Baltic J. Modern Computing, Vol. 13 (2025), No. 1, 32-66
https://doi.org/10.22364/bjmc.2025.13.1.03

Towards Universal Modeling Language for Neural Networks

Janis BARZDINS, Audris KALNINS, Paulis BARZDINS

Institute of Mathematics and Computer Science,University of Latvia,
Raiņa bulvaris 29, Riga, LV-1459, Latvia

{janis.barzdins, audris.kalnins, paulis.barzdins}@lumii.lv

ORCID 0009-0008-0040-5557, ORCID 0000-0003-0907-7496, ORCID 0009-0006-7186-9776

Abstract. Effective modeling is essential in both system and software development, serving as a key method for facilitating understanding, guiding design, and enabling communication among stakeholders. However, traditional universal system modeling languages like UML and SysML fall short when it comes to neural network modeling, where the structure, training, and deployment processes demand more detailed and specialized representations. Conversely, domain-specific languages like Keras, TensorFlow, PyTorch, and tools like Netron and Deep Learning Studio are too closely tied to specific implementation environments. This creates a significant challenge: the need to develop a universal modeling language specifically for neural networks that is both sufficiently simple (requiring a description of around ten pages) and capable of providing a detailed description of neural networks and their management. The main contribution of this paper is the introduction of such a language, called UM1NN, along with a detailed description and its application demonstrated through two important use cases: describing GPT-2 and defining the fine-tuning of GPT-2 for Question-Answering.

Keywords: Neural Networks, ML Systems, Graphical Modeling, UM1NN language.

1. Introduction

As is well known, modeling plays a critical role in both system and software development, serving as a foundational technique for understanding, design, and stakeholder communication. A good overview of the current state in this field can be found in the paper (Michael et al., 2024) with the noteworthy title "Quo Vadis Modeling?”.

The core of modeling is modeling languages. In general, modeling languages can be broadly classified into universal modeling languages and domain-specific modeling languages. Universal modeling languages are designed to provide a high-level, abstract view of various systems, facilitating stakeholder communication and overall system design. In contrast, domain-specific modeling languages (DSLs) offer detailed and precise tools tailored to the specific needs of particular domains, such as neural networks, ensuring effective implementation and management.

Over the past few decades, several universal system modeling languages have emerged, including UML (OMG, 2017; Rumbaugh et al., 2005), BPMN (Shapiro et al.,

2012; WEB (a)), SysML (Friedenthal et al., 2014; WEB (b)), KerML (OMG, 2024; Pires et al., 2024), conceptual modeling (Lukyanenko et al., 2024), and workflows (Aalst and Hee, 2022). These languages are primarily oriented towards business system modeling. At present, there are no widely recognized direct applications of these languages for neural network modeling. A more detailed discussion of the current state of neural network modeling will be provided in Section 2.

The problem addressed in this paper is the development of a universal modeling language specifically for neural networks that, on the one hand, is sufficiently simple (requiring a description of around ten pages, not the hundreds typically needed for languages like UML) and, on the other hand, allows for the detailed description of neural networks and their management. This language should be understandable to a typical stakeholder and usable as a communication tool among stakeholders.

The main contribution of this paper is the proposal of such a language, named UM1NN (with "1" signifying the initial version of the language). Section 3 provides a detailed description of UM1NN, while Sections 4 and 5 demonstrate its application through two significant use cases: describing GPT-2 (Section 4) and defining the fine-tuning of GPT-2 for Question-Answering (Section 5).

2. Overview of Neural Network Modeling Languages and UM1NN's Place

2.1. Universal Modeling Languages

The traditional universal system modeling languages mentioned in the Introduction are not sufficiently effective for neural network modeling. While these languages provide a variety of diagrams and notations to capture different aspects of system design, they are not optimized for the unique requirements of neural network modeling. Neural networks demand specialized and detailed representations, particularly for defining their structure, training processes, and deployment, which are not adequately supported by these general-purpose modeling languages.

2.1.1. Stereotypes and profiles

One way to make these universal modeling languages (primarily UML) more suitable for detailed neural network modeling is through the use of stereotypes and profiles. Stereotypes allow the customization of UML elements to represent domain-specific concepts. For instance, defining stereotypes like <<ConvolutionLayer>> or <<DenseLayer>>. Profiles are a collection of stereotypes and tagged values that cohesively extend UML for a specific domain. However, the introduction of stereotypes and profiles creates additional challenges:

- Complexity and Maintenance**:** Defining and maintaining stereotypes and profiles can become cumbersome, especially as the complexity of neural network architectures grows.
- Limited Expressiveness**:** Stereotypes and profiles may not capture all the nuances and specific details needed for precise neural network modeling.

- Performance**:** Using stereotypes for detailed neural network modeling can lead to performance issues in modeling tools.

Given these challenges, there is a need for a "hard extension" of UML to effectively model neural networks. A hard extension involves creating a new modeling language or significantly extending an existing one to directly support the requirements of neural network design, training, and deployment.

2.1.2. Our Proposed Solution: UM1NN

Our proposed modeling language, UM1NN, represents a hard extension of UML, specifically UML Activity diagrams, and is tailored for neural network modeling. UM1NN is designed to meet the specific demands of neural networks, providing both simplicity and comprehensive expressiveness:

- **Simplicity**: UM1NN is designed to be easy to understand and use, with a concise syntax and a limited set of core concepts. The entire language specification can be comprehensively described in about ten pages (see Section 3), and the description of a neural network in this language is almost self-explanatory (see Sections 4 and 5).
- **Effective Modeling:** UM1NN enables clear, stakeholder-friendly descriptions of neural networks, covering their architecture, training processes, hyperparameters, and deployment strategies, without aiming for formal specifications.
- **Enhanced Communication:** UM1NN provides a shared language for stakeholders, balancing the need for technical accuracy with accessibility, making it easy for both experts and non-experts to understand and communicate effectively.

The proposed modeling language UM1NN is, in some sense, domain-specific, but it differs from existing DSLs for neural networks in several key points:

- **Versatility**: It is not tied to a specific type of neural network. UM1NN can be used for all types of neural networks.
- **Framework Independence**: It is not tied to any particular neural network implementation framework. UM1NN specifies the operation of neural networks at a higher level of abstraction (similar to how UML is used for business systems).

According to the authors, this level of abstraction for describing neural networks holds an important place, much like UML descriptions for business systems, enabling stakeholders to communicate effectively without diving into the technical details of programming implementations.

