31][32].

In summary, transformer neural networks does not use recurrences and are attention-centric, and they are used for sequence modeling. Transformers differ from RNNs by processing sequences in parallel and using learned attention weights to decide witch tokens influence each other. These innovations allow transformers to be more effective than RNNs in maintaining context over a long sequence. Transformers have led to state-of-the-art results across a wide range of language tasks, and most of the modern large-scale language models (LLMs) and translation systems are built on transformer architectures, which place the transformer as the central model in today's deep learning.

3.3.2 Transformer Architecture

The Figure 1 illustrates the standard transformer architecture as originally proposed in the "Attention Is All You Need" in 2017. This model follows an encoder-decoder structure,

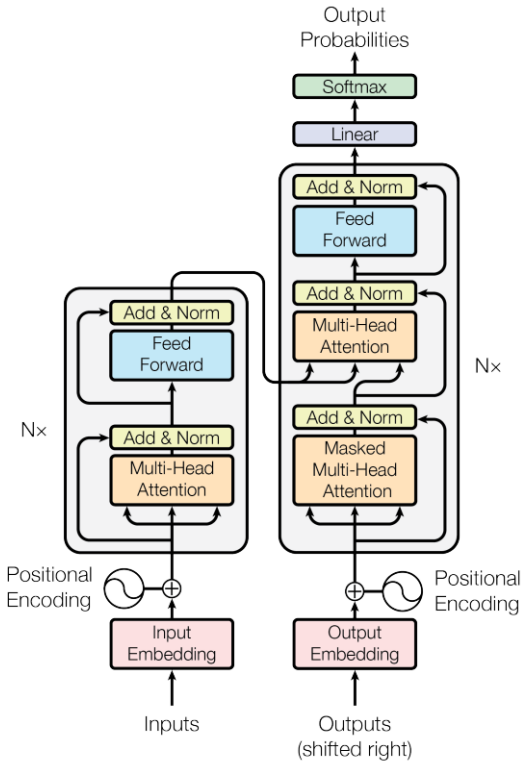


Figure 1. *The Transformer Model Architecture [9]*

which is common in many sequence-to-sequence models. The encoder (on the left in the Figure 1) is a stack of layers that reads the input sequence and maps it as a sequence of continuous vector representations. The decoder (on the right in the Figure 1) is another stack of layers that receives the output of the encoder and generates an output sequence one token at a time by using the encoder's representations as context. As seen in the Figure 1, the transformer does not use any recurrent connections; instead, both the encoder and decoder are built entirely from attention layers and feedforward layers. This means that the information flows trough the attention mechanism instead of sequential hidden states [9].

Encoder-Decoder Structure

In the full transformer model, the encoder and decoder both consist of multiple identical layers. In the original design, there were 6 layers for each. Each encoder layer has two main sub-layers: a multi-head self-attention layer and a position-wise feedforward layer. The self-attention sublayer allows each position in the input to attend to all other positions and thus see all of them, enabling the encoder to produce context-aware representations of each token. After the self-attention layer comes the feedforward sub-layer, which is a simple fully-connected network applied independently to each position's vector. This transforms the representation further at that position. Also in this structure, there are residual skip connections, and layer normalization is applied around each sub-layer to stabilize training, but the core idea still remains that each encoder layer mixes information across the sequence using attention and then refines it using a feedforward network. The output of the final encoder layer is a series of contextualized embeddings for every input token, and these encode the information of the whole input sequence [9][32].

The decoder is designed to generate an output sequence (e.g., translation or text generation) from the encoder embeddings given to the decoder. Each decoder layer similarly has a multi-head self-attention sub-layer and a feedforward sub-layer, but it has a third sub-layer between that differs from the encoder. This layer is a multi-head cross-attention layer, also called encoder-decoder attention, and this allows the decoder to attend to the encoder's output vectors. In the decoder's self-attention a masking mechanism is used to preserve the generation of tokens one-by-one. At decoding time, each position can only attend to earlier positions in the output sequence and not future ones that have not been generated yet. This is usually implemented by masking out future tokens positions in the attention computations, and this enables the model to generate tokens one-by-one in order using only past context. During the training process the output sequence is shifted and masked to simulate the step-by-step generation of tokens. The cross-attention sub-layer then allows the decoder to look at the encoder's output. In each timestep the decoder can attend to all positions of the encoded input sequence, which enables it to learn alignment between

input and output tokens. By stacking several decoder layers on top of each other (like in the original architecture, there were six), the model builds up the output representation, with each layer is able to refine the output both with the previously generated tokens using masked self-attention, and with context from the input using encoder cross-attention. Finally, a linear projection and softmax layer (see Figure 1 at the top gray and green layers) is used on the top decoder output to produce the next output token probability at each step [9][32].

In summary, the standard transformer architecture that uses an encoder-decoder structure uses the encoder to transform the input sequence into an abstract representation, and after that, the decoder uses that representation to generate the output sequence step by step. Both the encoder and decoder rely on multi-head self-attention and feedforward layers as the basic building blocks. This architecture was initially designed for semi-supervised tasks like machine translation, but it is also the template from which other transformer-based models are derived.

Variants of the Transformer Architecture

Most of the best models are essentially streamlined versions of the full encoder-decoder transformer, for example, using only the encoder part or only the decoder part to suit different tasks. Therefore there are three different transformer model configurations based on what they consist of:

- Encoder–Decoder Transformers (full model): This is the original architecture (further discussed in Section 3.3.2), featuring both the encoder and decoder. This architecture is usually used for sequence-to-sequence tasks where an input sequence is transformed into a different output sequence, for the classic example, machine translation being one of those tasks. This encoder-decoder setup allows the model to encode a source sequence into a rich representation and then decode that into a target sequence, paying attention to the source as needed. The majority of the translation models and text summarization models use this encoder-decoder architecture.
- Encoder-Only Transformers: These models consist of the transformer encoder stack without a decoder. Encoder-only transformer models are well-suited for tasks that involve understanding or classifying a single sequence, rather than generating a new sequence. A very good example is BERT (Bidirectional Encoder Representations from Transformers), which is an encoder-only model [33]. In an encoder-only model, there is no separate decoder module, therefore, input tokens attend to each other bidirectionally within each self-attention layer. Because no output needs to be generated autoregressively, self-attention in encoder-only models may utilize

all tokens on both the left and the right context freely, and this is often called bidirectional attention. For example, BERT's architecture is essentially a stack of encoder layers that process text bidirectionally, which enables the model to capture context from past and future words in a sentence. Such encoder-only models are used for language understanding tasks, typically after pre-training on large corpora (e.g., with masked word prediction), the encoder output may be fine-tuned on such tasks as classification, named entity recognition, or question answering. These encoder-only models like BERT have improved the NLP benchmarks, and they are now a ubiquitous baseline for many NLP tasks [33][34].

- Decoder-Only Transformers: These models consist of the transformer decoder stack without an encoder. Decoder-only transformers are used for generative tasks, which means that the goal is to produce an output sequence when given an input prompt. In this configuration, the model operates like a stand-alone language model; it uses masked self-attention to make sure that each position can only see earlier positions, and it predicts the next token in a sequence based on the previous tokens. The GPT (Generative Pre-trained Transformer) family of models by OpenAI is the best example of decoder-only transformers [35]. GPT models are essentially stacks of decoder layers, and they do not use a separate encoder module; they are trained on large text corpora to be able to predict the next word. Any provided input text, or in other words, a prompt, is added to the output being generated, all of which the single decoder stack processes. The stack is also equipped with appropriate masking to ensure the prompt tokens only attend among themselves and the generated tokens attend to the prompt and past output tokens. Because they attend only to earlier tokens, decoder-only models are unidirectional. Despite that, decoder-only transformers are unidirectional, they can generate extremely fluent and coherent text by learning the token sequence probability distribution. For example, GPT-3 with 175 billion parameters demonstrated that a decoder-only transformer, when scaled up and pre-trained on massive data, can achieve phenomenal performance in language generation and even few-shot question-answering tasks [36]. In summary, the decoder-only transformer architecture uses the transformer's powerful self-attention for autoregressive generation. This means that this architecture is well-suited for large language models that produce text, code, or other sequential outputs [35][36][37].

Each of these architecture variants is derived from the same fundamental building blocks of the original transformer. An encoder-only model can be seen as using just the encoder half of the full transformer model, and in this architecture, the cross-attention is not needed. A decoder-only model therefore, uses just the decoder half of the full transformer. In this architecture, the cross-attention to an encoder is removed. The flexibility of the transformer

architecture allows it to be adapted in different ways to fit different tasks, for example, encoder-only for analyzing sequences, decoder-only for generating new sequences, and encoder-decoder for converting one sequence into another. All of these variants benefit from the transformer's features of parallelizable self-attention and deep representation learning, and that's why transformers are the dominant architecture in modern natural language processing.

3.3.3 Splitting

In this chapter, the splitting refers to breaking down the transformer model's architecture into reusable building blocks that could be used to create different transformer model architectures. Focus was on three different transformer architectures, which were the encoder-decoder architecture, the encoder-only architecture, and the decoder-only architecture. Beyond these, there are hybrid and specialized variants (e.g. encoder–classifier, retrieval-augmented decoders), but encoder–decoder, encoder-only, and decoder-only remain the most widely adopted transformer design patterns, which is why these were chosen. All three of these transformer variants could be built as a linear stack of layers or blocks.

First Split

The first split of the transformer model was done using four different layer boxes and one compiler box (as seen in the Figure 2). This was the initial approach to gain a better

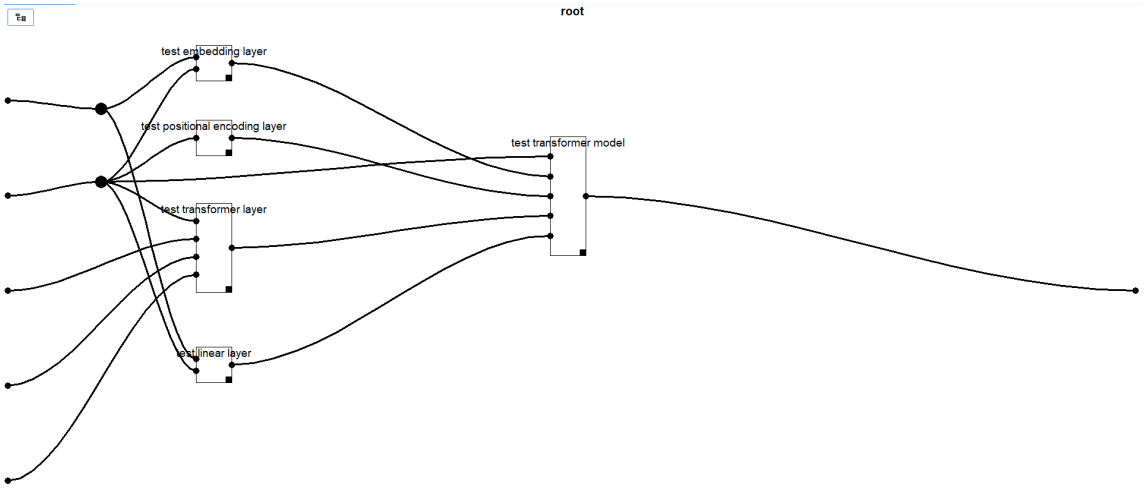


Figure 2. *The First Split Of The Transformer Model*

understanding of the transformer model and to test how it could be divided into boxes. First split used just boxes that created layers, which needed to be inside the transformer model. These layers were: embedding layer, positional encoding layer, transformer layer (that already had encoders and decoders initialized inside it), and final linear layer. The compiler box just took all the layers as inputs and compiled the transformer model. But this

split and representation on the diagram was not informative, and not positionally correct, what layer comes after which. Also the layer boxes needed each one a specific input to create each layer, the input wires became messy and hard to keep track of. With this split there could be created a little different transformer models for example, by changing the model dimension or vocabulary size. But the layers stayed the same and could not create an encoder-only or decoder-only architecture.

Second Split

The second split (see Figure 3) added some representation of what order the layers are in the model. In this split, there were still many input wires because each layer box still needed different inputs, but the top wire moved sequentially from each layer to the next, which gave some structural understanding to the diagram representation. Also in this split,

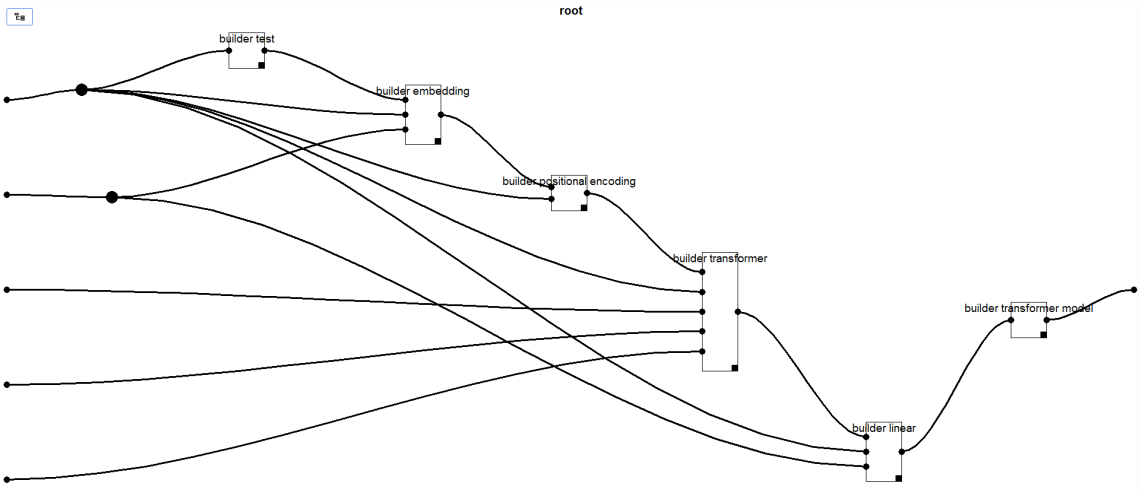


Figure 3. *The Second Split Of The Transformer Model*

there were used builders to add layers and create the final transformer model so that there would be more freedom in what layers could be added and in what order. But this was just an idea because when testing this, split up the layers needed to be at the exact positions that they are in the Figure 3, or otherwise the model would not work correctly. From this split came an idea to use builders for the next splits as well because they provided freedom and possibility to add layers sequentially how the layers should be positioned in the actual model. Overall the second split had the same layers as the first split, but instead of one compiler box, the layers were added between builder boxes. This split still did not support the encoder-only and decoder-only transformer architecture.

Third Split

The third split featured sixteen boxes from which six were builder boxes and the other ten were layer boxes (as seen in the Figure 4). In this split the most significant update was

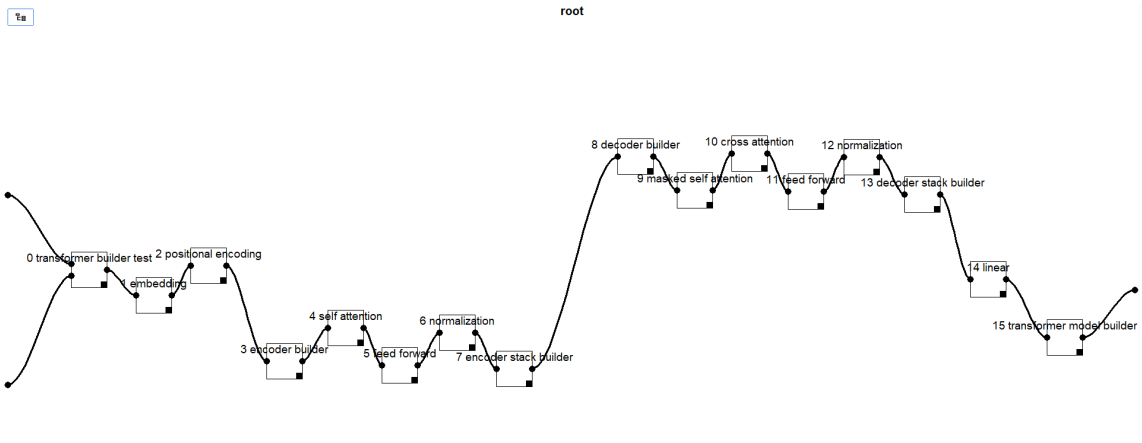


Figure 4. *The Third Split Of The Transformer Model*

that the previously featured transformer layer was split up into the encoder and decoder stacks. Also the sequential adding of layers was improved so that all the layers are placed sequentially and match the actual model layer's architecture. Input wires also only needed to go into the first builder box, which made the wiring clearer. The input builder box could take in two to seven inputs, depending on which variables would need to be changed from the default variables. But with this split, the encoder-only and decoder-only architectures were still not supported, but it made things clearer and gave an idea of how the encoder-only and decoder-only architectures could be created on the diagram and how the DSL should work in the background to support these. This split had one more problem with the encoder and decoder stack, and the problem was that one of those inputs to the first builder box was an integer that specified the number of encoder and decoder layers. This input was used in a for loop to create that many identical encoder and decoder layers for each stack. But creating them this way made the layers in the encoder and decoder stack share weights, which is not a classical approach, and this was an issue.

Fourth Split

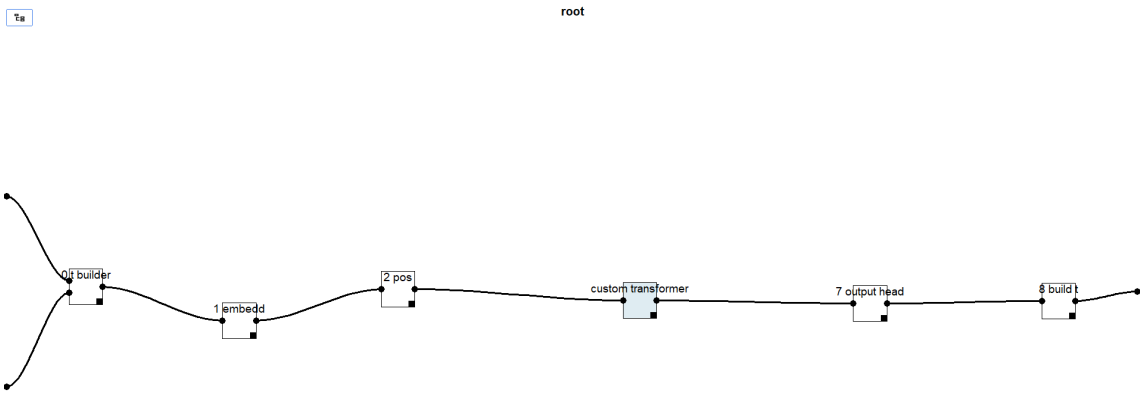


Figure 5. *The Fourth Split Of The Transformer Model*

In the fourth split (seen in the Figure 5), the builder boxes stayed because they proved

useful for representing the diagram sequentially, how the layers are added to the model, and it also gave freedom to create different kinds of layer stacks that could also be added to the final model. This split also divided the model layers even more, for example, there are new layer boxes like linear layer, dropout layer, and activation layer. Because the model was decomposed into so many different boxes this split uses sub-diagrams (see Appendix 3). So the sub-diagrams work like this that inside a box, for example, a custom transformer box (in the Figure 5), there are other boxes inside it. In this case, inside this box, there are two other boxes, and also inside both of these boxes are other boxes. The sub-diagram structure is also seen in the application UI (see sub-diagram structure in Appendix 3 in Figure 21). Using sub-diagrams allows to view the model from a high level and gradually go deeper into the lower levels of the model architecture. This split also created the possibility to create encoder-only and decoder-only architectures on the diagram, which was one of the goals achieved with this split. Also, using the sub-diagrams proved useful for adding many identical encoders or decoders to the model, witch is a common practice to use the same encoder and decoder layers inside the encoder or decoder block. This could be used like this that at the lowest level there could be created the encoder layer structure and after it is done, the box, which has the sub-diagram inside of it, could be copied and pasted as many times as needed. Creating as many encoders or decoders inside the corresponding block.

