DNNCASE – Visual Programming for Deep Learning Workflows

Report submitted in partial fulfillment of requirements for the B.Tech. degree in Computer Science and Engineering

By

Gurkaran Singh	*2020UCO1612*
Rohan Sharma	*2020UCO1617*
Rohan Ray	*2020UCO1618*

Under the supervision of

Dr. Shampa Chakraverty

Department of Computer Science and Engineering

Netaji Subhas University of Technology (NSUT)

New Delhi, India – 110078

May 2024

Certificate

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

This is to certify that the work embodied in the project thesis titled, “DNNCASE – Visual Programming for Deep Learning Workflows” by *Gurkaran Singh (2020UCO1612), Rohan Sharma (2020UCO1617) and Rohan Ray (2020UCO1618)* is the bonafide work of the group submitted to **Netaji Subhas University of Technology** for consideration in 8th Semester B. Tech. Project Evaluation.

The original research work was carried out by the team under my guidance and supervision in the academic year 2023 – 2024. This work has not been submitted for any other diploma or degree from any university. On the basis of the declaration made by the group, I recommend the project report for evaluation.

Dr. Shampa Chakraverty

Professor

Department of Computer Science and Engineering

Netaji Subhas University of Technology

Candidates' Declaration

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

We, *Gurkaran Singh(2020UCO1612), Rohan Sharma(20202UCO1617) and Rohan Ray(2020UCO1618)* of B.Tech. Department of Computer Science and Engineering, hereby declare that the project thesis titled "**DNNCASE – Visual Programming for Deep Learning Workflows**" which is submitted by us to the Department of Computer Science and Engineering, Netaji Subhas University of Technology (NSUT), Dwarka, New Delhi in partial fulfillment of the requirement for the award of the degree of Bachelor of Technology in original and not copied from any source without proper citation. The manuscript has been subjected to plagiarism check by Turnitin software. This work has not previously formed the basis for award of any Degree.

Place: New Delhi

Date:

Gurkaran Singh	**Rohan Sharma**	**Rohan Ray**
2020UCO1612	*2020UCO1617*	*2020UCO1618*

Certificate of Declaration

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

This is to certify that the project thesis titled "**DNNCASE – Visual Programming for Deep Learning Workflows**" which is being submitted by *Gurkaran Singh (2020UCO1612), Rohan Sharma (2020UCO1617) and Rohan Ray (2020UCO1618)* to the Department of Computer Science and Engineering, Netaji Subhas University of Technology (NSUT) Dwarka, New Delhi in partial fulfillment of the requirement for the award of the degree of Bachelor of Technology, is a record of the thesis work carried out by the students under my supervision and guidance. The content of this thesis, in full or in parts, has not been submitted for any other degree or diploma.

Place:

Date:

Dr. Shampa Chakraverty

Professor

Department of Computer Science and Engineering

Netaji Subhas University of Technology

Acknowledgement

We would like to express our gratitude and appreciation to all those who made it possible to complete this project. Special thanks to our project supervisor, Dr. Shampa Chakraverty whose help, stimulating suggestions and encouragement helped us in writing this report. We also sincerely thank our colleagues for the time spent proofreading and correcting our mistakes.

We would also like to acknowledge with much appreciation the crucial role of the staff of the Department of Computer Science and Engineering, who gave us the permission to use a lab and necessary systems.

Gurkaran Singh	**Rohan Sharma**	**Rohan Ray**
2020UCO1612	*2020UCO1617*	*2020UCO1618*

Abstract

Technology is an essential component of the functioning of our world. It enables us to perform various tasks to enhance our productivity. However, the huge amounts of data generated from the use of technology are considerably complex. To understand the processes that are at work in any system, various innovative modeling techniques are employed. Neural networks are one data modeling method that is not only highly flexible but also enjoys widespread use. To create these networks, various frameworks such as Keras have been developed. While these frameworks are a step towards an easier programming interface, there still remain difficulties in creating and visualizing complex architectures. These difficulties are especially pronounced in the case of developers who may have a weak understanding of deep learning techniques. Many recent advances in deep learning have motivated practitioners to experiment with various network architectures in different domains.

We describe a graphical language to depict neural network architectures on a canvas and generate corresponding code. We expand the capabilities of our system, by incorporating other deep learning workflows such as training, tuning, testing, etc. We achieve this by defining a visual language based on the dataflow and functional paradigms, and by creating an associated library for deep learning in that language.

KEYWORDS: *Deep Neural Network; Code generation; Weight Sharing; Keras; Visual Language; Education; Teaching Aids*

Table of Contents

List of Figures

Nomenclature and Abbreviation

CASE - Computer-Aided Software Engineering

DAG – Directed Acyclic Graph

DL – Deep Learning

DNN – Deep Neural Network

FBP – Flow Based Programming

FR – Functional Requirements

GUI – Graphic User Interface

HACUFS[1] – Hierarchical Agglomerative Clustering of Unbounded Feature Space for requirements elicitation

IDE – Integrated Development Environment

IPC – Inter Process Communication

ML – Machine Learning

NFR – Non - Functional Requirements

[1] See Chapter 4 and Appendix C

Chapter 1: Introduction

The functioning of the modern world is highly coupled with the use of technology. This has led to a lot of data being generated every day [1]. These data are collected from various processes with unknown mechanisms. These unknown mechanisms are a major concern for the stakeholders of their respective domains [2].

Various modeling techniques can be used to understand the data and make predictions from them. A lot of the data is unstructured [3]. There are various tasks done on such data, using neural networks among other algorithms.

Keras [4] is a popular and widely used framework for model construction and training. It is an API for the TensorFlow, JAX, and PyTorch libraries that commonly is used with the Python programming language. It aims to provide a flexible interface for modern ML problems, especially deep learning.

Although such tools provide ease of programmability when coding neural network architectures, there still are difficulties in creating and visualizing complex architectures. These difficulties are especially pronounced in the case of developers who may have a weak understanding of deep learning techniques. [5]

Recently, there have been many advances in deep learning research, and this has been a motivation for various developers and practitioners to come up and experiment with various network architectures in different domains. In this work, we have created a CASE (computer-aided software engineering) tool that allows users to create complex, modular, and wide-ranging deep neural network architectures. It will provide a canvas for creating the architecture diagrams and specifying schematics of associated workflows (such as training mechanisms). It also provides the python code of the corresponding deep neural network architecture made by the user.

We have created a visual programming language which uses the concepts of functional and dataflow programming languages to help the user create architectures for neural network using fundamental entities such as datatypes, function, if-else condition, loops and callbacks. The user can also train, test and perform hyperparameter tuning on their architecture using this language.

We have created a compiler that can generate python code for the language schematics. The compiler and basic yet scalable design allows our language to be used to design systems other than neural networks. We have also created a library focused on deep learning in this language.

Our Motivation

We wish to enable ML/DL professionals to focus on algorithmic complexities rather than grappling with coding intricacies, thus accelerating model development. One of our major goals in the development of this system is to allow users to model a neural network in the same way they would draw it on a whiteboard or notebook. By providing a canvas to the users, we can reduce the mismatch between the perceived understanding of a neural network by the user, and its specification in the system.

Architectural modifications will become straightforward, enabling agile experimentation and optimization. Different neural network architectures can include repeated sub-units. These present an opportunity for code reuse. The user should be able to specify a 'blueprint' of a single sub-unit and use it in various positions in a neural network diagram.

Moreover, currently, to the best of our knowledge, no graphical language exists that supports all possible network architectures and training workflows straightforwardly. An analysis is necessary to understand various constructs that such a language will need for a graphical description, given that those constructs may seem commonplace and simplistic when writing equivalent code.

By removing the programming obstacle, we will be able to make deep learning more accessible to school students and people from non-technical backgrounds. They will be able to understand and visualize the concepts of neural networks easily and quickly, without the need to first learn programming.

Chapter 2: Literature Survey and Statement of Problem

Lee et al. in their paper titled ‘A Deep Learning Model Generation Method for Code Reuse and Automatic Machine Learning’ [5] outline the need for a tool that can be used to develop and train deep neural networks. They focus on an autonomous approach to training neural networks, including autonomic hyperparameter tuning. Their tool allows users to train a model from within the system, but utilizing remote computing resources.

In their paper, they outline various difficulties in using the prevalent ML techniques, which include architecture selection and hyperparameter tuning. Because of this, their system has been centered around an easy-to-use and intuitive interface, with common functionality in-build and the complexities abstracted.

Their paper also refers to the various autonomicity levels [6] and associated trade-offs that arise as a result of making systems more and more autonomic. One of the major aspects outlined is that a more autonomic system places a burden on computing resources because of a large search space, while a lesser autonomic system is complex for its users.

They describe a methodology for loading a model as a graphical structure, from its Keras source code. This is central to making complex models’ source codes easy to understand and manipulate by novice users. This is achieved by converting each ‘sentence’ of code (i.e., one complete Python instruction) into its associated parse tree, and extracting relevant information such as various attributes. This information is later used to insert a node or an edge in the graphical depiction.

We analyzed the system described in this paper, to understand various features and use cases to start gathering our system requirements for DNNCASE. We were able to review various algorithms used in this paper and their proposed architecture.

This system incorporates various basic functionalities necessary in any deep learning system’s workflow. While it focuses on a rich subset of possible use cases, this system lacks support for more advanced use cases, architectures, and training workflows.

Based on the description of the system’s logic, we ascertained that it cannot support Siamese network architectures or other architectures involving weight–sharing [9]. Siamese networks are commonly used in one-shot learning. They can compare the similarity between 2 objects (such as face portraits). In Siamese networks, a subgraph of the neural network is used twice – once per input – during the inference time. However, while training such a network, gradients corresponding to both inputs must be backpropagated through the same subgraph and *not* its copy. Our system incorporates this kind of architecture creation through a layer reuse block.

Another facility provided by Keras, i.e., custom layers, though subtle, is not supported in this system.

Klemm, Scherzinger, et al. in their paper [7] introduce a similar tool called Barista that is meant to be an easy-to-use high-level system for designing, training, and testing deep neural networks (DNNs). It offers a GUI with a graph-based editor and provides features that allow researchers to focus on their core work, without having to manually edit text files or parse logs.

This paper mentions several other tools that were created in the past; however, those tools do not cover all aspects of a conventional deep learning pipeline in a GUI-based manner. One (Caffe GUI tool[2]) of those tools' codebase as available on GitHub has been archived. The paper expands on the functionalities provided by those tools and seeks to cover the entire end-to-end deep learning pipeline, starting from model creation to its training and evaluation.

The tool has been kept open-sourced by its authors and they use Caffe[3] as the underlying deep learning framework. Caffe was a prevalent framework at the time of this paper and enjoyed wide hardware support. It also has integrations built with multiprocessing frameworks such as Spark and MPI.

Barista allows imports of existing models and pre-trained weights if any into the system, and maintains a file structure that is easy to share and transferable across machines.

The interface provided by Barista is easy to use and provides a graph-based editor that incorporates context-sensitive options, thus reducing the need to access Caffe documentation. They provide support for transfer learning and training on remote machines.

[2] https://github.com/hctomkins/caffe-gui-tool
[3] https://caffe.berkeleyvision.org/

From an implementation perspective, Barista has been written in Python 2.7 using the Qt Framework (PyQT5) and Seaborn for visualization.

The various features provided by Barista form a major source of motivation for various system requirements for DNNCASE. The specifications of Barista also serve as a baseline description of what components a graphical user interface for deep learning must necessarily provide.

Lee et al. in their paper "Deep neural network model construction with interactive code reuse and automatic code transformation" [8] introduce a GUI-based modeling mechanism for DNNs, to easily transform models into code (Keras or TensorFlow). In addition to modeling DNNs, it also supports transfer learning, it provides facilities to import graph structure which can be modified.

This paper also states that for representing deep neural network models, a directed acyclic graph (DAG) can be employed at the layer level. The topological sorting of graph nodes facilitates the traversal of all layers based on their connection sequence.

This paper introduced the concept of grouping a sub-network of a DNN into a macroblock. Further macroblock can also be contained inside another macroblock. It states a sequence of building the DNN by first developing a top-level abstract network with macroblocks that contain lower-level sub-networks. The concept of macroblocks simplifies the process of visualization of DNNs for developers, especially for ones with less experience.

The tool discussed in the paper also allows users to import and modify existing code graphically.

When deep neural network model code is imported into a GUI-based tool, it can be challenging to present its structure and work with large and complex models. Hierarchical organization is necessary to make these models manageable and visually comprehensible within the tool's interface. Many excellent deep learning models are publicly available as source code. Developers often refer to these models when building their own. Thus, these publicly available models become valuable resources for reusing or extracting building blocks.

To facilitate model development, the text proposes a method to identify "articulable subgraphs" within the deep neural network model's graph representation. These subgraphs are considered potential building blocks that can simplify model construction.

In addition to articulable subgraphs, the text mentions "repeating subgraphs." These subgraphs represent parts of the model that are repeated. Identifying such repetitions can lead to a more compact and efficient visualization of the model.

Deep learning models often include specialized modules that perform specific tasks or extract distinguishing features. These modules are part of the model's architecture and can also be represented as articulable subgraphs. Frequent articulable subgraphs are suggested as useful building blocks for constructing deep learning models. They can be considered as abstract nodes within the model.

This concept has been used in the context of DNNCASE as well. We describe a module–oriented model creation, where the user creates reusable subgraph architectures, called artifacts in our system, but are also referred to as blocks by some authors [10].

Lee et al. in their paper [6] have emphasized the strong demands of automating the tuning process of machine learning architectures which makes the tuning task easy and efficient for developers. According to this paper, the main aim of autonomicity is to minimize developers' intervention in the development of machine learning applications.

This paper has defined 5 levels of autonomicity ranging from level 0 to level 4. These levels are defined by automating factors like data set(D), task(T), machine learning technique(A), hyperparameters(H), and attributes to be used(P). Search space for designing the machine learning model consists of a Cartesian product of all the above factors. A major task of each autonomicity level is to reduce the search space for the developer.

Autonomicity level 0 is the least autonomic level in which the developer is required to provide information about all the factors. This level doesn't reduce the work for the developer. The computations for learning and inference can be done automatically by the framework at this level which takes care of the distributed and parallel computations systems. Here the developer searches the complete search space. Interaction of the developer is most at the level when creating an ML application.

At autonomicity level 1, the developers are not required to select proper input attributes in the development of ML applications. The framework of this level needs to offer the functionality of selecting and extracting features for the provided training dataset. Here the search space includes all factors except the input attribute, thus reducing the search space.

Autonomicity level 2 aims to automate the hyperparameter tuning process. The framework will have the functionality to automatically determine the hyperparameters for the algorithm. Automating hyperparameter selection implies multiple trials of a hyperparameter combination on an ML algorithm and choosing the one that gives the best/optimal result. Here the search space includes all factors except the input attribute and hyperparameters thus reducing the search space.

At autonomicity level 3 the developer only provides the training dataset and the ML task to be done, the rest all the factors are taken care of by the platform itself. The platform chooses the optimal algorithm, hyperparameters, and input attributes based on the training dataset and the task provided. The search space includes only 2 factors: The training dataset and the task to be done.

Autonomicity level 4 is the highest and most automatic level which requires the developer to only provide the training data. All other work is done by the platform. This level requires the least interaction of a developer while creating an ML application. The search space only includes the training dataset.

After reading this paper we came to know about the different autonomicity levels. We intend to reach autonomicity level 2 in our system which would support automating the hyperparameter tuning process. We would also expand our system to autonomicity level 3 in the future.

Johnston et al. in their paper [12] describe the evolution of dataflow languages. They outline how research into the field of dataflow hardware, as opposed to von Neumann based processors proceeded, and the reasons for its hugely limited success. They also describe how textual and visual dataflow languages emerged and stated reasons for them being very similar to functional languages. They further proceed to outline a few open issues in data flow programming. Their paper also discusses how the view on dataflow programming shifted from one of obtaining high parallelism, to one where visual programming is supported by a visual dataflow language, and a runtime environment.

Statement of Problem

Deep learning projects follow iterative processes. They involve different steps. The following list shows a potential workflow [11]:

1. Task analysis
2. Model creation
3. Dataset procurement and set-up of associated functionality (such as augmentation)
4. Model training
5. Model hyperparameter tuning on a development set
6. Model testing on a training set
7. Error analysis to prioritize next steps
8. Implementing next steps and restarting from step 1

This shows that a deep learning system goes through a lot of steps that are logically sequential.

We define our problem statement as a multi–step approach that culminates in a system that allows the complete deep learning workflow to be done from within an intuitive and easy-to-use interface. To realize this system, we have identified the major top–level goals that need to be accomplished. They are as follows:

1. Analysis of existing deep learning frameworks to gain an insight into what are the common workflow structures and what kinds of architectural structures and processes the deep learning practitioners are already used to.
2. To decide what basic components are necessary in an end-to-end frontend-focused deep learning CASE tool. They should be well defined and their responsibilities must be clearly stated.
3. Features that aid the developer in performing various actions, that support a better way of direct manipulation such as undo-redo functionality must be enumerated and development must be done in their direction.
4. Creation of an algorithm for code generation from graphs of neural networks, and another algorithm that can create a graphical depiction from a stored Keras model object.
5. Specification of a graphical language for creating neural network graphs, for specifying training and inference flows, and for hyperparameter tuning.

6. Creation of an intuitive interface that is responsive and easy to understand for all users, irrespective of their knowledge level and technical expertise. The interface should also provide enough hints that allow the users to decide correct actions with minimal access to external documentation.

Objective

We analyzed various literature sources, programming libraries, existing systems, and our problem statement goals. Based on these, we came up with the requirements that serve as fundamental objectives of our system. These objectives serve to achieve an acceptable and usable solution to our problem statement. They are as follows:

1. To identify and formally define a graphical language with the capabilities of being able to describe any neural network, using a layer-based paradigm. This language must also support training and inference flow description along with a way of describing model functions that can be used in hyperparameter tuning.
2. To create a graphical interface that can be used easily by users. Our target audience also includes students of deep learning at various academic and expertise levels. We must ensure that they have enough context-sensitive hints and menus, along with informative and well-formatted documentation shown in tooltips on a hover event on some element.
3. To create a well-designed code creation mechanism, that can take in various user specifications such as model graphs, and create executable code from them. This should be accompanied by proper validation with messages shown in real-time to the user, along with error localization where possible.
4. To encourage experimentation by including reversibility of almost all user actions, including model training and tuning, by maintaining previous model instances.
5. To allow a file structure that actively supports a project–like representation of a user’s deep learning systems. It will allow de-centralized versioning and concurrent work on multiple files, by various people who may be a part of the same project team.

Chapter 3: Methodology

The methodology employed for this research project comprises a well-structured sequence of stages designed to guide the project's development process and evaluation. The following phases describe a summarized view of the methodology employed. Methodology is detailed in Chapters 4 – 8.

1. **Literature survey**: We began our project with a comprehensive literature survey to establish a strong foundation for our project's progression. By reading various research papers related to the work, we intended to explore the research done to date and compare, analyze, and build a system that eliminates the shortcomings of previous systems, while adding more features that allow DNNCASE to serve as a fully-fledged IDE for deep learning oriented visual programming. The steps we followed are as follows:
    a. We started the survey by visiting various research paper libraries and repositories, such as IEEE Xplore, ACM Digital Library, SpringerLink, arXiv, Google Scholar and others.
    b. We queried these libraries with keywords relevant to our initial understanding of the project. Some keywords we used were:
        i. Visual Language,
        ii. Dataflow Language,
        iii. Dataflow Programming,
        iv. Automated Code Generation, among others.
    c. Based on these, we were able to gain information about the field of visual programming, code generation, compilation, and dataflow programming. We also found various prominent researchers in the field, and were able to deepen our understanding by following their related research.
    d. We subsequently narrowed our focus on dataflow languages and the dataflow programming paradigm. We initially focused on a subset of dataflow programming, called flow-based programming (FBP) introduced by J. Paul Morrison in the late 1960s. He popularized this concept by publishing a book titled “Flow-Based Programming: A New Approach to Application Development” and providing various online resources on his website,

https://jpaulm.github.io/fbp/ [13]. Considering the time – limited nature of our project, we analyzed the content provided on the website and learned about the distinction between FBP and FBP – inspired systems. Whereas FBP itself is an alternate system architecture to the von Neumann architecture, our system fell under the category of FBP – inspired systems.

e. Morrison majorly discusses FBP in the cited sources. To get a deeper understanding of the FBP – inspired systems, we looked into the associated research to find several works on visual programming in computer network protocols.
f. The research on visual description of computer protocols is more focused on design diagram-based models and automatic protocol code generation, however, did not fall under our requirements.
g. We changed our focus from dataflow languages to visual languages in general and their use in deep learning. After much search, based on a now – refined criteria of keywords, researchers and institutes, we were able to locate papers [5, 6, 8] by Lee and others, and also [7] by Klemm and others. (These papers have been summarized in Chapter 2.)
h. We analyzed these papers deeply, and formed our background understanding of the field. We realized the need for a dataflow-oriented paradigm besides the visual paradigm.
i. Our new objective for the survey now was to revisit dataflow programming languages, outside the purview of FBP systems. We analyzed the paper [12] by Johnston and others that provides a survey overview of contemporary dataflow programming languages, both visual and textual. It includes various challenges and conditions that a language must meet to qualify as a dataflow language. We used the insights from this paper along with those from the other papers in our further system development.

2. **Initial Requirement Analysis and Mind Mapping**: Building upon the insights gained from the literature survey, we began an initial requirement analysis. This phase entailed a meticulous examination of project specifications and objectives. Furthermore, it we used mind-mapping techniques to visualize and structure these requirements, facilitating a

coherent comprehension of the project's overarching goals. We emphasized developing mind maps from the user’s perspective and adding all the necessary actions required by the user for efficient functioning and usage of the system. We have discussed requirements further in Chapter 4. Here, we present a sample image of the top-level domains of our mind map: (DNNLang refers to the visual language of our system; See Appendix A for an overview of the finalized language.)

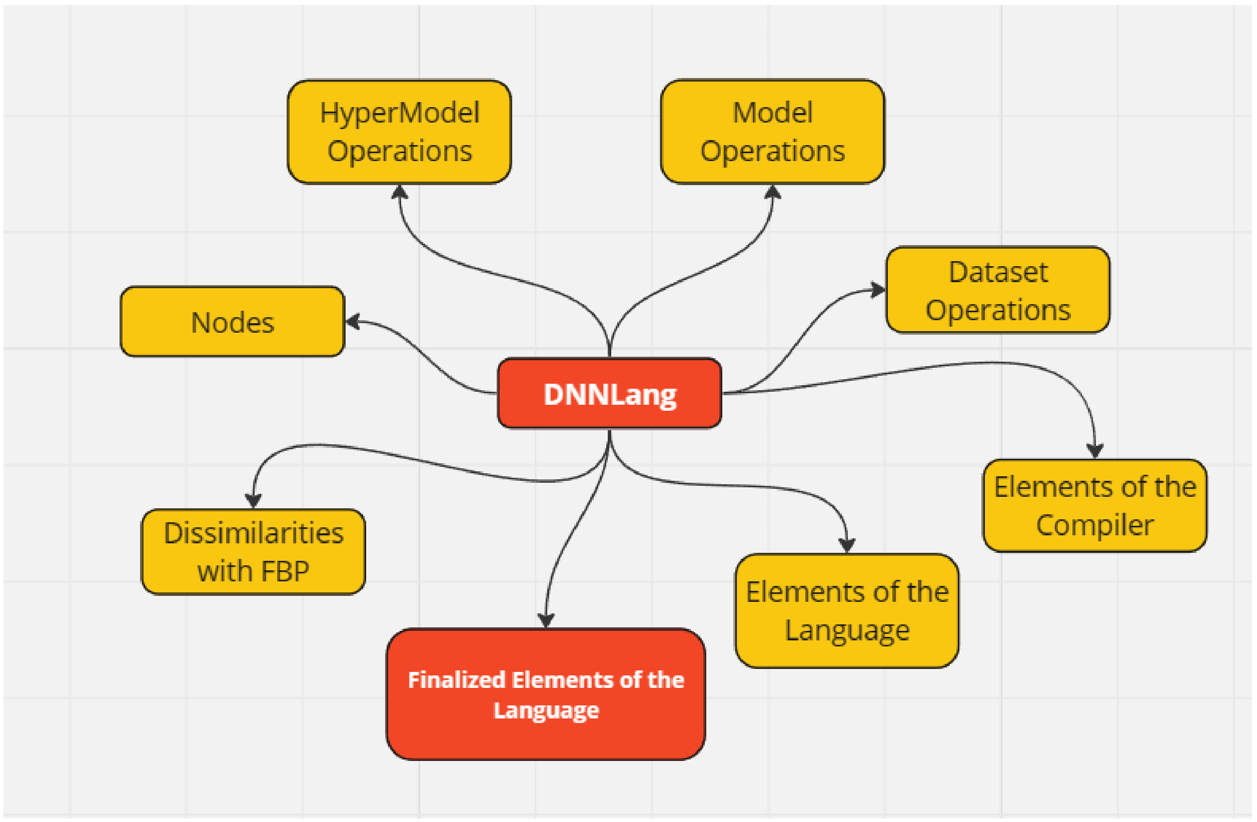


Figure 3.1 - Top level domains of mind map used for requirement analysis.

3. **Language and Compiler Creation:** Following the Requirement Analysis and Mind Mapping the subsequent steps involved designing a visual language and its associated complier. We have discussed the construction and working of the compiler for the language in great detail in Chapter 7. Here, we provide an overview of the compilation process:
    a. Our language is visual in nature, and therefore the “code” of this language is in the form of directed acyclic graphs. (See Appendix A for a brief description of the language.)
    b. The compiler has to traverse the code in a topologically sorted manner.

   c. As the compiler visits various nodes, analyses them as being data sources, data transformations, or data sinks. The compiler expects each node (except some special purpose nodes) to perform some execution, and return some data.
   d. According to the data dependencies present in the graph of the code, the compiler moves each data item from its source node to its target nodes.
   e. Once the compiler has resolved the graph of a code file, and the graphs of any other code files imported in the current code file, the compiler then generates executable textual Python code.

4. **Runtime Environment Design and Frontend Creation**: Based on our literature survey and problem domain, we surmised that one of the core requirements of any visual language is the software that enables users to view and code it. To this end, we greatly analyzed and designed toward a system that can be used to create programs in our language. We describe the design of our system and its working in depth in Chapter 6. As a preview, we here present the major aspects of the environment:
   a. The basic element of the environment is a *graph editor*. Similar to text editors, a graph editor allows its users to create and modify graphical data.
   b. We created a graph editor using a canvas that contains a 2 - dimensional coordinate system. The canvas allows placement of nodes and edges, dragging of nodes, a minimized top level viewing window of the whole canvas to allow the programmer to get a glimpse of their workspace, and also a control panel with buttons such as zoom in/out, enable or disable writes, etc. The editor also has controls to add nodes, and modify their properties. The following images show a

view of the graph editor:

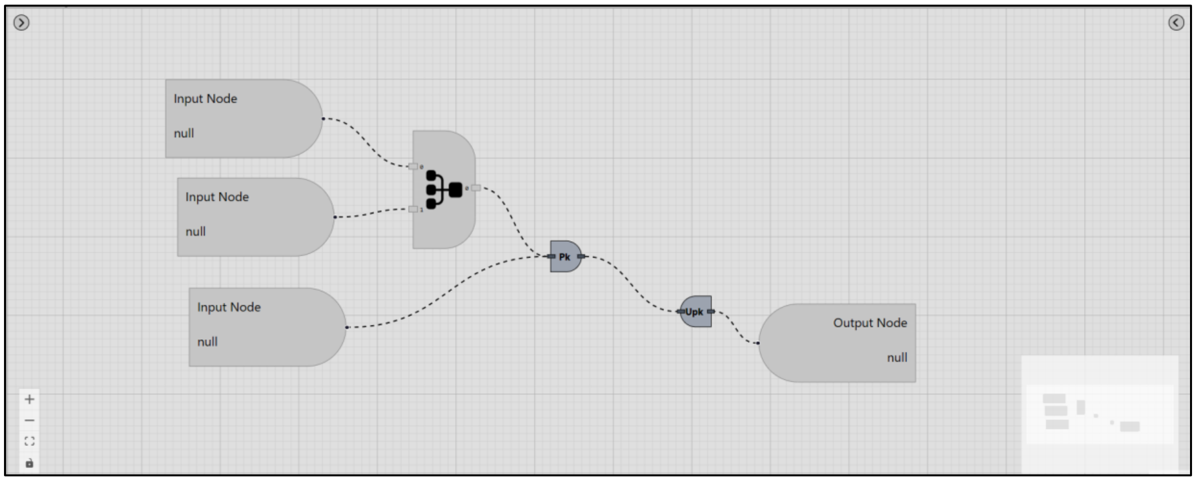


Figure 3.2 - The graph editor with a basic graph drawn in it. Observe the mini map on the right-bottom part of the editor, and the controls on the left-bottom of the editor.

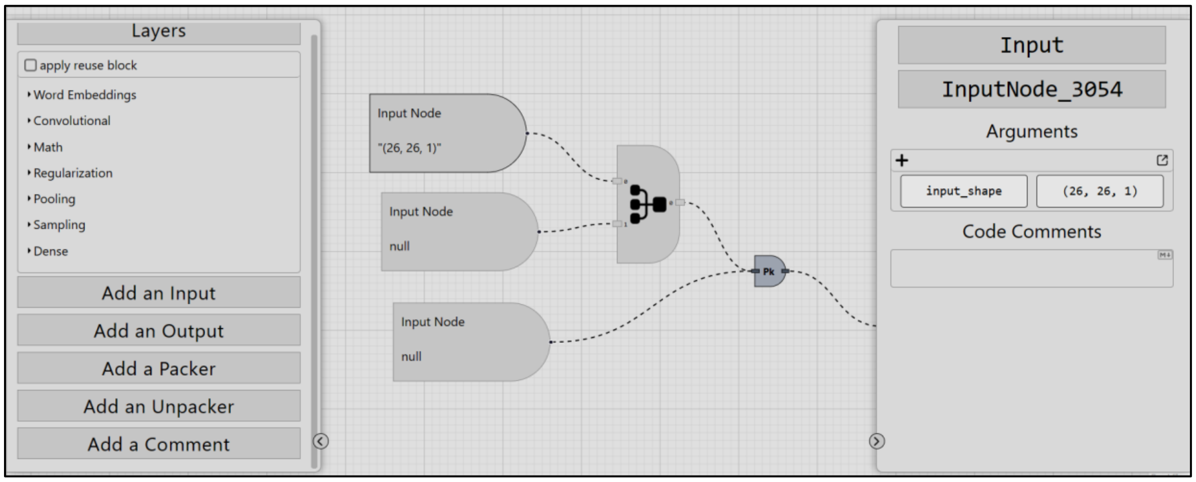
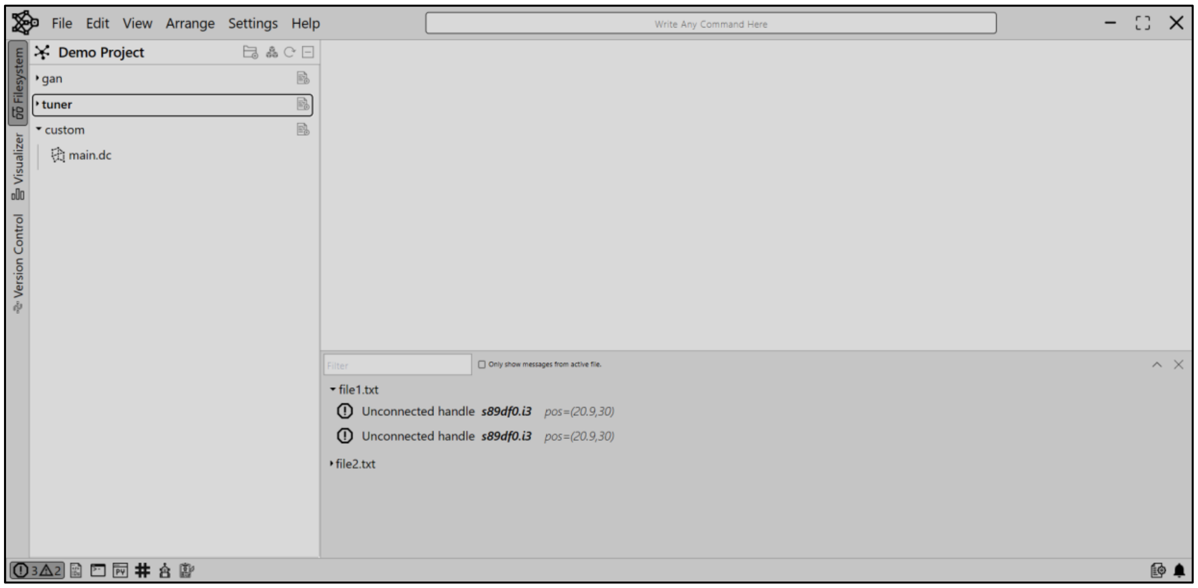


Figure 3.3 - The left and right panes of the graph editor. The left pane allows adding more nodes, and the right pane allows modifying the properties of individual nodes.

c. Once the editor was setup, we required an IDE to house various instances of this editor, and to provide an interface to the file system. For this, we created an application that could open and close files, manage various tabs, house different panels, and that has various programmer – assistance features, with more potential features on the way. We explain the IDE in Chapter 6. The following image is a view of the IDE:



```
file1.txt
  Unconnected handle s89df0.i3   pos=(20.9,30)
  Unconnected handle s89df0.i3   pos=(20.9,30)
file2.txt
```

Figure 3.4 - The DNNCASE IDE displaying the filesystem pane and errors box.

5. **Deep Learning Library creation:** In this phase, we incorporated various layers, optimizers, and additional features supported by Keras within the visual language framework of DNNCASE. This includes the integration of convolutional layers, recurrent layers, normalization techniques, activation functions, and other essential components that Keras provides. The objective was to ensure comprehensive support for Keras functionalities, empowering users to seamlessly import and leverage a wide array of deep learning constructs within our system. To analyze the core important functions to develop first, and to ascertain a priority order, we performed the following steps:
    a. We analyzed different commonly used neural network architectures and workflow functions.
    b. Based on these, we decided on the applicable and convenient functional nodes for the graph, that could perform a similar task, and were not cumbersome to use in a visual ecosystem.
    c. We then proceeded to construct applicable visual elements for these nodes, focusing on ease of usage when programming.
6. **Usability Testing**: To complete our project, an analysis of the usability was necessary. We conducted a survey on various undergraduate B. Tech. Computer Science students of

the 3rd and 4th academic years. The objective of this survey was to gauge the effectiveness of our system along the following 4 dimensions:

a. Ease or difficulty in understanding the concept of the reuse node.
b. Acceptability of this visual language in education.
c. Understandability of the generated neural network code.
d. Effectiveness of interaction with the user interface.

The various results we obtained, our survey methodology, and the survey questionnaire have been discussed in depth in Chapter 8. The survey was conducted to assess the power of DNNCASE, and the language it uses. As discussed later, we had at our disposal 2 possible language structures: a purely functional language, and a language that treated functions as top – level entities. We show that both the language structures are equally conducive to learnability in our results, by the statistics obtained in the 1st dimension (Ease or difficulty in understanding the concept of the reuse node.). As a context, the reuse node was necessary to implement functions that can return other functions, which is a requirement of top – level functions. This is an important observation, because both language paradigms can be merged into a hybrid paradigm with certain nodes emitting not just data, but functions as well. This is one of the future scopes of our project.