To provide further clarity, the following sections will explore traditional domain-specific languages (DSLs) for neural networks and their practical applications.

2.2. DSLs for Direct Execution in Neural Network Frameworks

In this section, we focus on domain-specific languages (DSLs) designed to define, build, and train neural networks directly within popular machine learning frameworks. These languages are primarily intended for coding and execution rather than abstract system

design, offering developers the tools needed to specify and run models in real-time environments.

2.2.1. Program Libraries for Deep Learning Models

When discussing domain-specific languages for neural networks, it's essential to first mention the program libraries available in Python for building deep learning models at the code level. Several well-established libraries, such as **Keras** (Chollet and Watson, 2024; WEB(c)), **TensorFlow** (Chollet and Watson, 2024; Abadi et al., 2016), and **PyTorch** (Chollet and Watson, 2024, Paszke et al, 2019), provide high-level operations for defining and executing deep learning models directly within Python. These libraries include built-in support for typical deep learning model elements such as layers (e.g., Linear layers, Convolutional layers, Transformer layers) with customizable parameters like dimensions, Weight, and Bias tensors.

Additionally, these libraries simplify the process of training and processing models by offering features like dynamic computation graphs and tools for handling complex data flows. They allow the user to create executable models efficiently, abstracting away much of the boilerplate code associated with neural network training and evaluation.

However, these libraries focus primarily on building and executing models at the code level. They do not provide tools for visualizing the overall structure of the model at a higher abstraction level, such as graphical diagrams that might be used to communicate a model's architecture more clearly to non-technical stakeholders or during the design phase.

2.2.2. Graphical Modeling Tools for Deep Learning

An alternative to program libraries are tools that use graphical modeling languages to represent deep learning models. Two notable tools in this area are **Netron** and **Deep Learning Studio (DLS)**. Both tools utilize a graphical workflow language that allows users to define models visually, using a straightforward sequence of actions, combined with facilities for specifying parameters.

2.2.3. Netron Tool

Netron (WEB (d)) is an open-source visualization tool for deep learning models. It supports a wide variety of formats, allowing models developed in environments (Chollet and Watson, 2024) like Keras, TensorFlow, PyTorch, Caffe, and MXNet to be visualized in a standardized format. The model's architecture is represented as a series of rounded rectangles (representing actions, which correspond to layers), with arrows indicating the flow of data between them. Each layer's detailed information—such as input/output tensor shapes, parameters, and operations—is displayed, making it easy to inspect the model’s components.

Netron is primarily used to review and verify the structure of existing neural network models, providing insights to both technical and non-technical users. However, it does not offer features for building new neural network models from scratch. Its main focus is on understanding and validating the architecture of pre-trained models.

2.2.4. Deep Learning Studio (DLS)

Deep Learning Studio (DLS) (WEB (e,f)) is a framework designed for building deep learning models using a graphical interface. DLS provides a drag-and-drop editor for constructing neural networks, represented as workflow diagrams. In this environment, layers of the model are visualized as rounded rectangles (actions), connected by lines that represent the flow of tensors between layers. The shapes of the tensors being passed between layers are also displayed, offering a visual insight into the model's data flow. The editor allows users to select basic layers common to deep learning models, such as convolutional layers, pooling layers, and normalization layers. However, the ability to construct more complex architectures is somewhat limited compared to full-featured coding libraries like TensorFlow or PyTorch.

2.3. DSLs for Model-Driven Engineering (MDE) of Machine Learning-Enabled Systems

While the DSLs discussed in the previous section focus on directly executing machine learning models, Model-Driven Engineering (MDE) (Brambilla et al., 2012) offers a more abstract and systematic approach to system development. MDE-based DSLs describe system architectures, which are then transformed into executable models through automated or semi-automated model transformations. This section reviews the use of domain-specific languages (DSLs) within MDE for Machine Learning-Enabled Systems. These systems typically encompass not only neural networks but also workflows for data preprocessing, feature engineering, and broader software system integration. DSLs in this context enable automated code generation through model transformations, facilitating the development and deployment of comprehensive machine learning pipelines. A comprehensive overview of the current state in this field is provided by papers from Rädler et al. (2024) and Naveed et al. (2024). Below, we highlight key aspects of the current situation in this area.

2.3.1. Ecore-Based DSLs for Machine Learning

The Ecore metamodel from the Eclipse Modeling Framework (Steinberg D., 2009) is a widely used foundation for defining DSLs within MDE environments, offering high-level abstractions for the design and integration of machine learning models. Several tools extend EMF to support machine learning workflows:

- **EMF-IncQuery** (Horvath et al., 2015)**:** This tool facilitates pattern-based queries within models, enabling advanced querying capabilities that simplify the manipulation and analysis of machine learning workflows.
- **Text-based DSLs** (Friese et al., 2008)**:** Custom languages can be developed to specify machine learning models and workflows, providing flexible, domain-specific approaches that help automate processes such as data preprocessing and model training.

2.3.2. MontiAnna: A DSL for Deep Learning Model Specification

MontiAnna (Kusmenko et al., 2019; Kirchhof et al., 2022) is a domain-specific language specifically designed for deep learning. Built on the MontiCore framework, an MDE platform, MontiAnna allows the specification of neural networks and deep learning models. It focuses on integrating machine learning into broader system architectures. MontiAnna offers both graphical and textual notations, providing flexibility for defining neural networks and their associated training processes. Additionally, the framework supports model transformations, enabling the generation of executable code from the defined models. Essentially, MontiAnna bridges the gap between high-level system models and the practical implementation of machine learning components.

2.3.3. AutoML: A DSL for Automating the Deep Learning Lifecycle

AutoML (Moin et al., 2022) presents a novel approach to automating the entire machine learning lifecycle, from model selection to optimization and deployment. The system automates the selection of the most appropriate model architecture (e.g., Fully-Connected Neural Networks (FCNN) or Long Short-Term Memories (LSTM)) using Bayesian Optimization. AutoML simplifies the development of AI-intensive systems by automating model and hyperparameter selection, significantly reducing the need for manual tuning. This DSL is built on the ML-Quadrat framework and includes a user-friendly, web-based interface, making it a valuable tool for streamlining ML development.