Fifth Split

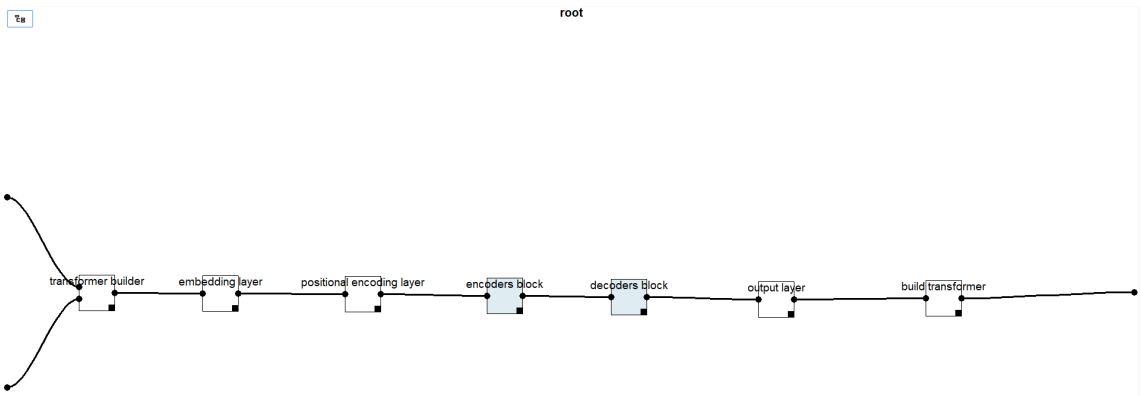


Figure 6. *The Fifth Split Of The Transformer Model*

The fifth split used all of the concepts and mechanisms of the fourth split, but several layer boxes were refined and renamed to more clearly illustrate their roles (as seen in the Figure 6). In this split, sub-diagrams were also used, which structure of the sub-diagrams and all individual sub-diagrams can be seen in the Appendix 4. Like also seen in the figures (see Appendix 4), the sub-diagram architecture is different from the fourth split, and this could also be changed. The diagram can be split into sub-diagrams any way the user wants, this is just an example of how it could be done and visualized by lower and higher layer overviews. This split also supports the encoder-only and decoder-only architectures (see Appendix 5 and Appendix 6). In the encoder-only architecture there is

only an encoders block box, and the decoders block box is removed. Inside the encoders block, there are the same layers as in the encoder-decoder architecture. In the decoder-only architecture, the encoders block box is removed entirely, and the decoders block box stays. But inside the basic decoder boxes, there is a difference between the original encoder-decoder architecture. This difference is that in the decoder-only architecture, there is no cross-attention block. That is because the cross attention block usually takes in the output from the encoder block, but in this architecture, there are no encoders, and that's why the cross attention is not needed. In the end, this split used twenty different layer boxes and builder boxes that can be used to represent different transformer model architectures in the diagram.

Throughout five iterative splits, the transformer decomposition was successfully refined until it supported a fully transparent, linear representation of the model's layer structure. The first split introduced only four layer boxes plus a single compiler box, which was too rough to convey layer order or allow encoder-only and decoder-only versions. In the second split, builder boxes were inserted to enforce a sequential top-wire flow, but individual layers still required many inputs at specific locations. The third split then separated the monolithic transformer layer into distinct encoder and decoder stacks and reorganized the diagram into sixteen boxes, which was closer to the true layer topology. In the fourth split, the builder boxes remained, but they were accompanied by more split-up layer boxes, so that different stacks can be built with clean sequencing still intact. Finally, the fifth split rebranded and optimized each box to its very purpose, to achieve a clear left-to-right flow that can match different transformer architectures and makes it easy to both see how it works and how the layers are added to the model.

3.3.4 Creating Boxes

As mentioned in the previous section (see Section 3.3.3), the transformer model was split up five times, each time improving the visual representation of the model architecture and the quality of generated code. The network was split up this way so that the layers could be added sequentially, so that the model would look clean in the diagram, and the boxes would be easy to add and connect. Every layer is represented in the diagram as a distinct box that can be placed, moved, and connected with other boxes. Users can use these boxes to visually assemble a transformer model layer at a time, and the underlying DSL will generate the code for the model, mirroring the exact order of the layers chosen on the diagram.

Each of these boxes encapsulates one PyTorch module or a small composition of modules.

Because the fifth split relies on builder boxes, all of the model's required parameters are passed as inputs to the very first box (the Transformer builder). Other following boxes can get parameters from the builder that already has all of the parameters. That also means that other boxes follow a single-input, single-output contract, and the box creation workflow does not need to account for branching or loops. The first box takes in two to seven inputs. The first two inputs are mandatory, but others have default values and can be added if the model needs specific parameters instead of the default ones. The single output that the diagram has represents the compiled model. So this means that when generating the code, it will have a main function that needs at least two inputs, and after calling the main function, the model will be compiled and returned. The following subsection provides specific box implementations for the fifth split of the transformer neural network.

Transformer Neural Network

In the transformer DSL, each fundamental component of the model is exposed as a distinct box that can be placed and connected in the diagramming canvas. By wiring together embedding, attention, normalization, builder boxes, etc, in a left-to-right order, users can visually assemble entire encoder–decoder, encoder-only, or decoder-only transformers. For the Fifth split, twenty different boxes were created that can be used to model a wide range of transformer architectures and generate code from the diagram.

Table 1. *Transformer Boxes Table*

Box Name	**Description**
Transformer Builder	This box creates the builder class for the transformer model that holds all of the parameters for other layers that come after this layer.
Embedding Layer	This box makes sure that raw token IDs are transformed into the proper vector format before any computation takes place.
Positional Encoding Layer	This box ensures that sequential position information is added before any attention or feedforward operations take place.
Encoder Builder	This box marks the start of an encoder block, so that the following layers would be added in the correct order.
Normalization Layer	This box can be used whenever the previous layer's outputs need normalization.

Table 1 – *Continues...*

Box Name	Description
Self-Attention Layer	This box lets each token in the sequence look at every other token and decide how much to focus on each token.
Build Encoder Self-Attention	This box is used in the encoder block to create the self-attention block.
Feedforward Builder	This box is used to mark the start of the feedforward block, and the next layers are added to this block.
Linear Layer	The linear layer box works like a regular feedforward dense layer.
Activation Layer	The activation layer box introduces non-linearity into the model computations.
Dropout Layer	The dropout layer box prevents the model from relying too heavily on any activation, reducing overfitting.
Build Encoder	This box marks the end of the encoder block layers.
Decoder Builder	This box should be used at the start of a decoder block so that the following layers would be added in the correct order.
Build Decoder Self-Attention	This box builds the self-attention block for the decoder block, and adds it to the decoder block.
Cross Attention Builder	This box is used to start the cross attention block.
Cross Attention Layer	The cross-attention layer box lets one sequence attend to a different sequence, typically the encoder's outputs.
Build Cross Attention	This box is used to mark the end of the cross-attention block in the decoder.
Build Decoder	This box marks the end of the decoder block layers.
Output Layer	The output layer box projects the final hidden vectors into the desired output space.
Build Transformer	This box should always be the last box of the transformer model in the diagram, because this builds the actual model that is created in the diagram.

In summary all of these distinct boxes could be used to create a wide variety of transformer model architectures. For further information about the different boxes, see Appendix 7. All of the layers that are going into the final model should be between the transformer builder

box and the build transformer box, and other blocks like encoder and decoder blocks also have their specific builders. So the start and end of a block should have the builders make sure that the generated code will mirror the exact architecture as modeled in the diagram.

3.4 Validating

In this section, there is brought out some validation methods that were used to validate if the models can be created with this application, and if the code can be generated easily from the diagram. The generated code should represent the intended neural network architecture that is drawn in the diagram. Specifically, in this chapter are brought out three validating step: first, creating different neural network models on the diagram, second, generating code from these diagrams, and third, training these models to get some results that the models are trainable and, with enough training, become usable.

3.4.1 Creating Models On Diagram

In this subsection there is brought out some examples of the diagrams that can be created in this application, and summarized what layers are in these models.

Feedforward Neural Networks

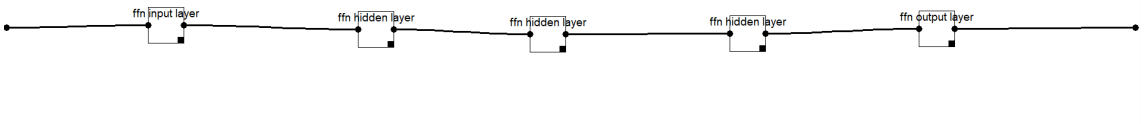


Figure 7. *Example One Feedforward Model*

This is a representation of a simple feedforward neural network model (see Figure 7), built from the DSL boxes. In this model architecture, there is an input layer, which defines the dimensionality of the data entering the network (given from the input port at the left). This is followed by three hidden dense layers, each of which encapsulates a linear layer and an activation (both of these can be set through the application UI), and one output layer at the end, which produces the final prediction. In this diagrammatic DSL, the layout of the boxes shows exactly in what order the layers are added to the model, and each layer box becomes one PyTorch call, so from left to right the diagram shows exactly how the data flows through the model.

This diagram (see Figure 8) shows another feedforward network extended with regularization and normalization between hidden layers. This network still begins with an input

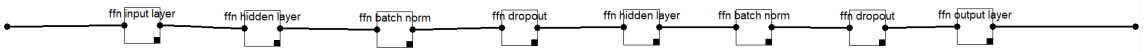


Figure 8. *Example Two Feedforward Model*

layer box, which defines the feature dimension, and still ends with an output layer box that produces the final prediction. Between those boxes are two repeats of a hidden layer box, followed by a batch normalization box and a dropout box. The hidden layer box still holds the linear transformation and activation, the batch norm box normalizes the previous hidden dimension, and the dropout box randomly zeros activations during training. By adding batch normalization and dropout boxes after each hidden layer, this stabilizes training and also reduces overfitting. Like in the previous feedforward network example, the left-to-right ordering of the boxes exactly matches the PyTorch construction. Each box becomes one module in a Sequential module, and data flows in the same sequence during the forward pass.

Convolutional Neural Networks

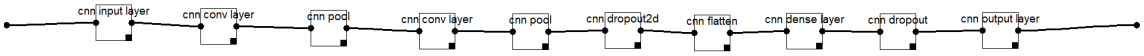


Figure 9. *Example Convolutional Model*

This diagram (see Figure 9) shows a simple example of a convolutional neural network built from the DSL boxes. The diagram starts with an input layer, which defines the incoming tensor shape (channels, height, width), and ends with an output layer that produces the final class scores. Between those two boxes, are positioned other layer boxes, like a convolutional layer box that applies a 2D convolution plus activation to extract local features. After this layer comes a pooling layer, which downsamples the feature maps. These two boxes are repeated one more time in the diagram to increase the model's depth. Next comes the dropout 2D layer box, which randomly zeros entire channels of the feature maps during training for regularization. Then comes the flatten layer that reshapes the 2D feature maps into a 1D vector to prepare for the final classifier. When the data is flattened, then the regular dense layer is added, and this applies a fully connected layer plus activation to combine the learned features. After the dense layer, one more dropout layer is added to create regularization in the dense portion. Because each box corresponds

one-for-one to a PyTorch module in a Sequential module, the left-to-right layout of the diagram both visualizes and is the network's execution order. This means that the data flows through the same sequence of layers that the generated code will execute.

Recurrent Neural Networks

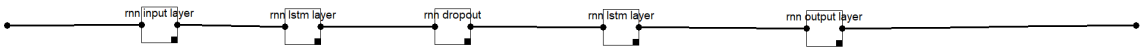


Figure 10. *Example One Recurrent Model*

This diagram (see Figure 10) shows a simple sequence-processing model built from RNN DSL boxes. The input layer defines the per-time-step feature size and also whether the sequences are treated as batch-first. The following box is an LSTM layer box with hidden-state recursion inside. Each LSTM cell loops over time, the box abstracts all of that into one module. After the LSTM box, the dropout layer box is added, which does what it always does and applies a fixed dropout probability to the output of the LSTM at each time step to help prevent overfitting. Then comes another LSTM layer that works the same way as the LSTM layer before, but with different parameters. The last layer is the output layer, which projects the final hidden-state vector down to the desired output size, e.g., class logits or regression scores. Even though each recurrent box contains internal time-looping logic, the left-to-right layout captures the layer stack in the same order that the generated PyTorch code will call them.

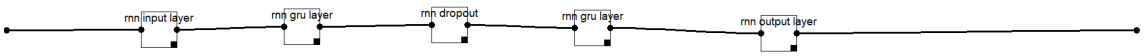


Figure 11. *Example Two Recurrent Model*

This diagram (see Figure 11) shows a simple GRU-based sequence model built from RNN DSL boxes. This model follows the same structure as the model before, but the LSTM layers are switched out for GRU layers. The input box sets the per-step feature size. Two GRU boxes wrap their internal recurrence, with a Dropout box in between. Finally, the output box extracts the last hidden state and projects it to the desired output dimension. Although GRU boxes hide their recurrence internally, their left-to-right layout matches exactly how the generated PyTorch code will invoke them.

This diagram (see Figure 12) shows the same sequential RNN pattern as the previous two examples of the RNN models, but in this the layers are switched with vanilla SimpleRNN

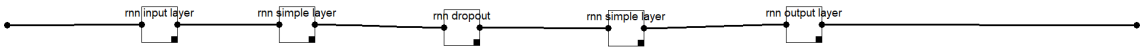


Figure 12. *Example Three Recurrent Model*

cells. The input layer fixes the per-time-step feature size. Then the two simple RNN boxes wrap their internal time-step loops, one before and one after the dropout box, witch randomly zeroes hidden-state activations. The final layer is the output box, which slices off the last hidden state and linearly projects it to the desired output dimension. Although each simple RNN layer box hides its recurrence internally, the left-to-right ordering directly corresponds to how the generated PyTorch code will invoke them in sequence.

Transformer Neural Networks

The encoder-decoder transformer diagram (see Figure 6) shows the top-level diagram of the transformer, but it has nested sub-diagrams inside it (see Appendix 4). This transformer model maps an input sequence to an output sequence via two major components, which are the encoder stack and the decoder stack. But before the tokens are passed into these stacks, the embedding and positional encoding layers are used. The embedding and positional encoding convert discrete tokens into continuous vectors and inject sequence-order information for the stacks. After these two layers comes the encoders block, which holds a stack of identical encoders inside of it (see Figure 30). Each basic encoder box consists of a multi-head self-attention block and a feedforward block (see Figure 31). In each sub-block, the necessary layer boxes (e.g., normalization, attention heads, linear layers, activations, dropout) are simply chained together in sequence to implement their functionality (see Figure 32 and Figure 33). After the encoder stack comes the decoder stack, which is inside the decoders block box (see Figure 34). Inside this block, there are chained-up identical basic decoder blocks. The basic decoder block consists of a multi-head self-attention block, a cross-attention block, and a feedforward block (see Figure 35). In each sub-block similarly to basic encoder, the necessary layer boxes (e.g., normalization, attention heads, cross attention layers, linear layers, activations, dropout) are simply chained together in sequence to implement their functionality (see Figure 36 and Figure 37 and Figure 38). Then finally, comes the output layer box, which projects the decoder's final hidden vectors into the target vocabulary space. By organizing the transformer this way, the top-level encoder and decoder blocks and nested attention sub-diagrams give a clear modular picture of every sub-component. This diagram reflects the order in which layers are composed into the final model rather than the exact runtime flow of token data. The internal loops

and parallel attention operations happen inside each block, but their boxes are laid out in the diagram exactly as the model is built step by step.

The encoder-only transformer diagram (see Figure 39) shows the top-level diagram of the transformer, but like the encoder-decoder transformer, this encoder-only transformer also has sub-diagrams inside it (see Figure 40). This architecture uses all of the same blocks and boxes as the encoder-decoder architecture, but the decoders block is removed entirely. Also, in this case left-to-right layout of the boxes exactly matches the order in which those layers would be added to the generated PyTorch model.

The decoder-only transformer diagram (see Figure 41) shows the top-level diagram of the transformer model, but also this diagram has sub-diagrams inside it (see Figure 42). This architecture uses all of the same blocks and boxes as the encoder-decoder architecture, but the encoders block is removed entirely, and also the cross attention block is removed from inside the basic decoder blocks (see Figure 43). Also, like in the previous architectures, in this case left-to-right layout of the boxes exactly matches the order in which those layers would be added to the generated PyTorch model.

In summary, in this application, it is possible to create a wide variety of different neural networks in the diagram. There could be created different classical neural networks like FFNs, CNNs, and RNNs, and it is also possible to change the architecture easily by adding more layers or different parameters to layers to create different models, specifically suited for different tasks. It is also possible to create transformer models, with architectures like encoder-decoder, encoder-only, or decoder-only. These models could also be modified in the diagram, creating custom encoder or decoder blocks at the lowest level, adding layer boxes one at a time to create the exact structure that is needed. In each case, the left-to-right layout of boxes exactly matches the order in which those layers would be added and invoked by the generated PyTorch model.

3.4.2 Generating Code From Diagram

In this subsection there is brought out the code generation validation for different neural networks that are created in the diagram and generated code from the diagram directly. All of the diagrams generate a PyTorch model, which can be compiled by calling the main function with the necessary inputs for each specific neural network. The code that is generated is validated by printing out the model summary and comparing this with the model diagram to see if the model is generated correctly.