Chapter 4: Requirement Analysis and System Design

We performed the requirements analysis for our system after the literature survey. We followed a 2 – stage approach at analyzing the requirements. The following schematic displays the approach we followed:

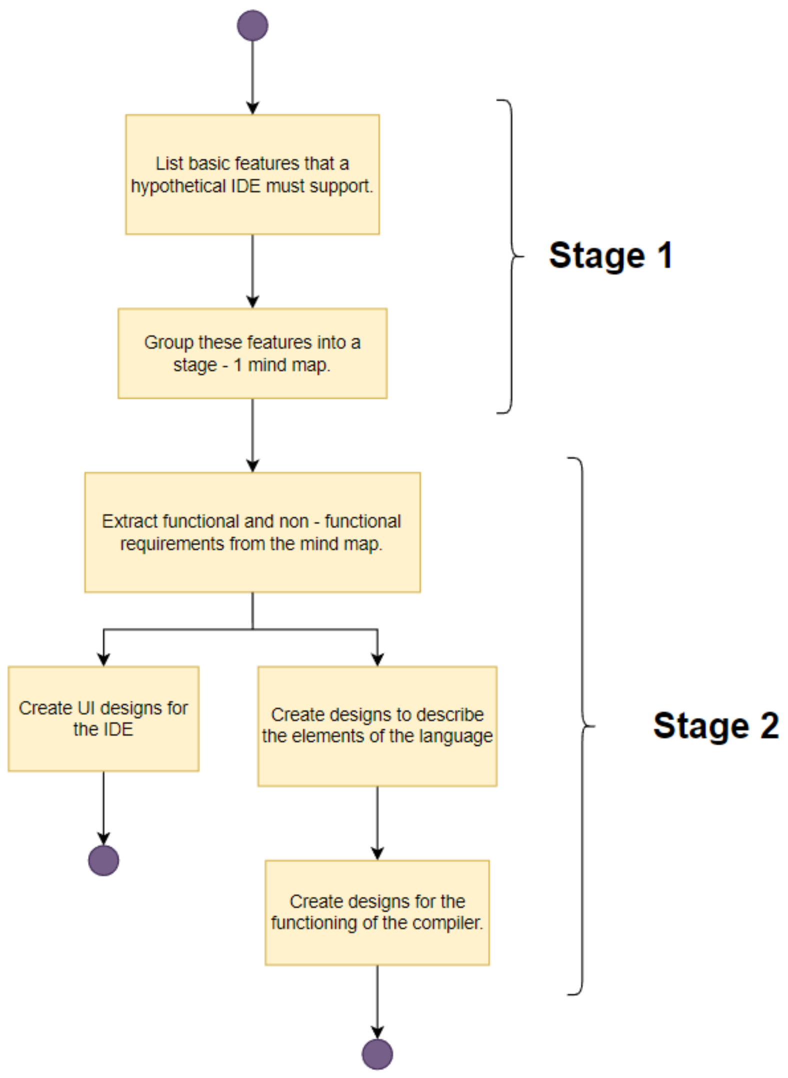

List basic features that a hypothetical IDE must support.
Group these features into a stage - 1 mind map.
Stage 1
Extract functional and non - functional requirements from the mind map.
Create UI designs for the IDE
Create designs to describe the elements of the language
Create designs for the functioning of the compiler.
Stage 2

Figure 4.1 - A flowchart displaying the requirements analysis process followed for DNNCASE.

Stage 1: Unbounded Features and Mind Map Grouping

In stage 1, we listed those features that a hypothetical unbounded – scope IDE must support. The aim of this task was to understand the scope of our project. As our time and resources were limited, we would only be able to develop a short subset of these features. However, this task helped us to achieve an understanding of the problem domain, to find what features were similar and what were distinct, and to develop a priority order among the features. As an example, one of the features was that IDE must apply validation rules to the coded program. However, this feature is meaningless until a canvas that supports graph editing is developed. Hence, this analysis helped us in understanding what features to develop first, and which to delay for a later time period. We then grouped the features into a mind map (the mind map along with the features is presented after this discussion). This grouping was done based on similarities between the features. The mind map grouping is a hierarchical agglomerative grouping, i.e., we initially viewed each feature as a distinct point, then we kept grouping together the similar features into hierarchical clusters, which resulted into a final mind map. We call this process **Hierarchical Agglomerative Clustering of Unbounded Feature Space for requirements elicitation (HACUFS)**.

The following list shows all the features that we initially compiled for stage 1:

1. A desktop app that can run as a self-contained system on any platform.
2. The application should have multiple toolbars.
3. One toolbar should contain the project name, app icon, close/minimize/maximize keys.
4. Another toolbar should contain the menus (`File`, `Edit`…). There should also be a `Help` menu, showing documentation links, and also updates if any.
5. A third toolbar should contain quick access options such as undo/redo and command palette.
6. Another toolbar should show the tabs currently open in the current workspace.
7. There should be a bottom bar showing warnings, errors, git pull/push status and output format (`.ipynb` or `.py`).
8. The main work area should display contents of the open file. In case the open file is a NN graph, it should show side bars for element selection (e.g., layer) and parameter selection (e.g., hyperparameters specification).
9. Users should be able to view and edit neural network graph diagrams.
10. As a future support, the users should be allowed to open their data as a preview.
11. To edit the graphs, users should be given a choice from all Keras API objects, such as layers, activations, losses, etc.
12. Users should be able to edit the hyperparameters for each layer.
13. Users should be given the allowance to undo/redo their changes and autosave their work.
14. The undo/redo should be separate for each editable tab and each input box.
15. The undo/redo should be supported across sessions (say I make some change, and then later open the graph. I should still be allowed to undo my changes).

16. Users should be given a command palette input where they can enter some text to display all associated allowed actions (say I want to auto-format the graph. The command palette should suggest it to me when I enter 'format').
17. Users should be allowed to auto format their graphs i.e., refactor the way the graph is being displayed.
18. Users should be allowed to move various graph entities around on an 'infinite' canvas.
19. Whenever a user adds an edge (u, v), the nodes u and v should exist.
20. Users should be allowed to create custom layers for their projects. The code for the custom layers should be given by the user. Our system should just import the layer as a black box, along with relevant metadata, such as hyperparameters, layer name, input-output count and so on.
21. Users should be able to import other graphs in their current graph as a black-box (i.e., the imported graph won't be editable from current tab) and use it as many times as they wish. *Important: each use is an independent and deep copy, i.e., separate instantiation of the imported graph*.
22. Users should be able to convert their graphs into python code (Keras based). This should be supported both as a python project, and as a Jupyter notebook.
23. Users should be able to have their graphs validated. This should be done to ensure there are no circular dependencies in their project, and also no cycles in their individual graphs, among other errors.
24. When a layer is hovered upon, users should get a docstring about that layer as a tooltip.
25. Users should be allowed to create a `SavedModel` object from within the app as well, so that they can just import it in their code, rather than use the python code files generated.
26. When users get their graphs validated, errors/warnings if any should be highlighted in the graph itself.
27. Users should be able to work on neural network projects. This means that multiple projects, each as a folder should be allowed to be opened in the system, BUT one at a time. This is similar to the `Open Folder...` functionality in VS Code.
28. Users should be provided with keyboard shortcuts, and as a future enhancement, these should be modifiable by the user, on a *per-action* basis. Clearly, this means **every user-facing action that can be mimicked by the push of a button should also be allowed to have a registered keybinding**.
29. Users should be allowed to import a model from its python code or its saved format. This will **only** be allowed if the input (code or saved format) *can be converted to a* `tf.keras.Model` *object*. The loaded models will be automatically formatted according to the formatting rules.
30. In case the model is loaded from a saved instance, its layer-level weights should be preserved, unless the user wants them purged (If the user does want the weights to be deleted, only the architecture should be retained). In case the weights are to be preserved, user should be allowed to decide the same for each layer (i.e., layer level granularity, not graph level). The weights should be preserved by default.

31. Users should be allowed to reorganize the layers of a loaded neural network's graph, and pre-trained weights if any must be preserved across this reorganization.
32. The language the users use to specify a graph structure should contain the `reuse block`, `repeater`, `edge`s, and `layer`s, as previously decided. A layer can have multiple inputs and outputs. Therefore, if another graph is imported into the current graph, the imported graph will also be treated as a layer (which doesn't have any hyperparameters).
33. Users should be allowed to train their models from within the system itself. This means that wherever the weights were preserved, transfer learning should take place.
34. Users should be able to set up data pipelines from within the system. They should be allowed to access remote data repositories as needed via URLs, and all `tf.data`, which includes `tf.data.Dataset`, functionalities should be given to the user for such creation.
35. Users should be able to test their model from within the system.
36. As the training/testing takes place, users should be shown the graphs depicting various user-chosen metrics, along with variation of loss.
37. Users should have the option to stop training/testing whenever they wish.
38. Users should be allowed to run inference on specific input data points.
39. Users should be allowed to create a version-controlled timeline of the neural net corresponding to their project. This means if I train my model today, and re-train it tomorrow, I should have a snapshot of the model after both the training sessions.
40. Users should be allowed to perform hyperparameter tuning. This should be provided separate from the model testing and training, to ensure no compatibility issues with any relevant libraries arise.
41. Users should be allowed to decide which version of Keras they'll use, in case there are major differences across upcoming and previous Keras versions. (I.e., in case forward- or backward- compatibility is missing)
42. Users should be provided a clear screen option. This should clear the currently visible graph on the screen (i.e., in the current tab). However, users should have the option to undo this change.
43. Users should be allowed to zoom in or out of the workspace of the graph, as visible on the screen.
44. Users should be given an option to choose their python interpreter, and TensorFlow version (in case there are more than one installed). Also, users should be allowed to choose what underlying hardware they will use, in case, say more than 1 GPUs are available, or where they want to store any downloaded data (if not in the project folder itself).
45. Users should be allowed to choose what theme they want (dark/light/etc...) via configurable files.

These features represent our project’s initial scope, which was unregulated as yet. To bring the scope down to a more manageable size, we performed the mind – map grouping. The method for developing this grouping - **HACUFS** has already been discussed above. The following figure shows the mind map. Please note that in the leaf nodes of the mind map, `Fi` refers to the ith feature in the above list.

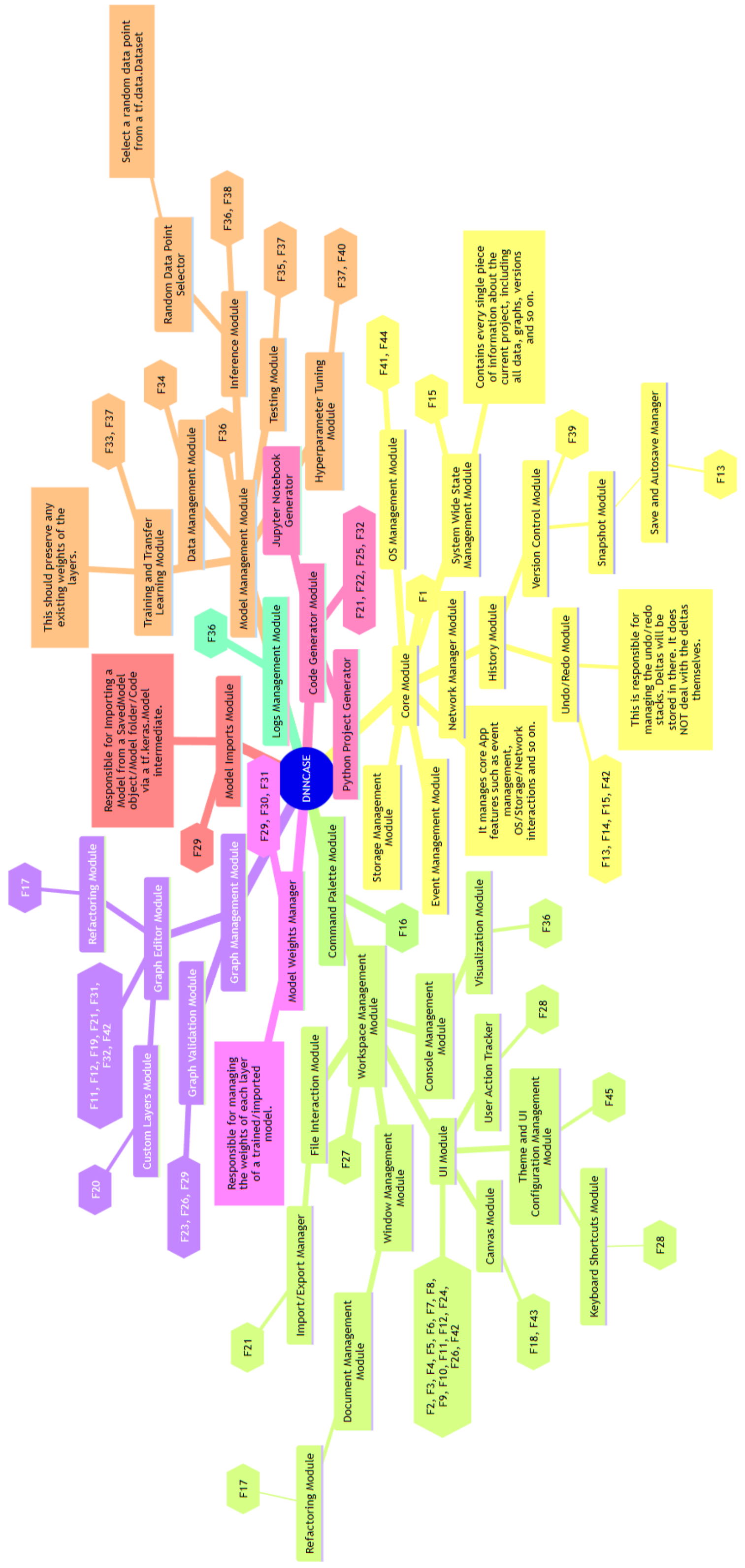

DNNCASE
Model Management Module
Training and Transfer Learning Module
This should preserve any existing weights of the layers.
F33, F37
Data Management Module
F34
F36
Inference Module
Random Data Point Selector
Select a random data point from a tf.data.Dataset
F36, F38
Testing Module
F35, F37
Hyperparameter Tuning Module
F37, F40
Jupyter Notebook Generator
Code Generator Module
F21, F22, F25, F32
Python Project Generator
Logs Management Module
F36
Model Imports Module
Responsible for importing a Model from a SavedModel object/Model folder/Code via a tf.keras.Model intermediate.
F29
F29, F30, F31
Graph Management Module
Graph Editor Module
Refactoring Module
F17
F11, F12, F19, F21, F31, F32, F42
Custom Layers Module
F20
Graph Validation Module
F23, F26, F29
Model Weights Manager
Responsible for managing the weights of each layer of a trained/imported model.
Core Module
OS Management Module
F41, F44
F1
System Wide State Management Module
F15
Contains every single piece of information about the current project, including all data, graphs, versions and so on.
Network Manager Module
History Module
Version Control Module
F39
Snapshot Module
Save and Autosave Manager
F13
Undo/Redo Module
This is responsible for managing the undo/redo stacks. Deltas will be stored in there. It does NOT deal with the deltas themselves.
F13, F14, F15, F42
It manages core App features such as event management, OS/Storage/Network interactions and so on.
Event Management Module
Storage Management Module
Command Palette Module
F16
Workspace Management Module
Visualization Module
F36
Console Management Module
User Action Tracker
F28
File Interaction Module
F27
UI Module
Theme and UI Configuration Management Module
F45
Keyboard Shortcuts Module
F28
Canvas Module
F18, F43
F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12, F24, F26, F42
Window Management Module
Document Management Module
Import/Export Manager
F21
Refactoring Module
F17

Figure 4.2 - The mind map grouping of the unbounded features for DNNCASE. The mind map has been created in an agglomerative hierarchical clustering mechanism (HACUFS) working up from the unbounded features.

Stage 2: Functional and Non – Functional Requirements Extraction, UI Designing, Language Designing, Compiler Designing

Based on the mind map, we were able to define abstractions that had to be formalized in the form of system requirements. As including every abstraction would have been out of the scope of our allotted time and available resources, we have retained the most important abstractions in the set of requirements. First, we mention the Functional Requirements (FRs) and then we mention the Non – Functional Requirements (NFRs):

FR1: The system should allow its users to visually create directed acyclic graphs that define some processing to be performed. The graphs should then be converted into executable python code.

FR2: The system must provide its users some graph editor on which the user can draw the graphs. The graph editor must have a canvas that is connected to the UI for inputs and outputs so that the user can not only view the graph but also:

1. Drag and drop nodes on the graph.
2. Connect various nodes via edges.
3. Pan and zoom the canvas to change the view.
4. Delete nodes and edges from the canvas.

FR3: The canvas must provide the user with ease-of-use features. At minimum, the canvas must provide the following features:

1. Dedicated buttons to zoom in and to zoom out of the canvas.
2. A dedicated button to mark the canvas as read – only; This is required if the user merely wants to browse the code.
3. A button to perform auto zoom and auto pan so that the entire graph is visible on the user's viewport.
4. A window to serve as a minor display of the canvas that allows panning and zooming, while displaying the entire canvas at any time to the user.

FR4: The graph editor must provide the user with access to all the nodes that can be inserted onto the canvas.

FR5: The graph editor must provide the user with facility to change any metadata of the nodes or of the edges.

FR6: A workspace environment must be set up in the form of an IDE. This IDE must provide the user with access to their file system, to create, rename, reorder and delete files and folders. This must be done in a way that the user can only modify entities within a project folder.

FR7: The IDE must provide the user with access to multiple open graph editors, one for a single file. The user must be able to switch the currently active editor. The user must also be able to close every editor.

FR8: The IDE must provide the user an option for each file to generate python code of that file and download it in a .py file to any location of the user's choice.

FR9: The system must have a well – defined graphical language that is easy to learn, and is based on the concept of dataflow instead of control flow. Even though the language is dataflow, it must contain certain control structures, namely branching of dataflow based on a boolean condition, and repeated flow of data through the same sub-graph (analogous to loops).

FR10: A compiler must be developed for the language of this system. The task of this compiler is to convert the graph into an executable python file. The compiler must be capable of loading stored files, and storing the generated python file.

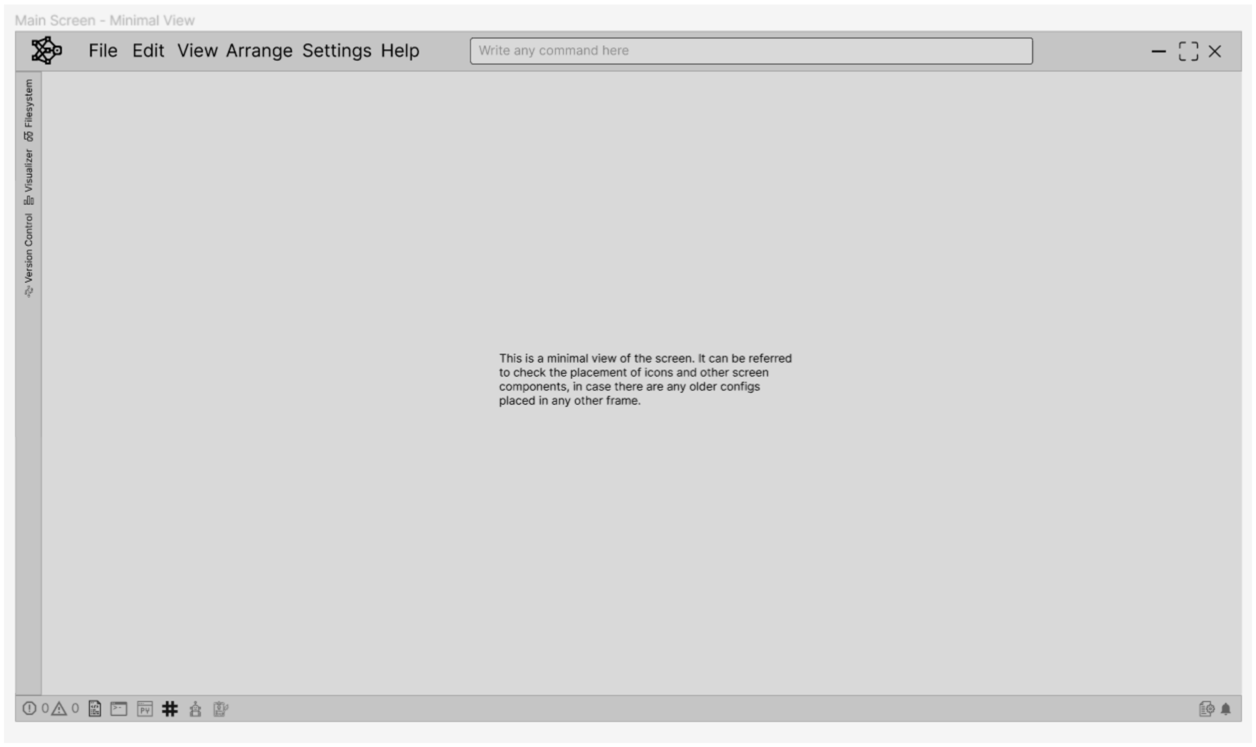
NFR1: Portability: The system must be portable across devices. Either the system must be web – based so that it can be accessed by a browser, or the system must have its compiled source code in formats that work in the 3 major operating systems: Windows 10 and above, Mac OS X, Linux Debian compliant flavors.

NFR2: Learnability: The core of the system is to ease deep learning education. For this purpose, the system must be easy to understand and its usage must be easy and quick to learn. The system must have a theme that makes it easy to view and intuitive to work with.

NFR3: Usability: The system must be able to create all workflows and architectures that a user wishes to create. The design of the system, of its language, or of the compiler must not present any hindrance to the user's work.

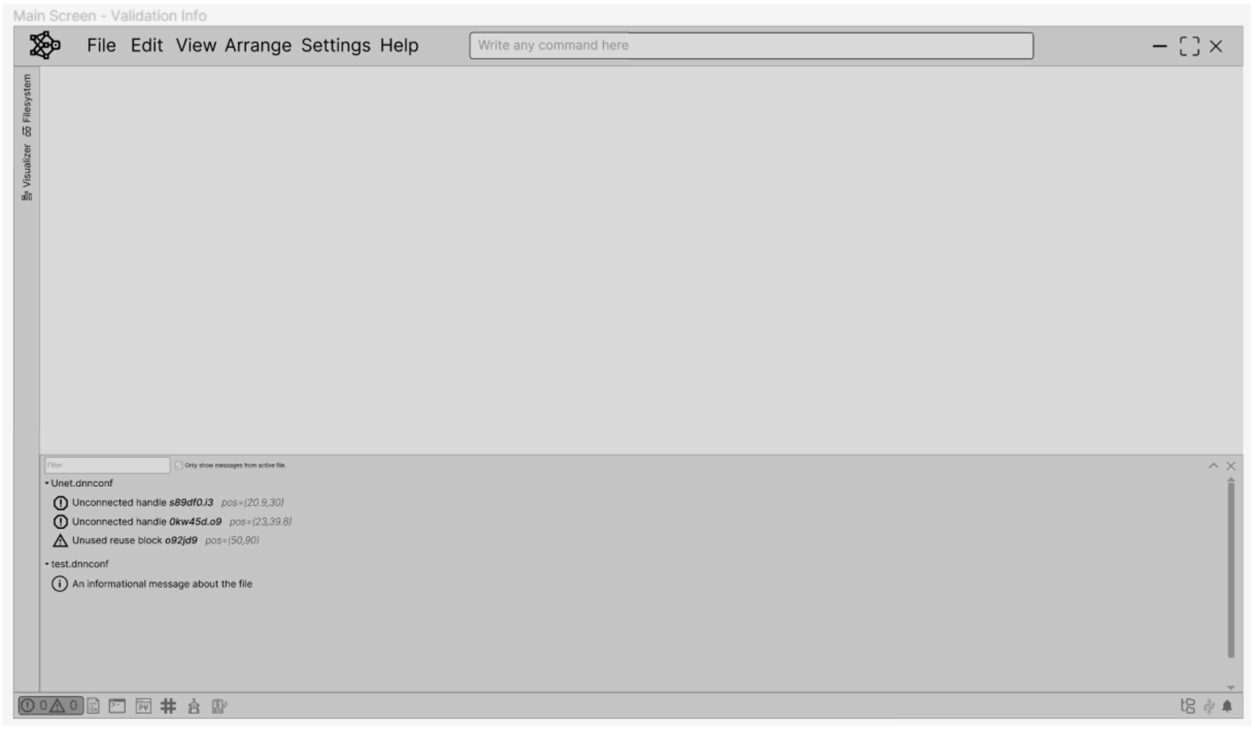
Based on these requirements, we proceeded on 2 different paths.

First, we created the design of the UI using Figma. Figma is a tool that simplifies creation of static prototypes for frontends. Once the designs were created, we developed a hierarchical component structure that distributed the display and interaction responsibilities of the UI elements among smaller more modular and reusable components. The following images provide snapshots of the Figma designs. Following these images, we show snapshots of the hierarchical component structure.



This is a minimal view of the screen. It can be referred to check the placement of icons and other screen components, in case there are any older configs placed in any other frame.

Figure 4.3 - A collapsed view of the DNNCASE IDE.



- Unet.dnnconf
  - Unconnected handle *s89df0.i3* pos=(20.9,30)
  - Unconnected handle *0kw45d.o9* pos=(23,39.8)
  - Unused reuse block *o92jd9* pos=(50,90)
- test.dnnconf
  - An informational message about the file

Figure 4.4 - The "Errors and Warnings" panel display of the IDE.

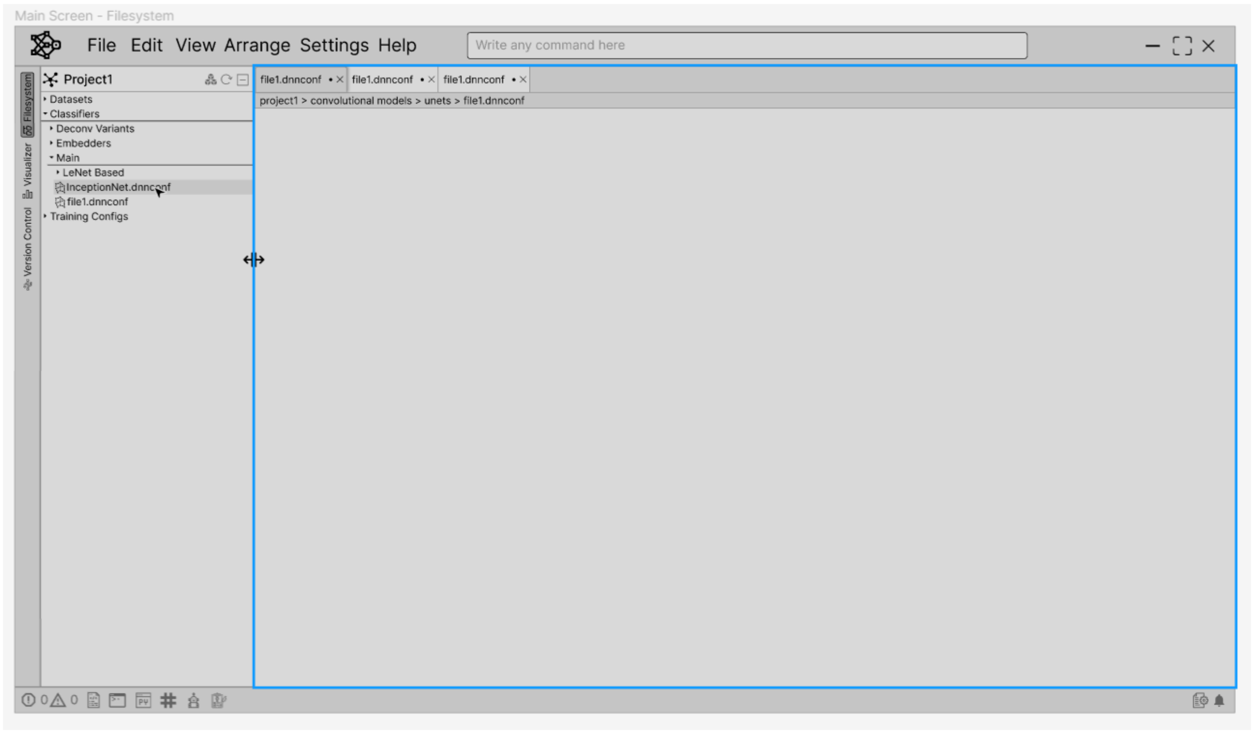


Figure 4.5 - The "Filesystem" pane of the IDE, providing access to user's project files. The outlined portion shows the workspace.

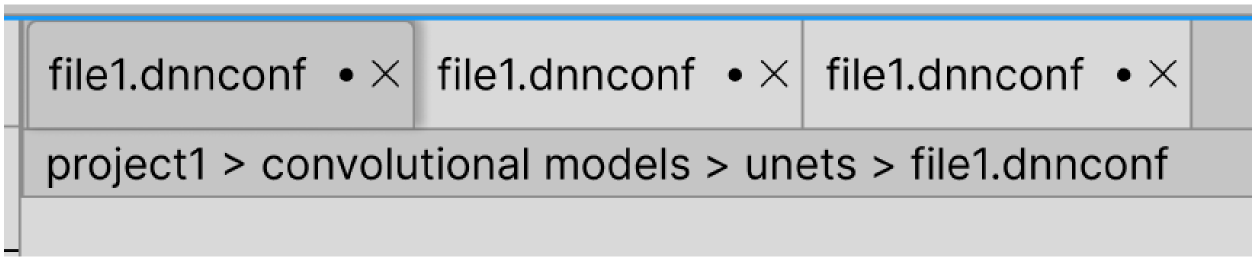
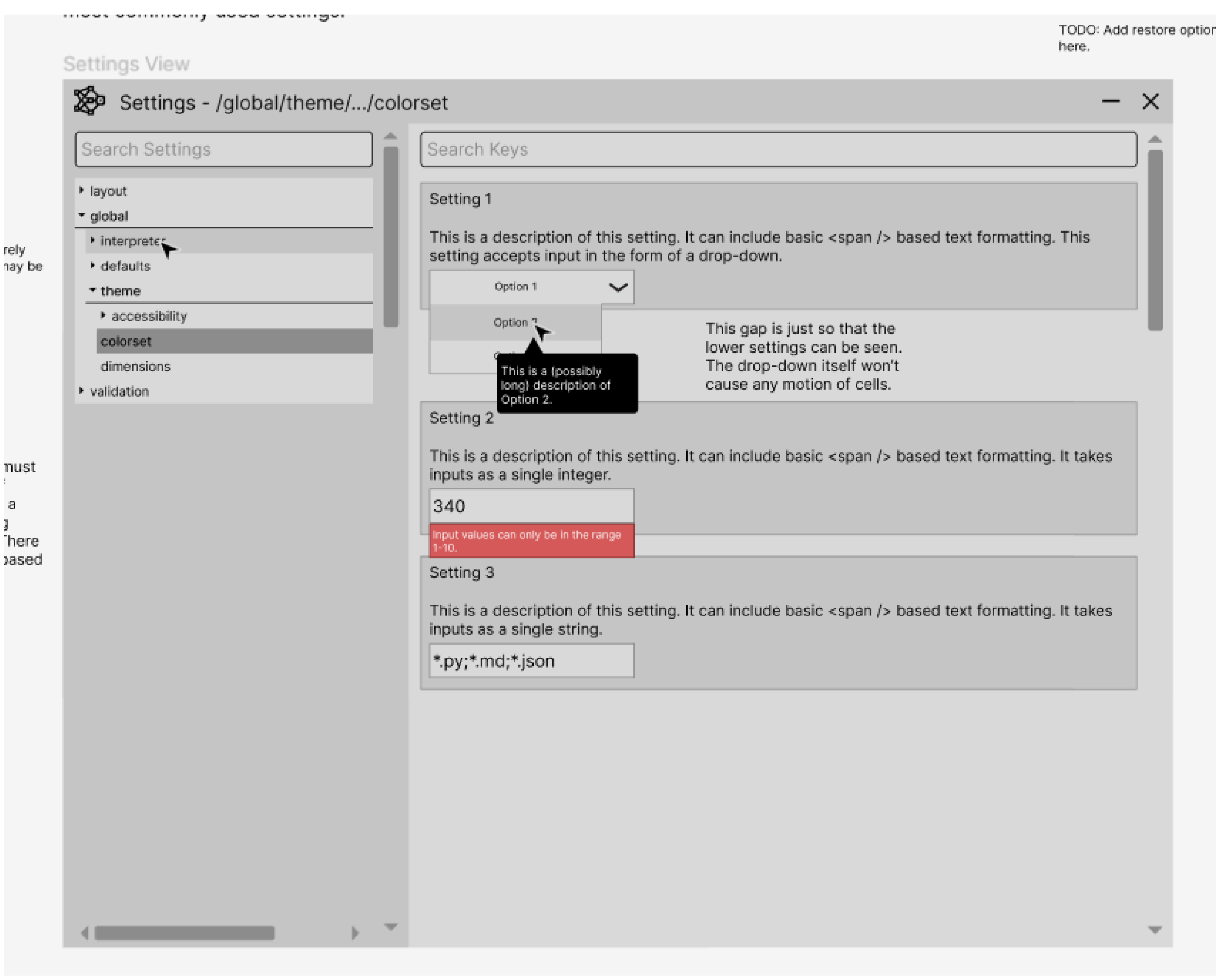


Figure 4.5 (Contd.) - A close up of the tabs and quick address bar section of the DNNCASE Workspace.



Setting 1

This is a description of this setting. It can include basic <span /> based text formatting. This setting accepts input in the form of a drop-down.

This is a (possibly long) description of Option 2.

This gap is just so that the lower settings can be seen. The drop-down itself won't cause any motion of cells.

Setting 2

This is a description of this setting. It can include basic <span /> based text formatting. It takes inputs as a single integer.

340

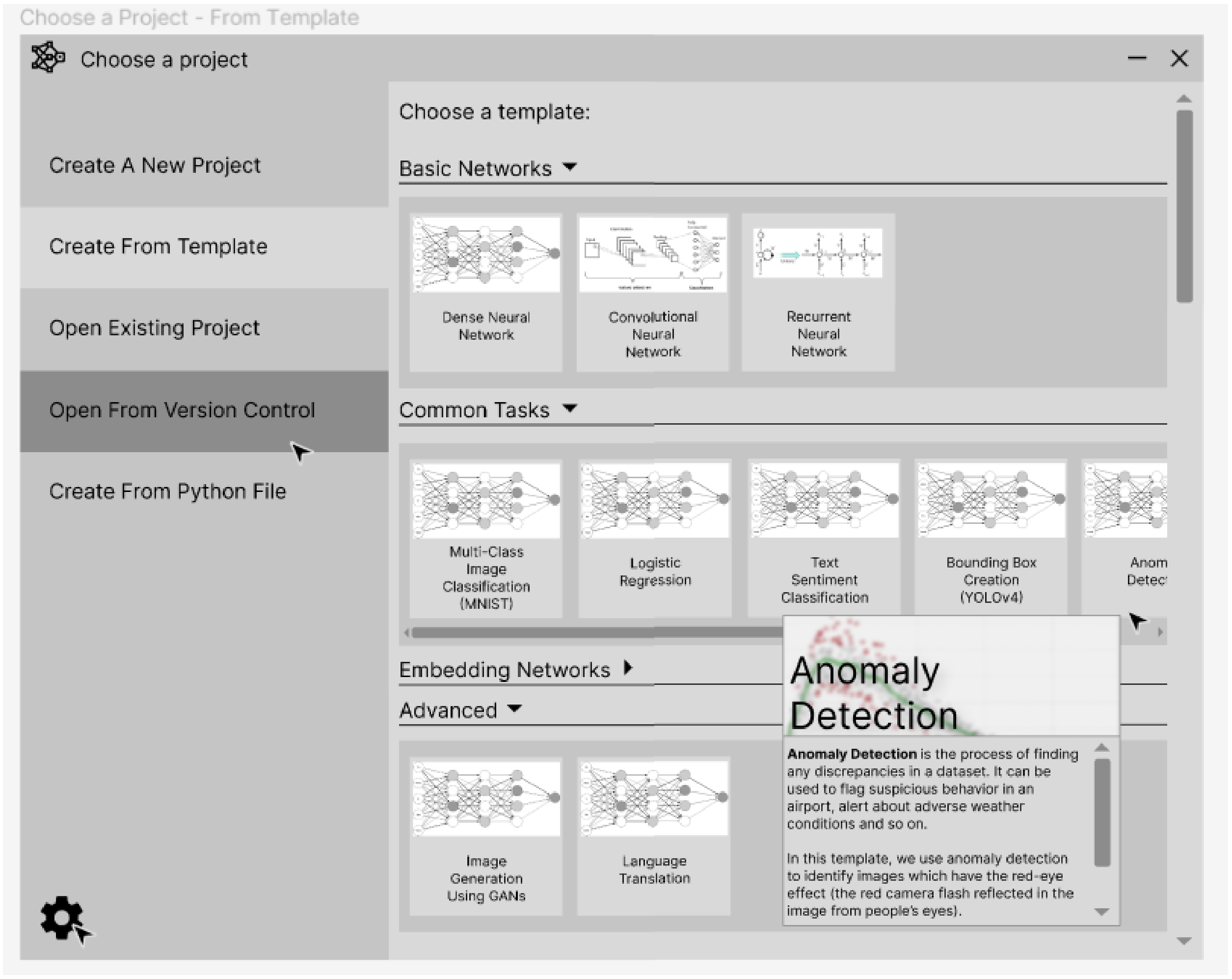
Input values can only be in the range 1-10.

Setting 3

This is a description of this setting. It can include basic <span /> based text formatting. It takes inputs as a single string.

.py;.md;*.json

Figure 4.6 - A proposed design of the settings dialog box for DNNCASE.



Anomaly Detection is the process of finding any discrepancies in a dataset. It can be used to flag suspicious behavior in an airport, alert about adverse weather conditions and so on.

In this template, we use anomaly detection to identify images which have the red-eye effect (the red camera flash reflected in the image from people's eyes).