2.3.4. Key Conclusions

Traditionally, MDE-based DSLs are formal languages that, at a given level of abstraction, can generate executable models through automated or semi-automated transformations. However, the proposed modeling language, UM1NN, is semi-formal. This raises the question: how does UM1NN relate to MDE-based DSLs? One potential research direction (outlined in the Conclusion section) is the exploration of methods to generate executable models from system descriptions written in semi-formal languages, leveraging Large Language Models (LLMs) such as ChatGPT. In this sense, we see a clear connection between UM1NN and traditional MDE approaches for Machine Learning-Enabled Systems.

3. Description of the UM1NN Modeling Language

UM1NN uses only one type of diagram, called UM1NN Activity Diagrams. The main concept in UM1NN, the **UM1NN System Model**, is defined as a set of thematically related UM1NN Activity Diagrams. An Activity Diagram represents a behavior composed of individual elements called actions. An action represents a single step within an activity. Therefore, defining the UM1NN language entails defining **UM1NN Activity Diagrams** (hereafter referred to simply as **UM1NN diagrams**).

3.1. UM1NNcore

As previously mentioned, in defining UM1NN, we will use UML Activity Diagrams as a foundation. The purpose of this section is to explain exactly what we are adopting from UML Activity Diagrams. Figure 1 shows a typical UML Activity Diagram taken from the official UML documentation (OMG, 2017). This diagram includes all the elements that we (with slight modifications) will directly incorporate into our modeling language UM1NN. (The official UML AD contains many other elements, but we will not use those.) However, this incorporation will come with one change: instead of Object Nodes, we will introduce a new type of node, which we will denote similarly but call a Data Node. An Object Node, according to UML AD semantics, is essentially a container that can hold many objects until they are processed by a subsequent action. In contrast, a Data Node (or Data Element) is analogous to a variable in programming languages, which can hold only one value at any given time, and this value can be used by multiple actions. To specify precisely which actions can modify and which can use the value of a data element, we will introduce dashed arrows as shown in Figure 2. We will retain the UML AD control flow arrows in our modeling language, but they will serve only to represent control flow (Fig.3, as an equivalent notation of Fig.2, we consider only as 'syntactic sugar'). Figure 2 also shows that action symbols can have a "Short Description" section in addition to their name, and data symbols can have a "Type Definition" section along with the name and "Short Description."

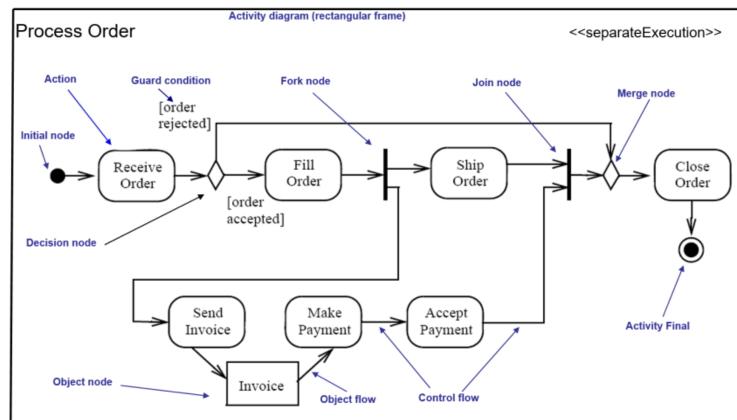

Figure 1. UML Activity Diagram.

To make UM1NN diagram notation more concise, we will agree on several defaults. First, using UML AD terminology, we will employ the <<separate execution>> semantics, meaning that a separate execution of the activity is created for each invocation. Second, when multiple control arrows converge into an action symbol, we will assume a merge symbol by default (and thus will not draw this symbol). Third, we will also omit the decision symbol (diamond); instead, we will agree that if an action symbol has more than one outgoing control arrow, each of these arrows must have an attached guard condition.

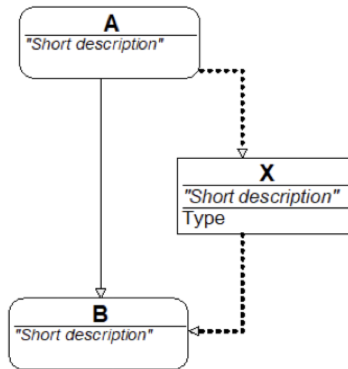

Figure 2. Dashed arrows.

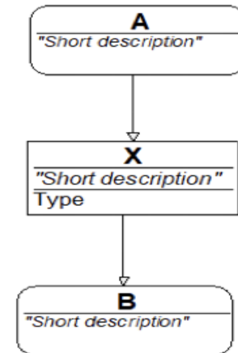

Figure 3. Equivalent representation of Fig. 2.

The resulting modeling language will be called UM1NNcore, and it will contain the following graphical elements:

- Activity diagram symbol (optional)
- Action symbol
- Data element symbol
- Control flow arrow (solid arrow)
- Control flow arrow with guard condition
- Data flow arrow (dashed arrow)
- Fork and Join symbols
- Start and End symbols
- (Decision and Merge symbols by default)

The UML Activity diagram shown in Figure 1 can be equivalently represented as an UM1NNcore diagram, as shown in Figure 4.

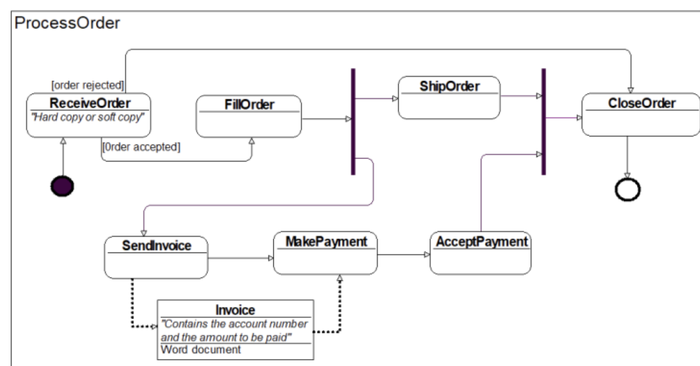

Figure 4. UM1NN Activity Diagram.

The following sections will focus on significantly extending UM1NNcore with new features tailored for neural network modeling applications, which could not be directly "borrowed" from UML AD.

3.2. Data Elements

In UM1NN Activity Diagrams, data elements play a crucial role in representing the information that flows through and is manipulated by the neural network. In UM1NN, data elements are classified into two main categories: **Flow Data** and **Internal Parameters**, providing a structured approach to represent the different types of information in the neural network.