Feedforward Neural Networks

In this section, the code generation from the feedforward neural networks is validated. The code that is validated comes from the two models that were drawn up in the previous section (see Section 3.4.1). To confirm that the generated code reproduces the exact layer stack as in the diagrams (see Figure 7 and Figure 8), the model is instantiated and its Sequential description is printed out:

```
model = main(16)
print(model)
```

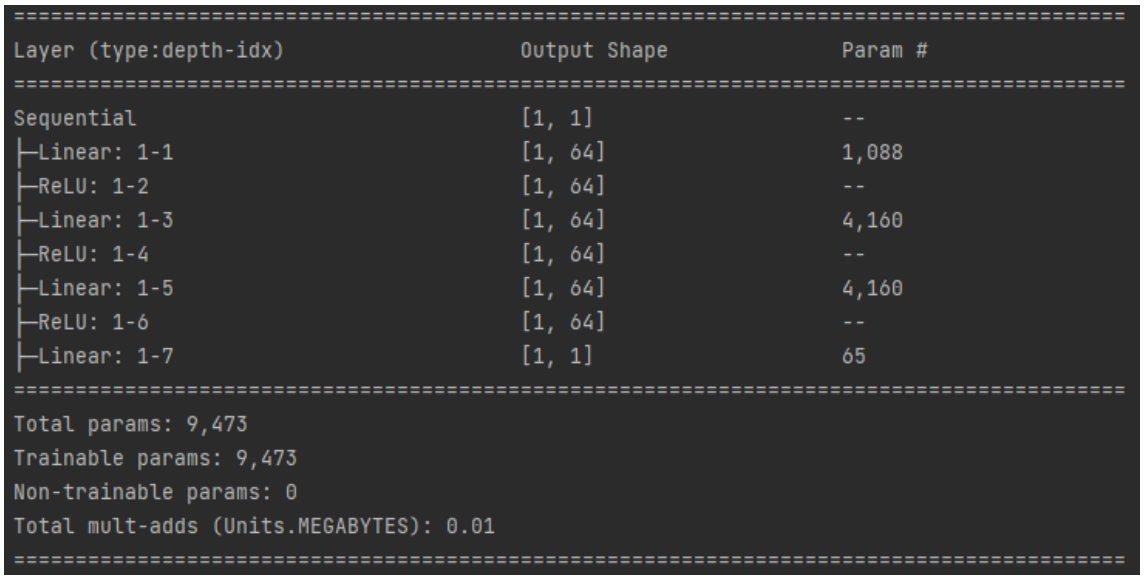
To verify layer by layer that the tensor shapes align with the diagram's expected flow and the parameters are correct, this is done by running:

```
from torchinfo import summary
summary(model, input_size=(1, 16))
```

The first model in the diagram (see Figure 7) has 64 neurons and ReLU activation in each of the hidden layers, and one output neuron for the output layer. Running these scripts gives:

```
Sequential(
  (0): Linear(in_features=16, out_features=64, bias=True)
  (1): ReLU()
  (2): Linear(in_features=64, out_features=64, bias=True)
  (3): ReLU()
  (4): Linear(in_features=64, out_features=64, bias=True)
  (5): ReLU()
  (6): Linear(in_features=64, out_features=1, bias=True)
)
```

Which matches exactly the diagram, an input-to-64 Linear, followed by three hidden 64-unit Linear+ReLU blocks, and finally a 64->1 output Linear. The other script, which prints a layer-by-layer breakdown:



```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
Sequential                               [1, 1]                    --
├─Linear: 1-1                            [1, 64]                   1,088
├─ReLU: 1-2                              [1, 64]                   --
├─Linear: 1-3                            [1, 64]                   4,160
├─ReLU: 1-4                              [1, 64]                   --
├─Linear: 1-5                            [1, 64]                   4,160
├─ReLU: 1-6                              [1, 64]                   --
├─Linear: 1-7                            [1, 1]                    65
==========================================================================================
Total params: 9,473
Trainable params: 9,473
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 0.01
==========================================================================================
```

Figure 13. *Script Two Output For First FFN Model*

The summary (see Figure 13) confirms that every layer's dimensions and parameter counts match the diagram. The first Linear layer goes from 16->64 (16×64 + 64 bias = 1088 params), then three identical hidden Linear layers each map 64->64 (64×64 + 64 = 4160 params) separated by ReLU's (0 params), and the final Linear layer projects 64->1 (64×1 + 1 = 65 params). Summing these gives exactly 1088 + 3×4160 + 65 = 9473 trainable parameters, just as shown in the printed and summarized outputs.

The second model in the diagram (see Figure 8) has 128 neurons and ReLU activation in the first hidden layer, 64 neurons and ReLU activation in the second hidden layer, both dropout boxes have 0.5 dropout probability, and the output layer has 10 output neurons. Running these scripts gives:

```
Sequential(
  (0): Linear(in_features=16, out_features=128, bias=True)
  (1): ReLU()
  (2): BatchNorm1d(128, ...)
  (3): Dropout(p=0.5, inplace=False)
  (4): Linear(in_features=128, out_features=64, bias=True)
  (5): ReLU()
  (6): BatchNorm1d(64, ...)
  (7): Dropout(p=0.5, inplace=False)
  (8): Linear(in_features=64, out_features=10, bias=True)
)
```

This matches the exact layer-by-layer order in the diagram (see Figure 8). The other script, which prints a layer-by-layer breakdown:

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
Sequential                               [1, 10]                   --
├─Linear: 1-1                            [1, 128]                  2,176
├─ReLU: 1-2                              [1, 128]                  --
├─BatchNorm1d: 1-3                       [1, 128]                  256
├─Dropout: 1-4                           [1, 128]                  --
├─Linear: 1-5                            [1, 64]                   8,256
├─ReLU: 1-6                              [1, 64]                   --
├─BatchNorm1d: 1-7                       [1, 64]                   128
├─Dropout: 1-8                           [1, 64]                   --
├─Linear: 1-9                            [1, 10]                   650
==========================================================================================
Total params: 11,466
Trainable params: 11,466
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 0.01
==========================================================================================
```

Figure 14. *Script Two Output For Second FFN Model*

The summary (see Figure 14) again confirms that every layer's dimensions and parameter counts match the diagram. The first Linear layer maps 16->128 (16×128 + 128 bias = 2176 params), followed by BatchNorm1d on 128 channels (2×128 = 256 params) and Dropout(p=0.5) (0 params), then a second Linear maps 128->64 (128×64 + 64 = 8256 params) followed by BatchNorm1d on 64 channels (2×64 = 128 params) and Dropout(p=0.5) (0 params), and finally an output Linear from 64->10 (64×10 + 10 = 650 params). Summing these gives exactly 2176 + 256 + 8256 + 128 + 650 = 11466 trainable parameters, just as shown in the printed and summarized outputs.

Convolutional Neural Networks

In this section, the code generation from the convolutional neural network is validated. The code is generated from the model that was brought out in the previous section (see Section 3.4.1 and see Figure 9).To confirm that the generated code reproduces the exact layer stack as in the diagram, the model is instantiated and its Sequential description is printed out:

```
model = main((3, 32, 32))
print(model)
```

To verify layer by layer that the tensor shapes align with the diagram's expected flow and

the parameters are correct, this is done by running:

```
from torchinfo import summary
summary(model, input_size=(1, 3, 32, 32))
```

The CNN model in the diagram (see Figure 9) has the first convolutional layer with parameters: out channels = 16, kernel size = 3, stride = 1, padding = 1, and activation is ReLU. The second convolutional layer has the same parameters, but the out channels is changed to 32. Both pool layers have parameters: kernel size = 2, stride = 1, and pool type is max. The dropout 2D has probability of 0.25, and the regular dropout has a probability of 0.5. The dense layer has 128 neurons and ReLU activation, and the output layer has 10 output neurons and ReLU activation as well. Running these scripts gives:

```
Sequential(
  (0): Conv2d(3, 16, kernel_size=(3, 3), ...)
  (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, ...)
  (2): ReLU()
  (3): MaxPool2d(kernel_size=2, stride=1, ...)
  (4): Conv2d(16, 32, kernel_size=(3, 3), ...)
  (5): BatchNorm2d(32, eps=1e-05, momentum=0.1, ...)
  (6): ReLU()
  (7): MaxPool2d(kernel_size=2, stride=1, ...)
  (8): Dropout2d(p=0.25, inplace=False)
  (9): Flatten(start_dim=1, end_dim=-1)
  (10): Linear(in_features=28800, out_features=128, bias=True)
  (11): BatchNorm1d(128, eps=1e-05, momentum=0.1, ...)
  (12): ReLU()
  (13): Dropout(p=0.5, inplace=False)
  (14): Linear(in_features=128, out_features=10, bias=True)
  (15): ReLU()
)
```

This matches the exact layer-by-layer order in the diagram (see Figure 9). The other script, which prints a layer-by-layer breakdown:

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
Sequential                               [1, 10]                   --
├─Conv2d: 1-1                            [1, 16, 32, 32]           448
├─BatchNorm2d: 1-2                       [1, 16, 32, 32]           32
├─ReLU: 1-3                              [1, 16, 32, 32]           --
├─MaxPool2d: 1-4                         [1, 16, 31, 31]           --
├─Conv2d: 1-5                            [1, 32, 31, 31]           4,640
├─BatchNorm2d: 1-6                       [1, 32, 31, 31]           64
├─ReLU: 1-7                              [1, 32, 31, 31]           --
├─MaxPool2d: 1-8                         [1, 32, 30, 30]           --
├─Dropout2d: 1-9                         [1, 32, 30, 30]           --
├─Flatten: 1-10                          [1, 28800]                --
├─Linear: 1-11                           [1, 128]                  3,686,528
├─BatchNorm1d: 1-12                      [1, 128]                  256
├─ReLU: 1-13                             [1, 128]                  --
├─Dropout: 1-14                          [1, 128]                  --
├─Linear: 1-15                           [1, 10]                   1,290
├─ReLU: 1-16                             [1, 10]                   --
==========================================================================================
Total params: 3,693,258
Trainable params: 3,693,258
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 8.61
==========================================================================================
```

Figure 15. *Script Two Output For CNN Model*

The summary (see Figure 15) confirms that every layer's dimensions and parameter counts match the diagram. The first Conv2d maps 3->16 (3×16×3×3 + 16 bias = 448 params), followed by BatchNorm2d(16) (2×16 = 32 params), ReLU and MaxPool2d (0 params); then a Conv2d 16->32 (16×32×3×3 + 32 = 4640 params), BatchNorm2d(32) (2×32 = 64), ReLU and MaxPool2d (0); a Dropout2d (0); Flatten; a Linear 28800->128 (28800×128 + 128 = 3686528 params), BatchNorm1d(128) (2×128 = 256), ReLU and Dropout (0); and finally a Linear 128->10 (128×10 + 10 = 1290 params) and ReLU (0). Summing these gives exactly 448 + 32 + 4640 + 64 + 3686528 + 256 + 1290 = 3693258 trainable parameters, just as shown in the printed and summarized outputs.

Recurrent Neural Networks

In this section, the code generation from the recurrent neural networks is validated. The code that is validated comes from the three models that were drawn up in the previous section (see Section 3.4.1). To confirm that the generated code reproduces the exact layer stacks as in the diagrams (see Figure 10, Figure 11, and Figure 12), the model is instantiated and its Sequential description is printed out:

```
model = main(16)
print(model)
```

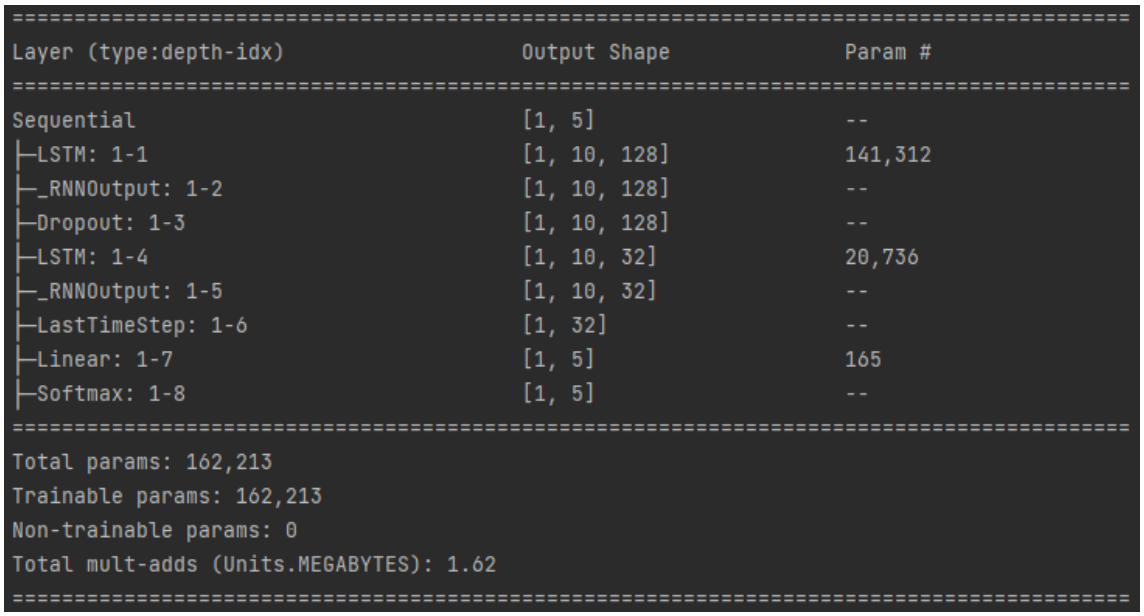
To verify layer by layer that the tensor shapes align with the diagram's expected flow and the parameters are correct, this is done by running:

```
from torchinfo import summary
summary(model, input_size=(1,10,16))
```

The first model in the diagram (see Figure 10) has the first LSTM layer with parameters: neurons = 64, number of layers = 2, bidirectional is true,e and dropout probability is 0.2. The second LSTM layer has 32 neurons, number of layers is 1, bidirectionality is false, and dropout is 0.1. The dropout box between the two LSTM layers has a probability of 0.5, and the output layer has 5 output neurons and Softmax activation. Running these scripts gives:

```
Sequential(
  (0): LSTM(16, 64, num_layers=2, batch_first=True, ...)
  (1): _RNNOutput()
  (2): Dropout(p=0.5, inplace=False)
  (3): LSTM(128, 32, batch_first=True, dropout=0.1)
  (4): _RNNOutput()
  (5): LastTimeStep()
  (6): Linear(in_features=32, out_features=5, bias=True)
  (7): Softmax(dim=None)
)
```

Which matches the exact layer-by-layer order in the diagram (see Figure 10). The other script, which prints a layer-by-layer breakdown:



```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
Sequential                               [1, 5]                    --
├─LSTM: 1-1                              [1, 10, 128]              141,312
├─_RNNOutput: 1-2                        [1, 10, 128]              --
├─Dropout: 1-3                           [1, 10, 128]              --
├─LSTM: 1-4                              [1, 10, 32]               20,736
├─_RNNOutput: 1-5                        [1, 10, 32]               --
├─LastTimeStep: 1-6                      [1, 32]                   --
├─Linear: 1-7                            [1, 5]                    165
├─Softmax: 1-8                           [1, 5]                    --
==========================================================================================
Total params: 162,213
Trainable params: 162,213
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 1.62
==========================================================================================
```

Figure 16. *Script Two Output For First RNN Model*

The summary (see Figure 16) confirms that every layer's dimensions and parameter counts match the diagram. The first bidirectional LSTM maps 16->64 over two layers (per layer per direction: 4×64×16 + 4×64×64 + 8×64 = 20992, ×2 directions, ×2 layers = 141312 params); RNNOutput, Dropout(p=0.5), and LastTimeStep add 0; the second unidirectional LSTM maps 128->32 over one layer (4×32×128 + 4×32×32 + 8×32 = 20736 params); and finally a Linear 32->5 (32×5 + 5 = 165 params) and Softmax (0). Summing these gives exactly 141312 + 20736 + 165 = 162213 trainable parameters, matching the printed and summarized outputs.

The second model in the diagram (see Figure 11) has the first GRU layer with parameters: neurons = 32, number of layers = 1, bidirectionality is false, and dropout probability is 0.1. The second GRU layer has the same parameters, but the dropout probability is 0.0. The dropout box between the GRU layers has a dropout probability of 0.5, and the output layer has 5 output neurons and Softmax activation. Running these scripts gives:

```
Sequential(
  (0): GRU(16, 32, batch_first=True, dropout=0.1)
  (1): _RNNOutput()
  (2): Dropout(p=0.5, inplace=False)
  (3): GRU(32, 32, batch_first=True)
  (4): _RNNOutput()
  (5): LastTimeStep()
```

```
    (6): Linear(in_features=32, out_features=5, bias=True)
    (7): Softmax(dim=None)
  )
```

Which matches the exact layer-by-layer order in the diagram (see Figure 11). The other script, which prints a layer-by-layer breakdown:

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
Sequential                               [1, 5]                    --
├─GRU: 1-1                               [1, 10, 32]               4,800
├─_RNNOutput: 1-2                        [1, 10, 32]               --
├─Dropout: 1-3                           [1, 10, 32]               --
├─GRU: 1-4                               [1, 10, 32]               6,336
├─_RNNOutput: 1-5                        [1, 10, 32]               --
├─LastTimeStep: 1-6                      [1, 32]                   --
├─Linear: 1-7                            [1, 5]                    165
├─Softmax: 1-8                           [1, 5]                    --
==========================================================================================
Total params: 11,301
Trainable params: 11,301
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 0.11
==========================================================================================
```

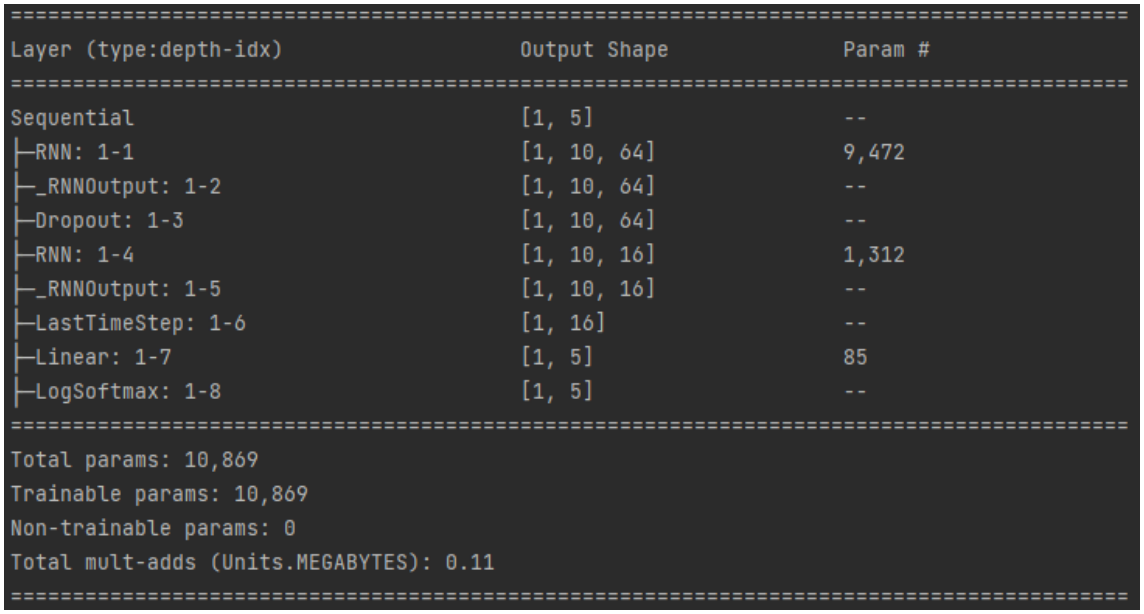
Figure 17. *Script Two Output For Second RNN Model*

The summary (see Figure 17) confirms that every layer's dimensions and parameter counts match the diagram. The first GRU maps 16->32 (weight-ih: 3×32×16 = 1536, weight-hh: 3×32×32 = 3072, biases: 2×3×32 = 192 -> total 4800 params), followed by RNNOutput and Dropout(p=0.5) (0 params); then a GRU 32->32 (weight-ih: 3×32×32 = 3072, weight-hh: 3×32×32 = 3072, biases: 2×3×32 = 192 -> total 6336 params), RNNOutput and LastTimeStep (0); and finally a Linear 32->5 (32×5 + 5 = 165 params) and Softmax (0). Summing these gives exactly 4800 + 6336 + 165 = 11301 trainable parameters, just as shown in the printed and summarized outputs.

The third model in the diagram (see Figure 12) has the first simple RNN layer with parameters: neurons = 32, number of layers = 2, non-linearity is ReLU, bidirectionality is True, and dropout = 0.1. The second simple RNN layer has therefore 16 neurons, number of layers is 1, non-linearity is Tanh, bidirectionality is False, and the dropout probability is 0.0. The dropout layer between the two simple RNN layers has a probability of 0.5, and the output layer has 5 output neurons and LogSoftmax activation. Running these scripts gives:

```
Sequential(
  (0): RNN(16, 32, num_layers=2, batch_first=True, ...)
  (1): _RNNOutput()
  (2): Dropout(p=0.5, inplace=False)
  (3): RNN(64, 16, batch_first=True)
  (4): _RNNOutput()
  (5): LastTimeStep()
  (6): Linear(in_features=16, out_features=5, bias=True)
  (7): LogSoftmax(dim=None)
)
```

Which matches the exact layer-by-layer order in the diagram (see Figure 12). The other script, which prints a layer-by-layer breakdown:



```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
Sequential                               [1, 5]                    --
├─RNN: 1-1                               [1, 10, 64]               9,472
├─_RNNOutput: 1-2                        [1, 10, 64]               --
├─Dropout: 1-3                           [1, 10, 64]               --
├─RNN: 1-4                               [1, 10, 16]               1,312
├─_RNNOutput: 1-5                        [1, 10, 16]               --
├─LastTimeStep: 1-6                      [1, 16]                   --
├─Linear: 1-7                            [1, 5]                    85
├─LogSoftmax: 1-8                        [1, 5]                    --
==========================================================================================
Total params: 10,869
Trainable params: 10,869
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 0.11
==========================================================================================
```

Figure 18. *Script Two Output For Third RNN Model*

The summary (see Figure 18) confirms that every layer's dimensions and parameter counts match the diagram. The first bidirectional SimpleRNN maps 16->32 over two layers (layer 0 per direction: 32×16 + 32×32 + 2×32 = 1600 -> ×2 directions = 3200; layer 1 per direction: 32×64 + 32×32 + 2×32 = 3136 -> ×2 directions = 6272 -> total 3200 + 6272 = 9472 params); RNNOutput and Dropout(p=0.5) add 0; the second unidirectional SimpleRNN maps 64->16 over one layer (16×64 + 16×16 + 2×16 = 1312 params); LastTimeStep adds 0; and finally a Linear 16->5 (16×5 + 5 = 85 params) and LogSoftmax add 0. Summing these gives exactly 9472 + 1312 + 85 = 10869 trainable parameters, matching the printed and summarized outputs.

Transformer Neural Networks

This section validates the code generation from the transformer neural networks. The code that will be validated comes from the three different transformer model architectures drawn up in the previous section (see Section 3.4.1). Each generated model is instantiated via its main function, and then the layer-by-layer architecture is printed out. After this, a dummy forward pass is performed to confirm that its output has the correct shape. All of this should ensure that the exported code compiles and implements the intended network topology, what is created in the diagram.

The encoder-decoder model in the diagram (see Figure 6) has six encoders and six decoders, which should be confirmed by the script:

```
from torchinfo import summary

vocab_size, model_dim = 1000, 512
encdec = main(model_dim, vocab_size)

print("encoder-decoder:")
print(encdec, "\n")

seq_len, batch = 16, 4
print("summary encdec:")
summary(encdec, input_data=(
    torch.zeros(seq_len, batch, dtype=torch.long),
    torch.zeros(seq_len, batch, dtype=torch.long)
))

src = torch.randint(0, vocab_size, (seq_len, batch))
tgt = torch.randint(0, vocab_size, (seq_len, batch))

out = encdec(src, tgt)
assert out.shape == (seq_len, batch, vocab_size)
print("forward-shape OK")
```

The outputs of this script can be seen in the Appendix 8 and the Appendix 9. In the Appendix 8 can be seen the structure and parameterization of the generated transformer model. The raw print confirms that the embedding, six encoder layers, six decoder layers, and output head are all correct. Then the summary call, which can be seen in the Appendix

9, verifies that every sub-module produces the expected intermediate tensor shapes and that the parameter counts match the architecture specification. Lastly a dummy forward pass confirms that the model runs end-to-end and outputs a correct tensor shape. Overall this proves that the generated code is structurally correct and matches the diagram (see Figure 6 and Appendix 4), and also functionally executable.

The encoder-only model in the diagram (see Figure 39) has six encoders and no decoders, which should be confirmed by the script:

```
from torchinfo import summary

vocab_size, model_dim = 1000, 512
enco = main(model_dim, vocab_size)

print("encoder-only:")
print(enco, "\n")

seq_len, batch = 16, 4
print("summary enc-only:")
summary(enco, input_data=(
    torch.zeros(seq_len, batch, dtype=torch.long),
    torch.zeros(1, 1, dtype=torch.long),
))

src = torch.randint(0, vocab_size, (seq_len, batch))

out = enco(src, None)
assert out.shape == (seq_len, batch, vocab_size)
print("encoder-only forward-shape OK")
```

The outputs of this script can be seen in the Appendix 10 and the Appendix 11. In the Appendix 10 can be seen the structure and parameterization of the generated encoder-only transformer model. The raw print confirms that the embedding, six encoder layers, and an output head are all correct, and follow the same topology as the diagram (see Figure 39 and Appendix 5). The summary call output can be seen in the Appendix 11, which verifies that every sub-module produces the expected tensor shapes and that the parameter counts match the specifications. The dummy forward pass also confirms that the model runs end-to-end and outputs a correct tensor shape. All this proves that the generated code

for the encoder-only model is structurally correct and functionally executable.

The decoder-only model in the diagram (see Figure 41) has no encoders and six decoders, which should be confirmed by the script:

```
from torchinfo import summary

vocab_size, model_dim = 1000, 512
deco = main(model_dim, vocab_size)

print("decoder-only:")
print(deco, "\n")

seq_len, batch = 16, 4
print("summary dec-only:")
summary(deco, input_data=(
    torch.zeros(1, 1, dtype=torch.long),
    torch.zeros(seq_len, batch, dtype=torch.long),
))

tgt = torch.randint(0, vocab_size, (seq_len, batch))