Figure 4.7 - A proposed design for the project selector dialog box for DNNCASE.

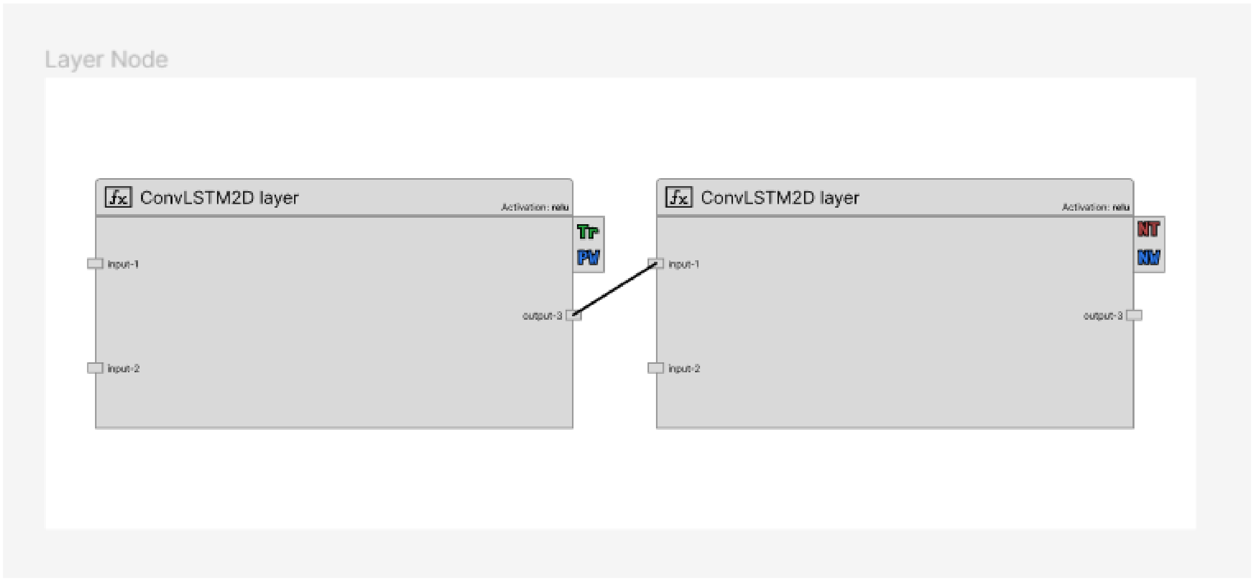


Figure 4.8 - A design for the function node of DNNCASE. This is a node to be used in the graph editor.

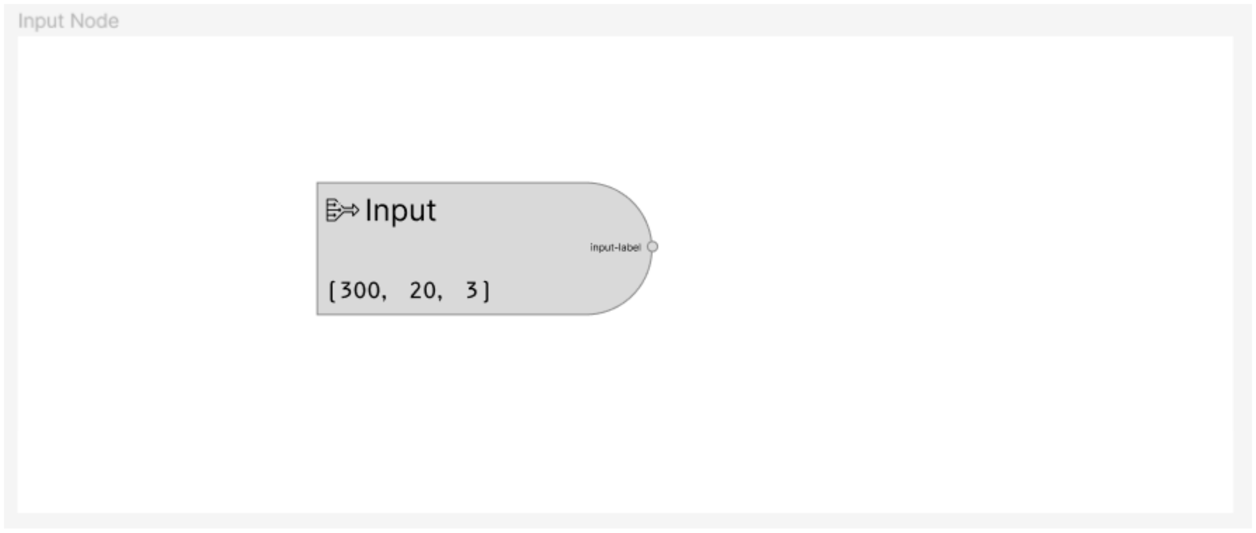


Figure 4.9 - A design for the input layer node for DNNCASE. This node is supposed to mark the beginning for forward propagation in a neural network.

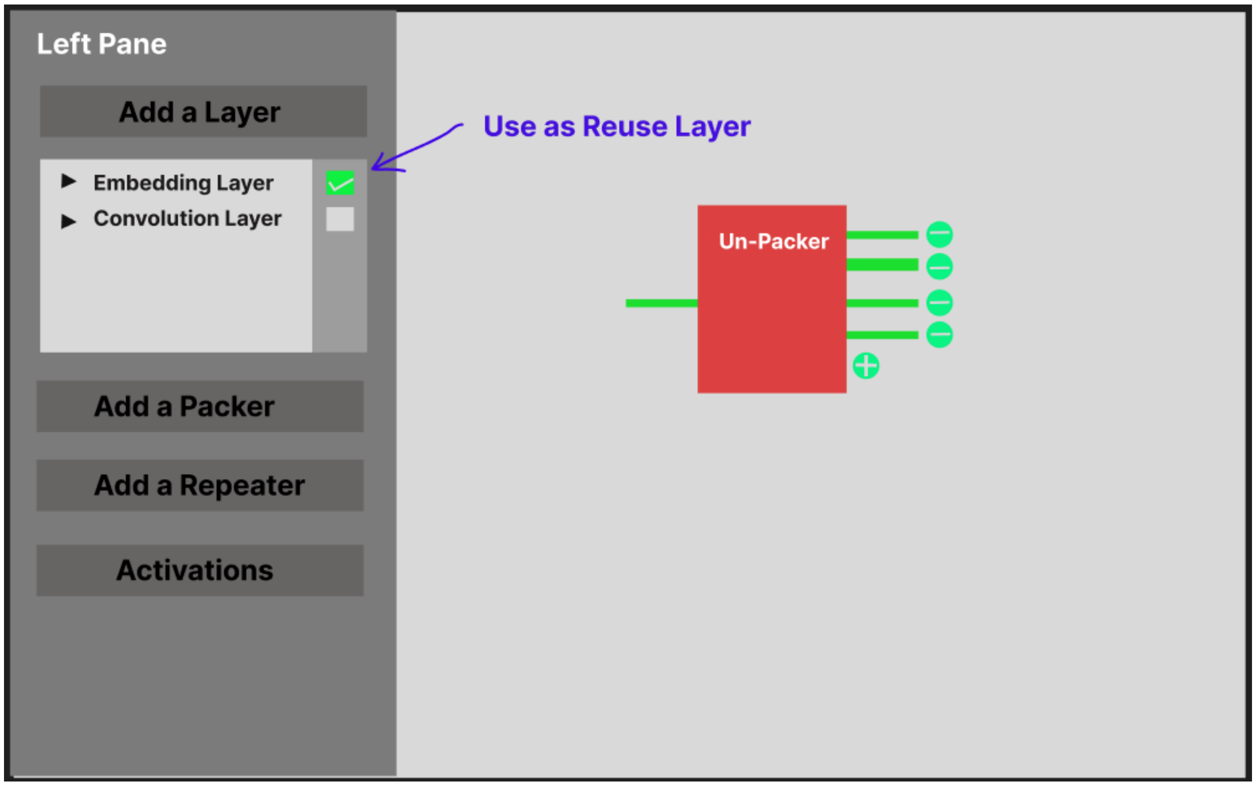


Figure 4.10 - An early design of the DNNCASE Canvas. The design also displays the left pane. The left pane is supposed to allow insertion of nodes onto the canvas.

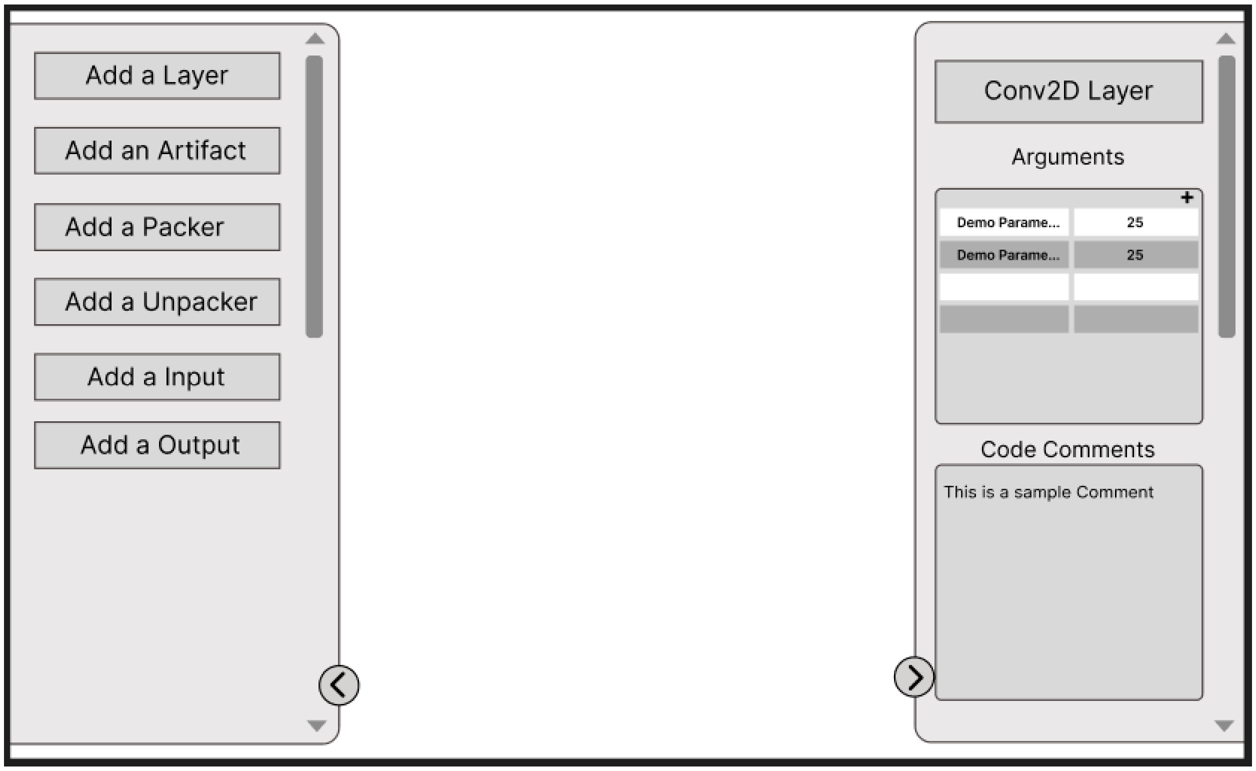


Figure 4.11 - Design of the Left and Right panes of DNNCASE Visual Canvas.

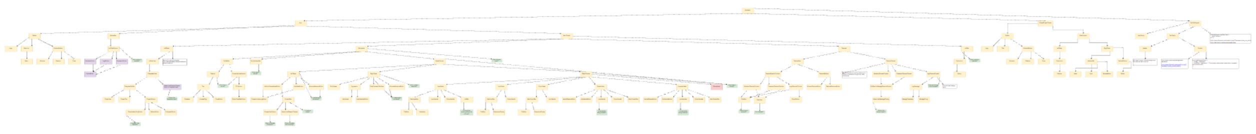


Figure 4.12 - The tree of UI components, hierarchically arranged. The diagram is too large to be printed. It is available on the project's public repository at: https://github.com/rohan843/dnncase/blob/c0b4761d1242ec0cba65841d3a6554e9129d09e7/component%20structure.drawio. The software "draw.io" must be used to open the file.

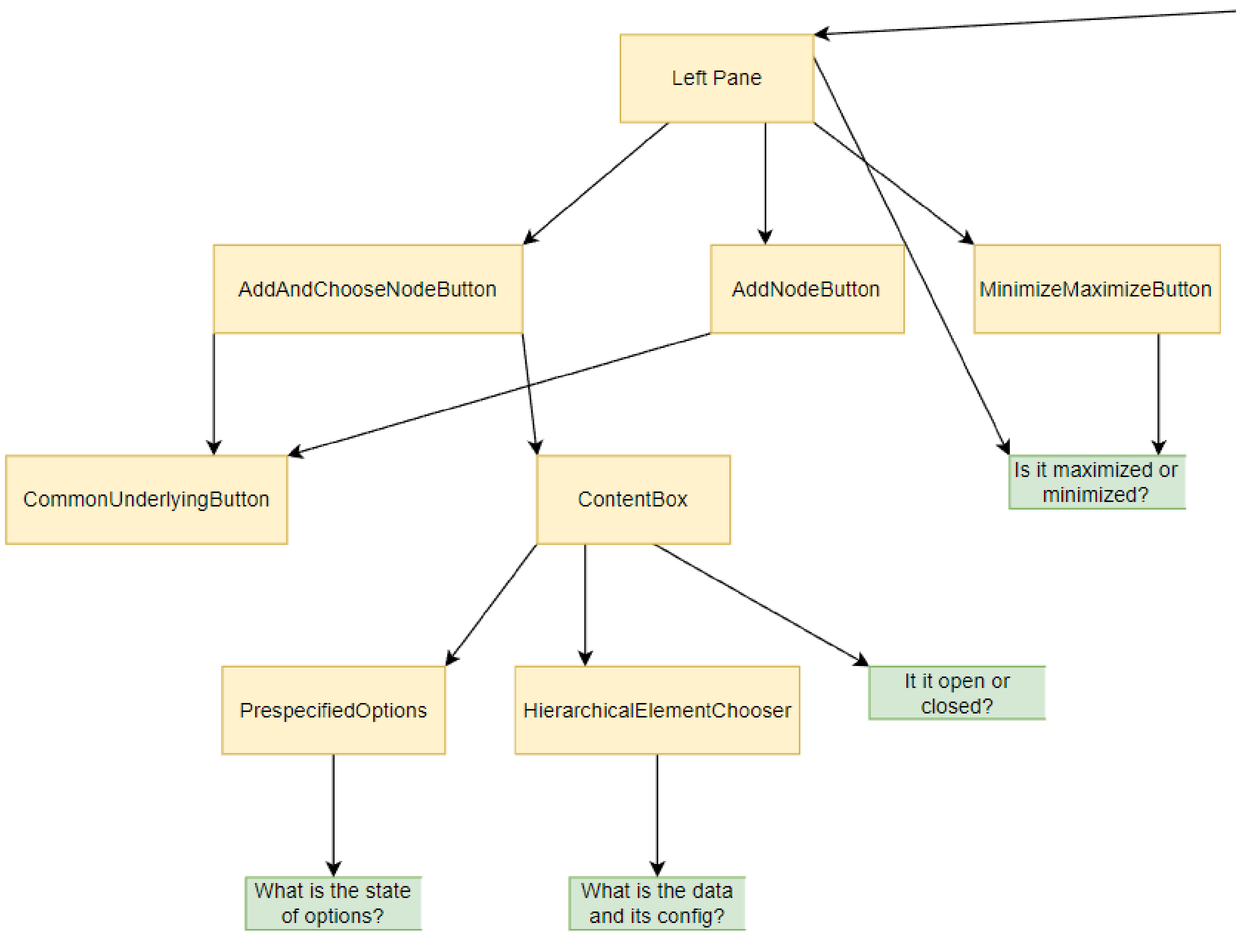


Figure 4.13 - A subtree of the component diagram displaying the components and their stateful information used to develop the Left Pane of DNNCASE Graph Editor.

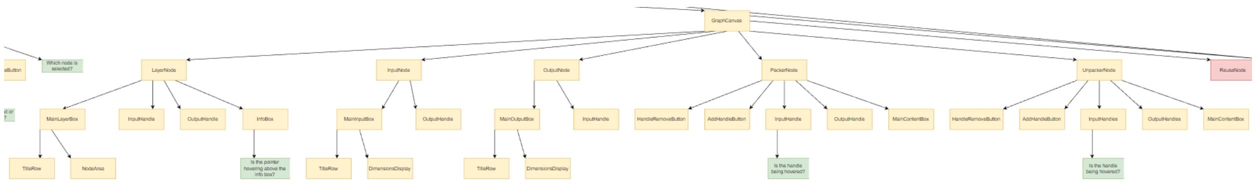


Figure 4.14 - A part of the component diagram displaying components for various nodes to be used on the Canvas in a code's graph.

Once the designs for the UI were ready, we prepared preliminary designs for the interaction between the UI and the system backend. While the most recent designs are elaborated in Chapter 6, we briefly mention the initial designs to convey our thought process and designing workflow.

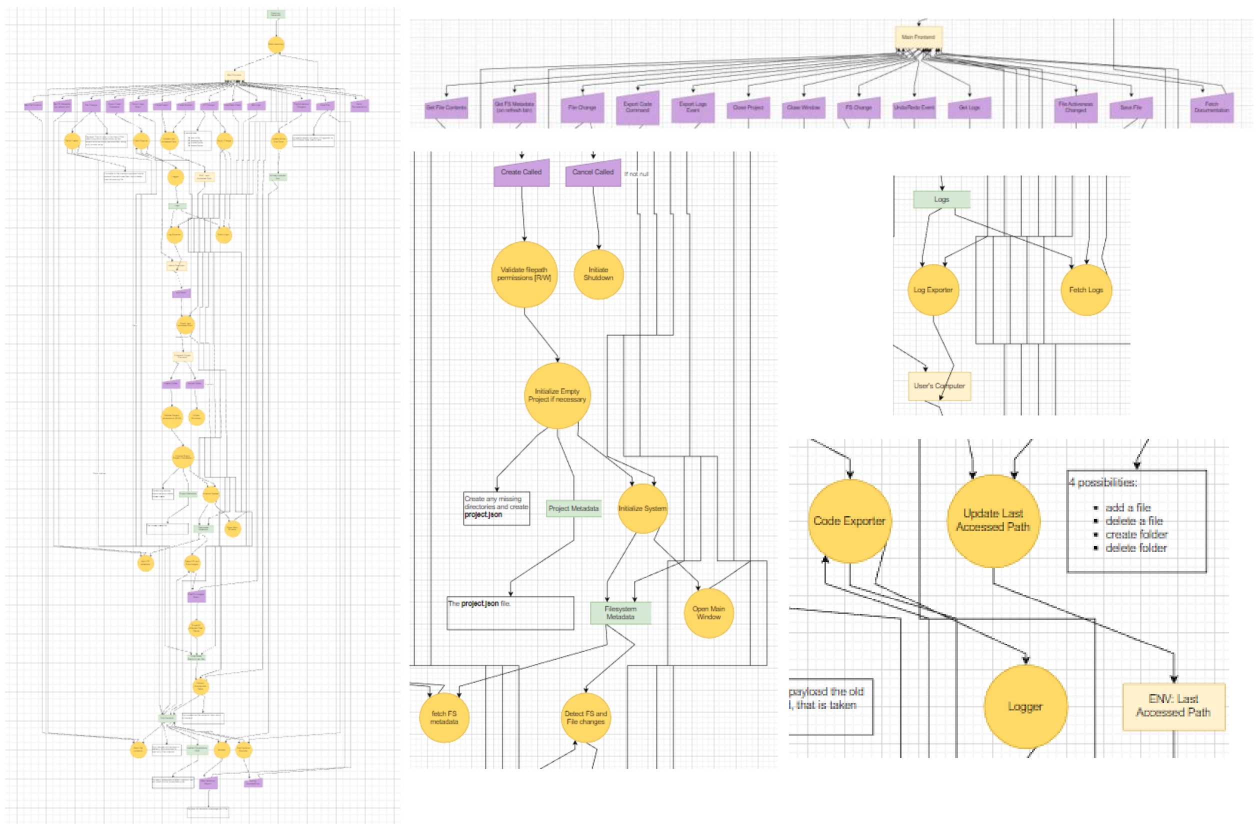


Figure 4.15 - Various snapshots of the Level-1 Dataflow Diagram displaying the UI and backend interaction. These images are for reference purposes only. To view the full schematic, please access it from the link: https://github.com/rohan843/dnncase/blob/c0b4761d1242ec0cba65841d3a6554e9129d09e7/processing%20and%20data%20store%20structure.drawio. The "draw.io" software must be used to open the file.

Now that the UI components were designed, we shifted our focus onto the second parallel path of development – the language and compiler design.

From the very beginning of the requirements elicitation process, our literature survey had shown us the need for a visual graphical dataflow language. We began the development of our visual language, **DNNLang** in a manner similar to what we followed for our overall system, as described so far. In the case of language development, we had a different set of challenges. Thanks to the multiple streamlined DL projects, libraries, documentations and courses, the tasks performed in DL were known to us beforehand. A brief outline of those tasks is as follows:

1. Model Architecture Creation
2. Model Compilation Procedure
3. Data loading and preprocessing
4. Within-epoch Model Training Workflow description
5. Overall Model Training Loop

6. Declaration of other functions such as loss functions.

We have presented this outline in our problem statement in Chapter 2 as well.

We began our process by surveying the TensorFlow and Keras documentation for understanding the python ecosystem. We organized our findings in another mind map. On account of this being too large for the page, we present salient snapshots of the mind map below. The complete mind map is available here: https://github.com/rohan843/dnncase/blob/2ebcade6e92bf0f29317b64f5d182c7d2e9e2352/DNNLang.pdf as a PDF file.

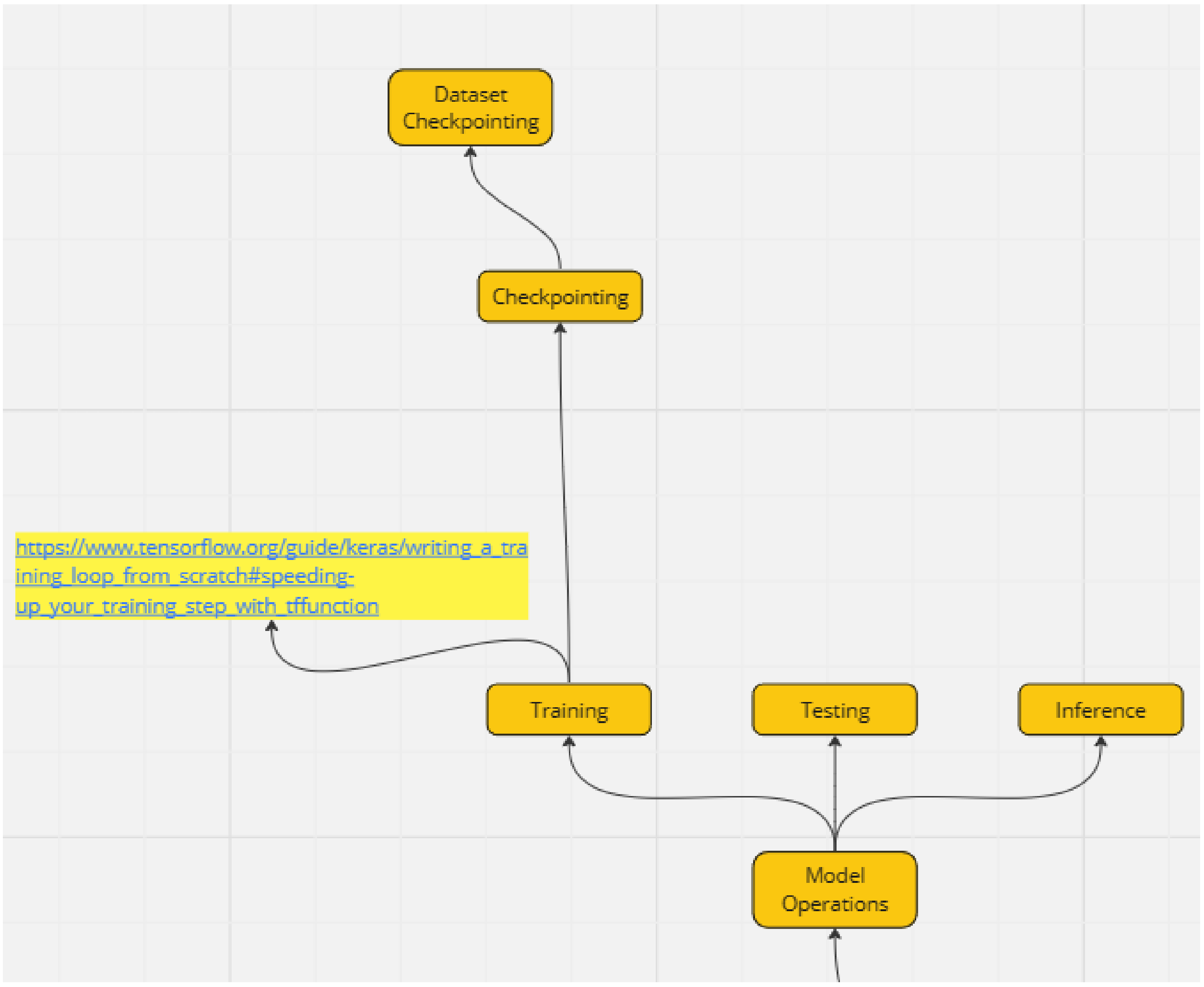

Dataset Checkpointing
Checkpointing
https://www.tensorflow.org/guide/keras/writing_a_training_loop_from_scratch#speeding-up_your_training_step_with_tffunction
Training
Testing
Inference
Model Operations

Figure 4.16 - Section of the Mind Map showing operations that a model can undergo.

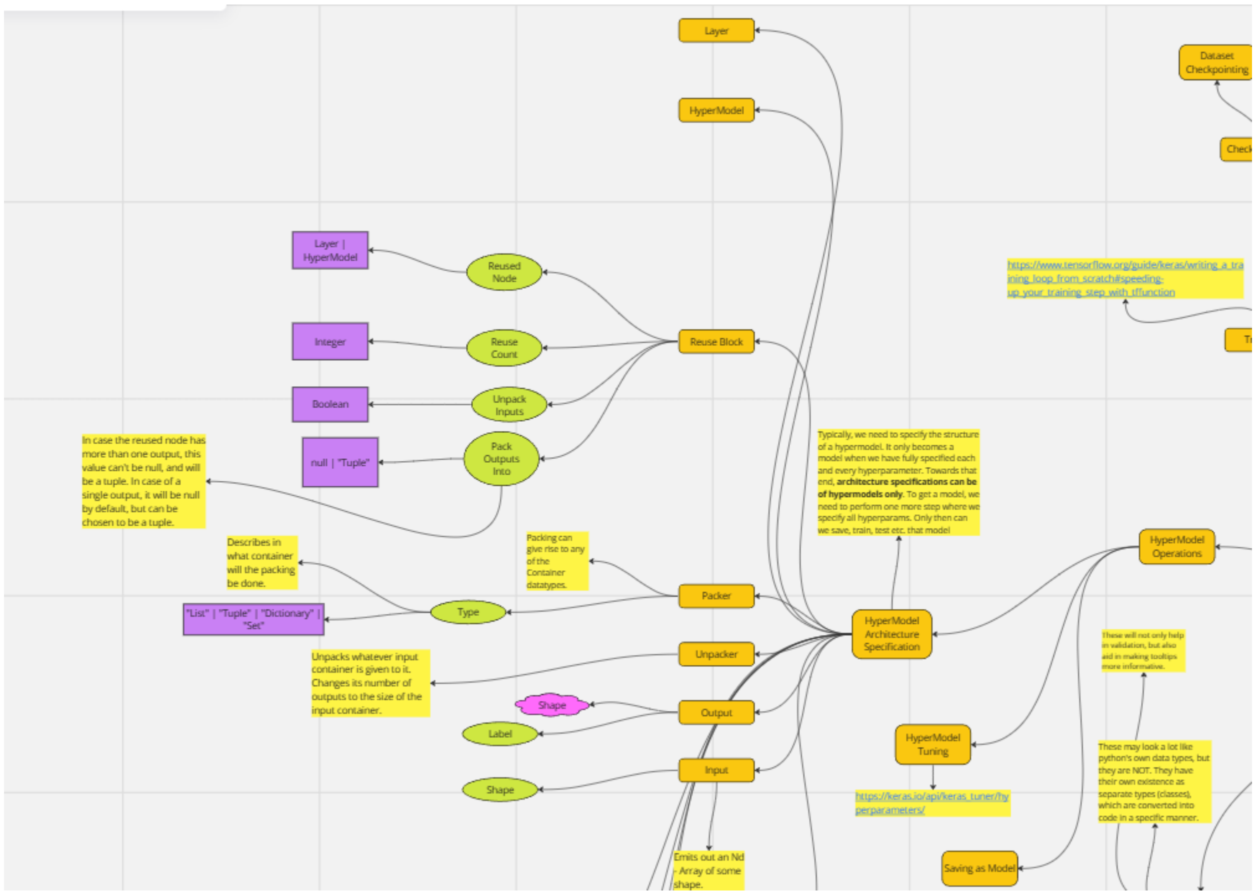

Layer
HyperModel
Dataset Checkpointing
Check
Layer | HyperModel
Reused Node
Integer
Reuse Count
Reuse Block
Boolean
Unpack Inputs
null | "Tuple"
Pack Outputs Into
In case the reused node has more than one output, this value can't be null, and will be a tuple. In case of a single output, it will be null by default, but can be chosen to be a tuple.
https://www.tensorflow.org/guide/keras/writing_a_training_loop_from_scratch#speeding-up_your_training_step_with_tffunction
Typically, we need to specify the structure of a hypermodel. It only becomes a model when we have fully specified each and every hyperparameter. Towards that end, **architecture specifications can be of hypermodels only**. To get a model, we need to perform one more step where we specify all hyperparams. Only then can we save, train, test etc. that model
Describes in what container will the packing be done.
Packing can give rise to any of the Container datatypes.
HyperModel Operations
Packer
"List" | "Tuple" | "Dictionary" | "Set"
Type
HyperModel Architecture Specification
These will not only help in validation, but also aid in making tooltips more informative.
Unpacker
Unpacks whatever input container is given to it. Changes its number of outputs to the size of the input container.
Shape
Output
Label
HyperModel Tuning
Input
Shape
These may look a lot like python's own data types, but they are NOT. They have their own existence as separate types (classes), which are converted into code in a specific manner.
https://keras.io/api/keras_tuner/hyperparameters/
Emits out an Nd - Array of some shape.
Saving as Model

Figure 4.17 - Mind Map section showing entities relevant to model and hypermodel architecture creation.

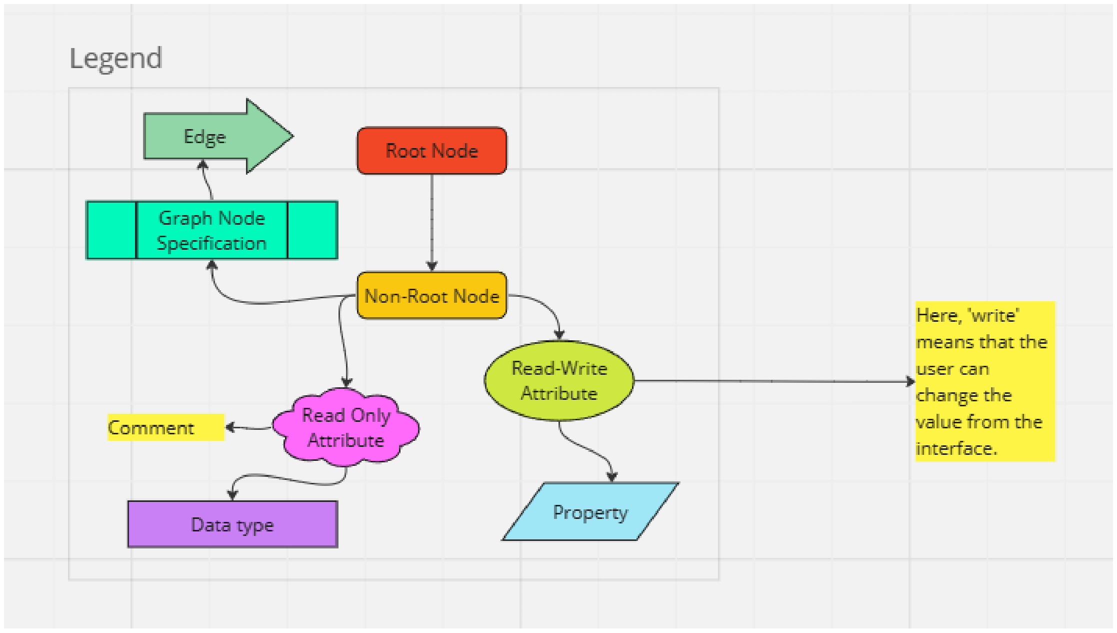
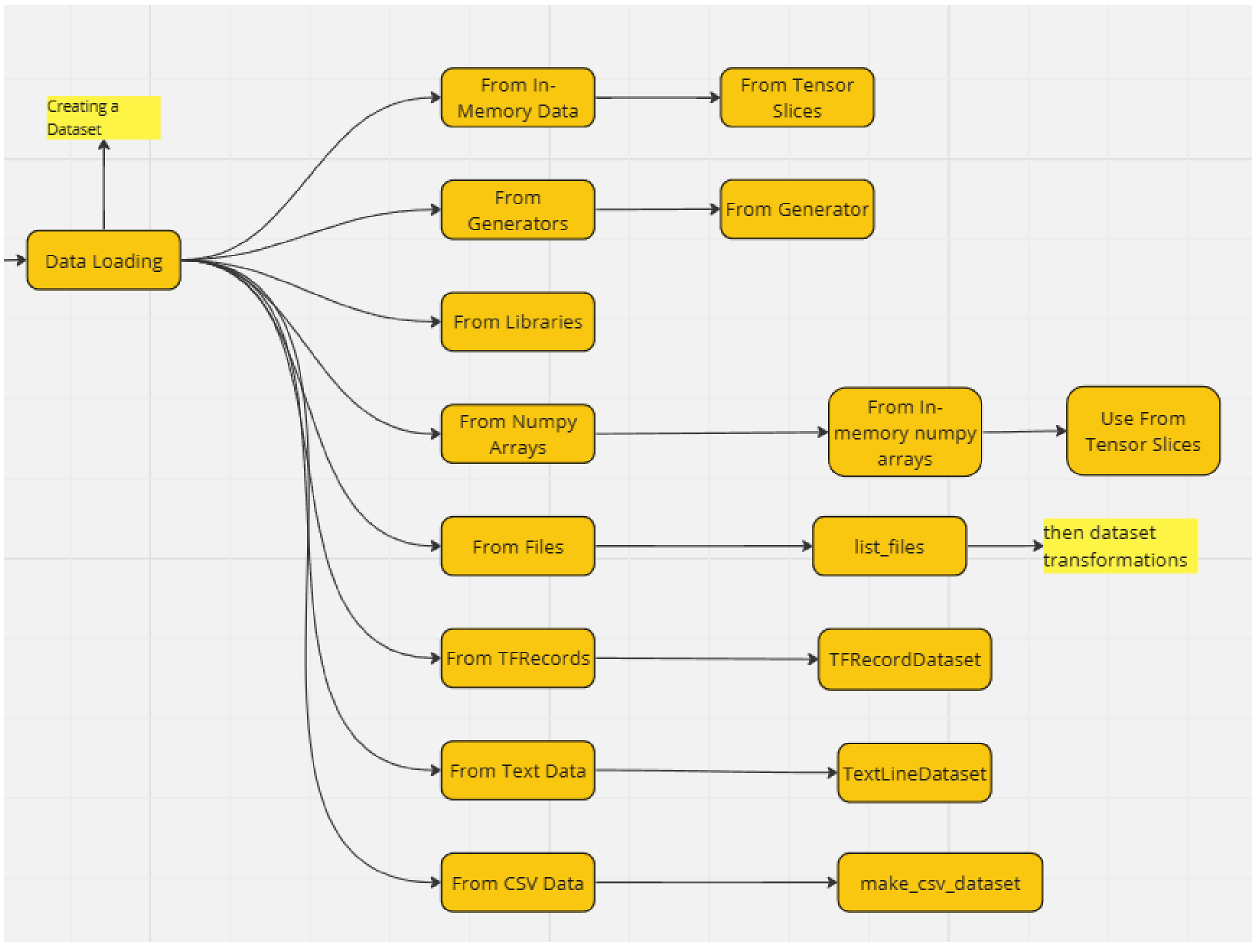


Figure 4.18 - The figure shows the notation followed in the mind map.