3.2.1. Flow Data

Flow data in neural networks represent dynamic information traveling through the network (e.g., input data and intermediate results). Flow data elements (see elements X and Y in Figure 5) are represented by rectangular frames with **black borders** and **white background** and can appear multiple times within a single UM1NN Activity Diagram. Each occurrence of a flow data element with the same name operates independently. This independence allows for flexibility, as actions in the diagram can use different data elements with the same name without causing conflicts.

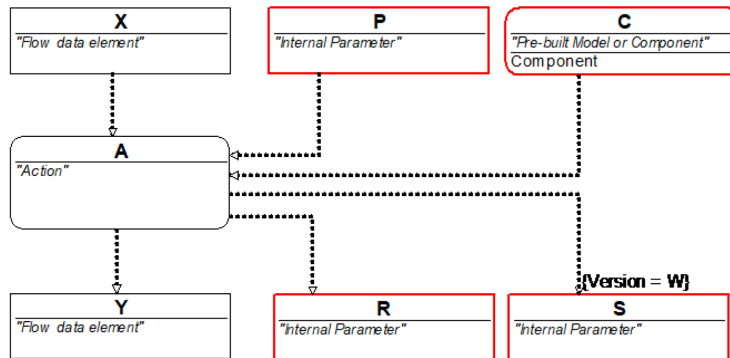

Figure 5. Action's dashed arrows.

3.2.2. Internal Parameters

Internal parameters are essential to the internal workings of the neural network. These include components like weight matrices and biases, which are adjusted during training to optimize the network's performance. In UM1NN diagrams, internal parameters (see elements P, R, S in Figure 5) are represented as rectangles with **bold red borders** to distinguish them from flow data elements. Internal parameters differ from flow data elements in that they maintain a consistent value across all diagrams in the UM1NN System Model. This means that if multiple internal parameters share the same name, they are automatically synchronized—any change made to one element is propagated to

all others with the same name. This ensures value consistency throughout the system model. Additionally, internal parameters can also have version labels for their values.

Internal parameters are typically used to represent model parameters (e.g., weights and biases) and hyperparameters (e.g., learning rate, batch size) and are depicted with bold red borders and colored backgrounds: **orange for model parameters** and **green for hyperparameters** (see Use Cases figures).

By integrating these concepts, UM1NN provides a structured way to represent and differentiate between the different types of data that flow through and govern the behavior of neural networks.

3.3. Pre-built Components

UM1NN Activity Diagrams can also include nodes that represent **pre-built components**. These nodes are graphically depicted as **rounded rectangles with bold red borders** and labeled with type = "Component" (see element C in Figure 5). These elements can be used in two ways:

- **Action using the pre-built component**: This refers to actions that utilize the pre-built component. A **dashed arrow** connects the pre-built component node to the action that makes use of it (see element C in Figure 5).
- **Direct use**: The pre-built component can also be directly used as an action, represented as shown in Figure 6 (further details will be discussed in the next section).

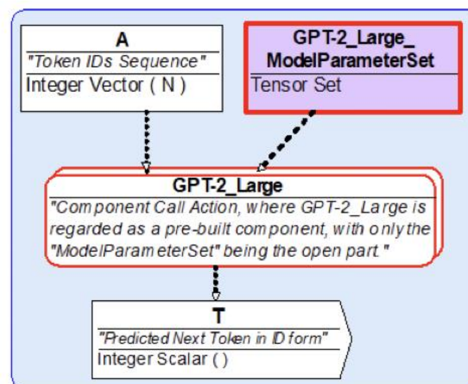

Figure 6. Component Call Action.

3.4. Actions

In UM1NN, **actions** represent individual steps or operations within an activity. These actions can be classified into four categories based on their function and scope: **Formal Actions**, **Informal Actions**, **Subdiagram Call Actions**, and **Component Call Actions**.

3.4.1. Formal Actions

Formal actions are well-defined operations with specific inputs and outputs. These actions typically involve mathematical operations essential to neural networks, such as tensor operations and matrix manipulations.

- **Examples**: Matrix multiplication, convolution, pooling, normalization.
- **Graphical Notation**: Formal actions are depicted as **rounded rectangles with a yellow background** (see Fig. 7).

3.4.2. Informal Actions

Informal actions represent higher-level processes that might encompass multiple steps or operations. Unlike formal actions, informal actions are more abstract, and their inputs and outputs may not be as clearly defined. These actions often describe broader processes in the neural network.

- **Examples**: Backpropagation, gradient descent, preparation of input data.
- **Graphical Notation**: Informal actions are represented as **rounded rectangles with a white background** (see Fig. 8).

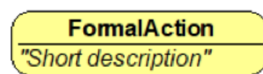

Figure 7. Formal Action.

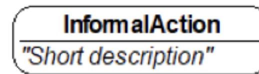

Figure 8. Informal Action.

The boundary between **Formal Actions** and **Informal Actions** can sometimes be fluid, depending on the stakeholder's level of expertise. For example, an action like backpropagation might be considered "formal" to someone with deep knowledge of neural networks, but "informal" to someone less familiar with the process. One useful criterion to distinguish between these two types of actions is whether an advanced model like **ChatGPT-4** can fully understand the action based solely on its **name** and **short description** (recall that action symbols in UM1NN diagrams contain both a "Name" and a "Short Description" section). If the action can be explained in detail by ChatGPT-4 based on this information alone, it is typically considered **formal**. In contrast, if the action is more abstract and requires broader contextual understanding, it leans towards being **informal**.

3.4.3. Subdiagram Call Actions

Subdiagram call actions refer to another diagram that is logically part of the current diagram. These actions allow for modularity by breaking down complex processes into smaller, manageable diagrams. Input and output parameters are clearly indicated, ensuring the proper flow of data between the main diagram and the subdiagram.

- **Graphical Notation**: Subdiagram call actions are represented as **rounded rectangles with a double border** and display both **input** and **output parameters**, as well as the corresponding **internal parameters** (see Fig. 9).

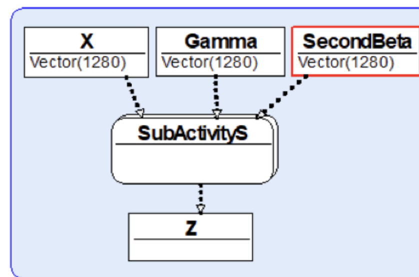

Figure 9. Subdiagram Call Action.

3.4.4. Component Call Actions

Component call actions represent the use of pre-built, sophisticated models or components. These actions are abstract in nature, meaning the internal workings of the component are hidden from the diagram, focusing instead on well-defined inputs, outputs, and exposed internal parameters. Component call actions are often used when applying pre-trained models or neural network architectures.