out = deco(None, tgt)
assert out.shape == (seq_len, batch, vocab_size)
print("decoder-only forward-shape OK")
```

The outputs of this script can be seen in the Appendix 12 and the Appendix 13. In the Appendix 12 can be seen the structure and parameterization of the generated decoder-only transformer model. The raw print confirms that the embedding, six decoder layers, and an output head are all present and correctly structured, like in the diagram (see Figure 41 and Appendix 6). The summary output can be seen in the Appendix 13, which verifies that every sub-module produces the expected intermediate tensor shapes and that the parameter counts match the architecture specification. Finally a dummy forward pass confirms that the model runs end-to-end and outputs a correct tensor shape. So every layer count, each sub-block's parameter count, and all tensor shapes are correct, which proves that the generated code for the decoder-only model is structurally correct and functionally executable.

In summary the code generation works like it should, and it follows the structure that is specified on the diagram for different neural network models. It also takes into account the parameters that have been set through UI or given as inputs for the main function, and makes sure that the tensor shapes and parameter counts are correct. All of the generated code is functionally executable, and the main function returns the compiled model like it should.

3.4.3 Training Generated Models

In this section there is brought out some of the models that were drawn up on the diagram and generated code from it and then trained for some tasks. The models were not trained to be perfect, but just to validate that the generated models can be trained, and that they can learn and improve through training. Also should be mentioned that the training process needs a lot of computing power, and that's why the training was cut short, just to see some features that each model had learned to validate that the models can be trained and they can learn.

The first model that was trained were the encoder-decoder transformer model (see Figure 6), which was trained to translate text from German to English. The data set that this model was trained on was IWSLT 2017 German-English and the model dimension was set to 512, and other values were left to default. Here is a table that shows how the trained transformer translated German sentences or phrases to English:

Table 2. *Transformer Translation Table*

Given Input	**Generated Output**	**Expected Output**
Guten Abend!	Good evening ! Good evening ! Good evening !	Good evening!
Gute Nacht!	Good night ! Good night ! Good night !	Good night!
Wie heißt du?	How do you do ? How do you ?	What is your name?
Willkommen!	Welcome ! Welcome . Welcome .	Welcome!
Wie komme ich dorthin?	How do I get there ? How do I get there ?	How do I get there?

Continues...

Table 2 – *Continues...*

Given Input	**Generated Output**	**Expected Output**
Bis zum nächsten Mal.	The next time , the next time .	See you next time.
Lass uns ein bisschen Spaß haben!	Let 's have a little fun !	Let's have some fun!
Wiederholen Sie bitte.	Give us a choice . Give me a hand . Give up .	Please repeat.
Tschüss!	Bye ! Bye ! Bye !	Bye!
Bitte schön.	There you go . There you go . There you go .	You're welcome.
Danke.	Thank you . Thank you . Thank you . Thank you . Thank you .	Thank you.
Entschuldigung.	Sorry . Sorry . Sorry . Sorry . Sorry .	Excuse me.
Wie alt bist du?	How old are you ?	How old are you?
Ich bin … Jahre alt.	I 'm – years old .	I'm … years old.
Keine Sorge.	Do n't worry . Do n't worry . No worry .	Don't worry.
Ich möchte … kaufen.	I want to buy . I want to buy .	I would like to buy …
Können Sie mir helfen?	Can you help me ?	Can you help me?
Ich feiere meinen Geburtstag.	I <unk>my birthday . I birthday my birthday .	I celebrate my birthday.
Nein, hungrig bin ich nicht.	No , I'm not hungry . I'm not .	No, I am not hungry.
Ich komme aus Rumänien.	I come from Romania . I come from Romania .	I come from Romania.
Hallo, Rezeption.	Hi , diamonds .	Hello, front desk.
Die Rechnung, bitte.	The bill , please .	The bill, please.
Prost!	Cheers . Cheers .	Cheers!
Wo ist der Bahnhof?	Where is the station ? Where 's the station ?	Where is the train station?
Entschuldigung.	Sorry . Sorry . Sorry . Sorry . Sorry .	Excuse me.
Einen Moment, bitte.	A moment , please .	One moment, please.

The model is clearly learned the basic mapping from German to English, but limitations show up as soon as the sentence is longer or has more complex words. The most obvious mistake that the model does is that it repeats the tokens, which is caused by training the model with a simple greedy decode and a lack of training overall. The model translates some sentences also completely wrong, and that is because the training data did not have enough examples, which explains the errors. However some easy sentences are translated correctly, which is a good sign. In summary, the model can be trained and used, but needs more training data and a longer training period, and a better training script as well, but this is enough to see that the model can learn, and with better training, it can be used, for example to translate text.

The second model that was trained was the decoder-only transformer model (see Figure 41), which was trained for language modeling. So at each position, it predicts the next token based on all the previous tokens. This model was trained on data WikiText-2 raw v1, which was obtained from Hugging Face. The model dimension was set to 512, and other parameters were left as defaults. Here is a table that shows some examples that the trained model generated when given an input prompt:

Table 3. *Transformer Generation Table*

Given Input	**Generated Output**
Once upon a time	Once upon a time record of his first appearance in the late 1940s...
Most popular people live in	most popular people live in the united states , and the united states...
Grades in high school does	grades in high school does not appear in the early 20th century...
There were a singer who was the	there were a singer who was the first to be the first to be...
In a world where AI	in a world where ai was abandoned around 21 may...
In Estonia there are many	in <unk> there are many of the kakapo . the kakapo is a common starling...
In 1492, Columbus sailed	in 1492 , columbus sailed off...

Taking into account that the model capacity was not massive and the dataset size was also small the model still learned something. The decoder-only model learned some basic syntactic and structural patterns, and the part-of-speech is roughly correct. But it still hallucinates and loops on low information prompts. It also struggles with more complex words or long-range coherence. In summary the model learned some of the most basic

things and needs more and better training, so that the model would become useful.

The third model that was trained was the encoder-only transformer model (see Figure 39), which was trained as a BERT-style masked-language model to predict missing tokens at the masked positions. So the model sees only the corrupted sequence and must predict the original token. This model was also trained on data WikiText-2 raw v1, which was obtained from Hugging Face. The model dimension was set to 512, and other parameters were left as defaults. Here is a table that shows some examples what tokens did the trained model predicted for the masked tokens:

Table 4. *Transformer Prediction Table*

Given Input	**Predicted Token**
She opened the door with her [MASK] hand.	other
The dog chased the [MASK] down the street.	man
Water [MASK] the most important fluid.	is
I have computer, mouse, [MASK] monitor.	and
Who [MASK] swimming?	are
They went to the [MASK] to buy some groceries.	idea
Monopoly is a good [MASK]	player

Taking into account that the model capacity was not massive and the dataset size was also small the model was able to learn something. The encoder-only model learned the broad syntactic patterns, what words tend to follow what contexts. It is good at general, easy grammar and frequent words, but very limited in real understanding. In summary the model learned some of the basics and can fill in the blanks, but needs more training to be useful for real-world tasks.

Overall Validation

Overall, the project largely met expectations, though it didn't unfold exactly as planned. So, in the developed application different neural networks can be visualized and modified in the diagram. This was implemented so that the layers could be added in linear order for all of the various architectures to keep the modeling as easy as possible. This linear representation aligned with the author's initial idea that the modeling of different neural network architectures would be simple and easy to follow, and which layers should be added after which. But focusing on the linear representation did not capture the flow of the data through forward passes for all of the different architectures. In a vanilla MLP (Multi-Layer Perceptron), each box's placement is the forward-pass order, so a single horizontal chain conveys both model construction and computation flow. However, in

the transformer encoder-decoder architecture, linear representation does not represent the data flow through the model, because the encoder's final representation must fan out into every decoder block's cross-attention layer. The model architecture could be drawn on the diagram to capture some idea of how the data will flow through the model (see the Appendix 14). So the data flow could be represented to some extent, but this could be improved as well.

A huge part of this whole thesis was learning about neural network architectures, identifying which layers make up multilayer perceptrons, convolutional networks, recurrent networks, and transformers. Through this work, insights were also gained into how the data actually flows through each model, beyond a simple left-to-right chain. Knowing all of this information now, if the project were to be started over, more time would be spent up-front designing a diagrammatic DSL that can express branching and fan-out, rather than relying mostly on the linear structure of adding layers to a model. The transformer flow would be prototyped early, with encoders, parallel attention heads, and decoders laid out on paper, so that the final UI and code generation pipeline would be designed to reflect the true shape of the computations and how the data will flow. In retrospect, a better flow design and a DSL constructed to follow this design in the code generation would have produced more informative model diagrams. That said, although the linear implementation is useful for easy model creation, it is viewed as less engaging by those more familiar with the neural network field.

In summary the whole project was a good learning experience, and the goals were achieved. With the benefit of hindsight, certain aspects would be approached differently. It was found that the sequential addition of layers enabled models to be constructed quickly and with minimal effort. Various neural network architectures were made creatable and modifiable on the diagram, and corresponding code generation was successfully implemented. However, it was recognized that the visual representation could be further refined so that even architectures featuring parallel paths or branching would clearly represent the data flow and compilation structure. The use of builder boxes to indicate where layer blocks should be inserted in the diagram, meaning between the builder boxes, was also appreciated. But also, for example, for users already well-versed in neural networks, these builder boxes may just make things more confusing, and in further development, maybe the builder boxes should be removed.

4. Summary

This thesis work has set up the design and implementation of a domain-specific extension to the Ivaldi string diagramming tool, which enables diagrammatic modeling and automatic code generation for a wide variety of neural network architectures. The model-building process allows users to construct models through drag-and-drop operations of layer primitives (dense, convolutional, recurrent, and transformer blocks), which mirror PyTorch equivalent modules. The system transforms each diagram into a hypergraph, which it traverses in topological order to insert user-defined hyperparameters into code templates before producing executable Python code.

Key contributions of this work include:

- DSL design based on category-theoretic string-diagram notation, which rigorously records sequential and partly non-linear compositions of layers.
- Implementation in Ivaldi, extending both frontend and backend to support layer-specific property editing, subdiagram nesting, and recursive code-generation logic.
- Classic and contemporary architecture support, starting from basic multilayer perceptions and convolutional classifiers to LSTMs, GRUs, and complete transformer encoder–decoder pipelines.
- Exhibitions demonstrating that diagrammatic model construction generates precise, trainable Python code for all network types supported.

The project implementation revealed essential trade-offs between linear left-to-right diagrammatic notation and the need to represent complex data-flow patterns, especially in encoder–decoder and multi-headed attention crossings. The experience shows that a more expressive DSL that enables explicit fan-out and branching would improve both readability and pedagogical value.

The extended Ivaldi tool enhances neural-network design workflows through its ability to merge high-level visual modeling with low-level code generation. The current implementation allows fast prototyping of different architectures, yet future work will focus on developing more detailed flow visualization and removing builder scaffolding nodes, and adding support for skip connections and weight sharing, and dynamic loops. This thesis provides a solid foundation for diagrammatic neural-network engineering while opening multiple avenues for future research and development.

References

[1] Vincent Thornton Abbott. *Robust Diagrams for Deep Learning Architectures: Applications and Theory*. [WWW]. URL: https://www.vtabbott.io/content/files/2023/11/Robust-Diagrams-for-Deep-Learning-Architectures.pdf.

[2] Tom Leinster. *Basic Category Theory*. [WWW]. URL: https://arxiv.org/abs/1612.09375.

[3] Sam Ezeh. *Graphical Rewriting for Diagrammatic Reasoning in Monoidal Categories in Lean4*. [WWW]. URL: https://drops.dagstuhl.de/storage/00lipics/lipics-vol309-itp2024/LIPIcs.ITP.2024.41/LIPIcs.ITP.2024.41.pdf.

[4] Celia Rubio-Madrigal and Jules Hedges. *Rendering string diagrams recursively*. [WWW]. URL: https://arxiv.org/pdf/2404.02679.

[5] Teppei Matsui Trung Quang Pham and Junichi Chikazoe. *Evaluation of the Hierarchical Correspondence between the Human Brain and Artificial Neural Networks: A Review*. [WWW]. URL: https://www.mdpi.com/2079-7737/12/10/1330.

[6] Tarun Kumar Gupta Sichen Pan and Khalid Raza. *BatTS: a hybrid method for optimizing deep feedforward neural network*. [WWW]. URL: https://peerj.com/articles/cs-1194/.

[7] Xia Zhao et al. *A review of convolutional neural networks in computer vision*. [WWW]. URL: https://link.springer.com/article/10.1007/s10462-024-10721-6.

[8] Theo G. Swart Ibomoiye Domor Mienye and George Obaido. *Recurrent Neural Networks: A Comprehensive Review of Architectures, Variants, and Applications*. [WWW]. URL: https://www.mdpi.com/2078-2489/15/9/517.

[9] Ashish Vaswani et al. *Attention Is All You Need*. [WWW]. URL: https://arxiv.org/pdf/1706.03762.

[10] Sanghyuk Roy Choi and Minhyeok Lee. *Transformer Architecture and Attention Mechanisms in Genome Data Analysis: A Comprehensive Review*. [WWW]. URL: https://www.mdpi.com/2079-7737/12/7/1033.

[11] FRANCESCO RICCARDO CRESCENZI. *TOWARDS A CATEGORICAL FOUNDATION OF DEEP LEARNING: A SURVEY*. [WWW]. URL: https://arxiv.org/pdf/2410.05353.

[12] Jade Master. *Deep Learning for the Working Category Theorist*. [WWW]. URL: https://math.ucr.edu/home/baez/mathematical/ACTUCR/Master_Deep_Learning_for_the_Working_Category_Theorist.pdf.

[13] Tom Xu amd Yoshihiro Maruyama. *Neural String Diagrams: A Universal Modelling Language for Categorical Deep Learning*. [WWW]. URL: https://link.springer.com/chapter/10.1007/978-3-030-93758-4_32.

[14] Jan Heering Marjan Mernik and Anthony M. Sloane. *When and how to develop domain-specific languages*. [WWW]. URL: https://dl.acm.org/doi/10.1145/1118890.1118892.

[15] Yoav Goldberg Graham Neubig and Chris Dyer. *On-the-fly Operation Batching in Dynamic Computation Graphs*. [WWW]. URL: https://arxiv.org/pdf/1705.07860.

[16] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. [WWW]. URL: https://arxiv.org/pdf/1912.01703.

[17] Jason Ansel et al. *PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation*. [WWW]. URL: https://dl.acm.org/doi/pdf/10.1145/3620665.3640366.

[18] PyTorch Contributors. *PyTorch documentation*. [WWW]. URL: https://pytorch.org/docs/main/.

[19] Seung Won Min et al. *PYTORCH-DIRECT: ENABLING GPU CENTRIC DATA ACCESS FOR VERY LARGE GRAPH NEURAL NETWORK TRAINING WITH IRREGULAR ACCESSES*. [WWW]. URL: https://arxiv.org/pdf/2101.07956.

[20] Marisa Kirisame et al. *DYNAMIC TENSOR REMATERIALIZATION*. [WWW]. URL: https://arxiv.org/pdf/2006.09616.

[21] Terrence L. Fine. *Feedforward Neural Network Methodology*. [WWW]. URL: https://books.google.ee/books?hl=en&lr=&id=s-PlBwAAQBAJ&oi=fnd&pg=PR5&dq=feedforward+neural+network&ots=adoICq4e7n&sig=pHSyL7ZZTZAsdtRKG78tD6qJOy8&redir_esc=y#v=onepage&q=feedforward%20neural%20network&f=false.

[22] Vladimír Kvasnicka Daniel Svozil and Jirí Pospichal. *Introduction to multi-layer feed-forward neural networks*. [WWW]. URL: https://staff.fmi.uvt.ro/~daniela.zaharie/dm2017/EN/projects/Algorithms/MLP/MLP%2BBP.pdf.

[23] Ian Goodfellow. *Deep Feedforward Networks*. [WWW]. URL: https://mnassar.github.io/deeplearninghandbook/slides/06_mlp.pdf.

[24] Andrew Ng and Tengyu Ma. *CS229 Lecture Notes*. [WWW]. URL: https://cs229.stanford.edu/main_notes.pdf.

[25] George Bebis and Michael Georgiopoulos. *OPTIMAL FEED-FORWARD NEURAL NETWORK ARCHITECTURES*. [WWW]. URL: https://www.cse.unr.edu/~bebis/optimal_arch.pdf.

[26] Zewen Li et al. *A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects*. [WWW]. URL: https://arxiv.org/pdf/2004.02806.

[27] Shuang Cong and Yang Zhou. *A review of convolutional neural network architectures and their optimizations*. [WWW]. URL: https://www.researchgate.net/profile/Shuang-Cong/publication/361477855_A_review_of_convolutional_neural_network_architectures_and_their_optimizations/links/663040e706ea3d0b7419945e/A-review-of-convolutional-neural-network-architectures-and-their-optimizations.pdf.

[28] Michael Nielsen. *Neural Networks and Deep Learning*. [WWW]. URL: https://aliosmangokcan.com/images/notes/yapay_sinir_agi_derin_ogrenme_pdf_ders_notu_e-book.pdf.

[29] Susmita Das et al. *Recurrent Neural Networks (RNNs): Architectures, Training Tricks, and Introduction to Influential Research*. [WWW]. URL: https://link.springer.com/protocol/10.1007/978-1-0716-3195-9_4.

[30] L.R. Medsker and L.C. Jain. *RECURRENT NEURAL NETWORKS Design and Applications*. [WWW]. URL: https://books.google.ee/books?hl=en&lr=&id=ME1SAkN0PyMC&oi=fnd&pg=PA1&dq=RECURRENT+NEURAL+NETWORKS+Design+and+Applications&ots=7dwxfS-USr&sig=k15Moqd8rR_fg20fZV0W2JT6hTk&redir_esc=y#v=onepage&q&f=false.

[31] NVIDIA Corporation. *Large Language Models*. [WWW]. URL: https://docs.nvidia.com/nemo-framework/user-guide/latest/nemotoolkit/nlp/nemo_megatron/intro.html.

[32] Colin Raffel et al. *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*. [WWW]. URL: https://www.jmlr.org/papers/volume21/20-074/20-074.pdf.

[33] Kenton Lee Jacob Devlin Ming-Wei Chang and Kristina Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. [WWW]. URL: https://aclanthology.org/N19-1423.pdf.

[34] Yinhan Liu et al. *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. [WWW]. URL: https://arxiv.org/pdf/1907.11692.

[35] Tim Salimans Alec Radford Karthik Narasimhan and Ilya Sutskever. *Improving Language Understanding by Generative Pre-Training*. [WWW]. URL: https://www.mikecaptain.com/resources/pdf/GPT-1.pdf.

[36] Tom Brown et al. *Language Models are Few-Shot Learners*. [WWW]. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

[37] Alec Radford et al. *Language Models are Unsupervised Multitask Learners*. [WWW]. URL: https://storage.prod.researchhub.com/uploads/papers/2020/06/01/language-models.pdf.

Appendix 1 – Non-Exclusive License for Reproduction and Publication of a Graduation Thesis[1]

I Joosep Lepland

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis "Diagrammatic Modeling of Neural Networks with Category Theory and String Diagrams", supervised by Pawel Maria Sobocinski and Niels Frits Willem Voorneveld
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

19.05.2025

[1]The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 - Info And Edit Parameter Pop-up Examples

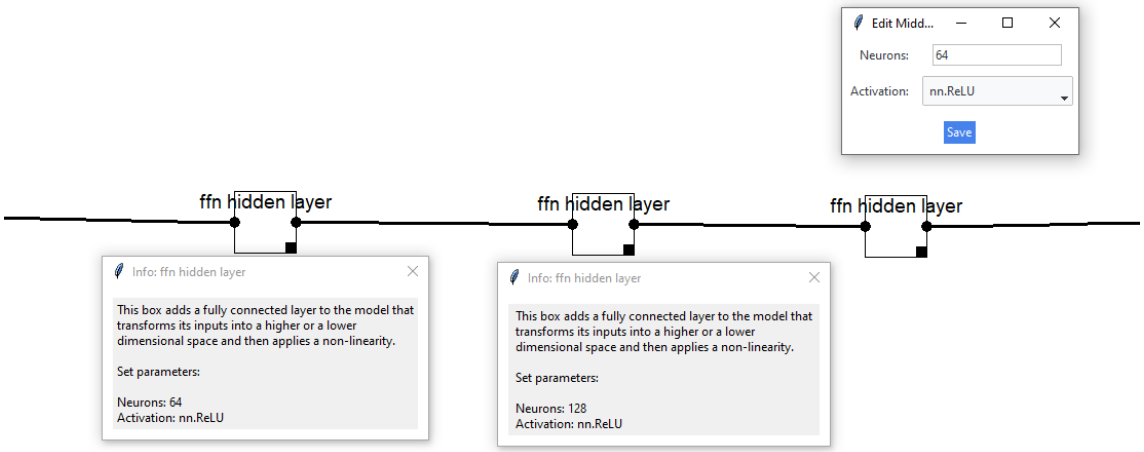