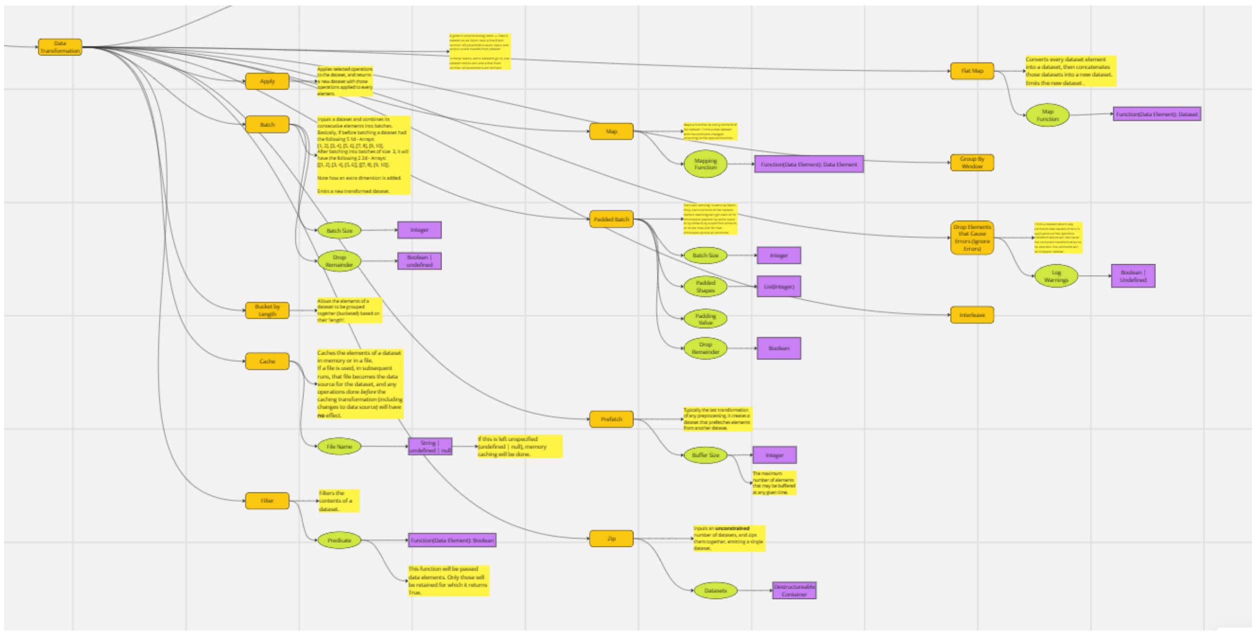


Figure 4.19 - Sections of the Mind Map displaying data loading and transformation operations.

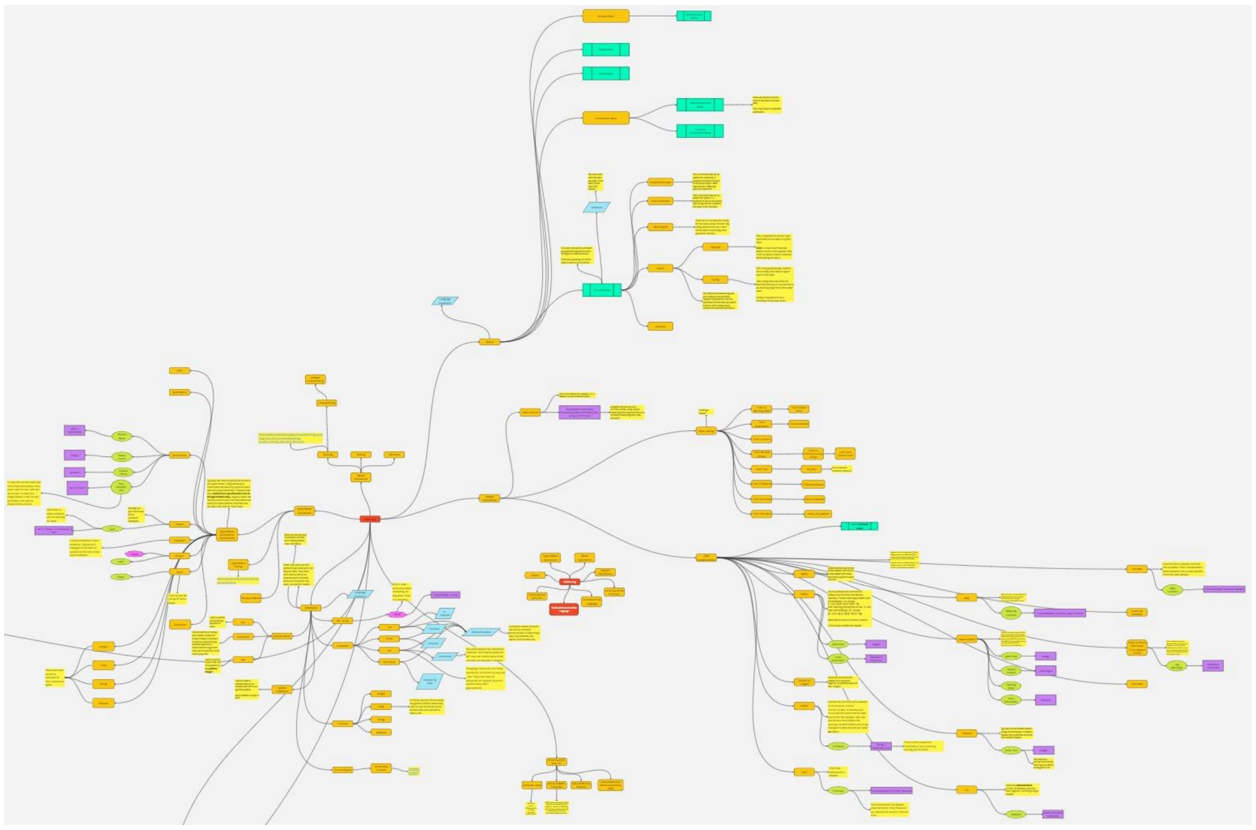
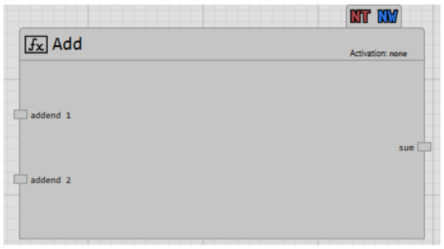


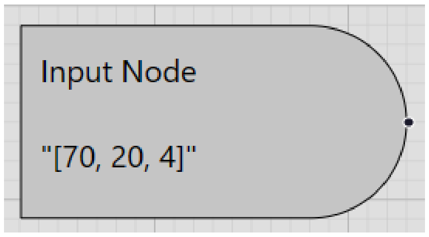
Figure 4.20 - Figure displaying all sections of the Mind Map relevant so far in the discussion. The complete Mind Map will follow later in this Chapter.

Based on the various tasks our language is meant to support, we began the construction of the most basic node in our language – the **function node**. We conceptualized this node as a node that takes in zero or more data streams, performs some processing based on them, and optionally emits a single data stream. We further analyzed our domain to conceptualize the notion of "**data**" and "**datatype**" in the context of our language. We further created various nodes to perform functions such as branching and iteration, in line with our functional requirements. We also conceptualized the fundamental entity that gets passed to the compiler – an **artifact**.

Further domain analysis (which we performed by analyzing sample Keras codes from Keras and TensorFlow websites) showed that the language still lacked important components. There were different kinds of responsibilities that different function nodes must display. To make the visual aspect in line with this, we created the concept of "**nodetype**". A node's nodetype defines its visual aspects. This means that a function node of type `*layer/add*` which processes input data to give outputs may look like:



While another function node of type `*layer/input*` which is a data source may look like:



This distinction in display makes it clear that one node is meant for data transformation, while the other is a data source. This is a visual distinction, while from a language perspective, both nodes are logically the same.

Another component that our language lacked as of now was the ability to treat different artefacts differently. To explain this point, we draw a parallel between DL workflows and website development. When a website is built, its structure is declaratively specified via HTML, much like the architecture of a model is specified by a graph of layers. The styling of the website is specified via CSS again declaratively after construction of the barebone HTML, much like the compilation of a model is specified after its architecture is described. Lastly, websites have functionality via JavaScript. Similarly, DL workflows have functionality such as defining custom losses, training loops and so on. This analogy shows that DL workflows are neither fully declarative, nor fully functional. They have both aspects.

Continuing this line of thought, we developed the idea of **declarative artefacts** and **procedural artefacts**. But issues persisted. There are different declarative and different procedural artefacts. To resolve this, we conceptualized the concept of "**artefacttype**". The artefacttype of an artefact is used to tell the compile what interpretation process to apply to any given artefact. Even though an artefact may have its own artefacttype, it is still treated as a function when imported in other artefacts. This solution enables both custom – processing of artefacts, and interoperability between different artefacts (such as by invoking one artefact within the graph of another).

For more information on the language aspects, please visit Appendix A, and also please view the Mind Map (link provided previously).

In line with the language developed, we constructed the logic and function documents and diagrams for the compiler. The discussion of the compiler is provided in – depth in Chapter 7.

The following images display the parts of the mind map relevant to the finalized DNNLang, and the compiler.

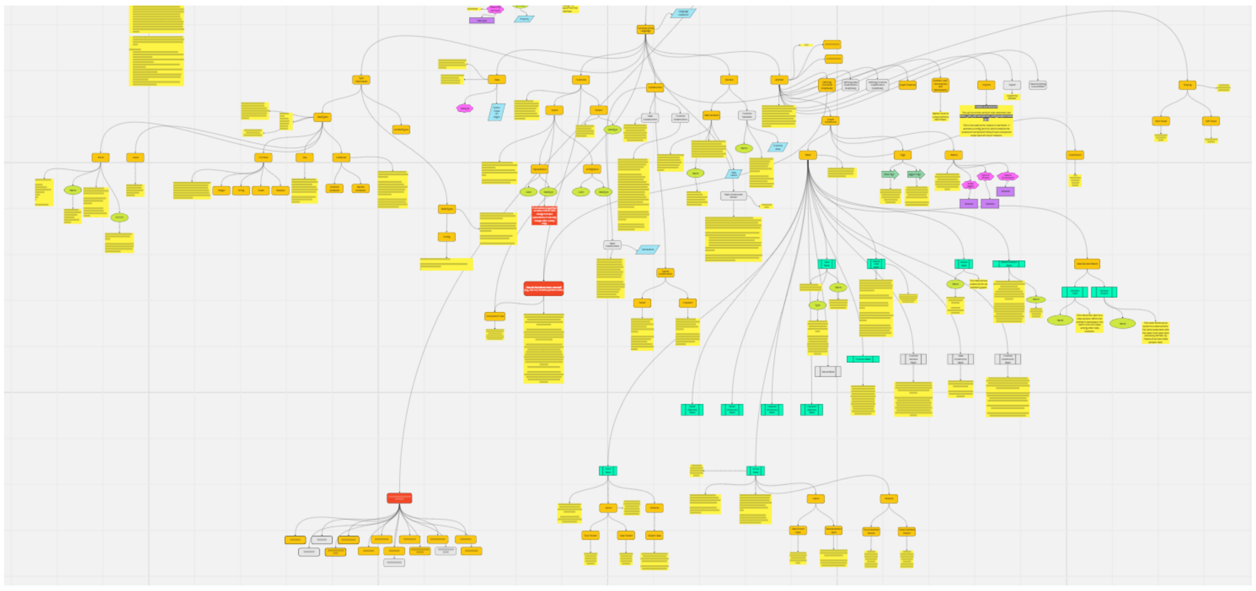


Figure 4.21 - The part of the Mind Map displaying elements of the language.

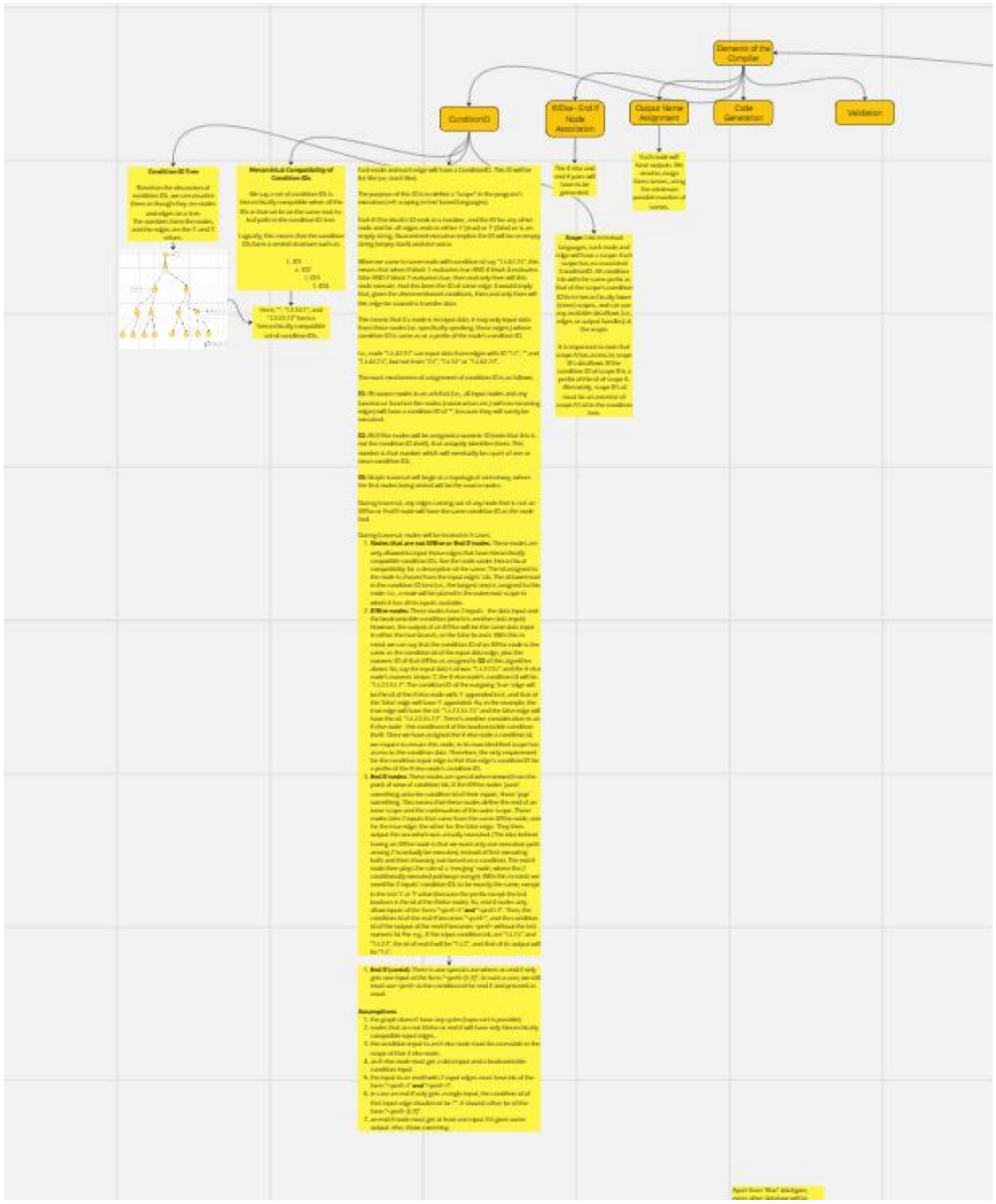


Figure 4.22 - The part of the Mind Map displaying elements of the compiler.

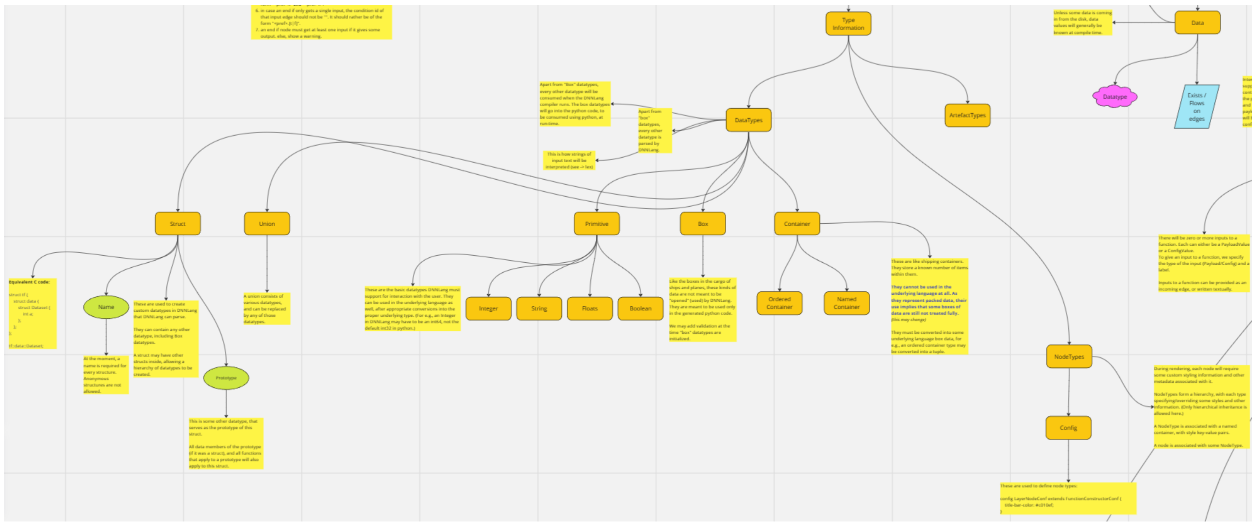
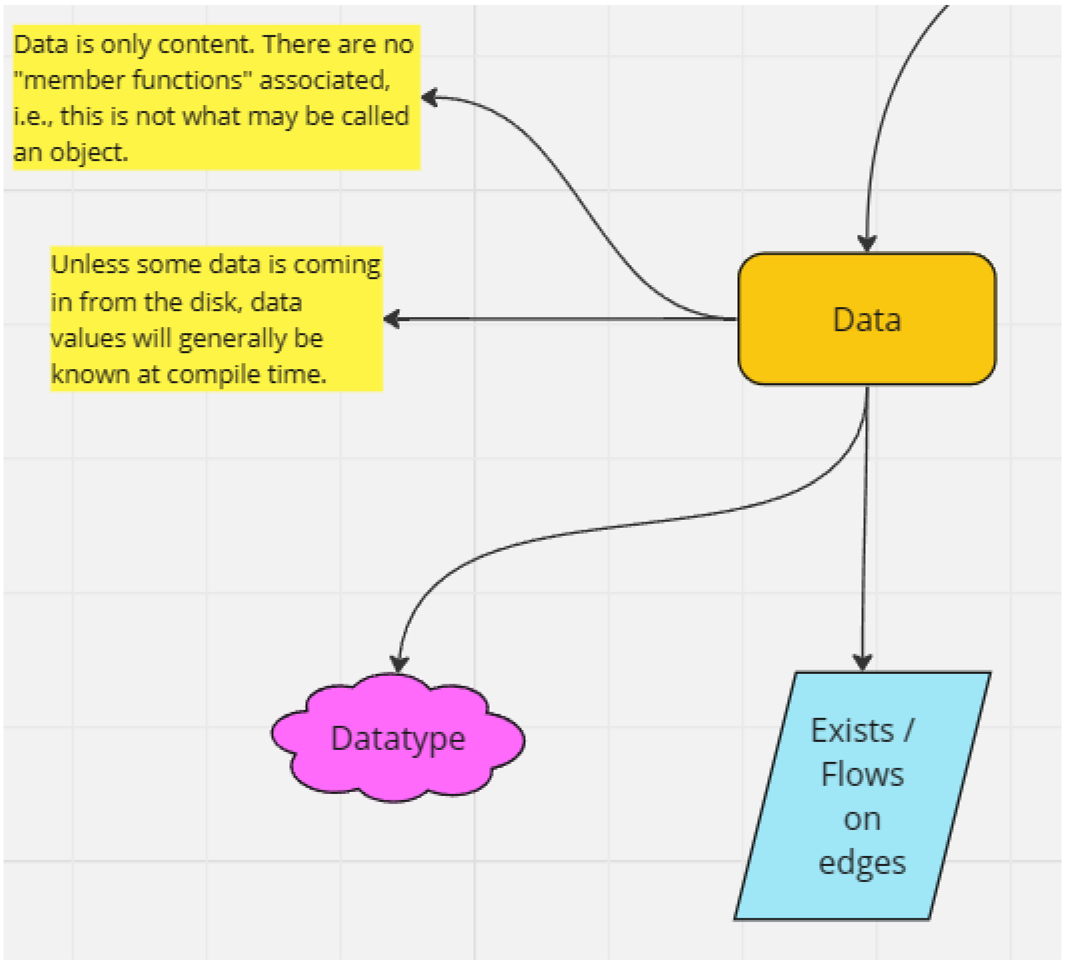
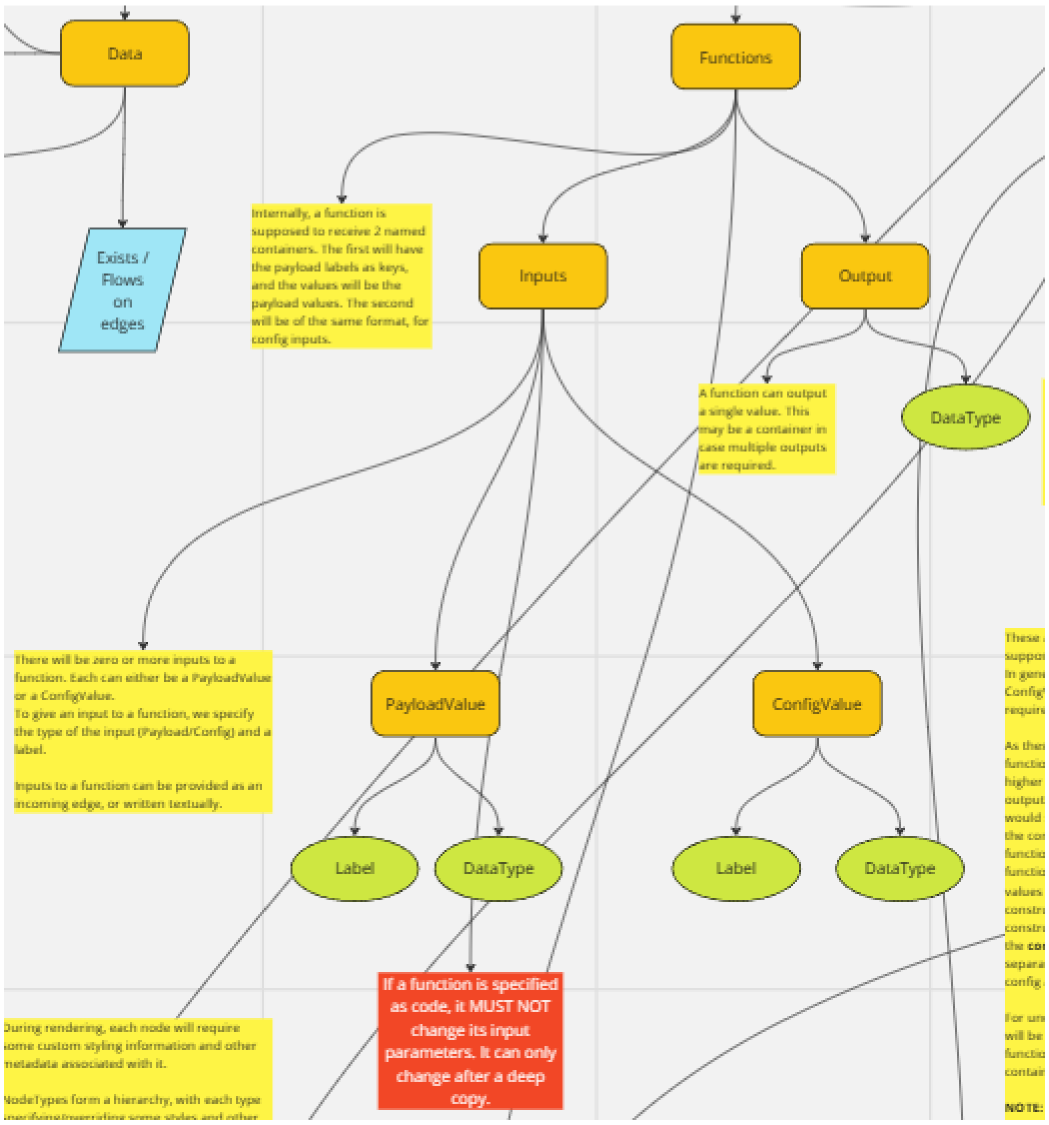


Figure 4.23 - The part of the Mind Map displaying information on datatypes, nodetypes, and artefacttypes.


Data is only content. There are no "member functions" associated, i.e., this is not what may be called an object.

Unless some data is coming in from the disk, data values will generally be known at compile time.

Figure 4.24 - The part of the Mind Map displaying the part on the concept of Data.



Internally, a function is supposed to receive 2 named containers. The first will have the payload labels as keys, and the values will be the payload values. The second will be of the same format, for config inputs.

A function can output a single value. This may be a container in case multiple outputs are required.

There will be zero or more inputs to a function. Each can either be a PayloadValue or a ConfigValue.
To give an input to a function, we specify the type of the input (Payload/Config) and a label.

Inputs to a function can be provided as an incoming edge, or written textually.

If a function is specified as code, it MUST NOT change its input parameters. It can only change after a deep copy.

During rendering, each node will require some custom styling information and other metadata associated with it.

NodeTypes form a hierarchy, with each type specifying/overriding some styles and other

Figure 4.25 - The part of the Mind Map displaying information about the concept of a function.

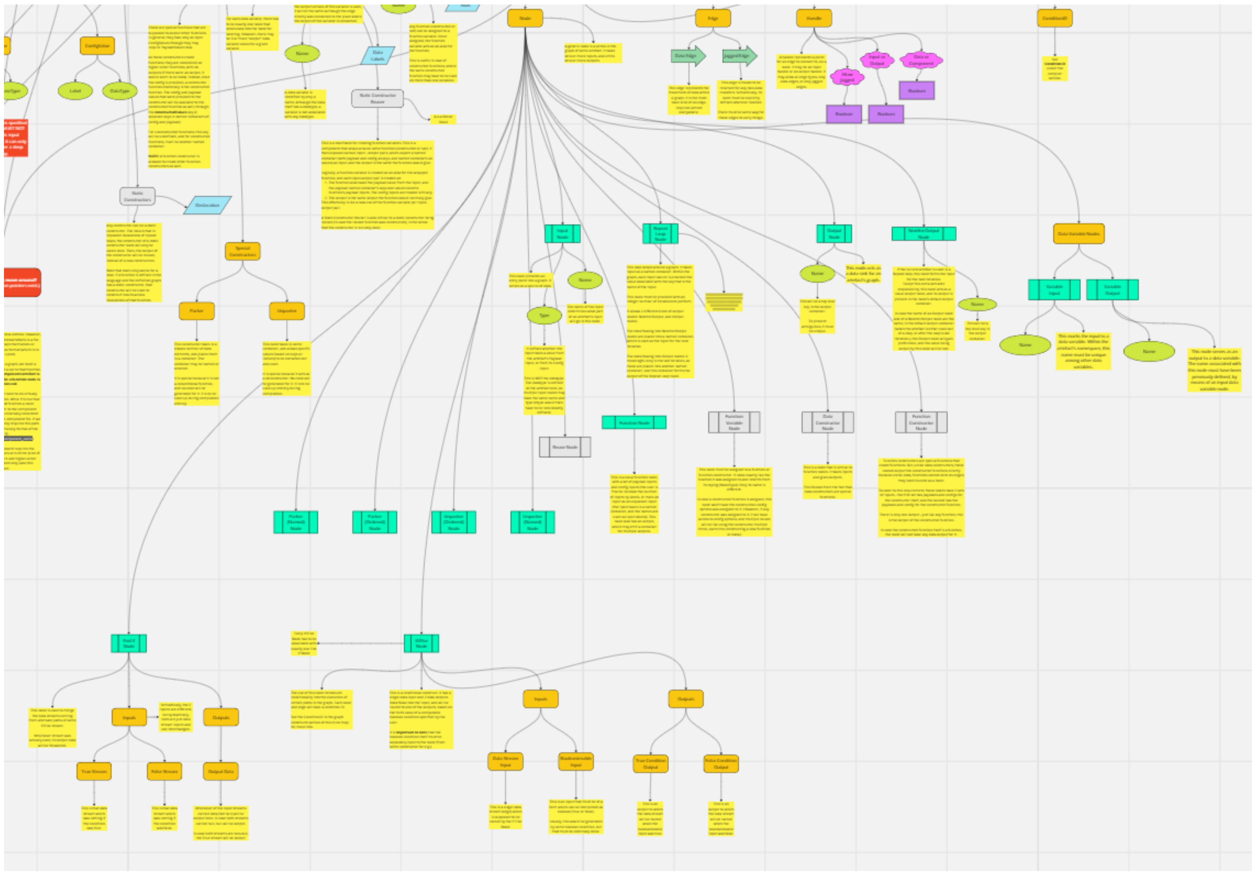


Figure 4.26 - The part of the mind map displaying information about nodes (cyan boxes).

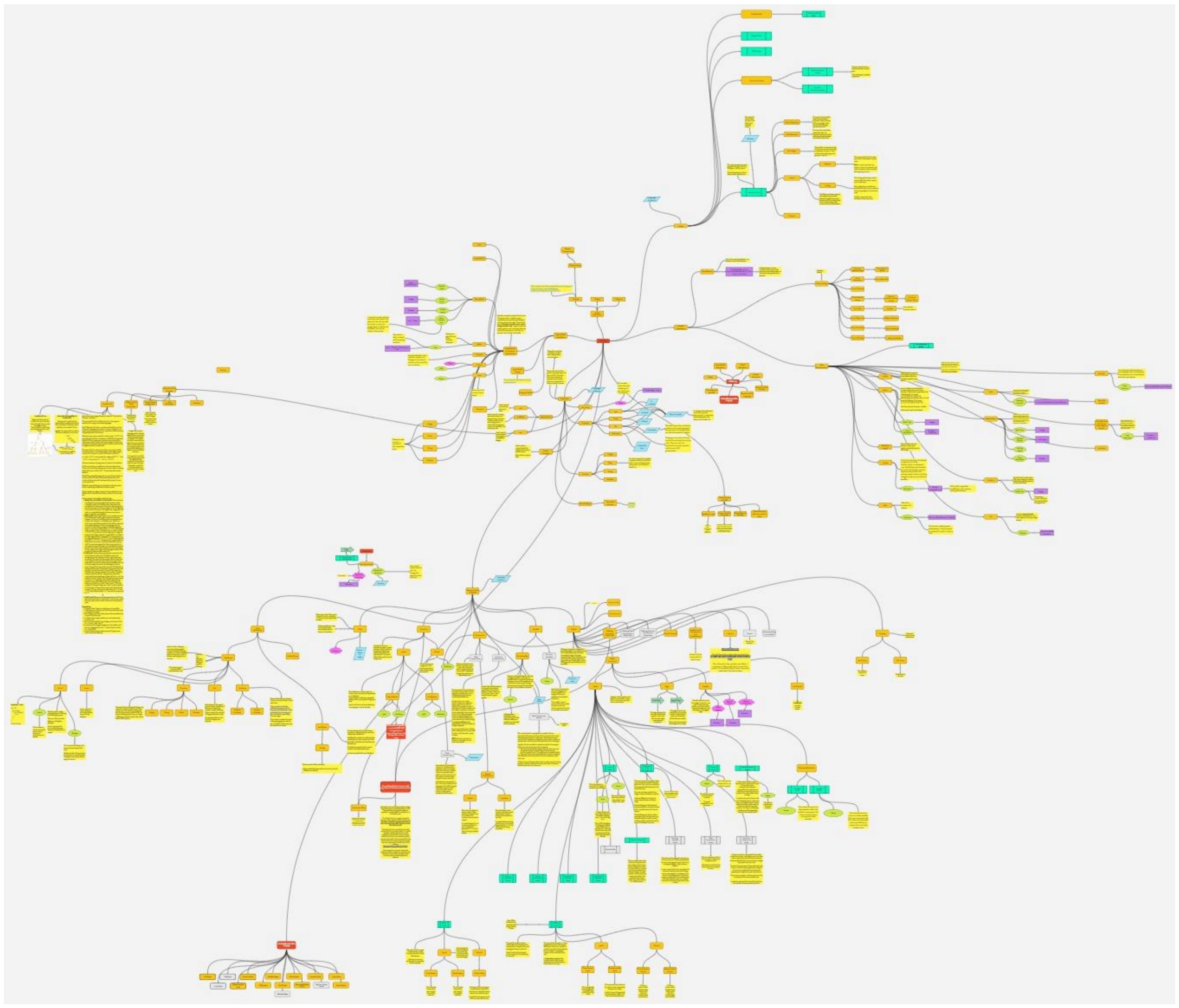


Figure 4.27 - The complete DNNLang Mind Map.

This concludes the discussion on requirements analysis and system design.

Chapter 5: Implementation and Challenges

Our Software Development Life Cycle

1. Our development cycle started with gathering the requirements (both functional and non-functional), post which we created mind-maps to add more enhanced set of requirements to each component and to have a general outline of what needs to be built.
2. After that we went forward with the development of system, in phase one we started focused on the UI and code generation algorithm. We thoroughly studied the Keras documentation to understand various constructs used in deep learning and making sure our architecture supports all of them. The System contains all the constructs of deep learning in the form of nodes and we can connect these nodes with edges. For e.g.: there are nodes of Conv2d, Dense Layer and other layers.
3. We also have nodes for input (specifying the dataset) and output (getting the output like probability, etc.).
4. While developing the system we stumbled upon a very rarely used network Architecture called Siamese network that is often used in tasks involving similarity or distance measurement. It consists of two identical subnetworks which share the same parameters and weights.
5. These subnetworks process two different inputs separately, and then some similarity metric is applied to the outputs of these subnetworks to determine the similarity or dissimilarity between the inputs. In a Siamese network, the two subnetworks share the same parameters.
6. This means that the weights and biases of the layers in each subnetwork are identical. For supporting this we came up with a Idea of Reuse blocks which helps us do the same, with Reuse block we can wrap a layer inside and have as many inputs and outputs to this as possible.
7. Siamese networks are commonly used in tasks such as signature verification, face recognition, similarity-based recommendation systems, and more.
8. They are particularly useful when dealing with tasks where labeled training data is scarce, as they can be trained using pairs of similar and dissimilar examples rather than relying solely on labeled data.
9. With this requirement, Reuse blocks can help us re-use the same layer parameters and apply on other data. Other important problem which we saw while development was that it was

difficult to create deep networks on UI as deep networks requires a large number of inputs travelling from one side to other which may cause confusion and to solve this we came up with concept of Packer/Unpacker Nodes, through which we can pack a set of inputs and now travel on the canvas with just one input line and unpack it where required, this makes the development of neural networks fast and clean.

10. Also, to support adding different parameters and hyper-parameters associated with Layers we introduced a Right Pane which helps us to manually put those specifically for each node, all you have to do is select the node and open Right pane, put the details.
11. Apart from development, we also focused on developing multiple dataflow diagrams for systematic development of front-end and back-end, specifically for code generation algorithm we created an entire workflow to analyze the working and deal with any edge cases if exists.
12. After first phase of development and testing we went forward with doing a survey to understand the usability of system and to collect feedbacks.
13. Once we were done with survey, we re-iterated on requirements and updated mind-maps to adapt the latest requirements and behaviors which we refined as part of survey.
14. One major point we got as feedback was to support the entire deep learning pipeline including the training, testing and hyper-parameter tuning of the networks.
15. In the process we came up with multiple new constructs on UI like concept of artifacts, for-loop node, if-else node etc. Our Aim is to support all types training, testing, basic and custom hyper-parameter tuning. Custom hyperparameter tuning would require repetition of certain code and taking decision based on some if/else condition so we introduced the constructs of if/else and for-loops, etc.

Our Research Survey

1. We conducted the research survey with college students to understand the usability and to get feedbacks about our system.

2. We started with an introduction of our system and explained what purpose it serves.

3. Post which we asked them to explore our system and ask if they have any doubts using it.

4. Then we gave them a Kaggle notebook with some Deep Learning tasks on MINST dataset and Fashion MINST dataset. They need to do the task by using our system i.e. use our system to draw a deep neural network architecture, generate code and use that code in the Kaggle notebook.