- **Examples**: Fine-tuning a pre-trained GPT-2 model, using ResNet for feature extraction, employing BERT for text encoding.
- **Graphical Notation**: Component call actions are depicted as **rounded rectangles with bold red double borders**. They also include **input**, **output**, and any exposed **internal parameters** (see Fig. 6).

3.4.5. One More Remark

Consider **Figure 5**, which illustrates a possible usage of dashed arrows in relation to **Action A**:

- **X ---> A** means that A uses the value of the flow element X.
- **A ---> Y** means that A modifies the value of the flow element Y.
- **P ---> A** means that A uses the value of the parameter P.
- **C ---> A** means that action A utilizes the component C.
- **A ---> R** means that A modifies the value of the parameter R.
- **A --{version=W}--> S** means that A modifies the value of the parameter S and assigns it the version label "W" (recall that internal parameter values can also be assigned version names).

This example demonstrates how UM1NN diagrams visually depict interactions between actions, flow elements, parameters, and components, ensuring clear tracking of value modifications and usage.

3.5. Dashed and Solid Arrows Between Flow Data Elements

Let us introduce a new type of operational mechanism between flow data elements (nodes) – the so-called dashed arrow mechanism (Fig. 10). If there is a dashed arrow

from data node X to another data node Y of the same type, it means that if any action at any point in time changes the value of node X (remember, X is formally a variable), this new value is immediately transferred to node Y. This mechanism allows us to freely supplement the existing activity diagram with new identical data nodes, but with different names. This, in turn, gives us the ability to graphically define both the graphical call mechanism and the diagram detailing mechanism (see the next section).

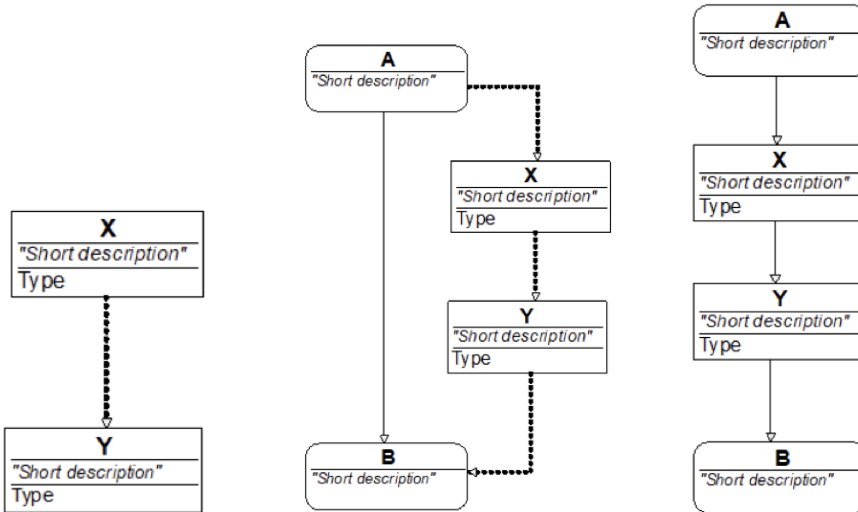

Figure 10. Dashed arrow. **Figure 11.** Diagram fragment. **Figure 12.** Equivalent representation of Fig. 11.

In many cases, it is natural to combine the control flow mechanism (control arrows, which are solid arrows) with the data flow mechanism (dashed arrows). Let's assume we have a fragment of an activity diagram corresponding to Fig. 2. Let's agree that it can equivalently be depicted as in Fig. 3. In this drawing, it is seen that the control arrow simultaneously serves as a data transfer arrow. At this level, it is just "syntactic sugar." However, let's add additional semantics to this construction: we will consider that the control token does not go "directly" to action B, but first to data node X, and from there further to action B. Let's combine this mechanism with the previously mentioned "dashed arrow" mechanism. As a result, the diagram fragment shown in Fig. 11 can naturally be depicted as in Fig. 12. In this case, the control token from action A first goes to data element X, then from data element X to data element Y, and finally from data element Y to action B. Now let's note that the control arrow from data element X to data element Y simultaneously serves the role of the "dashed arrow" – it instantly transfers the value of element X to element Y. As a result, our language will have two types of arrows between flow data elements: dashed and solid. Both types of arrows ensure the transfer of element values, but the solid arrow additionally ensures the transfer of control tokens.

Additionally, we will allow dashed arrows from internal parameters to flow data elements, with the same semantics as in the case of flow elements.

We will use these constructions in the next section for fragmentation call mechanism.

3.6. Graphical Call

First, let us note that a UM1NN System Model typically contains a single diagram called the **Main Diagram**, which serves as the central point of the model. This diagram, whose name starts with the word "Main," uses the call mechanism to invoke other diagrams within the model, known as **subdiagrams**, or predefined components.

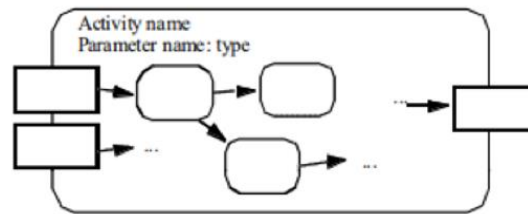

Figure 13. UML AD with parameters.

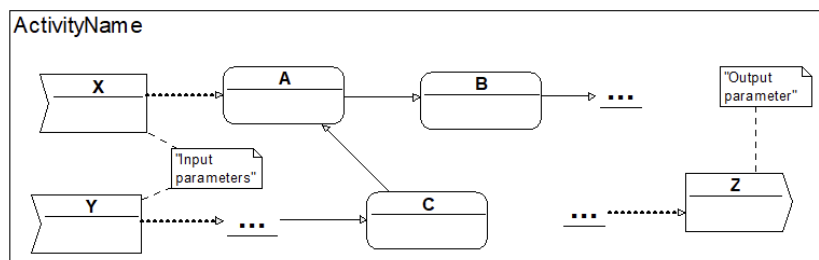

Figure 14. UM1NN AD with input and output parameters.