Figure 19. *Info And Edit Parameters Pop-up Example*

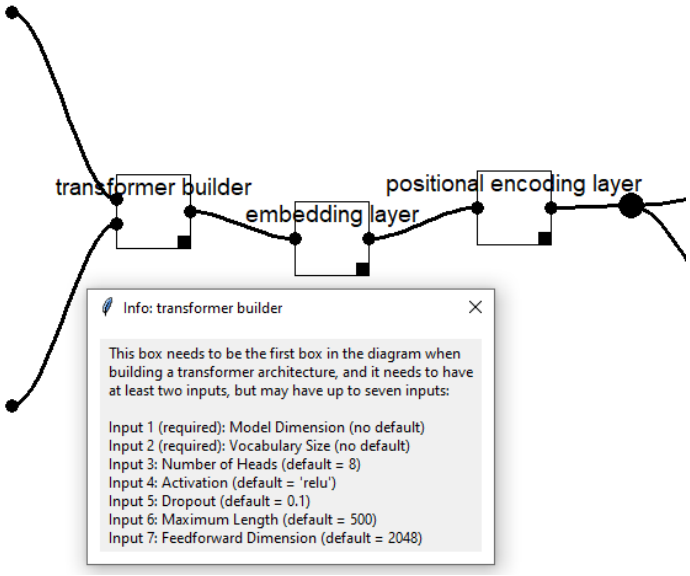


Figure 20. *Info Pop-up Example*

Appendix 3 - Transformer Fourth Split Sub-Diagrams

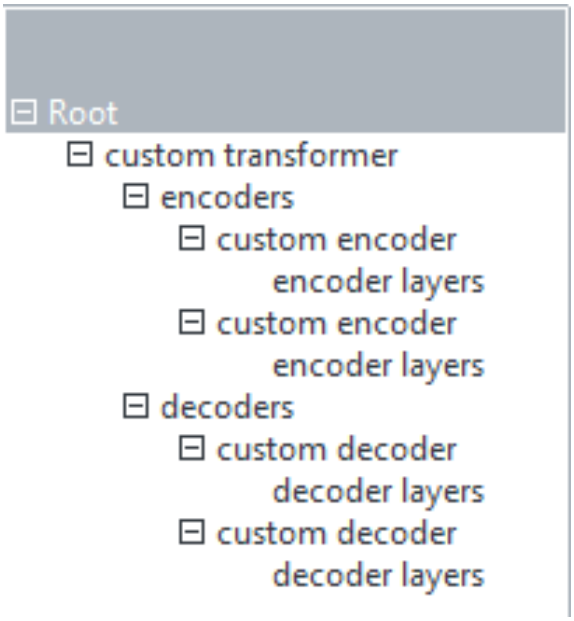


Figure 21. *Fourth Split Sub-Diagram Structure*

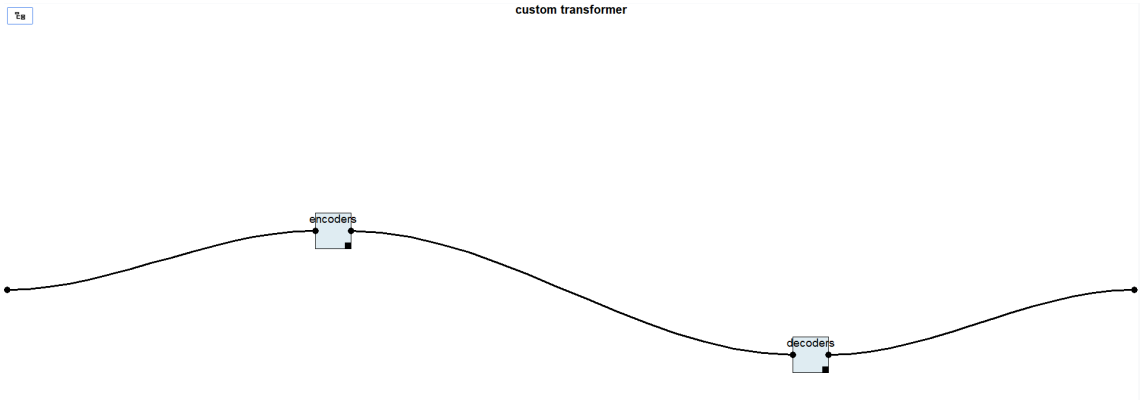


Figure 22. *Sub-Diagram Inside The Custom Transformer Box*

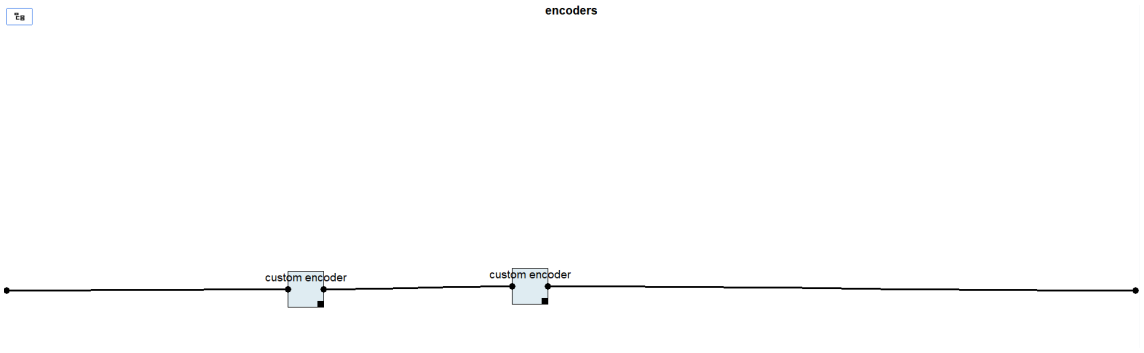


Figure 23. *Sub-Diagram Inside The Encoders Box*

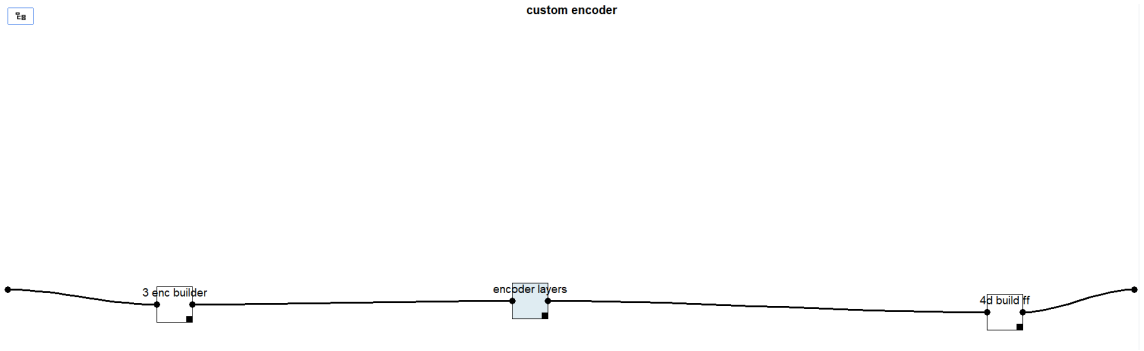


Figure 24. *Sub-Diagram Inside The Custom Encoder Box*

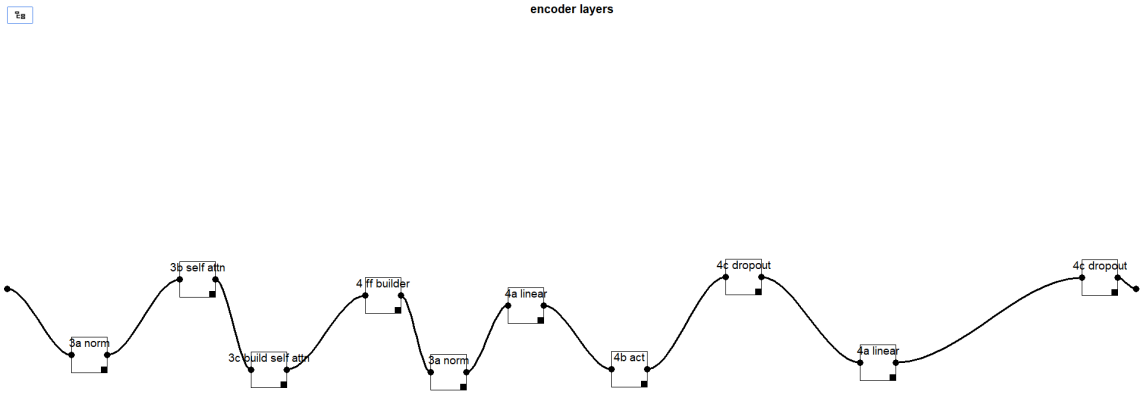


Figure 25. *Sub-Diagram Inside The Encoder Layers Box*

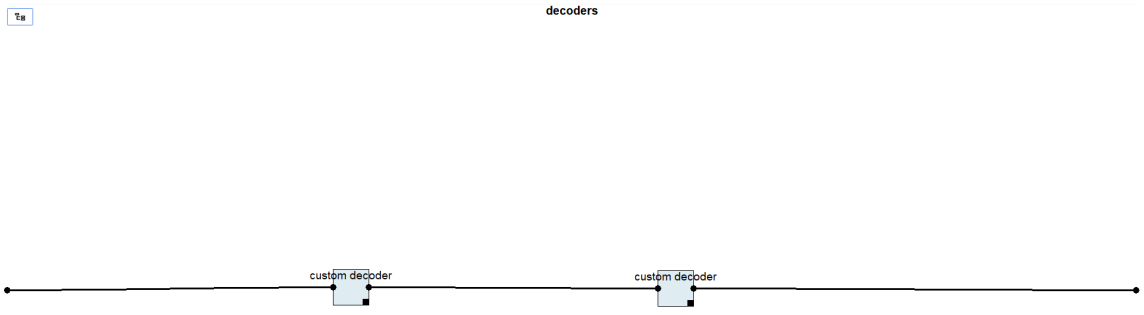


Figure 26. *Sub-Diagram Inside The Decoders Box*

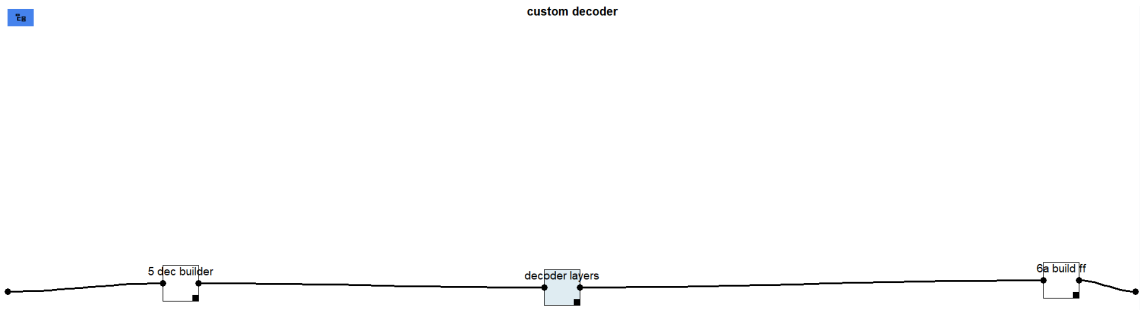


Figure 27. *Sub-Diagram Inside The Custom Decoder Box*

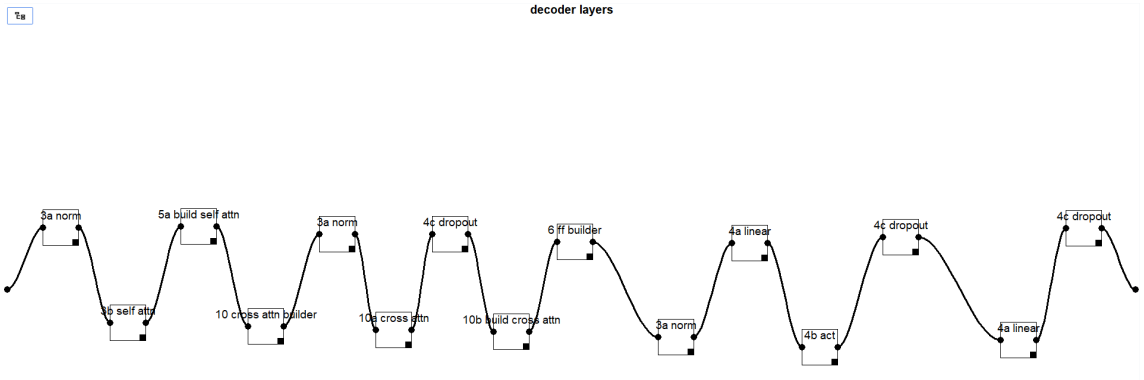


Figure 28. *Sub-Diagram Inside The Decoder Layers Box*

Appendix 4 - Transformer Fifth Split Sub-Diagrams

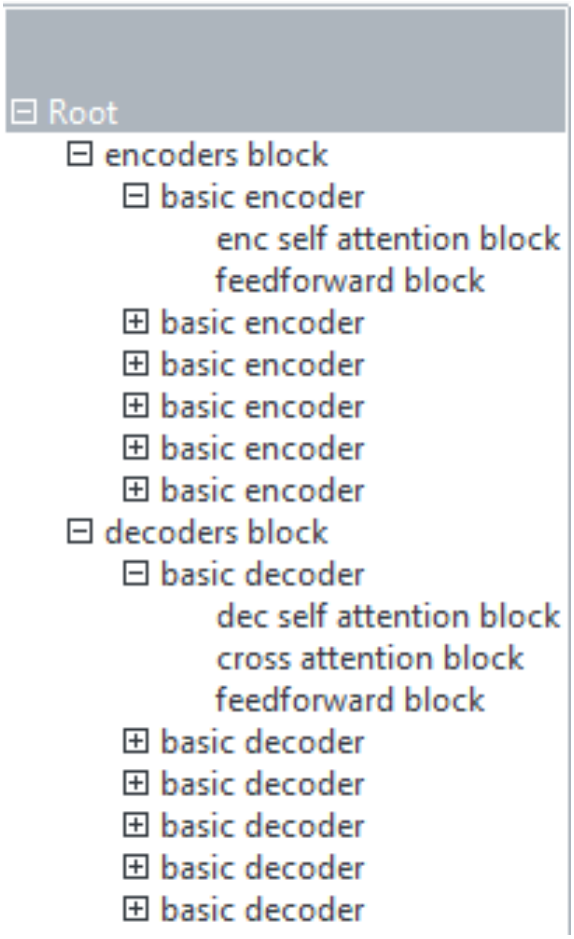


Figure 29. *Fifth Split Sub-Diagram Structure*

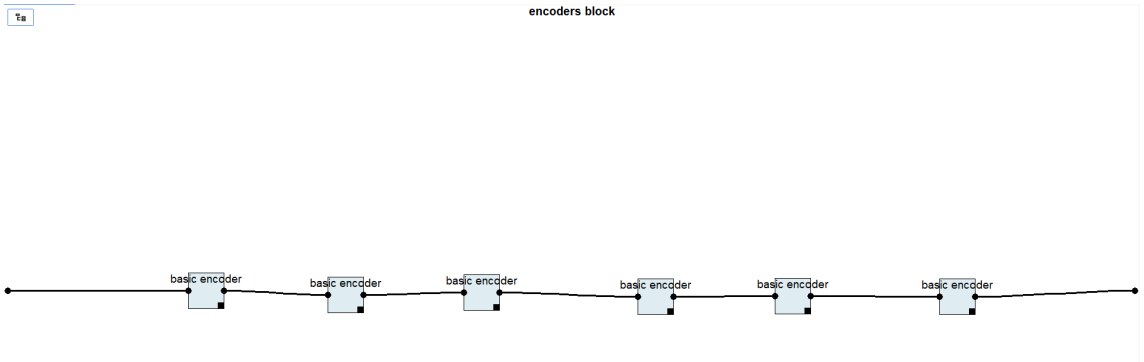


Figure 30. *Sub-Diagram Inside The Encoders Block Box*

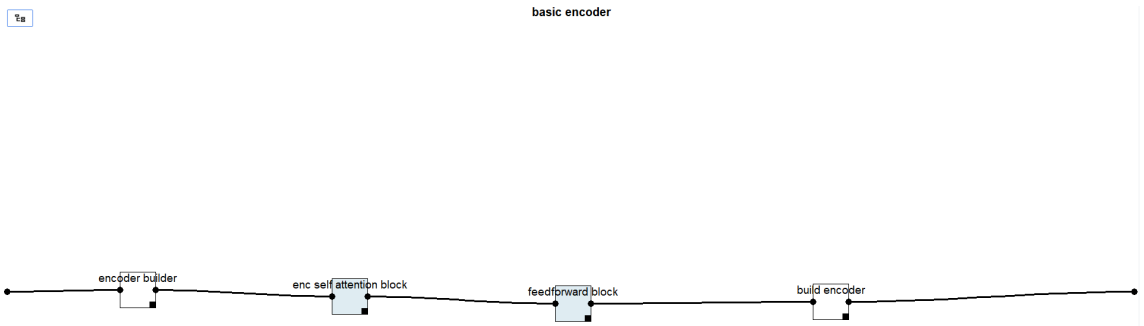


Figure 31. *Sub-Diagram Inside The Basic Encoder Box*

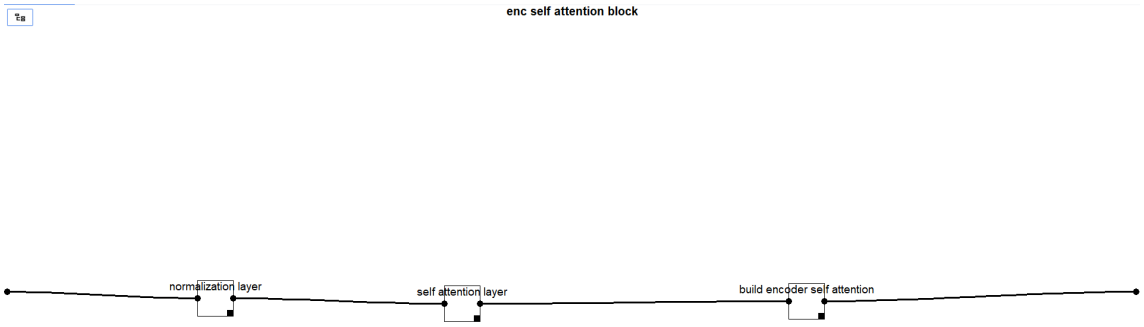


Figure 32. *Sub-Diagram Inside The Enc (Encoder) Self Attention Block Box*

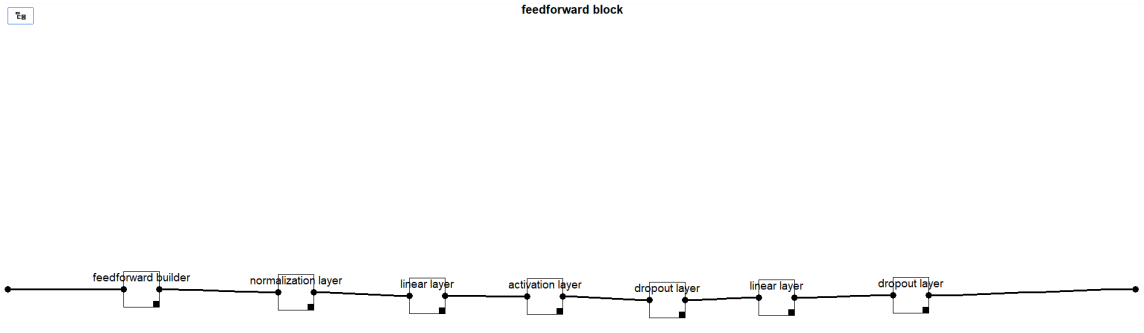


Figure 33. *Sub-Diagram Inside The Feedforward Block Box*

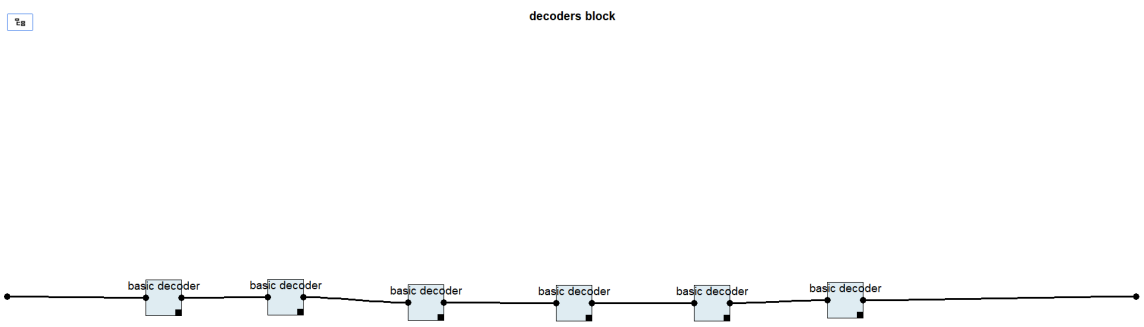


Figure 34. *Sub-Diagram Inside The Decoders Block Box*

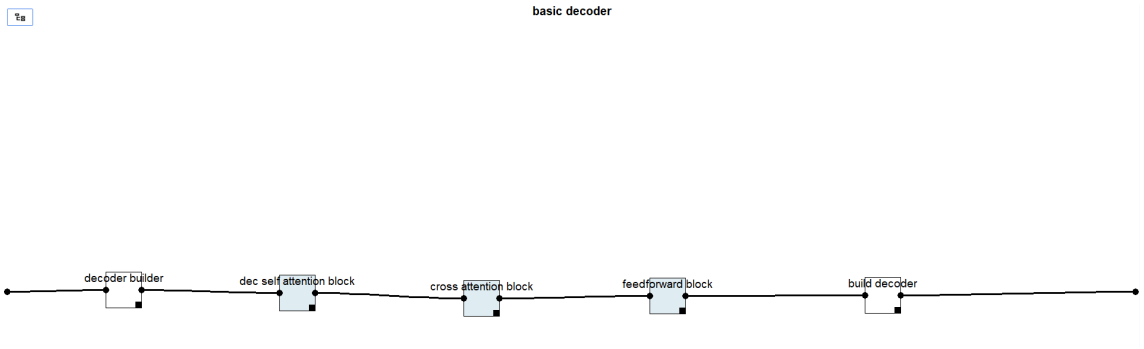


Figure 35. *Sub-Diagram Inside The Basic Decoder Box*

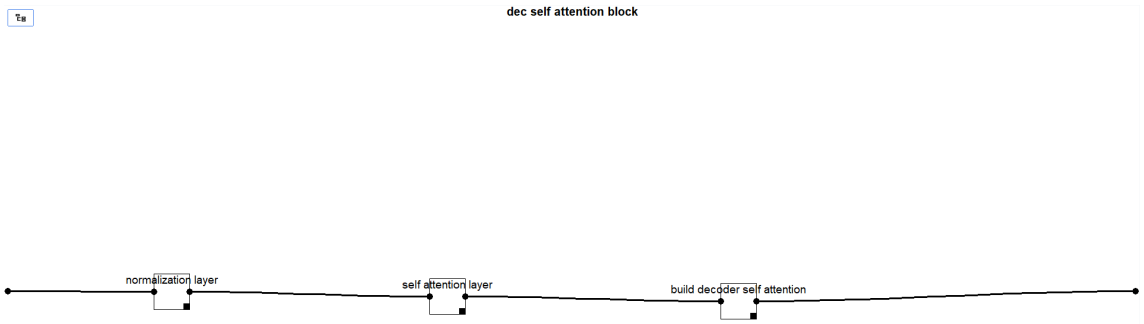


Figure 36. *Sub-Diagram Inside The Dec (Decoder) Self Attention Block Box*

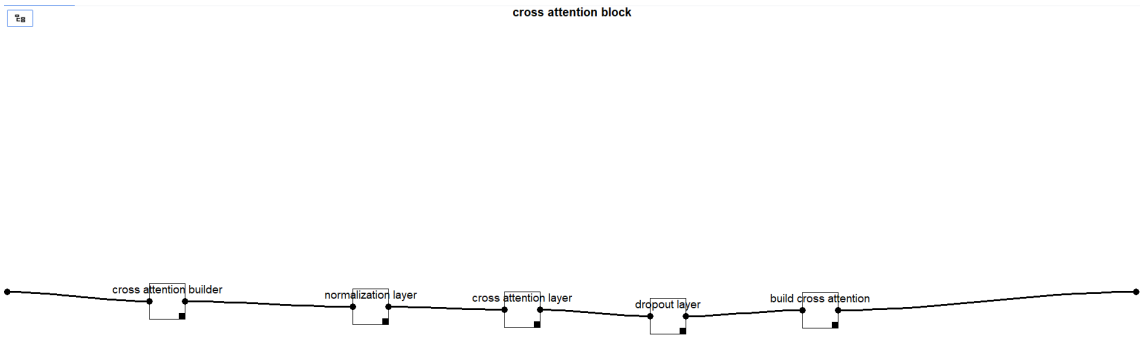


Figure 37. *Sub-Diagram Inside The Cross Attention Block Box*

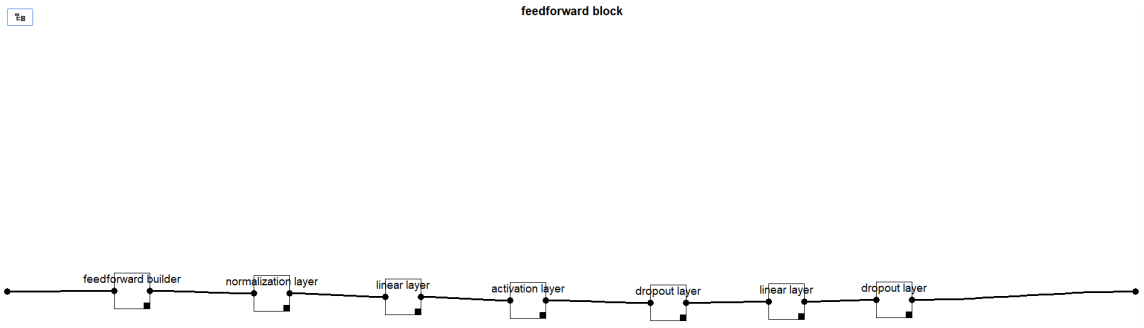


Figure 38. *Sub-Diagram Inside The Feedforward Block Box*

Appendix 5 - Transformer Fifth Split Encoder Only Architecture

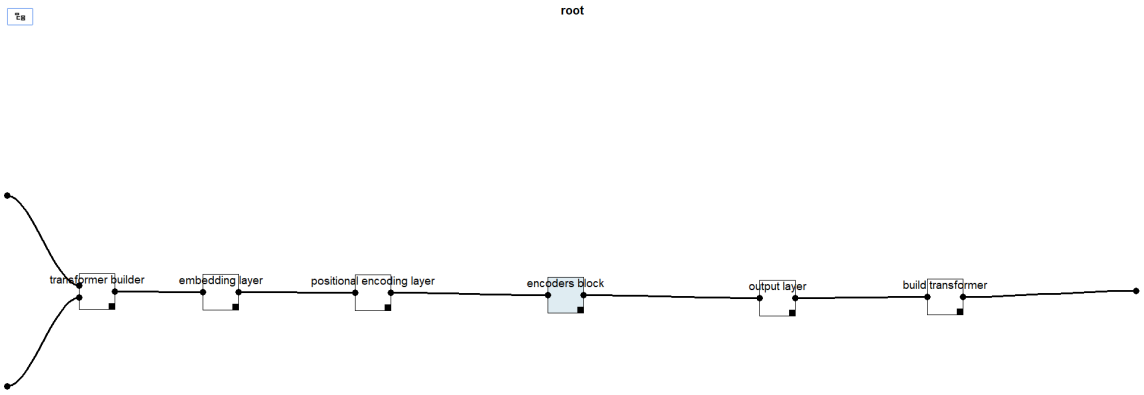


Figure 39. *Fifth Split Encoder Only Model*

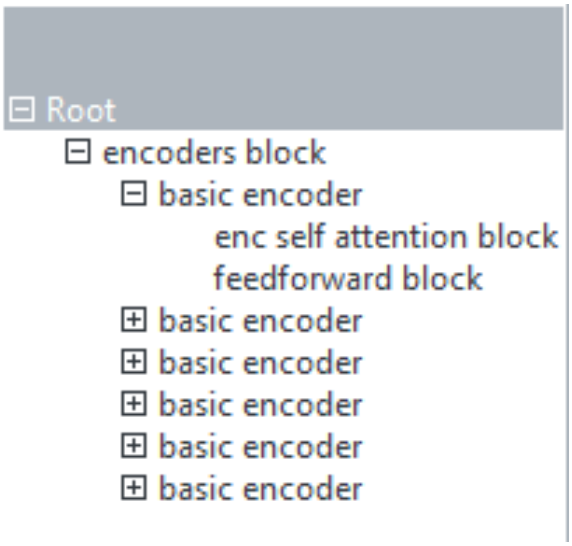


Figure 40. *Fifth Split Encoder Only Architecture Sub-Diagram Structure*

Appendix 6 - Transformer Fifth Split Decoder Only Architecture

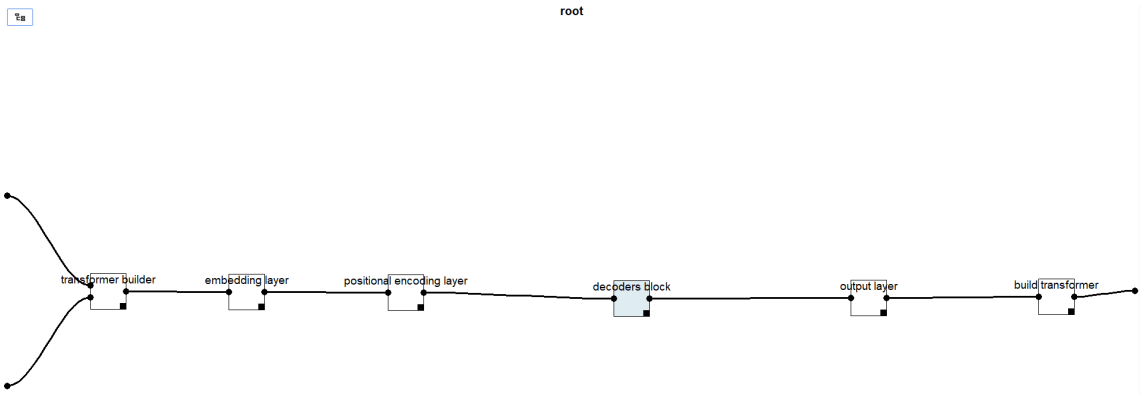


Figure 41. *Fifth Split Decoder Only Model*

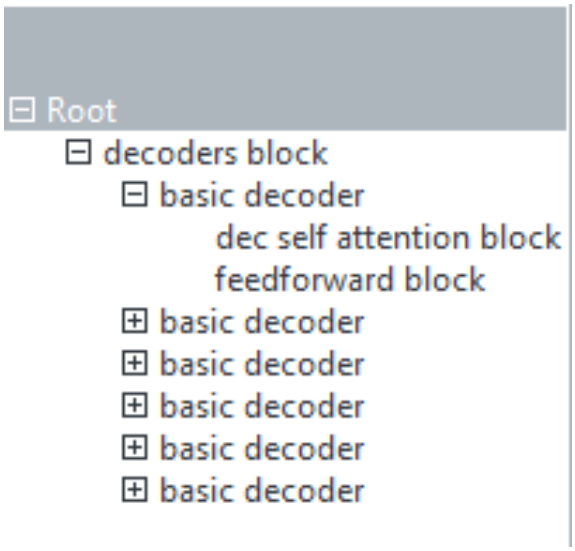


Figure 42. *Fifth Split Decoder Only Architecture Sub-Diagram Structure*

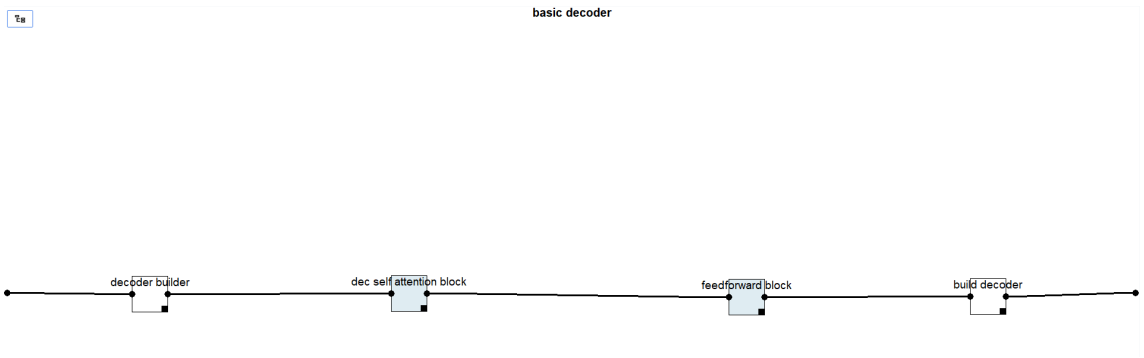


Figure 43. *Sub-Diagram Inside The Basic Decoder Box*

Appendix 7 - Further Information About Transformer Boxes

- Transformer Builder:
  - Inputs: Model dimension (an integer), vocabulary size (an integer), number of heads (an integer), activation (e.g., 'relu', 'gelu'), dropout (float between 0.0 and 1.0), maximum length (an integer), dimension of feedforward (an integer). All of these inputs can be given in as diagram inputs, and they should be connected to the transformer builder box. The model dimension and the vocabulary size inputs are mandatory, but others are not and can be set if the model needs specific parameters instead of the default values.
  - Purpose: This box creates the builder class for the transformer model that holds all of the parameters for other layers that come after this layer. But the main functionality of the transformer builder class is to hold the layers and layer stacks in the correct order so that when the transformer is built, the layers will be in the correct order. This box also holds other classes that support the creation of different layers.
  - Usage: This box needs to be the first box in the diagram when building a transformer architecture, and it needs to have at least two inputs, but may have up to seven inputs. This box is not used anywhere else in the diagram.
- Embedding Layer:
  - Inputs: Model dimension and vocabulary size are needed for this layer box. These values do not need to be set separately, but this box gets the values from the transformer builder box. The input wire that flows into this box should contain the builder itself.
  - Purpose: The embedding layer box adds the first layer to the architecture, which converts each input token (e.g., a word index) into a continuous vector. It converts each token ID into a fixed-size dense vector, and these embeddings serve as the input representations that the rest of the model can process. So this box holds a PyTorch Embedding module to supply meaningful numerical inputs to subsequent layers.
  - Usage: This box is used right after the transformer builder box, so this is the first actual layer of the model. By using this embedding layer box, the transformer DSL makes sure that raw token IDs are transformed into the proper vector format before any computation takes place.
- Positional Encoding Layer:

  - Inputs: Model dimension, dropout, and maximum length are needed for this layer, and these do not need to be set separately because this layer gets them also from the transformer builder box. The input wire that goes into this box should contain the builder.
  - Purpose: This box adds the positional encoding layer to the model. This positional encoding layer creates a unique pattern for each position in the input sequence. These position vectors are added to the token embeddings so that the transformer model knows the order of the tokens. So this box contains a PyTorch module that constructs and adds position encodings to the embeddings.
  - Usage: Positional encoding layer box is used after the embedding layer, and the transformer DSL ensures that sequential position information is added before any attention or feedforward operations take place.
- Encoder Builder:
  - Inputs: This box does not need any other inputs, just the wire that holds the builder.
  - Purpose: The purpose of this encoder builder box is to hold the encoder layer builder class, which holds the encoder layers in the position that they are added in the diagram, and when the encoder is built, it will be added to the transformer model encoder stack.
  - Usage: This box should be used at the start of an encoder block so that the following layers would be added in the correct order and the generated code would be correct as well. Each encoder block should start with this box, so for example, if the transformer architecture uses six encoder blocks, this box should also be used six times. This encoder builder box is used in the encoder-decoder architecture and the encoder-only architecture. This box marks the start of the encoder block.
- Normalization Layer:
  - Inputs: Model dimension or last layer output features, which is determined inside the box automatically based on the previous box, and the needed input is also added automatically, whichever one this box needs. Also, it needs the input wire that goes into this box from the previous box that holds the builder.
  - Purpose: The normalization layer box standardizes its inputs so each feature has zero mean and unit variance. This layer smooths out scale differences across hidden dimensions, and this helps stabilize and speed up the training process by keeping activations in a consistent range. This box contains a PyTorch LayerNorm module that normalizes each token's representation.
  - Usage: This normalization layer box is used in the encoder block and also in the decoder block. It can be used whenever the previous layer's outputs

need normalization. Transformer DSL makes sure that the model's inputs to attention and feedforward sublayers inside encoder and decoder blocks remain well conditioned, which improves the overall performance. The normalization layer box is used many times in most transformer architectures.

- Self-Attention Layer:
  - Inputs: Model dimension, number of heads, and dropout are the inputs that this layer box needs, but these do not need to be set separately because this layer gets them from a builder. The input wire that flows into this box should contain the builder.
  - Purpose: This self-attention layer box lets each token in the sequence look at every other token and decide how much to focus on each token. This layer computes attention scores between all pairs of input vectors and then uses those scores to take a weighted sum of the inputs. This computation produces new representations in which each position carries context from the entire sequence. This box uses the PyTorch MultiheadAttention module to project inputs into queries, keys, and values, computes a scaled dot-product (sum of the products of two vectors corresponding elements) attention across multiple heads in parallel, and then recombines the results.
  - Usage: This self-attention layer box is used in the encoder block and the decoder block as well, because both use self-attention. Which also means that this box is used in encoder-decoder, encoder-only, and decoder-only architectures, usually after the normalization layer box. The self-attention layer box is usually used once for each encoder and decoder block.
- Build Encoder Self-Attention:
  - Inputs: This box does not need any other inputs, just the input wire that holds the builder.
  - Purpose: Build encoder self-attention box, builds the self-attention block for the encoder block, and adds it to the encoder block. It takes all the layers that are added between the encoder builder box and this box, and creates the self-attention block with the exact layers that are added in the exact order.
  - Usage: This box is used in the encoder block to create the self-attention block. This means that if there is an encoder in the architecture and the encoder uses self-attention then this box is needed. This marks the end of the encoder self-attention block.
- Feedforward Builder:
  - Inputs: This box does not need any other inputs, just the input wire that holds the builder.
  - Purpose: Feedforward builder box main purpose is to add the feedforward block layers in the correct order to the model.

  - Usage: This box is used in the encoder and the decoder because both usually have the feedforward block inside of them. It is used to mark the start of the feedforward block, and the next layers are added to this block.
- Linear Layer:
  - Inputs: Model dimension or last layer output features, and dimension of feedforward. These values are assigned automatically from the builder, and if the previous layer output is needed, then it is taken from the last layer. Also, the input wire that holds the builder needs to be present.
  - Purpose: The linear layer box works like a regular feedforward dense layer, so that the network could reshape and reweight its representations as needed before passing them on to the next layer. This box typically takes the model dimension vectors and projects them up to a larger feedforward dimension, or down back to the original model dimension. It holds a PyTorch Linear module, which multiplies each input by a weight matrix and adds a bias.
  - Usage: This linear layer box is used in the feedforward block, and this block is used in both the encoder and decoder blocks. This means that the linear layer box is used in all of the main architectures (encoder-decoder, encoder-only, and decoder-only). The linear layer box is usually used many times in the feedforward block of the encoder and decoder blocks.
- Activation Layer:
  - Inputs: Activation, which is obtained from the builder automatically. The input wire should contain the builder itself.
  - Purpose: The activation layer box introduces nonlinearities into the model computations. This box applies a function such as ReLU or GELU to each element of its input vectors. By introducing nonlinearities, it enables the network to learn and represent more complex, nonlinear relationships in the data. This box holds a PyTorch activation module, like ReLU or GELU, which ensures that each transformed feature passes through the chosen nonlinear function before moving to the next layer.
  - Usage: This activation layer box is usually used after a linear transformation, which means that it is usually used after the linear layer box. It is mostly used in the feedforward block, which means that both the encoder and decoder use this box, and also all the different architectures use this box as well.
- Dropout Layer:
  - Inputs: Dropout, which is obtained from the builder automatically. Also, the input wire needs to contain the builder.
  - Purpose: The dropout layer box prevents the model from relying too heavily on any activation, reducing overfitting. This is done by randomly zeroing out a fraction of its input elements during training. This box uses the PyTorch

Dropout module, and it is given the dropout probability of dropping a unit.
  - Usage: This dropout layer box is used between other layers to improve generalization without changing the core model itself. This box is used in the feedforward block and cross-attention block, which means that this layer box is also used in most of the architectures to reduce overfitting.
- Build Encoder:
  - Inputs: Model dimension, which is obtained from the builder automatically. The input wire should contain the builder itself.
  - Purpose: The build encoder box builds the feedforward block, with the last layer output dimension as the model dimension, and adds both the self-attention block and feedforward block to the encoder builder. It builds the encoder itself with all the layers inside of it, mirroring the diagram representation. Also adds the built encoder block to the transformer's encoder stack, so this block would be used in the final transformer model architecture.
  - Usage: This box marks the end of the encoder block layers. So this means that after this box, should come another encoder builder box or encoders block are finished, and the decoder block starts. This build encoder box is used only in architectures where the encoder is present so in encoder-decoder and in encoder-only architectures.
- Decoder Builder:
  - Inputs: This box does not need any other inputs, just the wire that holds the builder.
  - Purpose: The main purpose of this decoder builder box is to hold the different decoder layers inside it so that a correctly ordered decoder block can be created based on the layer structure in the diagram. It holds the decoder builder class and also some other classes that are used to create cross-attention layers inside the decoder block.
  - Usage: This box should be used at the start of a decoder block so that the following layers would be added in the correct order and the generated code would be correct as well. If the transformer model, for example, uses six decoder blocks, then this box should be used six times as well. So this box marks the start of each decoder block.
- Build Decoder Self-Attention:
  - Inputs: This box does not need any other inputs, just the input wire that holds the builder.
  - Purpose: Build decoder self-attention box, builds the self-attention block for the decoder block, and adds it to the decoder block. It takes all the layers that are added between the decoder builder box and this box, and creates the self-attention block with the exact layers that are added in the exact order as in

the diagram.
  - Usage: This box is used in the decoder block to create the self-attention block. This means that if there is a decoder in the architecture and the decoder uses self-attention, like it usually does, then this box is needed. This marks the end of the decoder self-attention block.
- Cross Attention Builder:
  - Inputs: This box does not need any other inputs, just the input wire that holds the builder.
  - Purpose: This cross attention builder main purpose is to hold all the cross attention block layers in the correct order and compile this block correctly. This box also works together with the decoder builder box to create the cross-attention builder.
  - Usage: This box is used to start the cross attention block, and after this box comes layers that are in the cross attention block, until the block is built. This box is usually used after the decoder self-attention block, but the cross attention is not used for all the architectures. The cross-attention block is usually used only in the encoder-decoder architecture.
- Cross Attention Layer:
  - Inputs: Model dimension, number of heads, and dropout are the inputs that this layer box needs. These do not need to be set separately because this box gets them from a builder. The input wire that flows into this box should contain the builder.
  - Purpose: The cross-attention layer box lets one sequence attend to a different sequence, typically the encoder's outputs. It projects the decoder's inputs into queries, and the encoder's outputs into keys and values, then computes scaled dot-product attention to gather relevant encoder information for each decoder position. This box uses the PyTorch MultiheadAttention module so that queries come from the decoder and keys and values from the encoder. This enables the decoder to focus on the most relevant parts of the encoded input when generating output tokens.
  - Usage: This cross-attention layer box is used in the cross-attention block inside the decoder block itself. This layer box is the main part of the cross-attention block, and because the cross-attention block is usually only used in the encoder-decoder architecture, this means that this box is also usually used in only this architecture.
- Build Cross Attention:
  - Inputs: This box does not need any other inputs, just the input wire that holds the builder.
  - Purpose: This build cross attention box main purpose is to build and add the

cross-attention block to the decoder block structure. It takes all the layers that are between the cross-attention builder box and this box and creates the cross-attention block, mirroring the exact layer structure as in the diagram.
  - Usage: This box is used to mark the end of the cross-attention block in the decoder. This also means that this box is used only when the decoder is present and the decoder has a different sequence, like encoder's output to attend to. This box is usually only used in the encoder-decoder architecture.
- Build Decoder:
  - Inputs: Model dimension, which is obtained from the builder automatically. The input wire should contain the builder.
  - Purpose: The build decoder box builds the feedforward block, with the last layer output dimension as the model dimension, and adds the self-attention block, cross-attention block, and feedforward block to the decoder builder. This box builds the decoder block with all the layers in the same order as in the diagram. After the decoder is built, it also adds this block to the transformer builder so that this decoder block will be used in the final transformer architecture.
  - Usage: This box marks the end of the decoder block layers. This means that the build decoder box should be followed by another decoder builder box if the model has more decoder blocks, or should be followed by an output layer box. This box should be only used in the architectures where the decoder is present, so in encoder-decoder and decoder-only architectures.
- Output Layer:
  - Inputs: Model dimension and vocabulary size are the inputs needed for this box, but they are obtained automatically from the builder. The input wire should still contain the builder.
  - Purpose: The output layer box projects the decoder's final hidden vectors in encoder-decoder and decoder-only architectures, or the encoder's pooled representation in encoder-only architectures, into the desired output space. This box uses a PyTorch Linear module to map each vector to a set of logits. It gathers the last layer's outputs and turns them into predictions, which completes the model's forward pass.
  - Usage: This box should be used as the last layer of the transformer model architecture. This box should come after all of the encoder blocks in the encoder-only architecture, or after all of the decoder blocks in the encoder-decoder and decoder-only architectures.
- Build Transformer:
  - Inputs: This box does not need any other inputs, just the input wire that holds the builder.
  - Purpose: This build transformer box takes all the layers and blocks that are

added between the transformer builder box and this box, so all the layers and blocks overall, and compiles them together in the exact order as in the diagram. Then the transformer model is built and finished. The generated code will also use this box's output as the main function output, which is the compiled model. This box builds the corresponding transformer model based on which layers and boxes are added to the diagram.

- Usage: This box should always be the last box of the transformer model in the diagram, because this builds the actual model that is created in the diagram. After this box should be no more boxes, but the output wire of this box should be connected to the diagram output.

Appendix 8 - Encoder-Decoder Transformer Code Validation 1