5. At the end, we asked them to fill a google form which contains questions on usability and feedback.

Challenges That We Faced

1. Our first challenge was the requirements gathering process. Although we had studied it in Software Engineering, a real – life process for a project of such a large scope is a complex process that required a great amount of analysis and ideation.
2. We also found learning the intricacies of our chosen tech stack challenging, especially, understanding the process model and IPC of ElectronJS.
3. Once our system was developed, we had to conduct a survey for evaluating it. As this was a new experience for us, it required a lot of planning. This was a challenging task.
4. Creating a new language is not a common task. It was our objective from the beginning, but the process we followed in creating it was not a straightforward one. We were met with lots of difficulties along the way. Creating a compiler for this language was also a challenging task, which we spent a large amount of time designing and developing towards.

Software That We Used

This project has a well–defined scope, but within it lie various components, and different facets of the system require different technologies to implement. Based on this, we have identified various libraries and packages that we will require in the implementation:

1. **ReactJS**: This is a JavaScript framework that can be used to create various web-based interfaces in a modular manner. We will use this to create various frontends for our system.
2. **ElectronJS**: This is a JavaScript-based library that can be used to create windowed interfaces. We will use this, as we have in our prototypes, to create the main screens.
3. **React – Flow**: This is a library that allows us to render graphical data.
4. **Keras**: This is the main API based on which our system operates. We use its constructs to run generated backend code.

5. **Figma**: This software allows us to create graphical interfaces on a screen so that we can visualize them before we begin designing.
6. **Diagram.net (Draw.io)**: This is a diagramming software, wherein we can create schematics for various design diagrams.
7. **Miro**: This is a mind mapping software allowing us to create mind maps.
8. **GitHub/git**: We use these tools for version controlling and maintaining a centralized codebase.

Chapter 6: Runtime Environment Design and Frontend

As stated before, a runtime environment and a graph canvas are very important to visual languages. We developed a graph editor and an IDE. In this chapter, we aim to provide the details of both of them. We also provide a description of the way the environment treats the file system.

The IDE

The IDE consists of 6 major parts:

1. Left Bar
2. Left Panel
3. Bottom Bar
4. Bottom Panel
5. Top Bar
6. Workspace

The left bar and the left panel are pictured below:

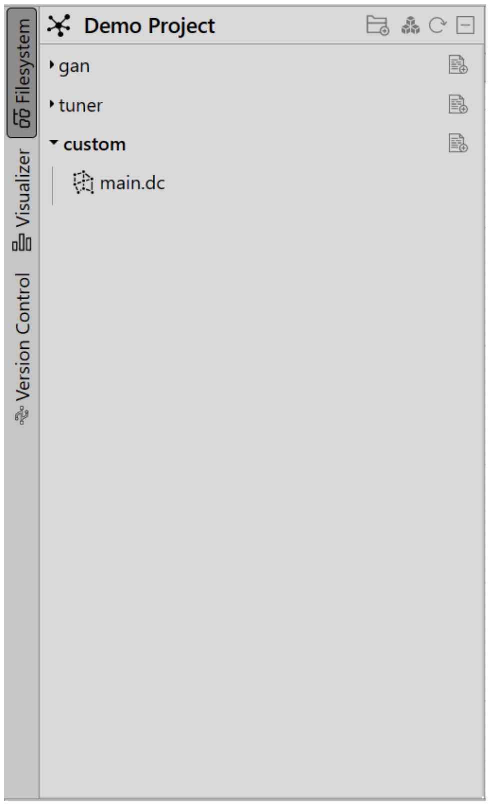


Figure 6.1 - The Left Bar and the Left Panel.

These contain access to the filesystem view, the visualizer, and the version control view. So far, our system requirements and therefore the aim for our project only necessitated the development of the filesystem view, however our system features do include visualizing and version controlling capabilities. Should DNNCASE be further developed, they will be accessible here.

The bottom bar and the bottom panel are pictured below:

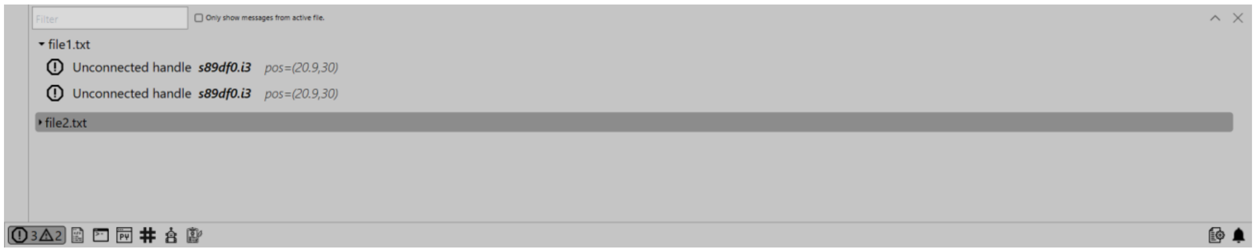

Filter
Only show messages from active file.
file1.txt
Unconnected handle s89df0.i3 pos=(20.9,30)
Unconnected handle s89df0.i3 pos=(20.9,30)
file2.txt
3 2

Figure 6.2 - The Bottom Bar and the Bottom Panel.

The bottom bar provides access to traditionally “landscape” information, such as validation messages, logs, etc. We outline a virtual assistant in Chapter 10 (Future Work). When created, the assistant will also be present in the bottom bar.

The top bar is pictured below:

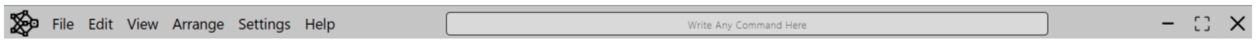


Figure 6.3 - The Top Bar

Besides the usual menus, logo and minimize/maximize/close window buttons, the top bar has a command palette. Here, users are allowed to enter some command from a set of commands for quick execution, rather than searching through menus and dialog boxes. The command palette is another feature that is not yet included in out system requirements, but has been analyzed in Chapter 4 as an unbounded feature and a Mind Map grouping.

The workspace is pictured below:

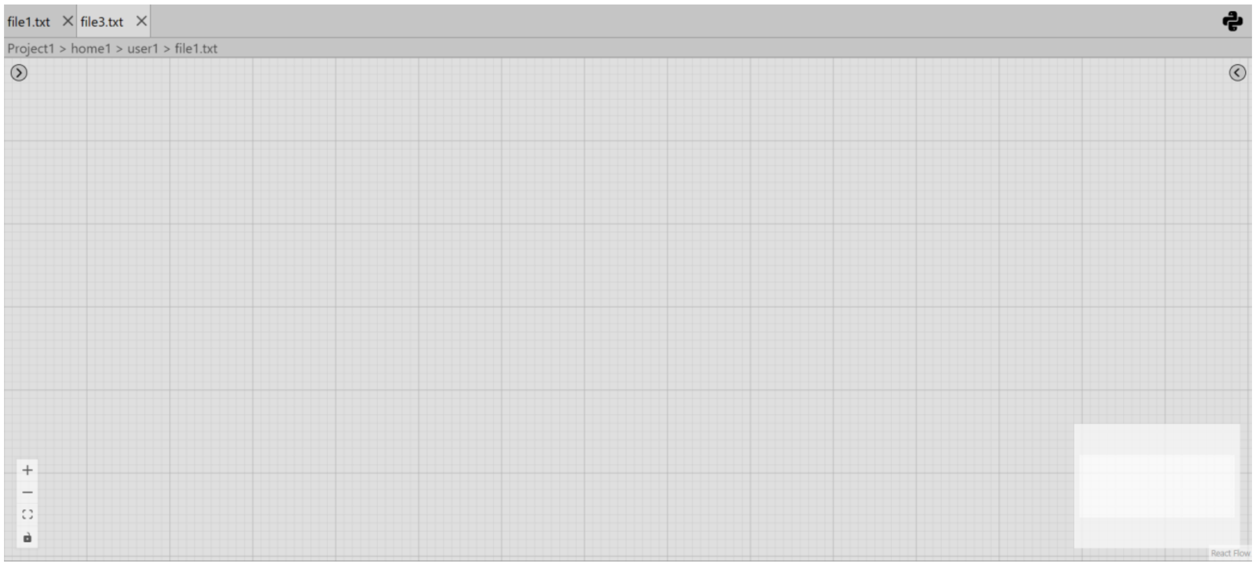


Figure 6.4 - The Workspace.

This is the workspace. The majority of its screen share is taken up by the graph editor. However, at its top, there is the tab bar, showing various tabs and the code generator button on the right.

The bar right below the tab bar is the address bar, displaying the address of the currently active file.

Below that is the graph editor. It is described in the following section.

The Graph Editor

The graph editor consists of the following 5 parts:

1. Visual Canvas
2. Control Buttons
3. Mini Map
4. Left Pane
5. Right Pane

The Visual Canvas is pictured below:

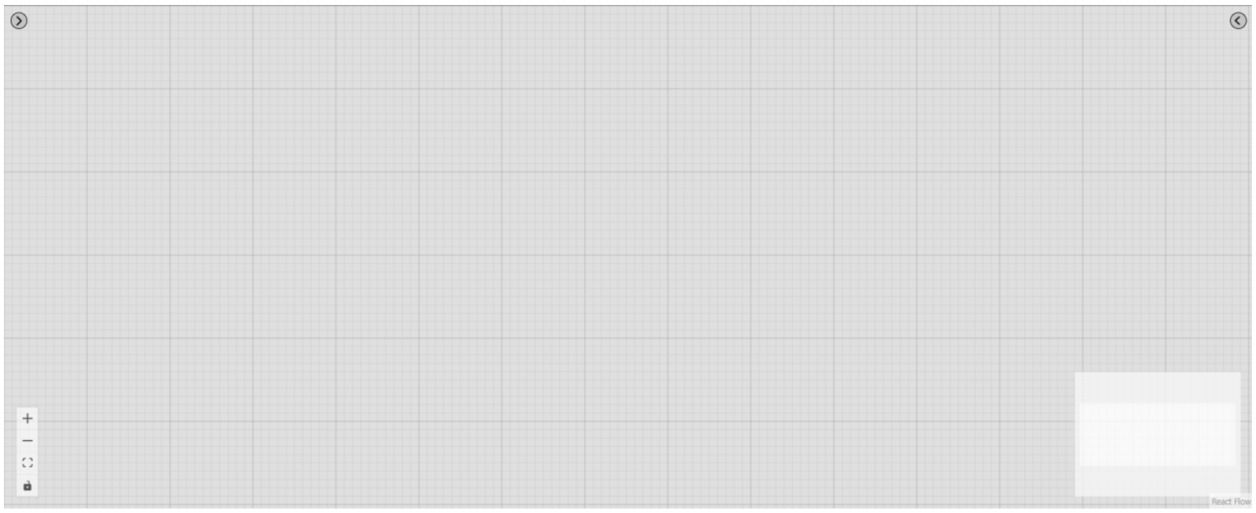


Figure 6.5 - The Visual Canvas.

The visual canvas is the area where the user creates graphs. It is a 2 – dimensional coordinate plane with an overlay grid. The canvas supports zooming and panning. It also supports drag – and – drop of nodes and creation of edges.

The Control Buttons are pictured below:

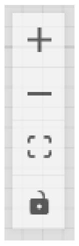


Figure 6.6 - The Control Buttons.

These 4 buttons are (top to bottom) zoom in, zoom out, fit to window and disable edits. The fit to window button makes the graph in the canvas be fully visible to the user at once. The disable edits button disables dragging/deletion of nodes and creation/deletion of edges, allowing for a simple code browsing.

The Mini Map is pictured below:

Figure 6.7 - The Mini Map.

This mini map is created as a visual aid to the programmer. It displays the entire graph in the background, and the user's current viewport position in the foreground. It also allows zooming and panning.

The left pane is pictured below:

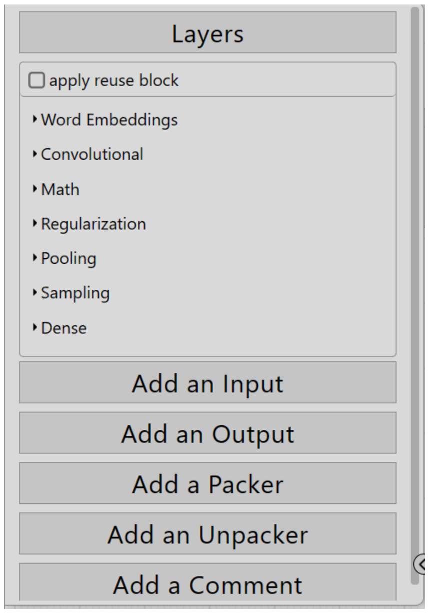
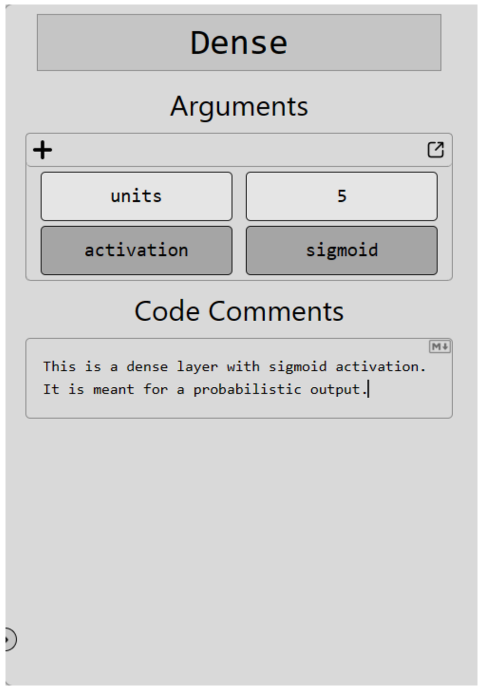


Figure 6.8 - The Left Pane.

This pane contains various nodes that can be added onto the Visual Canvas by the user.

The right pane is pictured below:



This is a dense layer with sigmoid activation.
It is meant for a probabilistic output.

Figure 6.9 - The Right Pane.

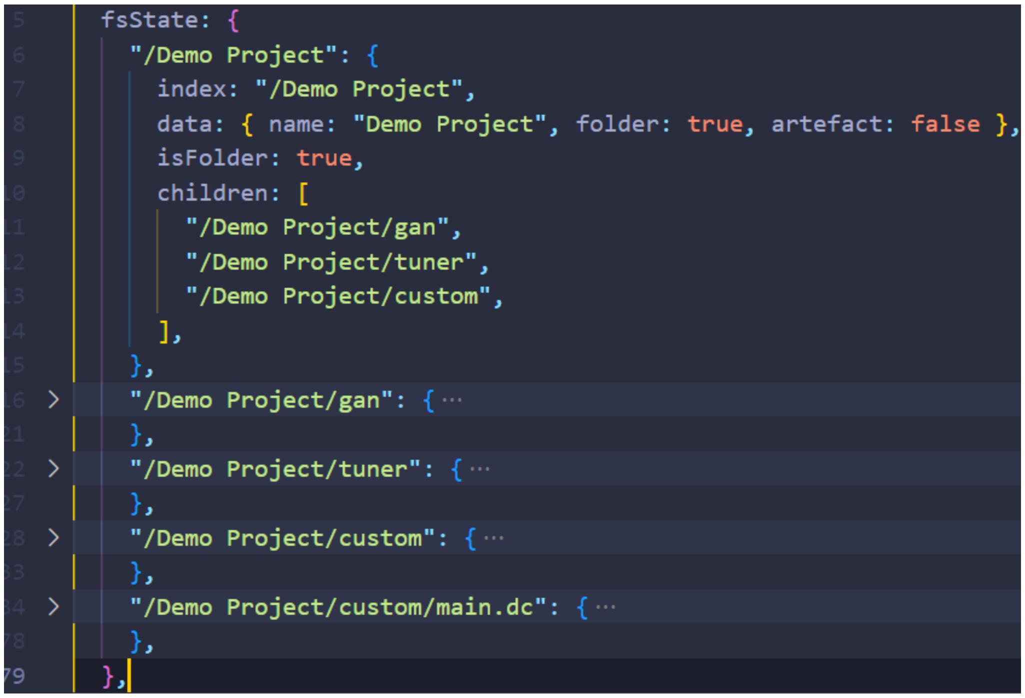
This pane allows the user to modify the properties of any specific node, and to also add any internal (code) comments as needed. The user can view and edit the properties of the activated (selected) node at a time.

The State of File System

The runtime environment requires 2 important fields of information about the file system:

1. What files come under the current project, and their contents.
2. What files are currently opened.

To track the files in the current project folder, the following code snippet shows the internal structure:

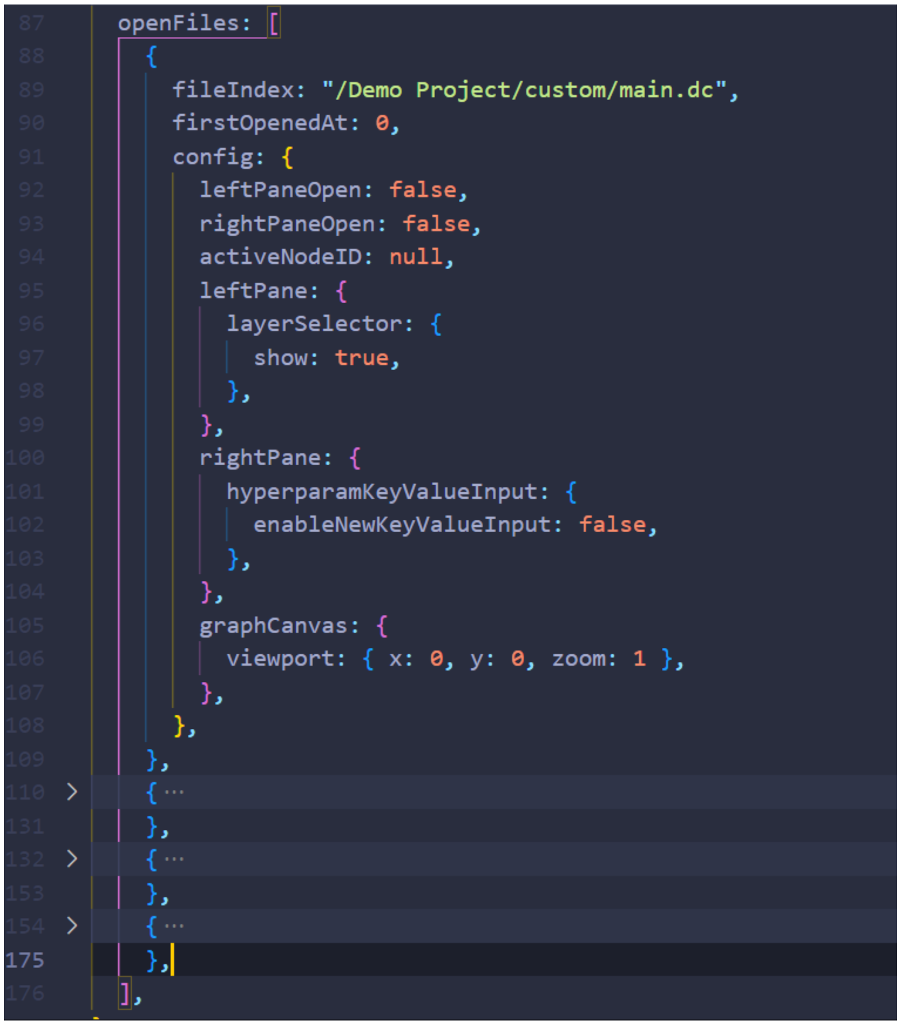


```
fsState: {
  "/Demo Project": {
    index: "/Demo Project",
    data: { name: "Demo Project", folder: true, artefact: false },
    isFolder: true,
    children: [
      "/Demo Project/gan",
      "/Demo Project/tuner",
      "/Demo Project/custom",
    ],
  },
  "/Demo Project/gan": {⋯
  },
  "/Demo Project/tuner": {⋯
  },
  "/Demo Project/custom": {⋯
  },
  "/Demo Project/custom/main.dc": {⋯
  },
},
```

Figure 6.10 - The way the state of the files in the project folder is saved.

We store each folder’s and each file’s information. The contents of some files are loaded into the system. Each folder stores the names of files and folders within it.

To track the currently opened files, we use the following array structure:



```
openFiles: [
  {
    fileIndex: "/Demo Project/custom/main.dc",
    firstOpenedAt: 0,
    config: {
      leftPaneOpen: false,
      rightPaneOpen: false,
      activeNodeID: null,
      leftPane: {
        layerSelector: {
          show: true,
        },
      },
      rightPane: {
        hyperparamKeyValueInput: {
          enableNewKeyValueInput: false,
        },
      },
      graphCanvas: {
        viewport: { x: 0, y: 0, zoom: 1 },
      },
    },
  },
  {…
  },
  {…
  },
  {…
  },
],
```

Figure 6.11 - The internal array to store currently opened files' information.

This array stores the names of the files user is currently working on and are open in the workspace. With each file, the system also stores the state of the graph editor for that file (see the `config` key). The first file object in this array is the file user currently is viewing.

Chapter 7: Compiler Working

We have created our own compiler for our system which will traverse all the architecture created by the user and generate its corresponding Python code. The code can then be used by the user on the required dataset. This will make it convenient for the user to focus on the concepts of Deep Neural Networks rather than the coding part. The compiler completes all the steps as done by general ones like lexical analysis, sematic analysis, syntactic analysis. The details steps/function of the compiler is as follows:

1) When the user asks the system to generate code, firstly the runtime environment (frontend) will send the set of the files to the compiler. The compiler's execution begins from here where there will be know set of files and the initial file to begin execution with. These files will be containing the information in textual format. It will be containing information about the artifacts, nodes, and edges. The compiler will individually load all the files available at the initial step. All these files loaded in in-memory structures can be accessed easily.

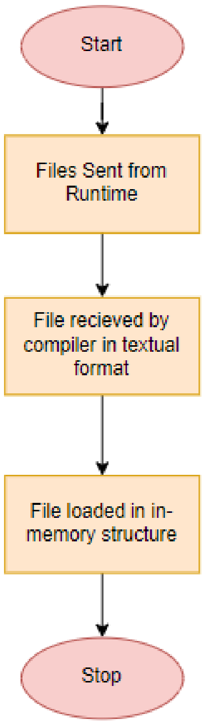

Start
Files Sent from Runtime
File recieved by compiler in textual format
File loaded in in-memory structure
Stop

Figure 7.1 - Flow Chart Depicting file loading in in-memory structure.

2) Now, to convert textual format to objects, compiler will parse each textual file from in-memory structures. These files will first be parsed, to create an object and then the contents in the object will be scanned. This is the lexical analysis part of the compiler where the objects are broken into smaller pieces. File scanning the contents if there is any file which is required, if the file is present in in-memory structure the above process will be repeated. This will be done for all file. If file is not present in in-memory structure then Step 1) will be repeated. After contents of all the required files are scanned, the contents of the files will be converted in key-value pairs in Map, which will make it easier to access the necessary information.

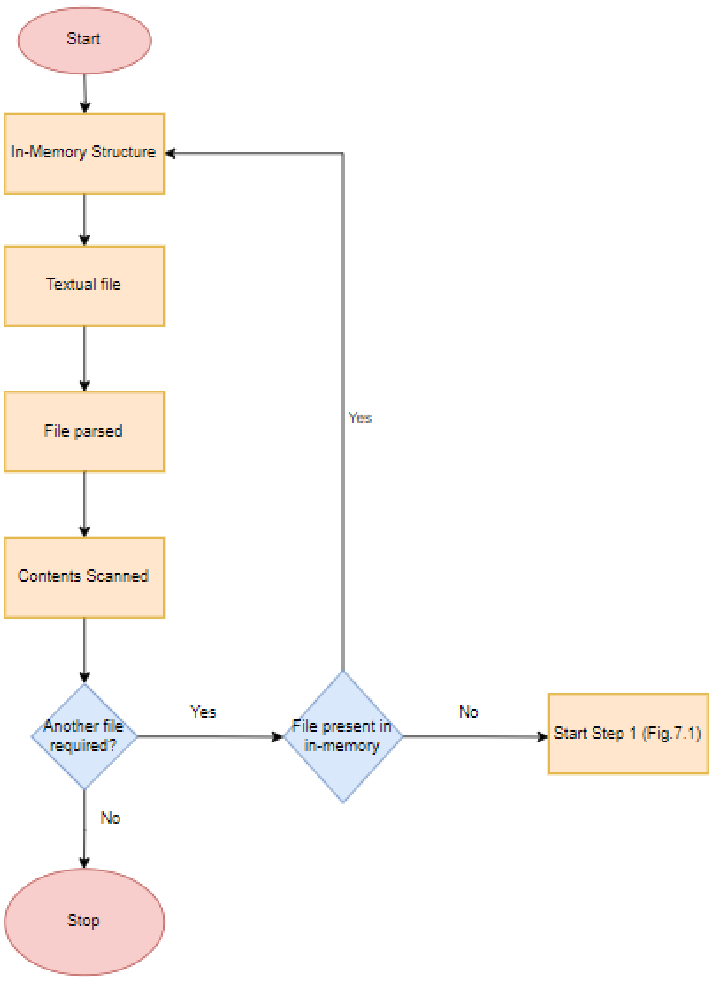


Figure 7.2 - Flow Chart Depicting the process in Step 2.

3) Now the compiler has to traverse all the artefacts. While traversing an artefact, it is possible that another artefact is required to be traversed before the current one. To solve this, the compiler first needs to find the correct order of traversing the artefact, such that the independent ones are traversed first and later on the dependent ones. The algorithm for the scheduling of the artefacts is:
   - i) We start by traversing all the artefact and finding on which other artefacts they are dependent on. A DAG of the artefacts is created. It will be an acyclic graph with information about the neighbors of an artefact.
   - ii) Now the compiler will apply topological sort on the graph created to find the ordering of artefacts. Depth-First-Search topological sort will be used.
   - iii) The compiler will be maintaining a visited array and stack. We will be traversing all the non-visited nodes and for each node, if a node has its neighbors and it is not visited, then recursive function will work on the neighbor. If a particular node has no neighbors or all neighbors have been visited, then the node is pushed into the stack.
   - iv) Finally, the elements of the stack are popped and added to the array which gives us the scheduling of the artefacts. This ensures that there will be no conflict while traversing any artefact.

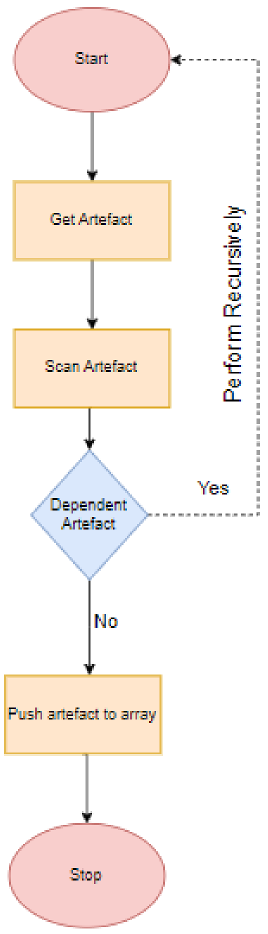


Figure 7.3 - Flow Chart depicting artefact scheduling algorithm.

4) The compiler will now be traversing the artefacts according to the schedule return in previous step (3). The steps in traversing each artefact are as follows:
   i) Depending on the type of the artefact, corresponding function will be called to generate Python function code for the artefacts including the parameters required. Then compiler will start by traversing the node which has zero indegree and is not a data variable node.
   ii) The compiler will first check the node type, then call the corresponding function which would return the Python code for this node. The Python code will then be appended to the string. After this, we move to the next node.
   iii) As we reach the next node, the first condition to check is the number of inputs required to execute the node. If the condition is meet Step ii will be executed, else the function will be returned and other nodes will be traversed. Basically, a recursive function will be called for each node.

iv) After all the nodes are traverse by repeating step ii and iii recursively, we get the Python code for the artefact.

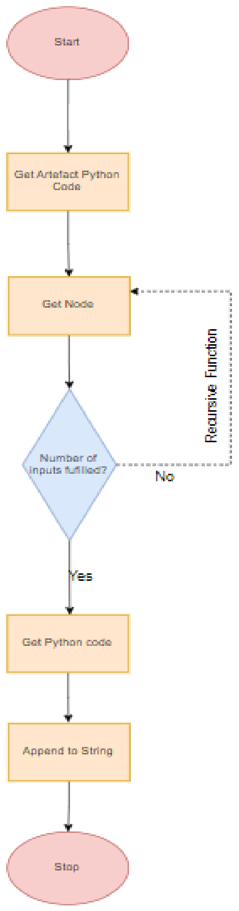


Figure 7.4 - Flow Chart depicting algorithm for traversing a single artefact.

After generating the code of all artefacts, we generate code of all the required imports. The artefact which was the initial artefact, its function must be called on the top level at the end of the file.

Chapter 8: Results and Discussion

To evaluate and test the effectiveness of our system we conducted a survey among college students. The aim of the survey was to know if students were able to conveniently use our system and were able to associate with the code generated by the compiler. The workflow of the survey was:

1. The very first step was to coordinate with students, so that we get a large group to conduct the survey. This process included talking to the professors, communicating with students, and taking the necessary permissions. We recruited 74 computer science undergraduate students (3rd and 4th years) who had some knowledge of neural networks.
2. During the survey, our first step was to explain our system to the students. We first explained our problem statement of our project and the end goal we want to achieve by conducting the survey. We gave a detailed explanation to the students about our system by explaining the different features in our system, how to add specific node, how to add edge between the nodes, how to add hyperparameters for a node, drag-drop feature. This was done to give a brief understanding about our system to the students.
3. After this, the students were given time to explore our system on their own. Simultaneously, they were given two tasks on Kaggle where they were asked to create the architecture on our system. The first task was a digit recognizer task where the students were required to create an architecture such that the neural network can recognize the digits. This was a normal task with digit recognizer dataset. The second task required the use of Siamese network, where the model was required to predict/tell the similarity between two images. For Siamese type networks, we created a concept of Weight Sharing called reuse node, which made it easier to draw the architecture.
4. The students then started to create the neural network architecture on our system. While most of the students were able to make the architecture conveniently, some students didn’t understand the task. So, we gave them an in-depth explanation of the task and also gave them a possible architecture they can create.
5. Then the students started to run their notebooks to check if the code generated by our system for their architecture works or not. The students also tested the model on test data.

6. Finally, a form was circulated to the students to get their feedback about the system, concepts, understandability and how much they were able to associate with the code generated. After this our survey ends.

In the survey, we majorly focused on the following 4 dimensions:

1. Ease or difficulty in understanding the concept of a reuse node. [Q1]
2. Acceptability of this visual language in education. [Q2, Q3]
3. Understandability of the generated neural network code. [Q4]
4. Effectiveness of interaction with the user interface. [Q5]

The results for the above section, along with the survey questions are:

Q1. To what extent did you understand the concept of a reuse block?

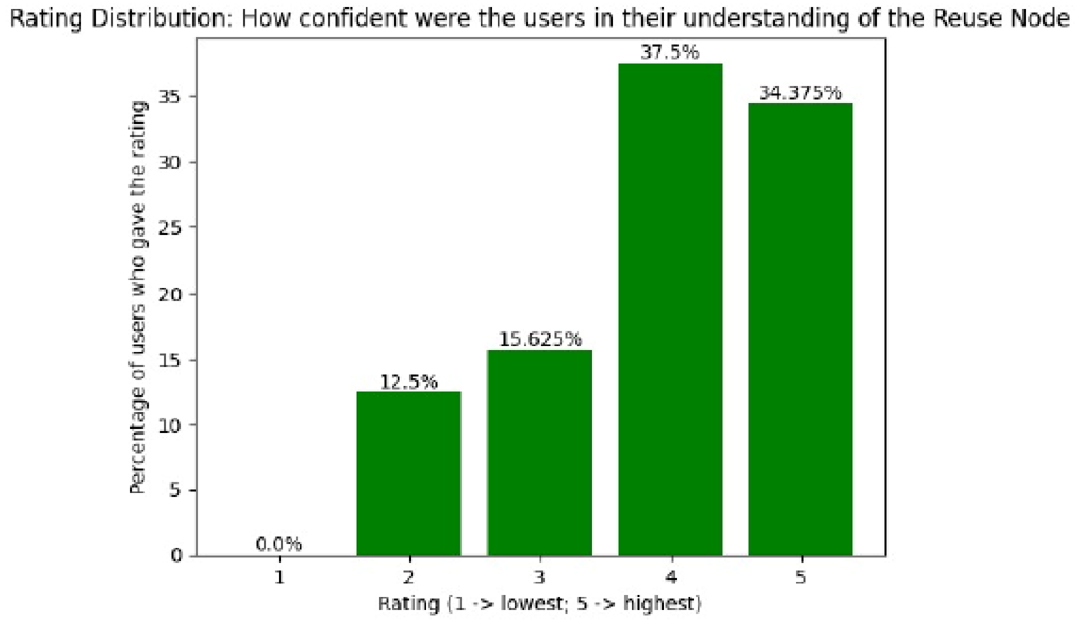


Figure 8.1 - A histogram displaying the frequency of user ratings on a 5 – point scale based on their self – assessment of their understanding of the reuse node.

We asked the users to rate on a scale of 1 (lowest) to 5 (highest) how confident they felt in their understanding of the reuse node. The graph in Figure 8.1 displays the results we obtained. The highest ratings given are 4 and 5, which shows that the users found it easy to understand the reuse node's operation.

Q2. What medium do you prefer for creating neural network architectures for practical DL tasks?

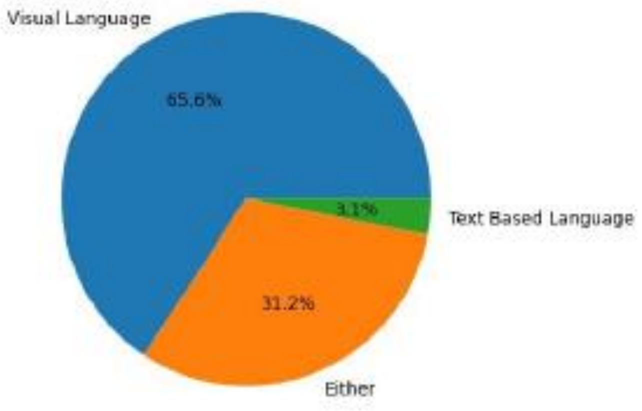


Figure 8.2 - A pie chart depicting the different mediums users prefer to code neural network architectures.

Q3. Would you prefer to use this language in a course about Deep Learning and Neural Networks?

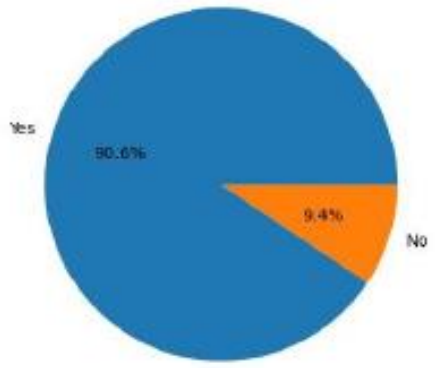


Figure 8.3 - A pie chart showing the distribution of students who prefer to use a visual language for a course on deep neural networks.

To assess the acceptability and applicability of the visual language for educational purposes, we asked the users about their preferences, once they finished the survey tasks. We have summarized the results in the figures Figure 8.2 and Figure 8.3. 65.6% of the users said they preferred the visual language to text based programming languages when creating neural network architectures. At the same time, 90.6% of the users said that they would prefer the visual language be used in a course teaching about deep neural networks.

To further probe into the reasons for the users' choices, we asked for their comments. An analysis of the comments showed that the major reasons were:

1. A visually drawn network is simpler to view and understand – the visual representation is closer to a graph-based representation that users utilize.
2. Users felt that the visual method will save time when building architectures.
3. Users were able to view the components they would insert into the canvas, which provided with recognition cues.

Q4. To what extent were you able relate and associate generated code with the visually drawn graph?

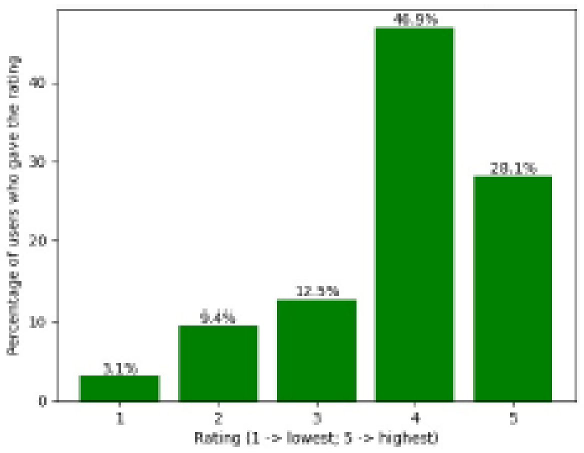


Figure 8.4 - A bar graph depicting the extent to which users could associate the code generated by our system with the neural network graph made by the user.

We asked the user to rate the extent to which they understood the code generated by our system with respect to the neural network graphically created by them. The graph in Figure 8.4 shows the results obtained. 75% of the users have given 4- and 5-star rating which shows users were satisfied and easily able to relate with the generated code.