In this section, we will examine the UM1NN subdiagram call mechanism. First, let's introduce the concept of an "Activity Diagram with Input and Output Parameters" (so that we have a use for the call mechanism). In UML AD, parameters are represented as shown in Fig. 13. However, this representation and its usage in graphical call notation present several inconveniences (in practice, we often need to switch from graphical to textual form). In the UM1NN language, we will take a different approach. First, we will introduce a new notation for input and output parameters: they will be depicted as flow data nodes (anywhere in the diagram), with slightly modified graphical symbols (**InArrows**, **OutArrows**) as shown in Fig. 14 (input parameters X and Y, output parameter Z). In fact, these symbols are introduced only for better visual clarity, as we will consistently follow one rule regarding parameters: input parameter nodes should not have incoming arrows, and output parameter nodes should not have outgoing arrows. Therefore, in the graphical notation of a call action (see Fig. 15), specific symbols for input and output nodes can be replaced with the standard data node symbols.

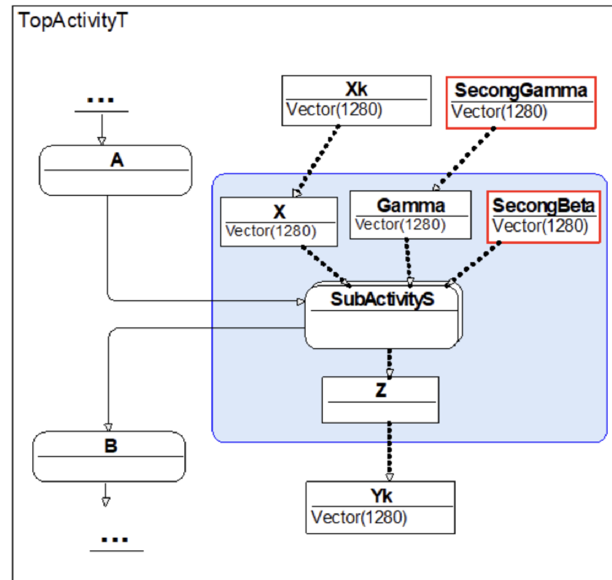

Figure 15. Subdiagram Call Action in the context of a Parent Diagram.

Now the most important part: the values of input parameters at the call action can be defined with the help of incoming dashed *arrows* from the corresponding data elements (nodes) that represent these values. Similarly, for output parameters, the dashed arrows will be outgoing (see Fig.15). In this notation, a call is represented not by the "fork" symbol as in UML AD, but as an action symbol with a double frame, from which dashed arrows indicate the output parameters of the called activity, and incoming dashed arrows indicate the input parameters. The name of the action with the double frame must match the name of the activity diagram we want to call. This activity diagram (which we want to call) must be from the same UM1NN system model in which this call occurs. We assume here that the activity (in the form of an action diagram) called through the call operation contains exactly one start symbol and exactly one end symbol. Under these conditions, the semantics of the call operation are clear.

Next, consider the case when the called diagram is only a fragment of a larger activity diagram and therefore might not contain start and end symbols directly. Recall the solid arrow mechanism introduced in the previous section, which serves both as a control flow and data element value transfer mechanism. This means that if we consider a large diagram W (Fig. 16) that logically contains a fragment S, we can depict this fragment S as a separate diagram (Fig. 17), where X can be considered as the start symbol and also as the input parameter symbol, and Y can be considered as the end symbol and also as the output parameter symbol. As a result, we can represent the original diagram W in a much more compact form using the new type of call operation (Fig. 18). This is a very convenient mechanism to logically divide large diagrams into manageable fragments (widely used in the section on GPT).

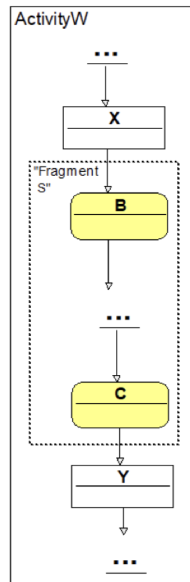

Figure 16.
"Large" diagram.

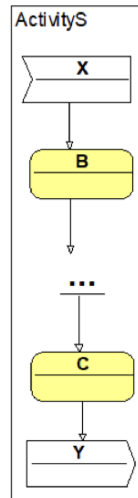

Figure 17.
Fragment as separate diagram.

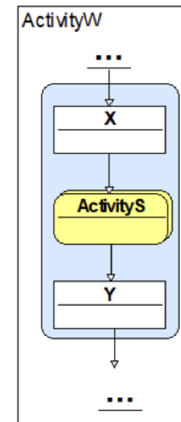

Figure 18.
"Large" diagram in more compact form.

3.7. Data element types and operations

For data elements, end-users can define any specific data types and their corresponding operations. However, it is up to the end-user to explain these to potential stakeholders. For applications in the field of neural networks, our language UM1NN will introduce the most crucial data type at its core - **tensors.**

More precisely, UM1NN proposes the following syntax for defining data element types, referred to as tensor types:

<dtype> Tensor <shape> (with "Float" as the default <dtype>)

If we specify a particular <dtype> (for example, Float) and a specific shape (for example, (1024, 512)), we then define a specific data element type:

Float Tensor (1024, 512)

whose values will be matrices of Float elements with 1024 rows and 512 columns.

Other examples include:

- Tensor () - alternatively called Scalar
- Tensor (25) - alternatively called Vector (25) (semantics: a vector of length 25, with element indexing starting at 0)
- Tensor (3, 5) - alternatively referred to as Matrix (3, 5) or 3 x 5 (semantics: a matrix with 3 rows and 5 columns, with row and column indexing starting at 0)

- Tensor (2, 3, 5) - tensors with shape (2, 3, 5) (element indexing starts at 0 for any dimension).

We also use the notation simply

Vector (without specifying the shape) – this means a vector of any length.

In these examples, we used the default dtype = float. If, for instance, we wanted to consider vectors of length 25 with Integer type elements, we would write

Integer Vector (25). Similarly, Integer Scalar, Integer Matrix (3, 5), etc.

Regarding operations with tensor-type data elements, PyTorch offers about 100 operations. We will list only the main ones here, in a slightly simplified syntax (noting that there is no uniform official syntax for tensor operations).