```
encoder-decoder:
TransformerModel(
  (embedding): Embedding(1000, 512)
  (pos_encoding): PositionalEncoding(
    (dropout): Dropout(p=0.1, inplace=False)
  )
  (encoder_stack): ModuleList(
    (0-5): 6 x CustomEncoderLayer(
      (self_attn_pipeline): Pipeline(
        (blocks): ModuleList(
          (0): NormBlock(
            (norm): LayerNorm((512,), eps=1e-05,
                    elementwise_affine=True)
          )
          (1): ResidualWrapper(
            (block): SelfAttentionBlock(
              (self_attn): MultiheadAttention(
                (out_proj): NonDynamicallyQuantizableLinear(
                    in_features=512, out_features=512, bias=True)
              )
            )
          )
        )
      )
      (ff_pipeline): ResidualWrapper(
        (block): Pipeline(
          (blocks): ModuleList(
            (0): NormBlock(
              (norm): LayerNorm((512,), eps=1e-05,
                        elementwise_affine=True)
            )
            (1): LinearBlock(
              (linear): Linear(in_features=512, out_features=2048,
```

```
                                bias=True)
            )
            (2): ActivationBlock(
              (act): ReLU()
            )
            (3): DropoutBlock(
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (4): LinearBlock(
              (linear): Linear(in_features=2048, out_features=512,
                                bias=True)
            )
            (5): DropoutBlock(
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
        )
      )
    )
  )
  (decoder_stack): ModuleList(
    (0-5): 6 x CustomDecoderLayer(
      (self_attn_pipeline): Pipeline(
        (blocks): ModuleList(
          (0): NormBlock(
            (norm): LayerNorm((512,), eps=1e-05,
                        elementwise_affine=True)
          )
          (1): ResidualWrapper(
            (block): SelfAttentionBlock(
              (self_attn): MultiheadAttention(
                (out_proj): NonDynamicallyQuantizableLinear(
                    in_features=512, out_features=512, bias=True)
              )
            )
          )
        )
      )
      (cross_attn_pipeline): Pipeline(
```