According to some users' comments, they found it easier to make graphs of neural networks and have the code automatically generated, rather than writing it.

Q5. How was your experience in creating neural network graphs? I.e., how favourably do you view the system interface?

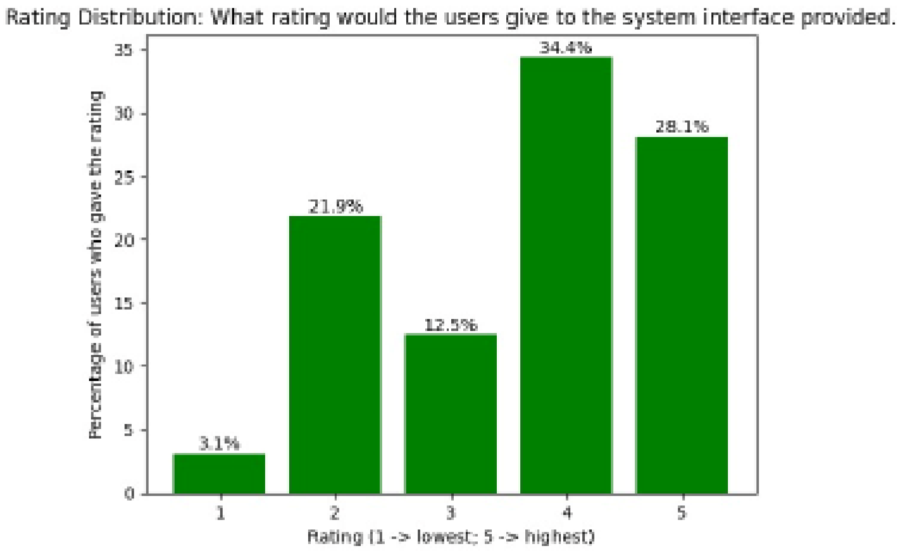


Figure 8.5: A bar graph depicting how favourably users view our system interface.

We asked our users to rate how favorably they viewed the interface. We present the results in the graph in Figure 8.5. 62% of the users gave a rating of 4 or above, however, a considerable number of users gave ratings below 4 as well. This suggests that there is still room for improvement in the design of a suitable interface for this visual language. We asked for the users' comments to infer the reasons for their ratings. Based on the results, we were able to find the following issues with the interface design:

1. Using a touchpad instead of a mouse (for instance, when using a laptop) can make it difficult to move around the canvas and place nodes.
2. Understanding and navigating the interface was challenging for some users. Notably, though, some users mentioned that once they did grasp the layout of the system, the controls were easy to access and use.
3. The procedure of inserting a reuse node into the graph was not convenient when multiple such nodes were needed.

Discussion

In this report, we described a graphical language capable of describing architectures of various neural networks and their associated workflows. This language has been designed to be similar to visual dataflow languages.

One critical and experimental aspect of the language is if we allow function nodes themselves to construct other functions. This pattern of a function constructing another function was not directly mappable to the visual interface. To test this aspect (which has not yet been included in DNNLang), we allowed functions to create other functions, which then "replace" the previous ones. We also created a reuse node to test this unconventional aspect. Based on the results above, we found that the reuse node did not prove very difficult to understand by new users.

We developed an interface that allows users to use the visual language. We presented the interface to multiple users to analyze its suitability, as described in section 4 (Results). Based on the results, we noticed that the users were inclined towards having this system integrated in educational courses about deep neural networks. We noted that the code generated by the system was easy to relate to the graph drawn. As regards the system interface, we note that the users found some issues related to ergonomics of design, such as difficulty in navigating the canvas using the touchpad. *This demonstrates that the frontend design of a programming environment for this language still requires further analysis.*

We found that users preferred to use laptops or desktops to use this system. We further found 4 major methods of interactions users preferred when using this system:

1. Using a mouse for pointing.
2. Using a touchpad for pointing.
3. Using a touchscreen (and optionally a stylus) for pointing.
4. Using a keyboard for entering information.

A suitable programming environment for this visual language would require addressing all these modalities.

As compared to the other available tools, our system seeks to extend the functionalities, by adding more language constructs. We described a set of constructs in the form of nodes and edges, that can be used as a formal basis for creating neural network architectures, and systems that facilitate the same. This system can therefore be used to develop much more complex architectures and workflows visually than what was possible before. The earlier possibilities are outlined in Chapter 2, but we present a brief comparison below:

1. Our system allows creation of Siamese architectures, where same layer's weights are reused. Previous systems do not allow this.
2. Our system allows model training with custom workflows, allowing architectures such as GANs.
3. Our system provides a very complex model tuning framework, where the model tuning can not only be used to tune layer hyperparameters, but the architecture of the model itself.

Chapter 9: Learnings

Technical Learnings

1. **SDLC Related:**
    1. **Clear Requirements Gathering**: In Our Software Development Life Cycle, we always focused on gathering clear requirements and finding effective ways to solve a problem. Throughout the process of development, we understood the importance of clear requirements and how we should be specific in requirement specification so that we can plan our goals and their deadlines. Unnecessary requirements delay our development process and hinder us to achieve the set target. Also, when multiple people work on the same project its crucial to maintain consistency between each individual's understanding of requirements. We also created multiple mind-maps to specify and document the requirements clearly all at one place. The HACUFS method we outlined before did not exist in literature previously. Coming up with it, and using it for requirements elicitation was an unprecedented learning experience.
    2. **Iterative Development:** We witnessed the power of iterative development; we first came up with a prototype or initial version of our system and went forward with feedbacks and it helped us see significant aspects which needs to be incorporated.
    3. **Flexibility and Adaptability:** SDLC taught us to incorporate the changes and be adaptive and flexible to changing requirements. Being open to adjustments and course corrections is essential for project success in dynamic environments.
    4. **Continuous Improvement:** SDLC taught us to continuously seek improvements by taking feedback.
    5. **Understanding version control system and its importance:** Since our development involved contribution from 3 people, we need to have a version control system and We leveraged Git to fortify our development process. We created a master branch to have the main code. The master branch in Git typically serves as the primary, stable branch of a project. It represents the main line of development and usually contains the most up-to-date, production-ready version of the code. Developers often use the master branch as a reference point for releases and deployments.

       On the other hand, feature branches are created to implement specific features or work on distinct tasks within the project. Each feature branch is typically branched off from the master branch and is dedicated to a particular feature or issue. We also created release tags for our two prototypes which we developed at an early stage of development. This whole version control helps to do parallel development, easy merging of code and to help us keep track of the development.

2. **React Related:** For development we used React as front-end and plain vanilla JavaScript for backend (compiler creation). React is one of the most popular libraries for front-end development and it became our choice as we have a vast set of libraries for React based development that eases out our work. We used react-flow which helped us create the graphical components in our system, it helps us manage our code and provides lot of abstracted functionality which we would have to write otherwise. We also used React state management tool: Redux, which helped us store multiple global states which in turn helped us create multiple features that enhance the user experience. In React, we learned out Component based development which helps us write code in the form of components that makes it all clean code and easy to navigate. It has large community support which helped us pass through many errors that we get while development.
3. **Portability Related:** In order to make the application portable we utilized a library called ElectronJS which helped us create a desktop application and a simple exe file that can be transported to other systems easily. Electron.js is an open-source framework that allows developers to build cross-platform desktop applications using web technologies such as HTML, CSS, and JavaScript. It combines the Chromium rendering engine and the Node.js runtime, enabling developers to create native-like desktop applications that can run on Windows, macOS, and Linux operating systems. Internally, ElectronJS uses a multi-process architecture to manage the application's functionality and user interface. This architecture consists of two main processes: the main process and renderer processes. The main process is responsible for managing the application's lifecycle, handling system-level events, and coordinating communication between different parts of the application. Renderer processes are responsible for rendering and displaying the application's user interface using web technologies.

Non – Technical Learnings

On the non – technical side, we were introduced to the various aspects of conducting a survey beyond just the preparation of tasks and questionnaires.

1. We understood the need of creating questions that the users would want to answer.
2. We also experienced first-hand how to explain concepts to a moderately sized group of students. We learned that it is not just about imparting knowledge, but also about presenting it in a way students would want to listen.
3. Finally, we were introduced to the academic research process. We understood how to structure and write research papers, the difference between conferences and journals, and how to submit research papers to journals.

Chapter 10: Future Work

The future work that we have planned is as follows:

1. **Comprehensive Dataset Management Library:** DNNCASE provides an easy method to create neural network architectures, training workflows and tuning workflows. However, the current data manipulation methods are not developed to their full potential. Our future work in this matter will be to analyze, tabulate and create a model for data loading. This model must incorporate multiple sources such as CSVs, databases, etc. Based on this model, we plan to create a library of nodes that can allow easy and visual methods of loading data into the system.
2. **Insights and Tips Assistant:** DNNCASE provides us an interface to create and execute deep neural networks but expects a certain level of knowledge to operate, but with a Virtual Assistant it can help its user to learn different aspects of it based on user's understanding and context. It will provide user information tips, suggested network architectures from user's drawings so far. It will provide a guided path to various common architectures to help users learn hands-on. Also, the assistant will display performance metrics, such as accuracy, loss curves, and confusion matrices, to help users evaluate model performance.
3. **Deeper research on HCI aspects of a graph editor UI:** In the course of our survey, we found that our UI for the graph editor, while easy to use with a mouse, presents difficulties with other kinds of pointing devices. Further research into user interaction with the UI of the graph editor can prove fruitful for the user experience.

Chapter 11: Conclusion

Our project describes a way in which a graphical language can be created to describe a neural network architecture and workflows visually. We have described an algorithm for converting the graphical specifications into Python code. Our current language can support the development use cases of most neural network workflows.

In this project report, we also presented the opinions of various users that interacted with the system. We demonstrated the acceptability of this language in learning and teaching deep neural networks, and the ease associated with using a visual interface to specify neural network architectures. We also described some issues associated with a programming interface created for this language. Development of a suitable interface for such a visual language remains an essential step towards using this language in an educational capacity.

The ease of use associated with a visual interface offers a more intuitive approach to create neural network architectures, which lowers the barrier to understanding deep neural networks for beginners and enhancing productivity for experienced practitioners.

References

[1] Cisco. (2023, Oct. 06). *Cisco Annual Internet Report (2018–2023) White Paper* [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html

[2] Indeed Editorial Team. (2023, Oct. 07). *What Is Data in Business? (Plus Importance and Examples)* [Online]. Available: https://www.indeed.com/career-advice/career-development/data-in-business

[3] Louie Andre (FinancesOnline). (2023, Oct. 07). *53 Important Statistics About How Much Data Is Created Every Day* [Online]. Available: https://financesonline.com/how-much-data-is-created-every-day/

[4] Keras. (2023, Oct. 08). *Keras* [Online]. Available: https://keras.io/

[5] K. M. Lee, K. S. Hwang, K. Kim, S. H. Lee, K. S. Park. "A Deep Learning Model Generation Method for Code Reuse and Automatic Machine Learning," in Proceedings of International Conference on Research in Adaptive and Convergent Systems, Honolulu, HI, USA, 2018.

[6] K. M. Lee, K. Kim, J. Yoo. "Autonomicity Levels and Requirements for Automated Machine Learning," in Proceedings of International Conference on Research in Adaptive and Convergent Systems, Honolulu, Krakow, Poland, 2017.

[7] S. Klemm, A. Scherzinger, D. Drees, X. Jiang. "Barista – a Graphical Tool for Designing and Training Deep Neural Networks," arXiv preprint arXiv:1802.04626.

[8] K.M. Lee, K.S. Park, K.S. Hwang, K.I. Kim. "Deep neural network model construction with interactive code reuse and automatic code transformation," *Concurrency and Computation: Practice and Experience,* e5480, 2019.

[9] J. Bromley, I. Guyon, Y. LeCun, E. Sackinger, R. Shah. "Signature verification using a siamese time delay neural network," *Advances in Neural Information Processing Systems*, 1993.

[10] Soumik Rakshit. (2023, Oct. 06). *Low-light image enhancement using MIRNet* [Online]. Available: https://keras.io/examples/vision/mirnet/

[11] DeepLearningAI (2023, Oct. 07). *Build First System Quickly, Then Iterate (C3W2L03)* [Online Video File]. Available: https://www.youtube.com/watch?v=HfM8UIohGE0&list=PLkDaE6sCZn6E7jZ9sN_xHwSHOdjUxUW_b&index=15

[12] Wesley M. Johnston, J. R. Paul Hanna, Richard J. Millar. "Advances in Dataflow Programming Languages," *ACM Computing Surveys, Vol. 36, No. 1, March 2004, pp. 1–34.*

[13] Morrison, J.P. (2020) Flow-based Programming, Flow-based programming. Available at: https://jpaulm.github.io/fbp/ (Accessed: 01 May 2024).

Appendix A: A Technical Description of DNNCASE Visual Language

This appendix presents a brief description of the language used in DNNCASE. We refer to the language as "**DNNLang**" in the following text. The code of this language is drawn on a **visual canvas**. We refer to this process as "writing" the code. The following are the salient points of the language:

1. DNNLang is a visual language that can describe program flows visually. The language focusses on ***data flow* instead of control flow**. This means that the language describes data dependencies among several components, called as nodes.
2. To compile DNNLang, a **compiler** is constructed. This compiler translates the language files into equivalent code of some **target language**. For the purposes of this project, the target language is python.
3. The basic element of execution in DNNLang is an **artefact**. This is similar to functions in other languages, such as C. An artefact is a directed acyclic graph with a finite number of inputs and outputs. The execution begins from a user-specified "main artefact". This is analogous to the *int main()* function in C programs.
4. The **edges** in the graph of an artefact primarily declare data flows, i.e., data dependencies.
5. The **nodes** in the graph of an artefact represent data sources, data sinks, or data transformations.
6. There are certain **special nodes**. These are: Input Node, Output Node, For Loop Node, While Loop Node, Repeat-For Loop Node, Data Variable In/Out Nodes, Boolean Branch Node, Branch End Node, Artefact Specifier Node, Named/Ordered Packer Nodes and Named/Ordered Unpacker Nodes.
7. The most commonly used node is a **function node**. When specified in a graph, this node can be a user-defined function of the target language, or it can be some artefact.
8. Codebases are structured as follows:
    a. Code is organized into *modules*. A module is a folder that contains a config file and several code files.
    b. A module may contain other modules, called sub-modules.

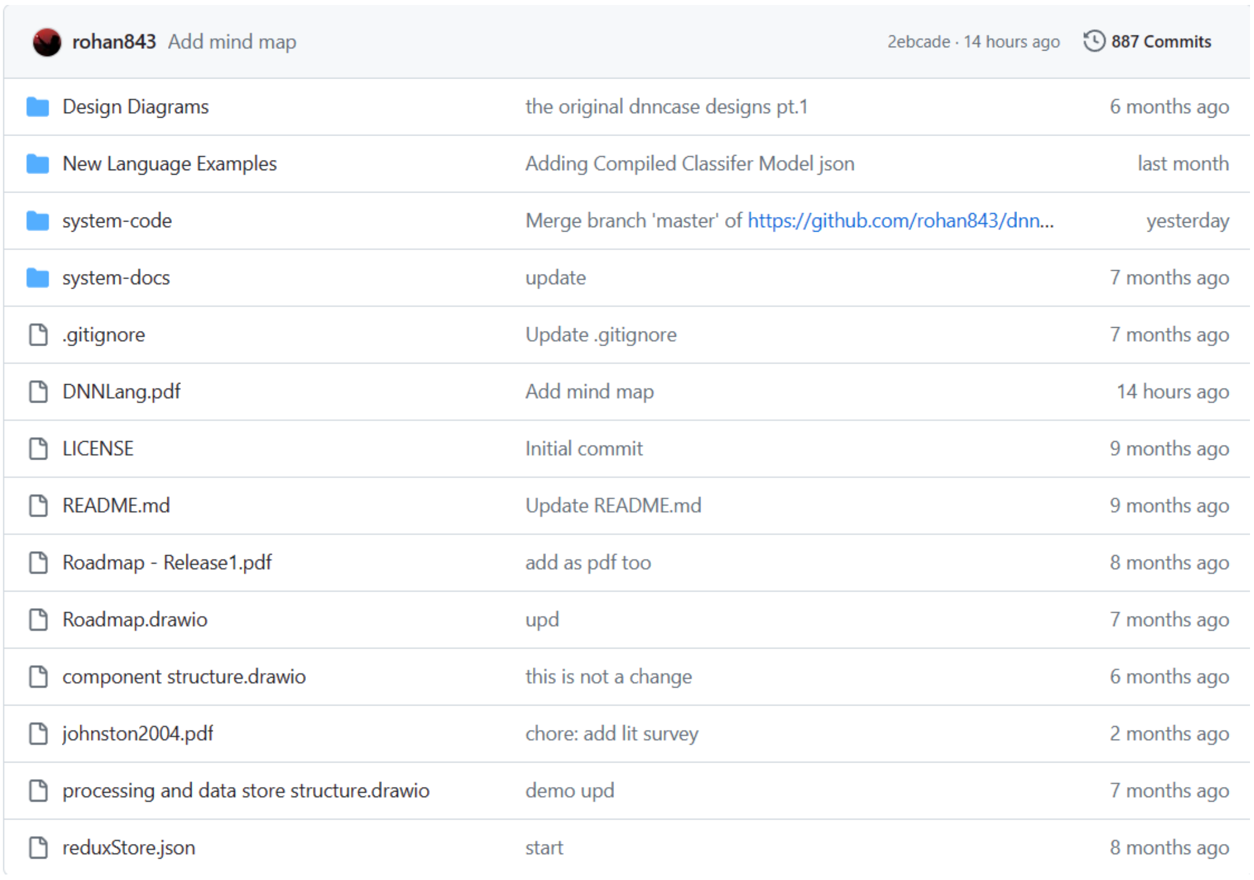
9. At times, it may be necessary to *specify* an artefact that needs to be used later, analogous to function pointers in C. This is done by specifying the relative path of the artefact within its module, and the artefact's module to an Artefact Specifier Node.
10. The language consists of **Pseudo-Nodes** as well. A pseudo-node is used to specify information regarding any code file to the programmer, much like comments in other languages. Code files in DNNLang are displayed on a visual canvas. A pseudo-node can be placed either at any position on the visual canvas using absolute coordinates, or it can be placed on the canvas *relative* to some other node or edge.
11. Some nodes may contain nested graphs. This occurs in the case of loops, where a loop node has the loop "body" specified within it as a graph.
12. The language has various **data types**:
    a. Primitive Datatypes: These are the most basic datatypes of the language. DNNLang has 4 primitives: Integer, String, Float, Boolean.
    b. Box Datatype: This is a special datatype. Data of this type is used to tell the compiler to treat it as textual code of the target language. This is useful in cases where say, an Integer in DNNLang may refer to a 32-bit int in the target language, but a 64-bit int is needed.
    c. Container Datatype: This is an aggregate datatype. It consists of 2 types: Named Container, where key-value data is stored, and Ordered Container, where arrays are stored.

Appendix B: Accessing the project's GitHub repository

This appendix describes the structure of our GitHub repository and instructs on how to browse, clone, and run the code.

Accessing the Repository

The repository can be accessed at: https://github.com/rohan843/dnncase. The following image shows the structure of the repository:



rohan843 Add mind map 2ebcade · 14 hours ago 887 Commits

Design Diagrams	the original dnncase designs pt.1	6 months ago
New Language Examples	Adding Compiled Classifer Model json	last month
system-code	Merge branch 'master' of https://github.com/rohan843/dnn...	yesterday
system-docs	update	7 months ago
.gitignore	Update .gitignore	7 months ago
DNNLang.pdf	Add mind map	14 hours ago
LICENSE	Initial commit	9 months ago
README.md	Update README.md	9 months ago
Roadmap - Release1.pdf	add as pdf too	8 months ago
Roadmap.drawio	upd	7 months ago
component structure.drawio	this is not a change	6 months ago
johnston2004.pdf	chore: add lit survey	2 months ago
processing and data store structure.drawio	demo upd	7 months ago
reduxStore.json	start	8 months ago

Figure B.1 - The structure of the DNNCASE repository.

Here, the folder titled system-code contains the source code of the system.

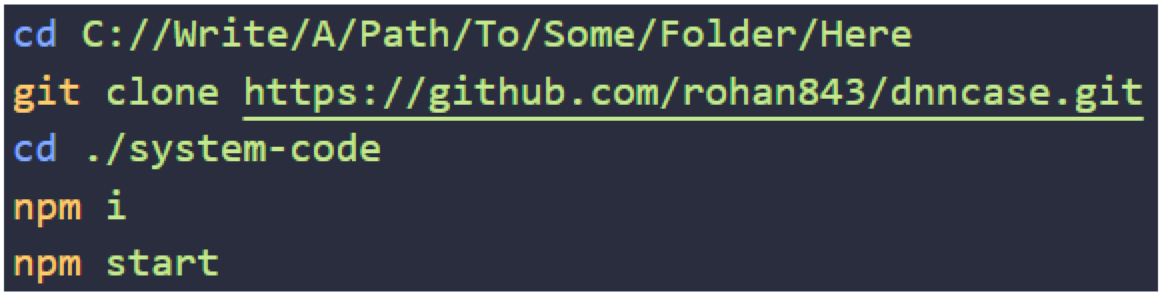
Various design diagrams are present within the Design Diagrams, New Language Examples, and system-docs folders.

Cloning the Repository and Executing the Code

The code can be executed by following these steps:

1. Install NodeJS from its official website here: https://nodejs.org/en/download.

2. Follow all instructions to ensure the `node` command works from the command line.
3. Next, install git from its official website here: https://git-scm.com/downloads.
4. Now, run the following commands on the command line:



```
cd C://Write/A/Path/To/Some/Folder/Here
git clone https://github.com/rohan843/dnncase.git
cd ./system-code
npm i
npm start
```

Figure B.2 - Commands to execute the code for DNNCASE.

Now, DNNCASE should be up and running!

Appendix C: HACUFS – Hierarchical Agglomerative Clustering of Unbounded Feature Space for requirements elicitation

We followed a requirements elicitation process which we call HACUFS. This process allowed us to define a scope and requirements for our project. This process was briefly discussed in the text in Chapter 4. Here, we elaborate on this process in its own right, so that it may be referred to by other software practitioners.

When to carry out HACUFS?

Consider you have a project that you have to deliver in a limited time, and with limited resources. Now, add to the already complex circumstance that the project by its nature has a large set of aspects and that the users may expect a diverse set of features. It can be very difficult in such a situation to decide which features to work on, which to combine, what modules to create, and what priority order to follow.

If your software project seems to fall in this category of projects, performing HACUFS may provide a solution.

What to expect as end deliverables once I've carried out HACUFS?

Now that you are considering using HACUFS for your requirements elicitation process, you can expect the following items to be available to you after the process is finished:

1. A set of features that are relevant to your project. These features may not all be possible to implement in your limited time availability. However, those that go un-implemented, serve as a good source of future directions of your project.
2. A hierarchical tree-like diagram (a mind map, for example) consisting of various internal nodes that can be mapped to independent modules that will constitute your system. Subsets of the features will be present at the leaf nodes of this tree structure.
3. A priority order that tells what modules are essential to the system, what are necessary but not essential and what are nice to have.

How do I carry out the HACUFS process?

Once you have decided to conduct HACUFS, you should follow these steps:

1. Assemble your team and **brief everyone about the project**, and any technical details related to it. (For example, if your project is about creation of a word processor, background knowledge about character sets, layout strategies etc. will be an advantage to know beforehand). Ensure all team members are on the same page as regards the objective (potentially informal at this point) and that they know *why* the project is being worked on.
2. Now, conduct a **brainstorming session**. In this session, there must be a conductor and a scribe. (One person may assume both the roles as well.) The conductor gets everyone assembled in the brainstorming room and goes around the group in a *circular* fashion. At a person's turn, they should give a set of ideas for the features or requirements they feel are relevant to the project. At this point in HACUFS, all ideas for features must be taken in and written down by the scribe. Preferably, all ideas written down so far must be available to be read by all participants. The brainstorming session goes on until the group is out of ideas.
3. After the session, the conductor (and possibly some other teammates) must number all features and write them down in a list document. The numbering here is arbitrary, and is only intended to give each feature a reference ID. At this point, some features may come out to be *too* similar in their wordings or meanings to each other. They may be combined. This list document is now our **set of unbounded features**. It contains all those features the teammates could think of, without any resource or other restrictions on the scope of the project.
4. Now, the team should perform an **agglomerative clustering** of the features. Do this process in the following manner:
    a. Read all the features, while writing those features that seem similar to each other together on some whiteboard.
    b. For those features that seem similar, write them closer to each other, assuming the whiteboard is a 2d coordinate system, and the features are points in this system.
    c. Now, we have various features arranged visually. Those features that seem to either require similar operations, or those that seek to achieve similar goals should be grouped together. (E.g., in a word processor, underlining text and italicizing text are very similar as both style text.) One feature can only be in a single group.

At this point, the whiteboard should have some groups of features, and other individual features. Treat the individual features as groups of size 1.

d. Some of the feature groups at this point must be assigned a module-based abstraction. Analyze any feature group, and try to decide if a single module should be responsible for these features. Appropriately name the module responsible for those features, and write down the module name on the whiteboard near the feature group it is responsible for, and make a directed edge from the module to its feature group.

e. As this process continues, it may happen that one module can serve as a submodule to another module, or that multiple modules may be clubbed together under a larger module. It is also possible that the team decides to assign more feature groups to some existing module. Keep updating the modules on the feature space, by making a directed edge from a larger module to its sub – modules, and to any feature groups that the module is responsible for.

f. As your team nears the end of this process, there will be several tree-like structures on the whiteboard. It is possible that some feature groups are not yet placed under any module. For these feature groups, perform a re – analysis of their relevance to the project, and the correctness of grouping them together. If any irrelevant feature can be identified at this stage, remove it from the feature space. If any incorrect grouping is found, the group may be broken down into smaller groups. These smaller groups may then also be merged into other feature groups or assigned to other modules based on relevance.

g. The process is said to end when no feature group is left unassigned, and all modules that could be sub – grouped under larger modules have been grouped likewise. Now, the whiteboard will contain several trees of hierarchically arranged modules and submodules. Create a new node, that is to be named as the “Project Node” (e.g., in case of a word processor, it may be called “Word Processor”) and make directed edges from it to all the root nodes of the module trees.

5. In following this process, your team now has various possible hierarchically arranged modules that your project needs to fulfill all the features of the feature set. We call this

structure a **Mind Map**. After this stage, we perform **assignment of priority** to the modules.

6. Your resources are limited, and therefore all the modules, to develop all the features will not be possible to design and code. Therefore, it is required to analyze the mind map.
    a. First, write in a few words what exactly is expected of each module.
    b. Then, to each module, assign one of 3 priority levels: **Core, Needed, Nice to Have**.
    c. The Core level must be assigned to those modules that your system absolutely needs to perform its most basic tasks. (For example, a word processor needs a text display and edit module.)
    d. The Needed level must be assigned to those modules that your system will highly benefit from having, and may increase your returns from your project. (For example, a word processor will highly benefit from a spelling checker module.)
    e. The Nice to Have level must be assigned to those modules that, if implemented, will make it easy to use your system, or will add some lesser relevant features to your system. (For example, an automatic mailing module and a thesaurus module in a word processor are nice to have, but most probably, the users can manage without the features these modules provide.)
    f. It is important to note that a module may have a higher or lower priority level than its sub – module. The hierarchy does not affect the priority level, but it may be beneficial to perform minor restructuring to the hierarchy after priority levels are assigned.
7. Now, you have a set of modules, hierarchically arranged, with 3 priority levels. Structure your work so as to implement the Core modules first, then the Needed modules, followed by the Nice to Have modules. Please note that when a sub – module is Core, but its super – module is of a lesser priority level, it will be necessary to design the sub – module with an interface that otherwise may be passed onto the super – module in later iterations.

You have now carried out the HACUFS process and can proceed to list out the most critical system requirements and design towards them first, based on the highest priority modules.

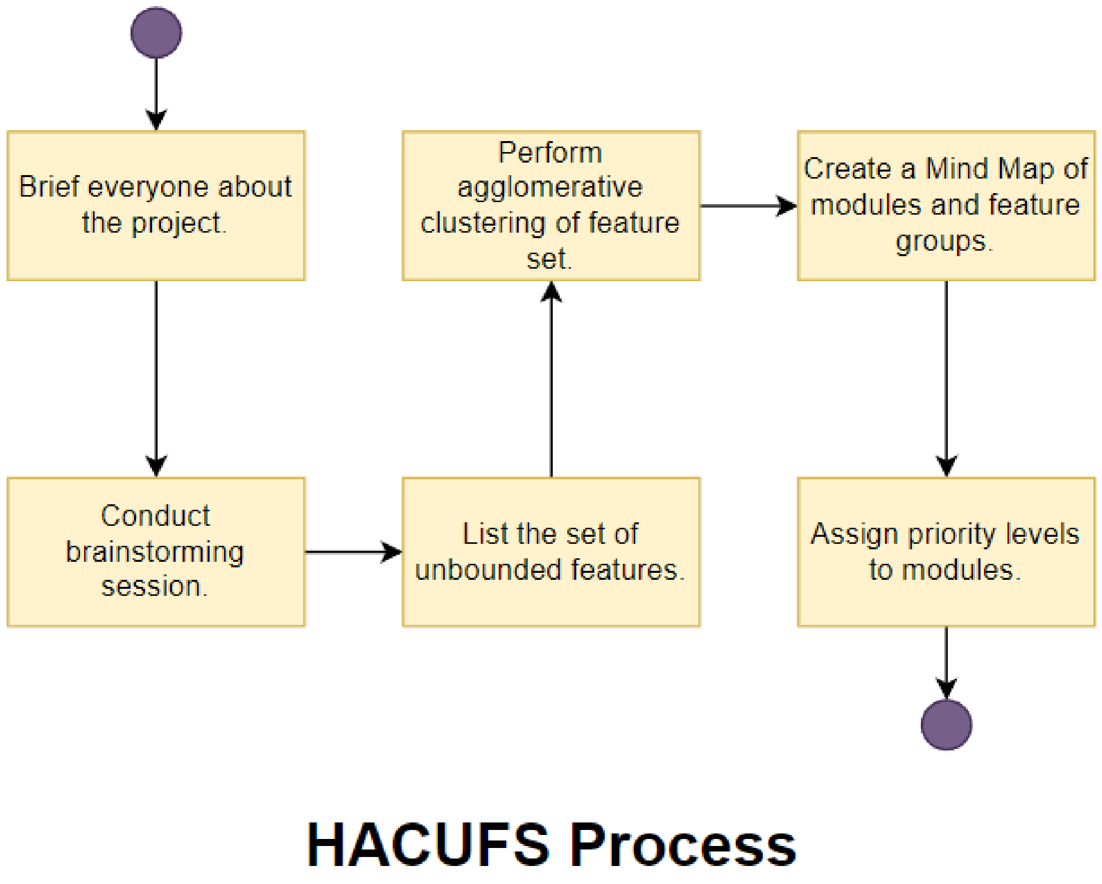

Brief everyone about the project.
Conduct brainstorming session.
List the set of unbounded features.
Perform agglomerative clustering of feature set.
Create a Mind Map of modules and feature groups.
Assign priority levels to modules.
HACUFS Process

Figure C.1 - A flowchart showing the HACUFS process.