Let c be a scalar of type float, u and v be vectors of equal length with the same element type, specifically float, and A and B be matrices with identical shape and element type, also float. Basic operations:

- **u + v** - This represents the vector addition of u and v. Each corresponding element of the vectors is added together.
- **u - v** - This is the vector subtraction of u from v. Each corresponding element of u is subtracted from the corresponding element in v.
- **A + B** - This indicates the matrix addition of A and B. Each corresponding element of the matrices is added together.
- **c * v** - This refers to the scalar multiplication of the vector v by the scalar c. Each element of v is multiplied by c.
- **c * A** - This represents the scalar multiplication of the matrix A by the scalar c. Each element of A is multiplied by c.
- **A + v** - Assuming the number of columns in A equals the length of the vector v, this expression would typically denote broadcasting the vector v across each row of A and then adding v to each row of A. In PyTorch, this is handled automatically if v is shaped correctly.
- **A * B** - This is the element-wise multiplication of A and B. Each element in A is multiplied by the corresponding element in B. This operation is known as Hadamard product.
- **concat(A, B)** - This function concatenates matrix B to the right of matrix A, forming a new matrix whose width is the sum of the widths of A and B, with the same number of rows. In PyTorch, this is performed using **torch.cat([A, B], dim=1)**. The **concat** operation can also be applied to three or more matrices, for example, **concat(A, B, C, ...)**.
- **A @ B** - If the number of columns in matrix A equals the number of rows in matrix B, then $A@B$ represents the traditional matrix multiplication of A and B (also called *dot product*). In PyTorch, this operation is denoted by the @ operator or by using **torch.matmul(A, B)**.

- **v @ A** - If v is a vector of length N and A is a matrix of shape $N\times M$, then $v@A$ represents the matrix multiplication of the vector v with the matrix A (also called *dot product*). This operation results in a new vector of length M. In PyTorch, this can be performed using the @ operator or **torch.matmul(v, A).**
- **u.dot(v)**: This represents the dot product of vectors u and v. It is the sum of the products of the corresponding elements of the vectors. In PyTorch, this can be computed using **torch.dot(u, v)**.
- **A^T -** This denotes the transpose of the matrix A. In matrix A, rows and columns are swapped, and in PyTorch, this is simply accessed with **A.T.**

These operations, extensively used in Sections 4 and 5, are crucial in our modeling language. They can be selected as predefined by end-users from a broader list of operations, which must be accompanied by clear explanations of each operation for stakeholders in the model documentation.

3.8. Specific Data Type: Tensor Set

Let's start with an example of a Tensor Set type data element shown in Fig. 19. It represents the entire set of weight and bias tensors used in GPT-2_Large. Similarly, Fig. 20 depicts another Tensor Set type element - the GPT-2_Large Hyperparameter Set. In this case, it is not a set of arbitrary tensors but rather a set of scalar tensors.

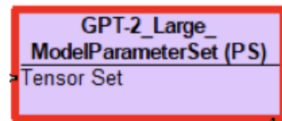

Figure 19. Tensor Set example.

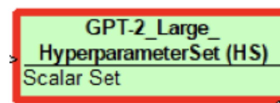

Figure 20. Another Tensor Set example.

These examples demonstrate the typical use of a Tensor Set for specifying specific collections of inner parameters in the UM1NN system model. Tensors have the concept of shape. Let's also introduce the concept of shape for a Tensor Set: it will be a set of pairs where each pair's first element is the tensor name, and the second element is the tensor's shape.

A tensor with a specific (fixed) shape has the concept of Value. Similarly, we can define the Value concept for a Tensor Set with a fixed shape as the values of the tensors it contains. Thus, when describing actions, we can use the term "assign values to a tensor set type element." For example, setting the "GPT-2_Large Model Parameter Set" involves assigning the pre-trained tensor values (weights and biases) to this set.

Let us also agree that in UM1NN diagrams, to indicate that a specific tensor (for example, "EmbeddingMatrix") belongs to a Tensor Set type element (for example, "GPT-2_Large Model Parameter Set"), we will use the same background color for both elements (in this case, orange).

3.9. Additional note on dashed arrows between data elements.

As mentioned in Section 3.5, if X and Y are nodes of the same type (for example, Integer Vector (1240)), a dashed arrow from node X to node Y means that any change in the value of node X is instantly transferred to node Y (i.e., X and Y, as variables, have the same value at any given moment). Since in Section 3.5 (see Fig. 10, Fig. 11, Fig. 12) we combined the dashed arrow with the control arrow (i.e., transformed it into a solid arrow), we will also apply the aforementioned automatic data transfer mechanism to solid arrows between data nodes.

For tensor-type data nodes, we will slightly extend this automatic value transfer mechanism: When a matrix A with m rows and n columns (i.e., a tensor A(m,n)) and a vector V of length n (i.e., a tensor $V(n)$) are connected by a dashed or solid arrow (represented graphically), the notation $V \to A(5,:)$ specifies that the values of vector V replace the values of the 5th row of matrix A. This action is symbolized by an arrow pointing from V to A, indicating the direction of data transfer. Similarly, the reverse operation is denoted as $(5,:) \to V$, where the 5th row of matrix A is used to update the values of vector V. See Fig. 21 and Fig. 22 for applications of these notations. This notation enables clear and concise representation of data flow between specific elements of matrices and vectors within the model.

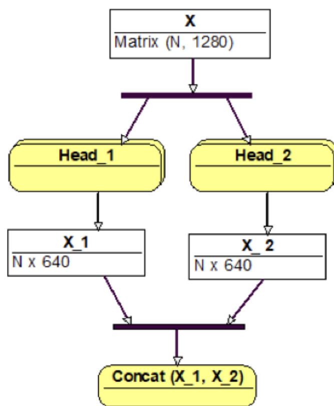

Figure 21. Concurrency example.

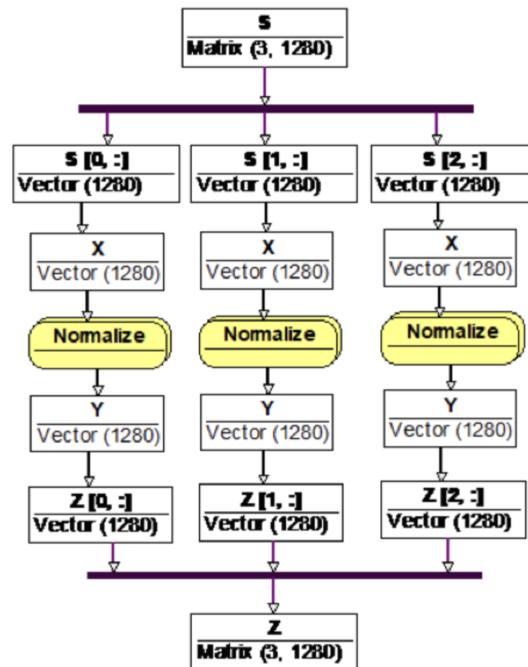

Figure 22. Another concurrency example.