```
  (blocks): ModuleList(
    (0): NormBlock(
      (norm): LayerNorm((512,), eps=1e-05,
                  elementwise_affine=True)
    )
    (1): ResidualWrapper(
      (block): CrossAttentionBlock(
        (cross_attn): MultiheadAttention(
          (out_proj): NonDynamicallyQuantizableLinear(
              in_features=512, out_features=512, bias=True)
        )
      )
    )
    (2): DropoutBlock(
      (dropout): Dropout(p=0.1, inplace=False)
    )
  )
)
(ff_pipeline): ResidualWrapper(
  (block): Pipeline(
    (blocks): ModuleList(
      (0): NormBlock(
        (norm): LayerNorm((512,), eps=1e-05,
                    elementwise_affine=True)
      )
      (1): LinearBlock(
        (linear): Linear(in_features=512, out_features=2048,
                         bias=True)
      )
      (2): ActivationBlock(
        (act): ReLU()
      )
      (3): DropoutBlock(
        (dropout): Dropout(p=0.1, inplace=False)
      )
      (4): LinearBlock(
        (linear): Linear(in_features=2048, out_features=512,
                         bias=True)
      )
```

```
            (5): DropoutBlock(
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
        )
      )
    )
    (output_head): OutputHead(
      (fc_out): Linear(in_features=512, out_features=1000, bias=True)
    )
)
```

Appendix 9 - Encoder-Decoder Transformer Code Validation 2

```
summary encdec:
==========================================================================================================================
Layer (type:depth-idx)                                  Output Shape              Param #
==========================================================================================================================
TransformerModel                                        [16, 4, 1000]             --
├─Embedding: 1-1                                        [16, 4, 512]              512,000
├─PositionalEncoding: 1-2                               [16, 4, 512]              --
│    └─Dropout: 2-1                                     [16, 4, 512]              --
├─Embedding: 1-3                                        [16, 4, 512]              (recursive)
├─PositionalEncoding: 1-4                               [16, 4, 512]              --
│    └─Dropout: 2-2                                     [16, 4, 512]              --
├─ModuleList: 1-5                                       --                        --
│    └─CustomEncoderLayer: 2-3                          [16, 4, 512]              --
│    │    └─Pipeline: 3-1                               [16, 4, 512]              1,051,648
│    │    └─ResidualWrapper: 3-2                        [16, 4, 512]              2,100,736
│    └─CustomEncoderLayer: 2-4                          [16, 4, 512]              --
│    │    └─Pipeline: 3-3                               [16, 4, 512]              1,051,648
│    │    └─ResidualWrapper: 3-4                        [16, 4, 512]              2,100,736
│    └─CustomEncoderLayer: 2-5                          [16, 4, 512]              --
│    │    └─Pipeline: 3-5                               [16, 4, 512]              1,051,648
│    │    └─ResidualWrapper: 3-6                        [16, 4, 512]              2,100,736
│    └─CustomEncoderLayer: 2-6                          [16, 4, 512]              --
│    │    └─Pipeline: 3-7                               [16, 4, 512]              1,051,648
│    │    └─ResidualWrapper: 3-8                        [16, 4, 512]              2,100,736
│    └─CustomEncoderLayer: 2-7                          [16, 4, 512]              --
│    │    └─Pipeline: 3-9                               [16, 4, 512]              1,051,648
│    │    └─ResidualWrapper: 3-10                       [16, 4, 512]              2,100,736
│    └─CustomEncoderLayer: 2-8                          [16, 4, 512]              --
│    │    └─Pipeline: 3-11                              [16, 4, 512]              1,051,648
│    │    └─ResidualWrapper: 3-12                       [16, 4, 512]              2,100,736
└─ModuleList: 1-6                                       --                        --
```

Figure 44. *Validation Script Output Part 2.1*

```
├─ModuleList: 1-6                                  --                        --
│    └─CustomDecoderLayer: 2-9                     [16, 4, 512]              --
│    │    └─Pipeline: 3-13                         [16, 4, 512]              1,051,648
│    │    └─Pipeline: 3-14                         [16, 4, 512]              1,051,648
│    │    └─ResidualWrapper: 3-15                  [16, 4, 512]              2,100,736
│    └─CustomDecoderLayer: 2-10                    [16, 4, 512]              --
│    │    └─Pipeline: 3-16                         [16, 4, 512]              1,051,648
│    │    └─Pipeline: 3-17                         [16, 4, 512]              1,051,648
│    │    └─ResidualWrapper: 3-18                  [16, 4, 512]              2,100,736
│    └─CustomDecoderLayer: 2-11                    [16, 4, 512]              --
│    │    └─Pipeline: 3-19                         [16, 4, 512]              1,051,648
│    │    └─Pipeline: 3-20                         [16, 4, 512]              1,051,648
│    │    └─ResidualWrapper: 3-21                  [16, 4, 512]              2,100,736
│    └─CustomDecoderLayer: 2-12                    [16, 4, 512]              --
│    │    └─Pipeline: 3-22                         [16, 4, 512]              1,051,648
│    │    └─Pipeline: 3-23                         [16, 4, 512]              1,051,648
│    │    └─ResidualWrapper: 3-24                  [16, 4, 512]              2,100,736
│    └─CustomDecoderLayer: 2-13                    [16, 4, 512]              --
│    │    └─Pipeline: 3-25                         [16, 4, 512]              1,051,648
│    │    └─Pipeline: 3-26                         [16, 4, 512]              1,051,648
│    │    └─ResidualWrapper: 3-27                  [16, 4, 512]              2,100,736
│    └─CustomDecoderLayer: 2-14                    [16, 4, 512]              --
│    │    └─Pipeline: 3-28                         [16, 4, 512]              1,051,648
│    │    └─Pipeline: 3-29                         [16, 4, 512]              1,051,648
│    │    └─ResidualWrapper: 3-30                  [16, 4, 512]              2,100,736
├─OutputHead: 1-7                                  [16, 4, 1000]             --
│    └─Linear: 2-15                                [16, 4, 1000]             513,000
==========================================================================================================
Total params: 45,163,496
Trainable params: 45,163,496
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 428.23
==========================================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 24.63
Params size (MB): 105.01
Estimated Total Size (MB): 129.64
==========================================================================================================
forward-shape OK
```

Figure 45. *Validation Script Output Part 2.2*

Appendix 10 - Encoder-Only Transformer Code Validation 1

```
encoder-only:
TransformerModel(
  (embedding): Embedding(1000, 512)
  (pos_encoding): PositionalEncoding(
    (dropout): Dropout(p=0.1, inplace=False)
  )
  (encoder_stack): ModuleList(
    (0-5): 6 x CustomEncoderLayer(
      (self_attn_pipeline): Pipeline(
        (blocks): ModuleList(
          (0): NormBlock(
            (norm): LayerNorm((512,), eps=1e-05,
                    elementwise_affine=True)
          )
          (1): ResidualWrapper(
            (block): SelfAttentionBlock(
              (self_attn): MultiheadAttention(
                (out_proj): NonDynamicallyQuantizableLinear(
                    in_features=512, out_features=512, bias=True)
              )
            )
          )
        )
      )
      (ff_pipeline): ResidualWrapper(
        (block): Pipeline(
          (blocks): ModuleList(
            (0): NormBlock(
              (norm): LayerNorm((512,), eps=1e-05,
                    elementwise_affine=True)
            )
            (1): LinearBlock(
              (linear): Linear(in_features=512, out_features=2048,
```

```
                                bias=True)
              )
              (2): ActivationBlock(
                (act): ReLU()
              )
              (3): DropoutBlock(
                (dropout): Dropout(p=0.1, inplace=False)
              )
              (4): LinearBlock(
                (linear): Linear(in_features=2048, out_features=512,
                                bias=True)
              )
              (5): DropoutBlock(
                (dropout): Dropout(p=0.1, inplace=False)
              )
            )
          )
        )
      )
    )
    (decoder_stack): ModuleList()
    (output_head): OutputHead(
      (fc_out): Linear(in_features=512, out_features=1000, bias=True)
    )
)
```

Appendix 11 - Encoder-Only Transformer Code Validation 2

```
summary enc-only:
==========================================================================================================================
Layer (type:depth-idx)                                   Output Shape              Param #
==========================================================================================================================
TransformerModel                                         [16, 4, 1000]             --
├─Embedding: 1-1                                         [16, 4, 512]              512,000
├─PositionalEncoding: 1-2                                [16, 4, 512]              --
│    └─Dropout: 2-1                                      [16, 4, 512]              --
├─ModuleList: 1-3                                        --                        --
│    └─CustomEncoderLayer: 2-2                           [16, 4, 512]              --
│    │    └─Pipeline: 3-1                                [16, 4, 512]              1,051,648
│    │    └─ResidualWrapper: 3-2                         [16, 4, 512]              2,100,736
│    └─CustomEncoderLayer: 2-3                           [16, 4, 512]              --
│    │    └─Pipeline: 3-3                                [16, 4, 512]              1,051,648
│    │    └─ResidualWrapper: 3-4                         [16, 4, 512]              2,100,736
│    └─CustomEncoderLayer: 2-4                           [16, 4, 512]              --
│    │    └─Pipeline: 3-5                                [16, 4, 512]              1,051,648
│    │    └─ResidualWrapper: 3-6                         [16, 4, 512]              2,100,736
│    └─CustomEncoderLayer: 2-5                           [16, 4, 512]              --
│    │    └─Pipeline: 3-7                                [16, 4, 512]              1,051,648
│    │    └─ResidualWrapper: 3-8                         [16, 4, 512]              2,100,736
│    └─CustomEncoderLayer: 2-6                           [16, 4, 512]              --
│    │    └─Pipeline: 3-9                                [16, 4, 512]              1,051,648
│    │    └─ResidualWrapper: 3-10                        [16, 4, 512]              2,100,736
│    └─CustomEncoderLayer: 2-7                           [16, 4, 512]              --
│    │    └─Pipeline: 3-11                               [16, 4, 512]              1,051,648
│    │    └─ResidualWrapper: 3-12                        [16, 4, 512]              2,100,736
├─OutputHead: 1-4                                        [16, 4, 1000]             --
│    └─Linear: 2-8                                       [16, 4, 1000]             513,000
==========================================================================================================================
Total params: 19,939,304
Trainable params: 19,939,304
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 218.17
==========================================================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 11.78
Params size (MB): 54.54
Estimated Total Size (MB): 66.33
==========================================================================================================================
encoder-only forward-shape OK
```

Figure 46. *Validation Script Output*

Appendix 12 - Decoder-Only Transformer Code Validation 1

```
decoder-only:
TransformerModel(
  (embedding): Embedding(1000, 512)
  (pos_encoding): PositionalEncoding(
    (dropout): Dropout(p=0.1, inplace=False)
  )
  (encoder_stack): ModuleList()
  (decoder_stack): ModuleList(
    (0-5): 6 x CustomDecoderLayer(
      (self_attn_pipeline): Pipeline(
        (blocks): ModuleList(
          (0): NormBlock(
            (norm): LayerNorm((512,), eps=1e-05,
                    elementwise_affine=True)
          )
          (1): ResidualWrapper(
            (block): SelfAttentionBlock(
              (self_attn): MultiheadAttention(
                (out_proj): NonDynamicallyQuantizableLinear(
                    in_features=512, out_features=512, bias=True)
              )
            )
          )
        )
      )
      (ff_pipeline): ResidualWrapper(
        (block): Pipeline(
          (blocks): ModuleList(
            (0): NormBlock(
              (norm): LayerNorm((512,), eps=1e-05,
                        elementwise_affine=True)
            )
            (1): LinearBlock(
```

```
            (linear): Linear(in_features=512, out_features=2048,
                                bias=True)
          )
          (2): ActivationBlock(
            (act): ReLU()
          )
          (3): DropoutBlock(
            (dropout): Dropout(p=0.1, inplace=False)
          )
          (4): LinearBlock(
            (linear): Linear(in_features=2048, out_features=512,
                                bias=True)
          )
          (5): DropoutBlock(
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
  )
  (output_head): OutputHead(
    (fc_out): Linear(in_features=512, out_features=1000, bias=True)
  )
)
```

Appendix 13 - Decoder-Only Transformer Code Validation 2

```
summary dec-only:
==========================================================================================================================
Layer (type:depth-idx)                                  Output Shape              Param #
==========================================================================================================================
TransformerModel                                        [16, 4, 1000]             --
├─Embedding: 1-1                                        [16, 4, 512]              512,000
├─PositionalEncoding: 1-2                               [16, 4, 512]              --
│    └─Dropout: 2-1                                     [16, 4, 512]              --
├─ModuleList: 1-3                                       --                        --
│    └─CustomDecoderLayer: 2-2                          [16, 4, 512]              --
│    │    └─Pipeline: 3-1                               [16, 4, 512]              1,051,648
│    │    └─ResidualWrapper: 3-2                        [16, 4, 512]              2,100,736
│    └─CustomDecoderLayer: 2-3                          [16, 4, 512]              --
│    │    └─Pipeline: 3-3                               [16, 4, 512]              1,051,648
│    │    └─ResidualWrapper: 3-4                        [16, 4, 512]              2,100,736
│    └─CustomDecoderLayer: 2-4                          [16, 4, 512]              --
│    │    └─Pipeline: 3-5                               [16, 4, 512]              1,051,648
│    │    └─ResidualWrapper: 3-6                        [16, 4, 512]              2,100,736
│    └─CustomDecoderLayer: 2-5                          [16, 4, 512]              --
│    │    └─Pipeline: 3-7                               [16, 4, 512]              1,051,648
│    │    └─ResidualWrapper: 3-8                        [16, 4, 512]              2,100,736
│    └─CustomDecoderLayer: 2-6                          [16, 4, 512]              --
│    │    └─Pipeline: 3-9                               [16, 4, 512]              1,051,648
│    │    └─ResidualWrapper: 3-10                       [16, 4, 512]              2,100,736
│    └─CustomDecoderLayer: 2-7                          [16, 4, 512]              --
│    │    └─Pipeline: 3-11                              [16, 4, 512]              1,051,648
│    │    └─ResidualWrapper: 3-12                       [16, 4, 512]              2,100,736
├─OutputHead: 1-4                                       [16, 4, 1000]             --
│    └─Linear: 2-8                                      [16, 4, 1000]             513,000
==========================================================================================================================
Total params: 19,939,304
Trainable params: 19,939,304
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 218.17
==========================================================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 11.78
Params size (MB): 54.54
Estimated Total Size (MB): 66.33
==========================================================================================================================
decoder-only forward-shape OK
```

Figure 47. *Validation Script Output*

Appendix 14 - Encoder-Decoder Transformer Better Flow Representation

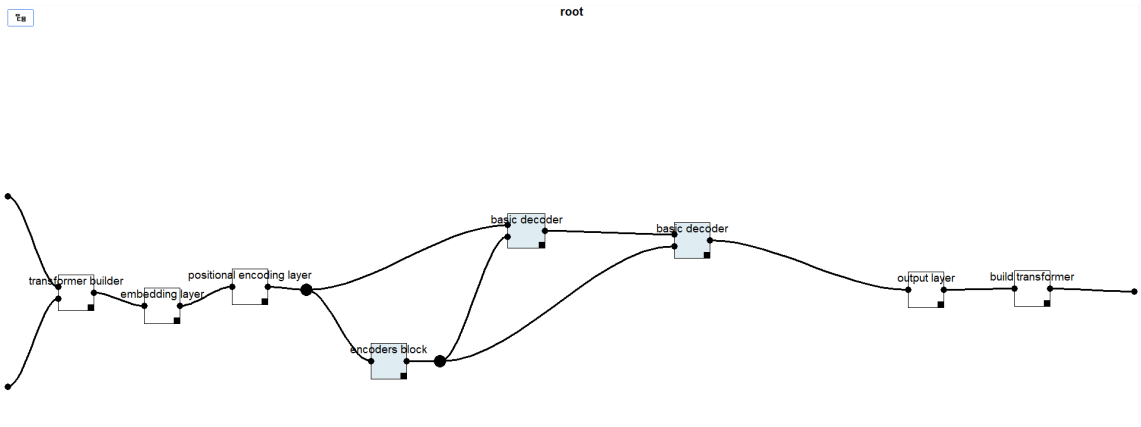


Figure 48. *Transformer Better Data Flow Representation*

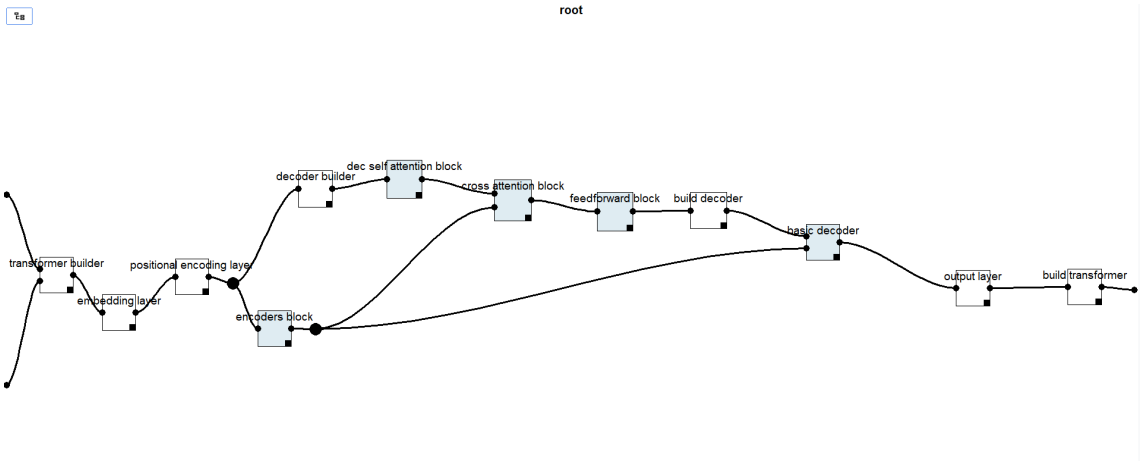


Figure 49. *One Basic Decoder Block Sub-Diagram Unfolded*