Turnitin Originality Report

Processed on: 02-May-2024 19:50 IST
ID: 2368845553
Word Count: 16538
Submitted: 1

BTP Endsem Report (3).docx By Gurkaran Singh

Similarity Index
3%

Similarity by Source

Internet Sources:	2%
Publications:	2%
Student Papers:	0%

1% match (Keon Myung Lee, Kwang Il Kim, Jaesoo Yoo. "Autonomicity Levels and Requirements for Automated Machine Learning", Proceedings of the International Conference on Research in Adaptive and Convergent Systems - RACS '17, 2017)
Keon Myung Lee, Kwang Il Kim, Jaesoo Yoo. "Autonomicity Levels and Requirements for Automated Machine Learning", Proceedings of the International Conference on Research in Adaptive and Convergent Systems - RACS '17, 2017

1% match (Internet from 05-Mar-2021)
https://archive.org/details/peel?and%5B%5D=firstCreator%3AD&sort=creatorSorter

< 1% match (Internet from 09-Mar-2024)
https://open-innovation-projects.org/blog/page/139

< 1% match (Internet from 17-Mar-2024)
https://open-innovation-projects.org/blog/page/112

< 1% match (Keon Myung Lee, Ki-Sun Park, Kyung-Soon Hwang, Kwang-Il Kim. "Deep neural network model construction with interactive code reuse and automatic code transformation", Concurrency and Computation: Practice and Experience, 2019)
Keon Myung Lee, Ki-Sun Park, Kyung-Soon Hwang, Kwang-Il Kim. "Deep neural network model construction with interactive code reuse and automatic code transformation", Concurrency and Computation: Practice and Experience, 2019

< 1% match (Internet from 06-Apr-2010)
http://www.rscr.cz/bulletinES/euraginfoen9102006.pdf

< 1% match (Internet from 02-Jan-2024)
https://www.tutorialspoint.com/building-desktop-gui-applications-with-javascript-and-electron-js

< 1% match (Internet from 26-Feb-2021)
https://www.mdpi.com/2078-2489/11/4/193/htm

< 1% match (Keon Myung Lee, Kyoung Soon Hwang, Kwang Il Kim, Sang Hyun Lee, Ki Sun Park. "A deep learning model generation method for code reuse and automatic machine learning", Proceedings of the 2018 Conference on Research in Adaptive and Convergent Systems - RACS '18, 2018)
Keon Myung Lee, Kyoung Soon Hwang, Kwang Il Kim, Sang Hyun Lee, Ki Sun Park. "A deep learning model generation method for code reuse and automatic machine learning", Proceedings of the 2018 Conference on Research in Adaptive and Convergent Systems - RACS '18, 2018

< 1% match (Blaney, F.E.. "MAMBAs: A real-time graphics environment for QSAR", Journal of Molecular Graphics, 199309)
Blaney, F.E.. "MAMBAs: A real-time graphics environment for QSAR", Journal of Molecular Graphics, 199309

< 1% match (Internet from 26-Feb-2023)
https://brainhub.eu/guides/electron-development

< 1% match (Clive Chatters. "Photographic and artwork credits", Walter de Gruyter GmbH, 2010)

Clive Chatters. "Photographic and artwork credits", Walter de Gruyter GmbH, 2010

< 1% match (Internet from 16-Jan-2023)
https://www.semanticscholar.org/paper/Barista-a-Graphical-Tool-for-Designing-and-Training-Klemm-Scherzinger/856b2931efb541bdc69c8fb940cca98f244a7c25

< 1% match (Internet from 15-Nov-2023)
https://www.youscribe.com/catalogue/documents/savoirs/oral-health-promoting-program-in-the-primary-school-elektronische-1428741

< 1% match (Internet from 21-Jun-2022)
https://github.com/KaanOzgencil/wetlandbirds

< 1% match (Internet from 14-Feb-2015)
http://premierafricanminerals.com/ul/PAM%20Updated%20Resource%20Statement_2014_Final.pdf

< 1% match ("Applied Physics, System Science and Computers II", Springer Science and Business Media LLC, 2019)
"Applied Physics, System Science and Computers II", Springer Science and Business Media LLC, 2019

< 1% match (Internet from 05-Jul-2023)
https://dev.to/gaius_001/to-install-react-and-create-your-first-app-50pp

< 1% match ()
Rimando, Ryan A.. "Development and analysis of security policies in security enhanced Android", Monterey, California. Naval Postgraduate School, 2012

< 1% match (Internet from 17-Oct-2020)
https://silo.pub/c-programming-from-problem-analysis-to-program-design.html

< 1% match (Internet from 10-Oct-2010)
http://www.coursehero.com/file/1603790/FinalThesis/

< 1% match (Internet from 01-Oct-2022)
https://www.ijariit.com/manuscripts/v7i4/V7I4-1315.pdf

Abstract Technology is an essential component of the functioning of our world. It enables us to perform various tasks to enhance our productivity. However, the huge amounts of data generated from the use of technology are considerably complex. To understand the processes that are at work in any system, various innovative modeling techniques are employed. Neural networks are one data modeling method that is not only highly flexible but also enjoys widespread use. To create these networks, various frameworks such as Keras have been developed. While these frameworks are a step towards an easier programming interface, there still remain difficulties in creating and visualizing complex architectures. These difficulties are especially pronounced in the case of developers who may have a weak understanding of deep learning techniques. Many recent advances in deep learning have motivated practitioners to experiment with various network architectures in different domains. We describe a graphical language to depict neural network architectures on a canvas and generate corresponding code. We expand the capabilities of our system, by incorporating other deep learning workflows such as training, tuning, testing, etc. We achieve this by defining a visual language based on the dataflow and functional paradigms, and by creating an associated library for deep learning in that language. KEYWORDS: Deep Neural Network; Code generation; Weight Sharing; Keras; Visual Language; Education; Teaching Aids Page |i Table of Contents S. No. Title Page Number 1 Certificate 2 Candidates' Declaration 3 Certificate of Declaration 4 Acknowledgement 5 Abstract & Keywords 6 Nomenclature and Abbreviations 7 Chapter 1: Introduction 8 Chapter 2: Literature Survey and Problem Statement 9 Chapter 3: Methodology 10 Chapter 4: Implementation 11 Chapter 5: Future Work 12 References 13 Appendix A: A Technical Description of DNNCASE Visual Language i ii iii iv v vii 1 3 10 12 15 16 17 Page |ii Nomenclature and Abbreviation CASE - Computer-Aided Software Engineering DAG – Directed Acyclic Graph DL – Deep Learning DNN – Deep Neural Network FBP – Flow Based Programming FR – Functional Requirements GUI – Graphic User Interface HACUFS1 – Hierarchical Agglomerative Clustering of Unbounded Feature Space for requirements elicitation IDE – Integrated Development Environment IPC – Inter

Process Communication ML – Machine Learning NFR – Non - Functional Requirements 1 See Chapter 4 Chapter 1: Introduction The functioning of the modern world is highly coupled with the use of technology. This has led to a lot of data being generated every day [1]. These data are collected from various processes with unknown mechanisms. These unknown mechanisms are a major concern for the stakeholders of their respective domains [2]. Various modeling techniques can be used to understand the data and make predictions from them. A lot of the data is unstructured [3]. There are various tasks done on such data, using neural networks among other algorithms. Keras [4] is a popular and widely used framework for model construction and training. It is an API for the TensorFlow, JAX, and PyTorch libraries that commonly is used with the Python programming language. It aims to provide a flexible interface for modern ML problems, especially deep learning. Although such tools provide ease of programmability when coding neural network architectures, there still are difficulties in creating and visualizing complex architectures. These difficulties are especially pronounced in the case of developers who may have a weak understanding of deep learning techniques. [5] Recently, there have been many advances in deep learning research, and this has been a motivation for various developers and practitioners to come up and experiment with various network architectures in different domains. In this work, we have created a CASE (computer-aided software engineering) tool that allows users to create complex, modular, and wide-ranging deep neural network architectures. It will provide a canvas for creating the architecture diagrams and specifying schematics of associated workflows (such as training mechanisms). It also provides the python code of the corresponding deep neural network architecture made by the user. We have created a visual programming language which uses the concepts of functional and dataflow programming languages to help the user create architectures for neural network using fundamental entities such as datatypes, function, if-else condition, loops and callbacks. The user can also train, test and perform hyperparameter tuning on their architecture using this language. Page |2 We have created a compiler that can generate python code for the language schematics. The compiler and basic yet scalable design allows our language to be used to design systems other than neural networks. We have also created a library focused on deep learning in this language. Our Motivation We wish to enable ML/DL professionals to focus on algorithmic complexities rather than grappling with coding intricacies, thus accelerating model development. One of our major goals in the development of this system is to allow users to model a neural network in the same way they would draw it on a whiteboard or notebook. By providing a canvas to the users, we can reduce the mismatch between the perceived understanding of a neural network by the user, and its specification in the system. Architectural modifications will become straightforward, enabling agile experimentation and optimization. Different neural network architectures can include repeated sub-units. These present an opportunity for code reuse. The user should be able to specify a 'blueprint' of a single sub-unit and use it in various positions in a neural network diagram. Moreover, currently, to the best of our knowledge, no graphical language exists that supports all possible network architectures and training workflows straightforwardly. An analysis is necessary to understand various constructs that such a language will need for a graphical description, given that those constructs may seem commonplace and simplistic when writing equivalent code. By removing the programming obstacle, we will be able to make deep learning more accessible to school students and people from non-technical backgrounds. They will be able to understand and visualize the concepts of neural networks easily and quickly, without the need to first learn programming. Chapter 2: Literature Survey and Statement of Problem Lee et al. in their paper titled 'A Deep Learning Model Generation Method for Code Reuse and Automatic Machine Learning' [5] outline the need for a tool that can be used to develop and train deep neural networks. They focus on an autonomous approach to training neural networks, including autonomic hyperparameter tuning. Their tool allows users to train a model from within the system, but utilizing remote computing resources. In their paper, they outline various difficulties in using the prevalent ML techniques, which include architecture selection and hyperparameter tuning. Because of this, their system has been centered around an easy-to-use and intuitive interface, with common functionality in-build and the complexities abstracted. Their paper also refers to the various autonomicity levels [6] and associated trade-offs that arise as a result of making systems more and more autonomic. One of the major aspects outlined is that a more autonomic system places a burden on computing resources because of a large search space, while a lesser autonomic system is complex for its users. They describe a methodology for loading a model as a graphical structure, from its Keras source code. This is central to making complex models' source codes easy to understand and manipulate by novice users. This is achieved by converting each 'sentence' of code (i.e., one complete Python instruction) into its associated parse tree, and extracting relevant information such as various attributes. This information is later used to insert a node or an edge in the graphical depiction. We analyzed the system described in this paper, to understand various features and use cases to start gathering our system requirements for DNNCASE. We were able to review various algorithms used in this paper and their proposed architecture. This system incorporates various basic functionalities necessary in any deep learning system's workflow. While it focuses on a rich subset of possible use cases, this system lacks support for more advanced use cases, architectures, and training workflows. Based on the description of the system's logic, we ascertained that it cannot support Siamese network architectures or other architectures involving weight–sharing [9]. Siamese networks are commonly used in one-shot learning. They can compare the similarity between 2 objects (such as face portraits). In Siamese networks, a subgraph of the neural network is used twice – once per input – during the inference time. However, while training such a network, gradients corresponding to both inputs must be backpropagated through the same subgraph and not its copy. Our system incorporates this kind of architecture creation through a layer reuse block. Another facility provided by Keras, i.e., custom layers, though subtle, is not supported in this system. Klemm, Scherzinger, et al. in their paper [7] introduce a similar tool called Barista that is meant to be an easy-to-use high-level system for designing, training, and testing deep neural networks (DNNs). It offers a GUI with a graph-based editor and provides features that allow researchers to focus on their core work, without having to manually edit text files or parse logs. This paper mentions several other tools that were created in the past; however, those tools do not cover all aspects of a conventional deep learning pipeline in a GUI-based manner. One (Caffe GUI tool2) of those tools' codebase as available on GitHub has been archived. The paper expands on the functionalities provided by those tools and seeks to cover the entire end-to-end deep learning pipeline, starting from model creation to its training and evaluation. The tool has been kept open-sourced by its authors and they use Caffe3 as the underlying deep learning framework. Caffe was a prevalent framework at the time of this paper and enjoyed wide hardware support. It also has integrations built with multiprocessing frameworks such as Spark and MPI. Barista allows imports of existing models and pre-trained weights if any into the system, and maintains a file structure that is easy to share and transferable across machines. The interface provided by Barista is easy to use and provides a graph-based editor that incorporates context-sensitive options, thus reducing the need to access Caffe documentation. They provide support for transfer learning and training on remote machines. 2 https://github.com/hctomkins/caffe-gui-tool 3 https://caffe.berkeleyvision.org/ From an implementation perspective, Barista has been written in Python 2.7 using the Qt Framework (PyQT5) and Seaborn for visualization. The various features provided by Barista form a major source of motivation for various system requirements for DNNCASE. The specifications of Barista also serve as a baseline description of what components a graphical user interface for deep learning must necessarily provide. Lee et al. in their paper "Deep neural network model construction with interactive code reuse and automatic code transformation" [8] introduce a GUI-based modeling mechanism for DNNs, to easily transform models into code (Keras or TensorFlow). In addition to modeling DNNs, it also supports transfer learning, it provides facilities to import graph structure which can be modified. This paper also states that for representing deep neural network models, a directed acyclic graph (DAG) can be employed at the layer level. The topological sorting of graph nodes facilitates the traversal of all layers based on their connection sequence. This paper introduced the concept of grouping a sub-network of a DNN into a macroblock. Further macroblock can also be contained inside another macroblock. It states a sequence of building the DNN by first developing a top-level abstract network with macroblocks that contain lower-level sub-networks. The concept of macroblocks simplifies the process of visualization of DNNs for developers, especially for ones with less experience. The tool discussed in the paper also allows users to import and modify existing code graphically. When deep neural network model code is imported into a GUI-based tool, it can be challenging to present its structure and work with large and complex models. Hierarchical organization is necessary to make these models manageable and visually comprehensible within the tool's interface. Many excellent deep learning models are publicly available as source code. Developers often refer to these models when building their own. Thus, these publicly available models become valuable resources for reusing or extracting building blocks. To facilitate model development, the text proposes a method to identify "articulable subgraphs" within the deep neural network model's graph representation. These subgraphs are considered potential building blocks that can simplify model construction. In addition to articulable subgraphs, the text mentions "repeating subgraphs." These subgraphs represent parts of the model that are repeated. Identifying such repetitions can lead to a more compact and efficient visualization of the model. Deep learning models often include specialized modules that perform specific tasks or extract distinguishing features. These modules are part of the model's architecture and can also be represented as articulable subgraphs. Frequent articulable subgraphs are suggested as useful building blocks for constructing deep learning models. They can be considered as abstract nodes within the model. This concept has been used in the context of DNNCASE as well. We describe a module–oriented model creation, where the user creates reusable subgraph architectures, called artifacts in our system, but are also referred to as blocks by some authors [10]. Lee et al. in their paper [6] have emphasized the strong demands of automating the tuning process of machine learning architectures which makes the tuning task easy and efficient for developers. According to this paper, the main aim of autonomicity is to minimize developers' intervention in the development of machine learning applications. This paper has defined 5 levels of autonomicity ranging from level 0 to level 4. These levels are defined by automating factors like data set(D), task(T), machine learning technique(A), hyperparameters(H), and attributes to be used(P). Search space for designing the machine learning model consists of a Cartesian product of all the above factors. A major task of each autonomicity level is to reduce the search space for the developer. Autonomicity level 0 is the least autonomic level in which the developer is required to provide information about all the factors. This level doesn't reduce the work for the developer. The computations for learning and inference can be done automatically by the framework at this level which takes care of the distributed and parallel computations systems. Here the developer searches the complete search space. Interaction of the developer is most at the level when creating an ML application. At autonomicity level 1, the developers are not required to select proper input attributes in the development of ML applications. The framework of this level needs to offer the functionality of selecting and extracting features for the provided training dataset. Here the search space includes all factors except the input attribute, thus reducing the search space. Autonomicity level 2 aims to automate the hyperparameter tuning process. The framework will have the functionality to automatically determine the hyperparameters for the algorithm. Automating hyperparameter selection implies multiple trials of a hyperparameter combination on an ML algorithm and choosing the one that gives the best/optimal result. Here the search space includes all factors except the input attribute and hyperparameters thus reducing the search space. At autonomicity level 3 the developer only provides the training dataset and the ML task to be done, the rest all the factors are taken care of by the platform itself. The platform chooses the optimal algorithm, hyperparameters, and input attributes based on the training dataset and the task provided. The search space includes only 2 factors: The training dataset and the task to be done. Autonomicity level 4 is the highest and most automatic level which requires the developer to only provide the training data. All other work is done by the platform. This level requires the least interaction of a developer while creating an ML application. The search space only includes the training dataset. After reading this paper we came to know about the different autonomicity levels. We intend to reach autonomicity level 2 in our system which would support automating the hyperparameter tuning process. We would also expand our system to autonomicity level 3 in the future. Johnston et al. in their paper [12] describe the evolution of dataflow languages. They outline how research into the field of dataflow hardware, as opposed to von Neumann based processors proceeded, and the reasons for its hugely limited success. They also describe how textual and visual dataflow languages emerged and stated reasons for them being very similar to functional languages. They further proceed to outline a few open issues in data flow programming. Their paper also discusses how the view on dataflow programming shifted from one of obtaining high parallelism, to one where visual programming is supported by a visual dataflow language, and a runtime environment. Statement of Problem Deep learning projects follow iterative processes. They involve different steps. The following list shows a potential workflow [11]: 1. Task analysis 2. Model creation 3. Dataset procurement and set-up of associated functionality (such as augmentation) 4. Model training 5. Model hyperparameter tuning on a development set 6. Model testing on a training set 7. Error analysis to prioritize next steps 8. Implementing next steps and restarting from step 1 This shows that a deep learning system goes through a lot of steps that are logically sequential. We define our problem statement as a multi–step approach that culminates in a system that allows the complete deep learning workflow to be done from within an intuitive and easy-to-use interface. To realize this system, we have identified the major top–level goals that need to be accomplished. They are as follows: 1. Analysis of existing deep learning frameworks to gain an insight into what are the common workflow structures and what kinds of architectural structures and processes the deep learning practitioners are already used to. 2. To decide what basic components are necessary in an end-to-end frontend-focused deep learning CASE tool. They should be well defined and their responsibilities must be clearly stated. 3. Features that aid the developer in performing various actions, that support a better way of direct manipulation such as undo-redo functionality must be enumerated and development must be done in their direction. 4. Creation of an algorithm for code generation from graphs of neural networks, and another algorithm that can create a graphical depiction from a stored Keras model object. 5. Specification of a graphical language for creating neural network graphs, for specifying training and inference flows, and for hyperparameter tuning. 6. Creation of an intuitive interface that is responsive and easy to understand for all users, irrespective of their knowledge level and technical expertise. The interface should also provide enough hints that allow the users to decide correct actions with minimal access to external documentation. Objective We analyzed various literature sources, programming libraries, existing systems, and our problem statement goals. Based on these, we came up with the requirements that serve as fundamental objectives of our system. These objectives serve to achieve an acceptable and usable solution to our problem statement. They are as follows: 1. To identify and formally define a graphical language with the capabilities of being able to describe any neural network, using a layer-based paradigm. This language must also support training and inference flow description along with a way of describing model functions that can be used in hyperparameter tuning. 2. To create a graphical interface that can be used easily by users. Our target audience also includes students of deep learning at various academic and expertise levels. We must ensure that they have enough context-sensitive hints and menus, along with informative and well-formatted documentation shown in tooltips on a hover event on some element. 3. To create a well-designed code creation mechanism, that can take in various user specifications such as model graphs, and create executable code from them. This should be accompanied by proper validation with messages shown in real-time to the user, along with error localization where possible. 4. To encourage experimentation by including reversibility of almost all user actions, including model training and tuning, by maintaining previous model instances. 5. To allow a file structure that actively supports a project–like representation of a user's deep learning systems. It will allow de-centralized versioning and concurrent work on multiple files, by various people who may be a part of the same project team. Chapter 3: Methodology The methodology employed for this research project comprises a well-structured sequence of stages designed to guide the project's development process and evaluation. The following phases describe a summarized view of the methodology employed. Methodology is detailed in Chapters 4 – 8. 1. Literature survey: We began our project with a comprehensive literature survey to establish a strong foundation for our project's progression. By reading various research papers related to the work, we intended to explore the research done to date and compare, analyze, and build a system that eliminates the shortcomings of previous systems, while adding more features that allow DNNCASE to serve as a fully-fledged IDE for deep learning oriented visual programming. The steps we followed are as follows: a. We started the survey by visiting various research paper libraries and repositories, such as IEEE Xplore, ACM Digital Library, SpringerLink, arXiv, Google Scholar and others. b. We queried these libraries with keywords relevant to our initial understanding of the project. Some keywords we used were: i. Visual Language, ii. Dataflow Language, iii. Dataflow Programming, iv. Automated Code Generation, among others. c. Based on these, we were able to gain information about the field of visual programming, code generation, compilation, and dataflow programming. We also found various prominent researchers in the field, and were able to deepen our understanding by following their related research. d. We subsequently narrowed our focus on dataflow languages and the dataflow programming paradigm. We initially focused on a subset of dataflow programming, called flow-based programming (FBP) introduced by J. Paul Morrison in the late 1960s. He popularized this concept by publishing a book titled "Flow-Based Programming: A New Approach to Application Development" and providing various online resources on his website, https://jpaulm.github.io/fbp/ [13]. Considering the time – limited nature of our project, we analyzed the content provided on the website and learned about the distinction between FBP and FBP – inspired systems. Whereas FBP itself is an alternate system architecture to the von Neumann architecture, our system fell under the category of FBP – inspired systems. e. Morrison majorly discusses FBP in the cited sources. To get a deeper understanding of the FBP – inspired systems, we looked into the associated research to find several works on visual programming in computer network protocols. f. The research on visual description of computer protocols is more focused on design diagram-based models and automatic protocol code generation, however, did not fall under our requirements. g. We changed our focus from dataflow languages to visual languages in general and their use in deep learning. After much search, based on a now – refined criteria of keywords, researchers and institutes, we were able to locate papers [5, 6, 8] by Lee and others, and also [7] by Klemm and others. (These papers have been summarized in Chapter 2.) h. We analyzed these papers deeply, and formed our background understanding of the field. We realized the need for a dataflow-oriented paradigm besides the visual paradigm. i. Our new objective for the survey now was to revisit dataflow programming languages, outside the purview of FBP systems. We analyzed the paper [12] by Johnston and others that provides a survey overview of contemporary dataflow programming languages, both visual and textual. It includes various challenges and conditions that a language must meet to qualify as a dataflow language. We used the insights from this paper along with those from the other papers in our further system development. 2. Initial Requirement Analysis and Mind Mapping: Building upon the insights gained from the literature survey, we began an initial requirement analysis. This phase entailed a meticulous examination of project specifications and objectives. Furthermore, it we used mind-mapping techniques to visualize and structure these requirements, facilitating a coherent comprehension of the project's overarching goals. We emphasized developing mind maps from the user’s perspective and adding all the necessary actions required by the user for efficient functioning and usage of the system. We have discussed requirements further in Chapter 4. Here, we present a sample image of the top-level domains of our mind map: (DNNLang refers to the visual language of our system; See Appendix A for an overview of the finalized language.) Figure 3.1 - Top level domains of mind map used for requirement analysis. 3. Language and Compiler Creation: Following the Requirement Analysis and Mind Mapping the subsequent steps involved designing a visual language and its associated complier. We have discussed the construction and working of the compiler for the language in great detail in Chapter 7. Here, we provide an overview of the compilation process: a. Our language is visual in nature, and therefore the “code” of this language is in the form of directed acyclic graphs. (See Appendix A for a brief description of the language.) b. The compiler has to traverse the code in a topologically sorted manner. c. As the compiler visits various nodes, analyses them as being data sources, data transformations, or data sinks. The compiler expects each node (except some special purpose nodes) to perform some execution, and return some data. d. According to the data dependencies present in the graph of the code, the compiler moves each data item from its source node to its target nodes. e. Once the compiler has resolved the graph of a code file, and the graphs of any other code files imported in the current code file, the compiler then generates executable textual Python code. 4. Runtime Environment Design and Frontend Creation: Based on our literature survey and problem domain, we surmised that one of the core requirements of any visual language is the software that enables users to view and code it. To this end, we greatly analyzed and designed toward a system that can be used to create programs in our language. We describe the design of our system and its working in depth in Chapter 6. As a preview, we here present the major aspects of the environment: a. The basic element of the environment is a graph editor. Similar to text editors, a graph editor allows its users to create and modify graphical data. b. We created a graph editor using a canvas that contains a 2 - dimensional coordinate system. The canvas allows placement of nodes and edges, dragging of nodes, a minimized top level viewing window of the whole canvas to allow the programmer to get a glimpse of their workspace, and also a control panel with buttons such as zoom in/out, enable or disable writes, etc. The editor also has controls to add nodes, and modify their properties. The following images show a view of the graph editor: Figure 3.2 - The graph editor with a basic graph drawn in it. Observe the mini map on the right- bottom part of the editor, and the controls on the left-bottom of the editor. Figure 3.3 - The left and right panes of the graph editor. The left pane allows adding more nodes, and the right pane allows modifying the properties of individual nodes. c. Once the editor was setup, we required an IDE to house various instances of this editor, and to provide an interface to the file system. For this, we created an application that could open and close files, manage various tabs, house different panels, and that has various programmer – assistance features, with more potential features on the way. We explain the IDE in Chapter 6. The following image is a view of the IDE: Figure 3.4 - The DNNCASE IDE displaying the filesystem pane and errors box. 5. Deep Learning Library creation: In this phase, we incorporated various layers, optimizers, and additional features supported by Keras within the visual language framework of DNNCASE. This includes the integration of convolutional layers, recurrent layers, normalization techniques, activation functions, and other essential components that Keras provides. The objective was to ensure comprehensive support for Keras functionalities, empowering users to seamlessly import and leverage a wide array of deep learning constructs within our system. To analyze the core important functions to develop first, and to ascertain a priority order, we performed the following steps: a. We analyzed different commonly used neural network architectures and workflow functions. b. Based on these, we decided on the applicable and convenient functional nodes for the graph, that could perform a similar task, and were not cumbersome to use in a visual ecosystem. c. We then proceeded to construct applicable visual elements for these nodes, focusing on ease of usage when programming. 6. Usability Testing: To complete our project, an analysis of the usability was necessary. We conducted a survey on various undergraduate B. Tech. Computer Science students of the 3rd and 4th academic years. The objective of this survey was to gauge the effectiveness of our system along the following 4 dimensions: a. Ease or difficulty in understanding the concept of the reuse node. b. Acceptability of this visual language in education. c. Understandability of the generated neural network code. d. Effectiveness of interaction with the user interface. The various results we obtained, our survey methodology, and the survey questionnaire have been discussed in depth in Chapter 8. The survey was conducted to assess the power of DNNCASE, and the language it uses. As discussed later, we had at our disposal 2 possible language structures: a purely functional language, and a language that treated functions as top – level entities. We show that both the language structures are equally conducive to learnability in our results, by the statistics obtained in the 1st dimension (Ease or difficulty in understanding the concept of the reuse node.). As a context, the reuse node was necessary to implement functions that can return other functions, which is a requirement of top – level functions. This is an important observation, because both language paradigms can be merged into a hybrid paradigm with certain nodes emitting not just data, but functions as well. This is one of the future scopes of our project. P a g e | 17 Chapter 4: Requirement Analysis and System Design We performed the requirements analysis for our system after the literature survey. We followed a 2 – stage approach at analyzing the requirements. The following schematic displays the approach we followed: Figure 4.1 - A flowchart displaying the requirements analysis process followed for DNNCASE. Stage 1: Unbounded Features and Mind Map Grouping In stage 1, we listed those features that a hypothetical unbounded – scope IDE must support. The aim of this task was to understand the scope of our project. As our time and resources were limited, we would only be able to develop a short subset of these features. However, this task helped us to achieve an understanding of the problem domain, to find what features were similar and what were distinct, and to develop a priority order among the features. As an example, one of the features was that IDE must apply validation rules to the coded program. However, this feature is meaningless until a canvas that supports graph editing is developed. Hence, this analysis helped us in understanding what features to develop first, and which to delay for a later time period. We then grouped the features into a mind map (the mind map along with the features is presented after this discussion). This grouping was done based on similarities between the features. The mind map grouping is a hierarchical agglomerative grouping, i.e., we initially viewed each feature as a distinct point, then we kept grouping together the similar features into hierarchical clusters, which resulted into a final mind map. We call this process Hierarchical Agglomerative Clustering of Unbounded Feature Space for requirements elicitation (HACUFS). The following list shows all the features that we initially compiled for stage 1: 1. A desktop app that can run as a self-contained system on any platform. 2. The application should have multiple toolbars. 3. One toolbar should contain the project name, app icon, close/minimize/maximize keys. 4. Another toolbar should contain the menus (File, Edit...). There should also be a Help menu, showing documentation links, and also updates if any. 5. A third toolbar should contain quick access options such as undo/redo and command palette. 6. Another toolbar should show the tabs currently open in the current workspace. 7. There should be a bottom bar showing warnings, errors, git pull/push status and output format (.ipynb or .py). 8. The main work area should display contents of the open file. In case the open file is a NN graph, it should show side bars for element selection (e.g., layer) and parameter selection (e.g., hyperparameters specification). 9. Users should be able to view and edit neural network graph diagrams. 10. As a future support, the users should be allowed to open their data as a preview. 11. To edit the graphs, users should be given a choice from all Keras API objects, such as layers, activations, losses, etc. 12. Users should be able to edit the hyperparameters for each layer. 13. Users should be given the allowance to undo/redo their changes and autosave their work. 14. The undo/redo should be separate for each editable tab and each input box. 15. The undo/redo should be supported across sessions (say I make some change, and then later open the graph. I should still be allowed to undo my changes). 16. Users should be given a command palette input where they can enter some text to display all associated allowed actions (say I want to auto-format the graph. The command palette should suggest it to me when I enter 'format'). 17. Users should be allowed to auto format their graphs i.e., refactor the way the graph is being displayed. 18. Users should be allowed to move various graph entities around on an 'infinite' canvas. 19. Whenever a user adds an edge (u, v), the nodes u and v should exist. 20. Users should be allowed to create custom layers for their projects. The code for the custom layers should be given by the user. Our system should just import the layer as a black box, along with relevant metadata, such as hyperparameters, layer name, input- output count and so on. 21.

Users should be able to import other graphs in their current graph as a black-box (i.e., the imported graph won't be editable from current tab) and use it as many times as they wish. Important: each use is an independent and deep copy, i.e., separate instantiation of the imported graph. 22. Users should be able to convert their graphs into python code (Keras based). This should be supported both as a python project, and as a Jupyter notebook. 23. Users should be able to have their graphs validated. This should be done to ensure there are no circular dependencies in their project, and also no cycles in their individual graphs, among other errors. 24. When a layer is hovered upon, users should get a docstring about that layer as a tooltip. 25. Users should be allowed to create a SavedModel object from within the app as well, so that they can just import it in their code, rather than use the python code files generated. 26. When users get their graphs validated, errors/warnings if any should be highlighted in the graph itself. 27. Users should be able to work on neural network projects. This means that multiple projects, each as a folder should be allowed to be opened in the system, BUT one at a time. This is similar to the Open Folder... functionality in VS Code. 28. Users should be provided with keyboard shortcuts, and as a future enhancement, these should be modifiable by the user, on a per-action basis. Clearly, this means every user- facing action that can be mimicked by the push of a button should also be allowed to have a registered keybinding. 29. Users should be allowed to import a model from its python code or its saved format. This will only be allowed if the input (code or saved format) can be converted to a tf.keras.Model object. The loaded models will be automatically formatted according to the formatting rules. 30. In case the model is loaded from a saved instance, its layer-level weights should be preserved, unless the user wants them purged (If the user does want the weights to be deleted, only the architecture should be retained). In case the weights are to be preserved, user should be allowed to decide the same for each layer (i.e., layer level granularity, not graph level). The weights should be preserved by default. 31. Users should be allowed to reorganize the layers of a loaded neural network's graph, and pre-trained weights if any must be preserved across this reorganization. 32. The language the users use to specify a graph structure should contain the reuse block, repeater, edges, and layers, as previously decided. A layer can have multiple inputs and outputs. Therefore, if another graph is imported into the current graph, the imported graph will also be treated as a layer (which doesn't have any hyperparameters). 33. Users should be allowed to train their models from within the system itself. This means that wherever the weights were preserved, transfer learning should take place. 34. Users should be able to set up data pipelines from within the system. They should be allowed to access remote data repositories as needed via URLs, and all tf.data, which includes tf.data.Dataset, functionalities should be given to the user for such creation. 35. Users should be able to test their model from within the system. 36. As the training/testing takes place, users should be shown the graphs depicting various user-chosen metrics, along with variation of loss. 37. Users should have the option to stop training/testing whenever they wish. 38. Users should be allowed to run inference on specific input data points. 39. Users should be allowed to create a version-controlled timeline of the neural net corresponding to their project. This means if I train my model today, and re-train it tomorrow, I should have a snapshot of the model after both the training sessions. 40. Users should be allowed to perform hyperparameter tuning. This should be provided separate from the model testing and training, to ensure no compatibility issues with any relevant libraries arise. 41. Users should be allowed to decide which version of Keras they'll use, in case there are major differences across upcoming and previous Keras versions. (I.e., in case forward- or backward- compatibility is missing) 42. Users should be provided a clear screen option. This should clear the currently visible graph on the screen (i.e., in the current tab). However, users should have the option to undo this change. 43. Users should be allowed to zoom in or out of the workspace of the graph, as visible on the screen. 44. Users should be given an option to choose their python interpreter, and TensorFlow version (in case there are more than one installed). Also, users should be allowed to choose what underlying hardware they will use, in case, say more than 1 GPUs are available, or where they want to store any downloaded data (if not in the project folder itself). 45. Users should be allowed to choose what theme they want (dark/light/etc...) via configurable files. These features represent our project's initial scope, which was unregulated as yet. To bring the scope down to a more manageable size, we performed the mind – map grouping. The method for developing this grouping - HACUFS has already been discussed above. The following figure feature in the above list. Figure 4.2 - The mind map grouping of the unbounded features for DNNCASE. The mind map has been created in an agglomerative hierarchical clustering mechanism (HACUFS) working up from the unbounded features. shows the mind map. Please note that in the leaf nodes of the mind map, `Fi` refers to the ith Page |21 Stage 2: Functional and Non – Functional Requirements Extraction, UI Designing, Language Designing, Compiler Designing Based on the mind map, we were able to define abstractions that had to be formalized in the form of system requirements. As including every abstraction would have been out of the scope of our allotted time and available resources, we have retained the most important abstractions in the set of requirements. First, we mention the Functional Requirements (FRs) and then we mention the Non – Functional Requirements (NFRs): FR1: The system should allow its users to visually create directed acyclic graphs that define some processing to be performed. The graphs should then be converted into executable python code. FR2: The system must provide its users some graph editor on which the user can draw the graphs. The graph editor must have a canvas that is connected to the UI for inputs and outputs so that the user can not only view the graph but also: 1. Drag and drop nodes on the graph. 2. Connect various nodes via edges. 3. Pan and zoom the canvas to change the view. 4. Delete nodes and edges from the canvas. FR3: The canvas must provide the user with ease-of-use features. At minimum, the canvas must provide the following features: 1. Dedicated buttons to zoom in and to zoom out of the canvas. 2. A dedicated button to mark the canvas as read – only; This is required if the user merely wants to browse the code. 3. A button to perform auto zoom and auto pan so that the entire graph is visible on the user’s viewport. 4. A window to serve as a minor display of the canvas that allows panning and zooming, while displaying the entire canvas at any time to the user. P a g e | 23 FR4: The graph editor must provide the user with access to all the nodes that can be inserted onto the canvas. FR5: The graph editor must provide the user with facility to change any metadata of the nodes or of the edges. FR6: A workspace environment must be set up in the form of an IDE. This IDE must provide the user with access to their file system, to create, rename, reorder and delete files and folders. This must be done in a way that the user can only modify entities within a project folder. FR7: The IDE must provide the user with access to multiple open graph editors, one for a single file. The user must be able to switch the currently active editor. The user must also be able to close every editor. FR8: The IDE must provide the user an option for each file to generate python code of that file and download it in a .py file to any location of the user’s choice. FR9: The system must have a well – defined graphical language that is easy to learn, and is based on the concept of dataflow instead of control flow. Even though the language is dataflow, it must contain certain control structures, namely branching of dataflow based on a boolean condition, and repeated flow of data through the same sub-graph (analogous to loops). FR10: A compiler must be developed for the language of this system. The task of this compiler is to convert the graph into an executable python file. The compiler must be capable of loading stored files, and storing the generated python file. NFR1: Portability: The system must be portable across devices. Either the system must be web – based so that it can be accessed by a browser, or the system must have its compiled source code in formats that work in the 3 major operating systems: Windows 10 and above, Mac OS X, Linux Debian compliant flavors. P a g e | 24 NFR2: Learnability: The core of the system is to ease deep learning education. For this purpose, the system must be easy to understand and its usage must be easy and quick to learn. The system must have a theme that makes it easy to view and intuitive to work with. NFR3: Usability: The system must be able to create all workflows and architectures that a user wishes to create. The design of the system, of its language, or of the compiler must not present any hindrance to the user’s work. Based on these requirements, we proceeded on 2 different paths. First, we created the design of the UI using Figma. Figma is a tool that simplifies creation of static prototypes for frontends. Once the designs were created, we developed a hierarchical component structure that distributed the display and interaction responsibilities of the UI elements among smaller more modular and reusable components. The following images provide snapshots of the Figma designs. Following these images, we show snapshots of the hierarchical component structure. Figure 4.3 - A collapsed view of the DNNCASE IDE. Figure 4.4 - The "Errors and Warnings" panel display of the IDE. Figure 4.5 - The "Filesystem" pane of the IDE, providing access to user's project files. The outlined portion shows the workspace. Figure 4.5 (Contd.) - A close up of the tabs and quick address bar section of the DNNCASE Workspace. Figure 4.6 - A proposed design of the settings dialog box for DNNCASE. Figure 4.7 - A proposed design for the project selector dialog box for DNNCASE. Figure 4.8 - A design for the function node of DNNCASE. This is a node to be used in the graph editor. Figure 4.9 - A design for the input layer node for DNNCASE. This node is supposed to mark the beginning for forward propagation in a neural network. Figure 4.10 - An early design of the DNNCASE Canvas. The design also displays the left pane. The left pane is supposed to allow insertion of nodes onto the canvas. Figure 4.11 - Design of the Left and Right panes of DNNCASE Visual Canvas. Figure 4.12 - The tree of UI components, hierarchically arranged. The diagram is too large to be printed. It is available on the project's public repository at: https://github.com/rohan843/dnncase/blob/c0b4761d1242ec0cba65841d3a6554e9129d09e7/componen structure.drawio. The software "draw.io" must be used to open the file. Figure 4.13 -

A subtree of the component diagram displaying the components and their stateful information used to develop the Left Pane of DNNCASE Graph Editor. Figure 4.14 - A part of the component diagram displaying components for various nodes to be used on the Canvas in a code's graph. Once the designs for the UI were ready, we prepared preliminary designs for the interaction between the UI and the system backend. While the most recent designs are elaborated in Chapter 6, we briefly mention the initial designs to convey our thought process and designing workflow. Figure 4.15 - Various snapshots of the Level-1 Dataflow Diagram displaying the UI and backend interaction. These images are for reference purposes only. To view the full schematic, please access it from the link:
https://github.com/rohan843/dnncase/blob/c0b4761d1242ec0cba65841d3a6554e9129d09e7/processing
and data s tore structure.drawio. The "draw.io" software must be used to open the file. Now that the UI components were designed, we shifted our focus onto the second parallel path of development – the language and compiler design. From the very beginning of the requirements elicitation process, our literature survey had shown us the need for a visual graphical dataflow language. We began the development of our visual language, DNNLang in a manner similar to what we followed for our overall system, as described so far. In the case of language development, we had a different set of challenges. Thanks to the multiple streamlined DL projects, libraries, documentations and courses, the tasks performed in DL were known to us beforehand. A brief outline of those tasks is as follows: 1. Model Architecture Creation 2. Model Compilation Procedure 3. Data loading and preprocessing 4. Within-epoch Model Training Workflow description 5. Overall Model Training Loop 6. Declaration of other functions such as loss functions. We have presented this outline in our problem statement in Chapter 2 as well. We began our process by surveying the TensorFlow and Keras documentation for understanding the python ecosystem. We organized our findings in another mind map. On account of this being too large for the page, we present salient snapshots of the mind map below. The complete mind map is available here:
https://github.com/rohan843/dnncase/blob/2ebcade6e92bf0f29317b64f5d182c7d2e9e2352/DNN Lang.pdf as a PDF file. Figure 4.16 - Section of the Mind Map showing operations that a model can undergo. Figure 4.17 - Mind Map section showing entities relevant to model and hypermodel architecture creation. Figure 4.18 - The figure shows the notation followed in the mind map. Figure 4.19 - Sections of the Mind Map displaying data loading and transformation operations. Figure 4.20 - Figure displaying all sections of the Mind Map relevant so far in the discussion. The complete Mind Map will follow later in this Chapter. Based on the various tasks our language is meant to support, we began the construction of the most basic node in our language – the function node. We conceptualized this node as a node that takes in zero or more data streams, performs some processing based on them, and optionally emits a single data stream. We further analyzed our domain to conceptualize the notion of "data" and "datatype" in the context of our language. We further created various nodes to perform functions such as branching and iteration, in line with our functional requirements. We also conceptualized the fundamental entity that gets passed to the compiler – an artifact. Further domain analysis (which we performed by analyzing sample Keras codes from Keras and TensorFlow websites) showed that the language still lacked important components. There were different kinds of responsibilities that different function nodes must display. To make the visual aspect in line with this, we created the concept of "nodetype". A node's nodetype defines its visual aspects. This means that a function node of type `layer/add` which processes input data to give outputs may look like: While another function node of type `layer/input` which is a data source may look like: This distinction in display makes it clear that one node is meant for data transformation, while the other is a data source. This is a visual distinction, while from a language perspective, both nodes are logically the same. Another component that our language lacked as of now was the ability to treat different artefacts differently. To explain this point, we draw a parallel between DL workflows and website development. When a website is built, its structure is declaratively specified via HTML, much like the architecture of a model is specified by a graph of layers. The styling of the website is specified via CSS again declaratively after construction of the barebone HTML, much like the compilation of a model is specified after its architecture is described. Lastly, websites have functionality via JavaScript. Similarly, DL workflows have functionality such as defining custom losses, training loops and so on. This analogy shows that DL workflows are neither fully declarative, nor fully functional. They have both aspects. Continuing this line of thought, we developed the idea of declarative artefacts and procedural artefacts. But issues persisted. There are different declarative and different procedural artefacts. To resolve this, we conceptualized the concept of "artefacttype". The artefacttype of an artefact is used to tell the compile what interpretation process to apply to any given artefact. Even though an artefact may have its own artefacttype, it is still treated as a

function when imported in other artefacts. This solution enables both custom – processing of artefacts, and interoperability between different artefacts (such as by invoking one artefact within the graph of another). P a g e | 37 For more information on the language aspects, please visit Appendix A, and also please view the Mind Map (link provided previously). In line with the language developed, we constructed the logic and function documents and diagrams for the compiler. The discussion of the compiler is provided in – depth in Chapter 7. The following images display the parts of the mind map relevant to the finalized DNNLang, and the compiler. Figure 4.21 - The part of the Mind Map displaying elements of the language. Figure 4.22 - The part of the Mind Map displaying elements of the compiler. Figure 4.23 - The part of the Mind Map displaying information on datatypes, nodetypes, and artefacttypes. Figure 4.24 - The part of the Mind Map displaying the part on the concept of Data. Figure 4.25 - The part of the Mind Map displaying information about the concept of a function. Figure 4.26 - The part of the mind map displaying information about nodes (cyan boxes). Figure 4.27 - The complete DNNLang Mind Map. This concludes the discussion on requirements analysis and system design. Chapter 5: Implementation and Challenges Our Software Development Life Cycle 1. Our development cycle started with gathering the requirements (both functional and non- functional), post which we created mind-maps to add more enhanced set of requirements to each component and to have a general outline of what needs to be built. 2. After that we went forward with the development of system, in phase one we started focused on the UI and code generation algorithm. We thoroughly studied the Keras documentation to understand various constructs used in deep learning and making sure our architecture supports all of them. The System contains all the constructs of deep learning in the form of nodes and we can connect these nodes with edges. For e.g.: there are nodes of Conv2d, Dense Layer and other layers. 3. We also have nodes for input (specifying the dataset) and output (getting the output like probability, etc.). 4. While developing the system we stumbled upon a very rarely used network Architecture called Siamese network that is often used in tasks involving similarity or distance measurement. It consists of two identical subnetworks which share the same parameters and weights. 5. These subnetworks process two different inputs separately, and then some similarity metric is applied to the outputs of these subnetworks to determine the similarity or dissimilarity between the inputs. In a Siamese network, the two subnetworks share the same parameters. 6. This means that the weights and biases of the layers in each subnetwork are identical. For supporting this we came up with a Idea of Reuse blocks which helps us do the same, with Reuse block we can wrap a layer inside and have as many inputs and outputs to this as possible. 7. Siamese networks are commonly used in tasks such as signature verification, face recognition, similarity-based recommendation systems, and more. 8. They are particularly useful when dealing with tasks where labeled training data is scarce, as they can be trained using pairs of similar and dissimilar examples rather than relying solely on labeled data. 9. With this requirement, Reuse blocks can help us re-use the same layer parameters and apply on other data. Other important problem which we saw while development was that it was difficult to create deep networks on UI as deep networks requires a large number of inputs travelling from one side to other which may cause confusion and to solve this we came up with concept of Packer/Unpacker Nodes, through which we can pack a set of inputs and now travel on the canvas with just one input line and unpack it where required, this makes the development of neural networks fast and clean. 10. Also, to support adding different parameters and hyper-parameters associated with Layers we introduced a Right Pane which helps us to manually put those specifically for each node, all you have to do is select the node and open Right pane, put the details. 11. Apart from development, we also focused on developing multiple dataflow diagrams for systematic development of front-end and back-end, specifically for code generation algorithm we created an entire workflow to analyze the working and deal with any edge cases if exists. 12. After first phase of development and testing we went forward with doing a survey to understand the usability of system and to collect feedbacks. 13. Once we were done with survey, we re-iterated on requirements and updated mind-maps to adapt the latest requirements and behaviors which we refined as part of survey. 14. One major point we got as feedback was to support the entire deep learning pipeline including the training, testing and hyper-parameter tuning of the networks. 15. In the process we came up with multiple new constructs on UI like concept of artifacts, for- loop node, if-else node etc. Our Aim is to support all types training, testing, basic and custom hyper-parameter tuning. Custom hyperparameter tuning would require repetition of certain code and taking decision based on some if/else condition so we introduced the constructs of if/else and for-loops, etc. Our Research Survey 1. We conducted the research survey with college students to understand the usability and to get feedbacks about our system. 2. We started with an introduction of our system and explained what purpose it serves. 3. Post which we asked them to explore our system and ask if they have any doubts using it. 4. Then we gave them a Kaggle notebook with some Deep Learning tasks on MINST dataset and Fashion MINST dataset. They need to do the task by using our system i.e. use our system to draw a deep neural network architecture, generate code and use that code in the Kaggle notebook. 5. At the end, we asked them to fill a google form which contains questions on usability and feedback. Challenges That We Faced 1. Our first challenge was the requirements gathering process. Although we had studied it in Software Engineering, a real – life process for a project of such a large scope is a complex process that required a great amount of analysis and ideation. 2. We also found learning the intricacies of our chosen tech stack challenging, especially, understanding the process model and IPC of ElectronJS. 3. Once our system was developed, we had to conduct a survey for evaluating it. As this was a new experience for us, it required a lot of planning. This was a challenging task. 4. Creating a new language is not a common task. It was our objective from the beginning, but the process we followed in creating it was not a straightforward one. We were met with lots of difficulties along the way. Creating a compiler for this language was also a challenging task, which we spent a large amount of time designing and developing towards. Software That We Used This project has a well–defined scope, but within it lie various components, and different facets of the system require different technologies to implement. Based on this, we have identified various libraries and packages that we will require in the implementation: 1. ReactJS: This is a JavaScript framework that can be used to create various web-based interfaces in a modular manner. We will use this to create various frontends for our system. 2. ElectronJS: This is a JavaScript-based library that can be used to create windowed interfaces. We will use this, as we have in our prototypes, to create the main screens. 3. React – Flow: This is a library that allows us to render graphical data. 4. Keras: This is the main API based on which our system operates. We use its constructs to run generated backend code. 5. Figma: This software allows us to create graphical interfaces on a screen so that we can visualize them before we begin designing. 6. Diagram.net (Draw.io): This is a diagramming software, wherein we can create schematics for various design diagrams. 7. Miro: This is a mind mapping software allowing us to create mind maps. 8. GitHub/git: We use these tools for version controlling and maintaining a centralized codebase. Chapter 6: Runtime Environment Design and Frontend As stated before, a runtime environment and a graph canvas are very important to visual languages. We developed a graph editor and an IDE. In this chapter, we aim to provide the details of both of them. We also provide a description of the way the environment treats the file system. The IDE The IDE consists of 6 major parts: 1. Left Bar 2. Left Panel 3. Bottom Bar 4. Bottom Panel 5. Top Bar 6. Workspace The left bar and the left panel are pictured below: Figure 6.1 - The Left Bar and the Left Panel. These contain access to the filesystem view, the visualizer, and the version control view. So far, our system requirements and therefore the aim for our project only necessitated the development of the filesystem view, however our system features do include visualizing and version controlling capabilities. Should DNNCASE be further developed, they will be accessible here. The bottom bar and the bottom panel are pictured below: Figure 6.2 - The Bottom Bar and the Bottom Panel. The bottom bar provides access to traditionally "landscape" information, such as validation messages, logs, etc. We outline a virtual assistant in Chapter 10 (Future Work). When created, the assistant will also be present in the bottom bar. The top bar is pictured below: Figure 6.3 - The Top Bar Besides the usual menus, logo and minimize/maximize/close window buttons, the top bar has a command palette. Here, users are allowed to enter some command from a set of commands for quick execution, rather than searching through menus and dialog boxes. The command palette is another feature that is not yet included in out system requirements, but has been analyzed in Chapter 4 as an unbounded feature and a Mind Map grouping. The workspace is pictured below: Figure 6.4 - The Workspace. This is the workspace. The majority of its screen share is taken up by the graph editor. However, at its top, there is the tab bar, showing various tabs and the code generator button on the right. The bar right below the tab bar is the address bar, displaying the address of the currently active file. Below that is the graph editor. It is described in the following section. The Graph Editor The graph editor consists of the following 5 parts: 1. Visual Canvas 2. Control Buttons 3. Mini Map 4. Left Pane 5. Right Pane The Visual Canvas is pictured below: Figure 6.5 - The Visual Canvas. The visual canvas is the area where the user creates graphs. It is a 2 – dimensional coordinate plane with an overlay grid. The canvas supports zooming and panning. It also supports drag – and – drop of nodes and creation of edges. The Control Buttons are pictured below: Figure 6.6 - The Control Buttons. These 4 buttons are (top to bottom) zoom in, zoom out, fit to window and disable edits. The fit to window button makes the graph in the canvas be fully visible to the user at once. The disable edits button disables dragging/deletion of nodes and creation/deletion of edges, allowing for a simple code browsing. The Mini Map is pictured below: Figure 6.7 - The Mini

Map. This mini map is created as a visual aid to the programmer. It displays the entire graph in the background, and the user's current viewport position in the foreground. It also allows zooming and panning. The left pane is pictured below: Figure 6.8 - The Left Pane. This pane contains various nodes that can be added onto the Visual Canvas by the user. The right pane is pictured below: Figure 6.9 - The Right Pane. This pane allows the user to modify the properties of any specific node, and to also add any internal (code) comments as needed. The user can view and edit the properties of the activated (selected) node at a time. The State of File System The runtime environment requires 2 important fields of information about the file system: 1. What files come under the current project, and their contents. 2. What files are currently opened. To track the files in the current project folder, the following code snippet shows the internal structure: Figure 6.10 - The way the state of the files in the project folder is saved. We store each folder's and each file's information. The contents of some files are loaded into the system. Each folder stores the names of files and folders within it. To track the currently opened files, we use the following array structure: Figure 6.11 - The internal array to store currently opened files' information. This array stores the names of the files user is currently working on and are open in the workspace. With each file, the system also stores the state of the graph editor for that file (see the `config` key). The first file object in this array is the file user currently is viewing. Chapter 7: Compiler Working We have created our own compiler for our system which will traverse all the architecture created by the user and generate its corresponding Python code. The code can then be used by the user on the required dataset. This will make it convenient for the user to focus on the concepts of Deep Neural Networks rather than the coding part. The compiler completes all the steps as done by general ones like lexical analysis, sematic analysis, syntactic analysis. The details steps/function of the compiler is as follows: 1) When the user asks the system to generate code, firstly the runtime environment (frontend) will send the set of the files to the compiler. The compiler's execution begins from here where there will be know set of files and the initial file to begin execution with. These files will be containing the information in textual format. It will be containing information about the artifacts, nodes, and edges. The compiler will individually load all the files available at the initial step. All these files loaded in in-memory structures can be accessed easily. Figure 7.1 - Flow Chart Depicting file loading in in-memory structure. 2) Now, to convert textual format to objects, compiler will parse each textual file from in- memory structures. These files will first be parsed, to create an object and then the contents in the object will be scanned. This is the lexical analysis part of the compiler where the objects are broken into smaller pieces. File scanning the contents if there is any file which is required, if the file is present in in-memory structure the above process will be repeated. This will be done for all file. If file is not present in in-memory structure then Step 1) will be repeated. After contents of all the required files are scanned, the contents of the files will be converted in key-value pairs in Map, which will make it easier to access the necessary information. Figure 7.2 - Flow Chart Depicting the process in Step 2. 3) Now the compiler has to traverse all the artefacts. While traversing an artefact, it is possible that another artefact is required to be traversed before the current one. To solve this, the compiler first needs to find the correct order of traversing the artefact, such that the independent ones are traversed first and later on the dependent ones. The algorithm for the scheduling of the artefacts is: i) We start by traversing all the artefact and finding on which other artefacts they are dependent on. A DAG of the artefacts is created. It will be an acyclic graph with information about the neighbors of an artefact. ii) Now the compiler will apply topological sort on the graph created to find the ordering of artefacts. Depth-First-Search topological sort will be used. iii) The compiler will be maintaining a visited array and stack. We will be traversing all the non-visited nodes and for each node, if a node has its neighbors and it is not visited, then recursive function will work on the neighbor. If a particular node has no neighbors or all neighbors have been visited, then the node is pushed into the stack. iv) Finally, the elements of the stack are popped and added to the array which gives us the scheduling of the artefacts. This ensures that there will be no conflict while traversing any artefact. Figure 7.3 - Flow Chart depicting artefact scheduling algorithm. 4) The compiler will now be traversing the artefacts according to the schedule return in previous step (3). The steps in traversing each artefact are as follows: i) Depending on the type of the artefact, corresponding function will be called to generate Python function code for the artefacts including the parameters required. Then compiler will start by traversing the node which has zero indegree and is not a data variable node. ii) The compiler will first check the node type, then call the corresponding function which would return the Python code for this node. The Python code will then be appended to the string. After this, we move to the next node. iii) As we reach the next node, the first condition to check is the number of inputs required to execute the node. If the condition is meet Step ii will be executed, else the function will be returned and other nodes will be traversed. Basically, a recursive function will be called for each node. iv) After all the nodes are traverse by repeating step ii and iii recursively, we get the Python code for the artefact. Figure 7.4 - Flow Chart depicting algorithm for traversing a single artefact. After generating the code of all artefacts, we generate code of all the required imports. The artefact which was the initial artefact, its function must be called on the top level at the end of the file. Chapter 8: Results and Discussion To evaluate and test the effectiveness of our system we conducted a survey among college students. The aim of the survey was to know if students were able to conveniently use our system and were able to associate with the code generated by the compiler. The workflow of the survey was: 1. The very first step was to coordinate with students, so that we get a large group to conduct the survey. This process included talking to the professors, communicating with students, and taking the necessary permissions. We recruited 74 computer science undergraduate students (3rd and 4th years) who had some knowledge of neural networks. 2. During the survey, our first step was to explain our system to the students. We first explained our problem statement of our project and the end goal we want to achieve by conducting the survey. We gave a detailed explanation to the students about our system by explaining the different features in our system, how to add specific node, how to add edge between the nodes, how to add hyperparameters for a node, drag-drop feature. This was done to give a brief understanding about our system to the students. 3. After this, the students were given time to explore our system on their own. Simultaneously, they were given two tasks on Kaggle where they were asked to create the architecture on our system. The first task was a digit recognizer task where the students were required to create an architecture such that the neural network can recognize the digits. This was a normal task with digit recognizer dataset. The second task required the use of Siamese network, where the model was required to predict/tell the similarity between two images. For Siamese type networks, we created a concept of Weight Sharing called reuse node, which made it easier to draw the architecture. 4. The students then started to create the neural network architecture on our system. While most of the students were able to make the architecture conveniently, some students didn’t understand the task. So, we gave them an in-depth explanation of the task and also gave them a possible architecture they can create. 5. Then the students started to run their notebooks to check if the code generated by our system for their architecture works or not. The students also tested the model on test data. 6. Finally, a form was circulated to the students to get their feedback about the system, concepts, understandability and how much they were able to associate with the code generated. After this our survey ends. In the survey, we majorly focused on the following 4 dimensions: 1. Ease or difficulty in understanding the concept of a reuse node. [Q1] 2. Acceptability of this visual language in education. [Q2, Q3] 3. Understandability of the generated neural network code. [Q4] 4. Effectiveness of interaction with the user interface. [Q5] The results for the above section, along with the survey questions are: Q1. To what extent did you understand the concept of a reuse block? Figure 8.1 - A histogram displaying the frequency of user ratings on a 5 – point scale based on their self – assessment of their understanding of the reuse node. We asked the users to rate on a scale of 1 (lowest) to 5 (highest) how confident they felt in their understanding of the reuse node. The graph in Figure 8.1 displays the results we obtained. The highest ratings given are 4 and 5, which shows that the users found it easy to understand the reuse node's operation. Q2. What medium do you prefer for creating neural network architectures for practical DL tasks? Figure 8.2 - A pie chart depicting the different mediums users prefer to code neural network architectures. Q3. Would you prefer to use this language in a course about Deep Learning and Neural Networks? Figure 8.3 - A pie chart showing the distribution of students who prefer to use a visual language for a course on deep neural networks. To assess the acceptability and applicability of the visual language for educational purposes, we asked the users about their preferences, once they finished the survey tasks. We have summarized the results in the figures Figure 8.2 and Figure 8.3. 65.6% of the users said they preferred the visual language to text based programming languages when creating neural P a g e | 62 network architectures. At the same time, 90.6% of the users said that they would prefer the visual language be used in a course teaching about deep neural networks. To further probe into the reasons for the users' choices, we asked for their comments. An analysis of the comments showed that the major reasons were: 1. A visually drawn network is simpler to view and understand – the visual representation is closer to a graph-based representation that users utilize. 2. Users felt that the visual method will save time when building architectures. 3. Users were able to view the components they would insert into the canvas, which provided with recognition cues. Q4. To what extent were you able relate and associate generated code with the visually drawn graph? Figure 8.4 - A bar graph depicting the extent to which users could associate the code generated by our system with the neural network graph made by the user.

We asked the user to rate the extent to which they understood the code generated by our system with respect to the neural network graphically created by them. The graph in Figure 8.4 shows the results obtained. 75% of the users have given 4- and 5-star rating which shows users were satisfied and easily able to relate with the generated code. According to some users' comments, they found it easier to make graphs of neural networks and have the code automatically generated, rather than writing it. Q5. How was your experience in creating neural network graphs? I.e., how favourably do you view the system interface? Figure 8.5: A bar graph depicting how favourably users view our system interface. We asked our users to rate how favorably they viewed the interface. We present the results in the graph in Figure 8.5. 62% of the users gave a rating of 4 or above, however, a considerable number of users gave ratings below 4 as well. This suggests that there is still room for improvement in the design of a suitable interface for this visual language. We asked for the users' comments to infer the reasons for their ratings. Based on the results, we were able to find the following issues with the interface design: 1. Using a touchpad instead of a mouse (for instance, when using a laptop) can make it difficult to move around the canvas and place nodes. 2. Understanding and navigating the interface was challenging for some users. Notably, though, some users mentioned that once they did grasp the layout of the system, the controls were easy to access and use. 3. The procedure of inserting a reuse node into the graph was not convenient when multiple such nodes were needed. Discussion In this report, we described a graphical language capable of describing architectures of various neural networks and their associated workflows. This language has been designed to be similar to visual dataflow languages. One critical and experimental aspect of the language is if we allow function nodes themselves to construct other functions. This pattern of a function constructing another function was not directly mappable to the visual interface. To test this aspect (which has not yet been included in DNNLang), we allowed functions to create other functions, which then “replace” the previous ones. We also created a reuse node to test this unconventional aspect. Based on the results above, we found that the reuse node did not prove very difficult to understand by new users. We developed an interface that allows users to use the visual language. We presented the interface to multiple users to analyze its suitability, as described in section 4 (Results). Based on the results, we noticed that the users were inclined towards having this system integrated in educational courses about deep neural networks. We noted that the code generated by the system was easy to relate to the graph drawn. As regards the system interface, we note that the users found some issues related to ergonomics of design, such as difficulty in navigating the canvas using the touchpad. This demonstrates that the frontend design of a programming environment for this language still requires further analysis. We found that users preferred to use laptops or desktops to use this system. We further found 4 major methods of interactions users preferred when using this system: 1. Using a mouse for pointing. 2. Using a touchpad for pointing. 3. Using a touchscreen (and optionally a stylus) for pointing. 4. Using a keyboard for entering information. A suitable programming environment for this visual language would require addressing all these modalities. As compared to the other available tools, our system seeks to extend the functionalities, by adding more language constructs. We described a set of constructs in the form of nodes and edges, that can be used as a formal basis for creating neural network architectures, and systems that facilitate the same. This system can therefore be used to develop much more complex architectures and workflows visually than what was possible before. The earlier possibilities are outlined in Chapter 2, but we present a brief comparison below: 1. Our system allows creation of Siamese architectures, where same layer’s weights are reused. Previous systems do not allow this. 2. Our system allows model training with custom workflows, allowing architectures such as GANs. 3. Our system provides a very complex model tuning framework, where the model tuning can not only be used to tune layer hyperparameters, but the architecture of the model itself. Chapter 9: Learnings Technical Learnings 1. SDLC Related: 1. Clear Requirements Gathering: In Our Software Development Life Cycle, we always focused on gathering clear requirements and finding effective ways to solve a problem. Throughout the process of development, we understood the importance of clear requirements and how we should be specific in requirement specification so that we can plan our goals and their deadlines. Unnecessary requirements delay our development process and hinder us to achieve the set target. Also, when multiple people work on the same project its crucial to maintain consistency between each individual’s understanding of requirements. We also created multiple mind-maps to specify and document the requirements clearly all at one place. The HACUFS method we outlined before did not exist in literature previously. Coming up with it, and using it for requirements elicitation was an unprecedented learning experience. 2. Iterative Development: We witnessed the power of iterative development; we first came up with a prototype or initial version of our system and went forward with feedbacks and it helped us see significant aspects which needs to be incorporated. 3. Flexibility and Adaptability: SDLC taught us to incorporate the changes and be adaptive and flexible to changing requirements. Being open to adjustments and course corrections is essential for project success in dynamic environments. 4. Continuous Improvement: SDLC taught us to continuously seek improvements by taking feedback. 5. Understanding version control system and its importance: Since our development involved contribution from 3 people, we need to have a version control system and We leveraged Git to fortify our development process. We created a master branch to have the main code. The master branch in Git typically serves as the primary, stable branch of a project. It represents the main line of development and usually contains the most up-to- date, production-ready version of the code. Developers often use the master branch as a reference point for releases and deployments. On the other hand, feature branches are created to implement specific features or work on distinct tasks within the project. Each feature branch is typically branched off from the master branch and is dedicated to a particular feature or issue. We also created release tags for our two prototypes which we developed at an early stage of development. This whole version control helps to do parallel development, easy merging of code and to help us keep track of the development. 2. React Related: For development we used React as front-end and plain vanilla JavaScript for backend (compiler creation). React is one of the most popular libraries for front-end development and it became our choice as we have a vast set of libraries for React based development that eases out our work. We used react-flow which helped us create the graphical components in our system, it helps us manage our code and provides lot of abstracted functionality which we would have to write otherwise. We also used React state management tool: Redux, which helped us store multiple global states which in turn helped us create multiple features that enhance the user experience. In React, we learned out Component based development which helps us write code in the form of components that makes it all clean code and easy to navigate. It has large community support which helped us pass through many errors that we get while development. 3. Portability Related: In order to make the application portable we utilized a library called ElectronJS which helped us create a desktop application and a simple exe file that can be transported to other systems easily. Electron.js is an open-source framework that allows developers to build cross-platform desktop applications using web technologies such as HTML, CSS, and JavaScript. It combines the Chromium rendering engine and the Node.js runtime, enabling developers to create native-like desktop applications that can run on Windows, macOS, and Linux operating systems. Internally, ElectronJS uses a multi-process architecture to manage the application's functionality and user interface. This architecture consists of two main processes: the main process and renderer processes. The main process is responsible for managing the application's lifecycle, handling system-level events, and coordinating communication between different parts of the application. Renderer processes are responsible for rendering and displaying the application's user interface using web technologies. Non – Technical Learnings On the non – technical side, we were introduced to the various aspects of conducting a survey beyond just the preparation of tasks and questionnaires. 1. We understood the need of creating questions that the users would want to answer. 2. We also experienced first-hand how to explain concepts to a moderately sized group of students. We learned that it is not just about imparting knowledge, but also about presenting it in a way students would want to listen. 3. Finally, we were introduced to the academic research process. We understood how to structure and write research papers, the difference between conferences and journals, and how to submit research papers to journals. Chapter 10: Future Work The future work that we have planned is as follows: 1. Comprehensive Dataset Management Library: DNNCASE provides an easy method to create neural network architectures, training workflows and tuning workflows. However, the current data manipulation methods are not developed to their full potential. Our future work in this matter will be to analyze, tabulate and create a model for data loading. This model must incorporate multiple sources such as CSVs, databases, etc. Based on this model, we plan to create a library of nodes that can allow easy and visual methods of loading data into the system. 2. Insights and Tips Assistant: DNNCASE provides us an interface to create and execute deep neural networks but expects a certain level of knowledge to operate, but with a Virtual Assistant it can help its user to learn different aspects of it based on user's understanding and context. It will provide user information tips, suggested network architectures from user's drawings so far. It will provide a guided path to various common architectures to help users learn hands-on. Also, the assistant will display performance metrics, such as accuracy, loss curves, and confusion matrices, to help users evaluate model performance. 3. Deeper research on HCI aspects of a graph editor UI: In the course of our survey, we found that our UI for the graph editor, while easy to use with a mouse, presents difficulties with other kinds of pointing devices. Further research into user interaction with the UI of the graph editor can prove fruitful for the user experience. Chapter 11: Conclusion Our project describes a way in which a graphical language can be created to describe a neural network architecture and workflows visually. We have described an algorithm for converting the graphical specifications into Python code. Our current language can support the development use cases of most neural network workflows. In this project report, we also presented the opinions of various users that interacted with the system. We demonstrated the acceptability of this language in learning and teaching deep neural networks, and the ease associated with using a visual interface to specify neural network architectures. We also described some issues associated with a programming interface created for this language. Development of a suitable interface for such a visual language remains an essential step towards using this language in an educational capacity. The ease of use associated with a visual interface offers a more intuitive approach to create neural network architectures, which lowers the barrier to understanding deep neural networks for beginners and enhancing productivity for experienced practitioners. Appendix A: A Technical Description of DNNCASE Visual Language This appendix presents a brief description of the language used in DNNCASE. We refer to the language as "DNNLang" in the following text. The code of this language is drawn on a visual canvas. We refer to this process as "writing" the code. The following are the salient points of the language: 1. DNNLang is a visual language that can describe program flows visually. The language focusses on data flow instead of control flow. This means that the language describes data dependencies among several components, called as nodes. 2. To compile DNNLang, a compiler is constructed. This compiler translates the language files into equivalent code of some target language. For the purposes of this project, the target language is python. 3. The basic element of execution in DNNLang is an artefact. This is similar to functions in other languages, such as C. An artefact is a directed acyclic graph with a finite number of inputs and outputs. The execution begins from a user-specified "main artefact". This is analogous to the int main() function in C programs. 4. The edges in the graph of an artefact primarily declare data flows, i.e., data dependencies. 5. The nodes in the graph of an artefact represent data sources, data sinks, or data transformations. 6. There are certain special nodes. These are: Input Node, Output Node, For Loop Node, While Loop Node, Repeat-For Loop Node, Data Variable In/Out Nodes, Boolean Branch Node, Branch End Node, Artefact Specifier Node, Named/Ordered Packer Nodes and Named/Ordered Unpacker Nodes. 7. The most commonly used node is a function node. When specified in a graph, this node can be a user-defined function of the target language, or it can be some artefact. 8. Codebases are structured as follows: a. Code is organized into modules. A module is a folder that contains a config file and several code files. b. A module may contain other modules, called sub-modules. 9. At times, it may be necessary to specify an artefact that needs to be used later, analogous to function pointers in C. This is done by specifying the relative path of the artefact within its module, and the artefact's module to an Artefact Specifier Node. 10. The language consists of Pseudo-Nodes as well. A pseudo-node is used to specify information regarding any code file to the programmer, much like comments in other languages. Code files in DNNLang are displayed on a visual canvas. A pseudo-node can be placed either at any position on the visual canvas using absolute coordinates, or it can be placed on the canvas relative to some other node or edge. 11. Some nodes may contain nested graphs. This occurs in the case of loops, where a loop node has the loop "body" specified within it as a graph. 12. The language has various data types: a. Primitive Datatypes: These are the most basic datatypes of the language. DNNLang has 4 primitives: Integer, String, Float, Boolean. b. Box Datatype: This is a special datatype. Data of this type is used to tell the compiler to treat it as textual code of the target language. This is useful in cases where say, an Integer in DNNLang may refer to a 32-bit int in the target language, but a 64-bit int is needed. c. Container Datatype: This is an aggregate datatype. It consists of 2 types: Named Container, where key-value data is stored, and Ordered Container, where arrays are stored. P a g e | 73 Appendix B: Accessing the project's GitHub repository This appendix describes the structure of our GitHub repository and instructs on how to browse, clone, and run the code. Accessing the Repository The repository can be accessed at: https://github.com/rohan843/dnncase. The following image shows the structure of the repository: Figure B.1 - The structure of the DNNCASE repository. Here, the folder titled system-code contains the source code of the system. Various design diagrams are present within the Design Diagrams, New Language Examples, and system-docs folders. Cloning the Repository and Executing the Code The code can be executed by following these steps: 1. Install NodeJS from its official website here: https://nodejs.org/en/download. 2. Follow all instructions to ensure the `node` command works from the command line. 3. Next, install git from its official website here: https://git-scm.com/downloads. 4. Now, run the following commands on the command line: Figure B.2 - Commands to execute the code for DNNCASE. Now, DNNCASE should be up and running! Appendix C: HACUFS – Hierarchical Agglomerative Clustering of Unbounded Feature Space for requirements elicitation We followed a requirements elicitation process which we call HACUFS. This process allowed us to define a scope and requirements for our project. This process was briefly discussed in the text in Chapter 4. Here, we elaborate on this process in its own right, so that it may be referred to by other software practitioners. When to carry out HACUFS? Consider you have a project that you have to deliver in a limited time, and with limited resources. Now, add to the already complex circumstance that the project by its nature has a large set of aspects and that the users may expect a diverse set of features. It can be very difficult in such a situation to decide which features to work on, which to combine, what modules to create, and what priority order to follow. If your software project seems to fall in this category of projects, performing HACUFS may provide a solution. What to expect as end deliverables once I've carried out HACUFS? Now that you are considering using HACUFS for your requirements elicitation process, you can expect the following items to be available to you after the process is finished: 1. A set of features that are relevant to your project. These features may not all be possible to implement in your limited time availability. However, those that go un-implemented, serve as a good source of future directions of your project. 2. A hierarchical tree-like diagram (a mind map, for example) consisting of various internal nodes that can be mapped to independent modules that will constitute your system. Subsets of the features will be present at the leaf nodes of this tree structure. 3. A priority order that tells what modules are essential to the system, what are necessary but not essential and what are nice to have. How do I carry out the HACUFS process? Once you have decided to conduct HACUFS, you should follow these steps: 1. Assemble your team and brief everyone about the project, and any technical details related to it. (For example, if your project is about creation of a word processor, background knowledge about character sets, layout strategies etc. will be an advantage to know beforehand). Ensure all team members are on the same page as regards the objective (potentially informal at this point) and that they know why the project is being worked on. 2. Now, conduct a brainstorming session. In this session, there must be a conductor and a scribe. (One person may assume both the roles as well.) The conductor gets everyone assembled in the brainstorming room and goes around the group in a circular fashion. At a person's turn, they should give a set of ideas for the features or requirements they feel are relevant to the project. At this point in HACUFS, all ideas for features must be taken in and written down by the scribe. Preferably, all ideas written down so far must be available to be read by all participants. The brainstorming session goes on until the group is out of ideas. 3. After the session, the conductor (and possibly some other teammates) must number all features and write them down in a list document. The numbering here is arbitrary, and is only intended to give each feature a reference ID. At this point, some features may come out to be too similar in their wordings or meanings to each other. They may be combined. This list document is now our set of unbounded features. It contains all those features the teammates could think of, without any resource or other restrictions on the scope of the project. 4. Now, the team should perform an agglomerative clustering of the features. Do this process in the following manner: a. Read all the features, while writing those features that seem similar to each other together on some whiteboard. b. For those features that seem similar, write them closer to each other, assuming the whiteboard is a 2d coordinate system, and the features are points in this system. c. Now, we have various features arranged visually. Those features that seem to either require similar operations, or those that seek to achieve similar goals should be grouped together. (E.g., in a word processor, underlining text and italicizing text are very similar as both style text.) One feature can only be in a single group. At this point, the whiteboard should have some groups of features, and other individual features. Treat the individual features as groups of size 1. d. Some of the feature groups at this point must be assigned a module-based abstraction. Analyze any feature group, and try to decide if a single module should be responsible for these features. Appropriately name the module responsible for those features, and write down the module name on the whiteboard near the feature group it is responsible for, and make a directed edge from the module to its feature group. e. As this process continues, it may happen that one module can serve as a submodule to another module, or that multiple modules may be clubbed together under a larger module. It is also possible that the team decides to assign more feature groups to some existing module. Keep updating the modules on the feature space, by making a directed edge from a larger module to its sub – modules, and to any feature groups that the module is responsible for. f. As your team nears the end of this process, there will be several tree-like structures on the whiteboard. It is possible that some feature groups are not yet placed under any module. For these feature groups, perform a re – analysis of their relevance to the project, and the correctness of grouping them together. If any irrelevant feature can be identified at this stage, remove it from the feature space. If any incorrect grouping is found, the group may be broken down into smaller groups. These smaller groups may then also be merged into other feature groups or assigned to other modules based on relevance. g. The process is said to end when no feature group is left unassigned, and all modules that could be sub – grouped under larger modules have been grouped likewise. Now, the whiteboard will contain several trees of hierarchically arranged modules and submodules. Create a new node, that is to be named as the “Project Node” (e.g., in case of a word processor, it may be called “Word Processor”) and make directed edges from it to all the root nodes of the module trees. 5. In following this process, your team now has various possible hierarchically arranged modules that your project needs to fulfill all the features of the feature set. We call this structure a Mind Map. After this stage, we perform assignment of priority to the modules. 6. Your resources are limited, and therefore all the modules, to develop all the features will not be possible to design and code. Therefore, it is required to analyze the mind map. a. First, write in a few words what exactly is expected of each module. b. Then, to each module, assign one of 3 priority levels: Core, Needed, Nice to Have. c. The Core level must be assigned to those modules that your system absolutely needs to perform its most basic tasks. (For example, a word processor needs a text display and edit module.) d. The Needed level must be assigned to those modules that your system will highly benefit from having, and may increase your returns from your project. (For example, a word processor will highly benefit from a spelling checker module.) e. The Nice to Have level must be assigned to those modules that, if implemented, will make it easy to use your system, or will add some lesser relevant features to your system. (For example, an automatic mailing module and a thesaurus module in a word processor are nice to have, but most probably, the users can manage without the features these modules provide.) f. It is important to note that a module may have a higher or lower priority level than its sub – module. The hierarchy does not affect the priority level, but it may be beneficial to perform minor restructuring to the hierarchy after priority levels are assigned. 7. Now, you have a set of modules, hierarchically arranged, with 3 priority levels. Structure your work so as to implement the Core modules first, then the Needed modules, followed by the Nice to Have modules. Please note that when a sub – module is Core, but its super – module is of a lesser priority level, it will be necessary to design the sub – module with an interface that otherwise may be passed onto the super – module in later iterations. You have now carried out the HACUFS process and can proceed to list out the most critical system requirements and design towards them first, based on the highest priority modules. Figure C.1 - A flowchart showing the HACUFS process. P a g e | 3 P a g e | 4 P a g e | 5 P a g e | 6 P a g e | 7 Page |8 Page |9 Page |10 Page |11 Page |12 Page |13 Page |14 Page |15 Page |16 Page |18 Page |19 Page |20 Page |22 Page |25 Page |26 Page |27 Page |28 Page |29 Page |30 Page |31 Page |32 Page |33 Page |34 Page |35 Page |36 Page |38 Page |39 Page |40 Page |41 Page |42 Page |43 Page |44 Page |45 Page |46 Page |47 Page |48 Page |49 Page |50 Page |51 Page |52 Page |53 Page |54 Page |55 Page |56 Page |57 Page |58 Page |59 Page |60 Page |61 Page |63 Page |64 Page |65 Page |66 Page |67 Page |68 Page |69 Page |70 Page |71 Page |72 Page |74 Page |75 Page |76 Page |77 Page |78 Page |79