3.10. Representing and Managing Concurrency

By **concurrent activities**, we mean activities that can execute independently and potentially in parallel with each other. In UML Activity Diagrams (AD), this is achieved through the use of fork and join nodes, which enable parallel execution and synchronization of these activities. In our language, we also adopt the fork-join mechanism, but with an additional requirement that each fork node corresponds to exactly one join node, and between them are independent concurrent activities (Fig. 21, Fig. 22, Fig. 23).

If the number of concurrent activities between the fork and join nodes is small, it is easy to represent them in our graphical modeling language. However, in neural network scenarios, it is typical to have a large number of concurrent activities, making it impractical to draw all of them directly. These activities often differ only slightly, specifically by the value of an integer parameter that appears in the definitions of actions and/or data elements. The aim of this section is to propose a special type of graphical loops (called **concurrency graphical loops**) for defining such uniform concurrent activities.

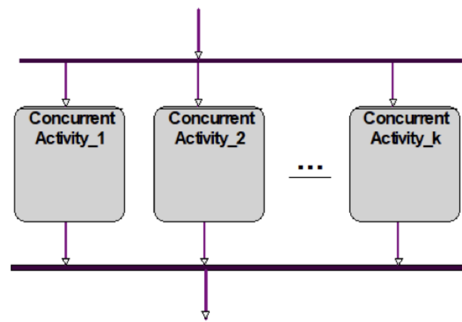

Figure 23. Concurrent activities.

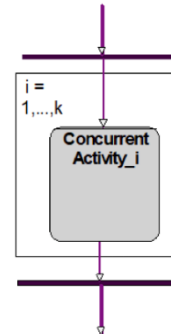

Figure 24. Concurrency loop.

First, let's agree that integer-type variable parameters will be denoted by lowercase Latin letters, for example, *i*, *j*, etc. Next, if we want to indicate that a letter, such as *i*, should be interpreted as a loop parameter in a syntactic loop, we will represent it in the form *_i<space>, _i<underscore>, _i+1<space>, or _i+1<underscore>*. The notation of Fig. 23 using this type of loop is shown in Fig. 24. If the range of the parameter *i* is small, the cyclic construction can be omitted. However, in neural network models, the range of the aforementioned parameter values is usually very large - several hundred or thousand, and the use of these loops is essential. The GPT model discussed in Section 4 relies heavily on the extensive use of concurrency graphical loops.

By the way, our concurrency graphical loops share similarities with UML graphical loops that use expansion regions (OMG, 2017: WEB (j)), but offer different capabilities.

Finally, one more clarification about data flow semantics in the case of fork and join nodes. First, we syntactically exclude the case where dashed arrows enter or exit fork or join nodes. However, it is permissible for solid arrows to enter or exit fork and join nodes, which can connect not only to action nodes but also to data nodes. In such cases, these arrows serve both control transfer and data transfer functions. Consider Fig. 21. In

this diagram, a solid arrow goes from data node X to the fork node, and from there to actions Head1 and Head2. Since this arrow originates from data node X, it means that it carries the value of node X with it. This, in turn, implies that actions Head1 and Head2 can directly use node X in defining their operations. A similar situation occurs with the join node—solid arrows "carry" the values of data elements X1 and X2 through this node. A more complex situation is shown in Fig. 22. Here, the row transfer mechanism for matrices mentioned in previous Section is used. Additionally, it demonstrates our agreement on flow data elements from Section 3.2—each new occurrence of a data element, even with the same name, is treated as a separate data element. Therefore, we repeat the data element named X three times (not X1, X2, X3), and similarly, Y three times (not Y1, Y2, Y3). This allows us to make the call mechanism significantly more versatile. For example, in this case, the Normalize activity is defined only for the pair (X, Y), but Normalize (X1, Y1), Normalize (X2, Y2), Normalize (X3, Y3) are obtained through the graphical parameter transfer mechanism (data and control transfer arrows S[0,:]->X, S[1,:]->X, S[2,:]->X).

Thus, the main constructions of the UM1NN language are outlined. The next section will cover a few more technical additional features.

3.11. Additional Features

In modeling languages, comments play an important role. In UM1NN, two types of comments and their respective notations are provided:

- **Element Comments**: Similar to UML, any diagram element can have an attached comment (rectangle with a folded corner), as shown in Fig.14.
- **Fragment Comments**: UM1NN also allows adding comments to an entire diagram fragment. This is done by drawing a dashed rectangle around the fragment and including explanatory text inside the rectangle using quotation marks (" "). This text typically serves as a meaningful name at a higher level of abstraction, explaining the defined content of the fragment (see, for example, Fig.16).

Additionally, neural network models often consist of many similar fragments. Therefore, in some cases, it is useful to use **ellipses** in the usual intuitive sense. In our proposed modeling language, such use of ellipses is allowed (see, for example, Fig. 16, 17, 18, 23), provided that the potential reader (stakeholders) can understand or infer their meaning in the given context.

3.12. Concluding Remark

The proposed neural network graphical modeling language provides a comprehensive framework for describing the detailed structure and operations of neural networks, including training processes and tensor-based data manipulations. In the following sections, we will utilize this modeling language to represent two significant use cases: the GPT model and the Fine-Tuning model for Question Answering based on GPT. To facilitate understanding, each section will be accompanied by a natural language explanation of the key concepts behind GPT-2 and Fine-Tuning, generated with the assistance of ChatGPT-4.

4. Use Case 1: GPT-2 Large Description in UM1NN Language

GPT-2_Large (Farris et al., 2024; Alammar, 2019) is a powerful language model designed to generate coherent and contextually relevant text from input data. Its functionality relies on multiple interconnected components, each handling distinct phases of the text generation process. This structure is represented in the UM1NN modeling language, spanning from Fig. GPT-1 to Fig. GPT-9, where each diagram illustrates a specific aspect of the model's workflow. Below is a brief overview of these models.

4.1. Main Model of GPT-2_Large (Fig. GPT-1)

The **Main Model** manages the overall context processing and data flow. It receives a sequence of token IDs as input and produces the predicted next token in ID form. Key stages include:

- **Embedding and Positioning**: Converts token IDs into embeddings and adds positional information to prepare the input for the Transformer.
- **Call Transformer Model**: Invokes the Transformer to generate a probability distribution over possible next tokens.
- **Select Predicted Token**: Chooses the token with the highest probability from the Transformer output.

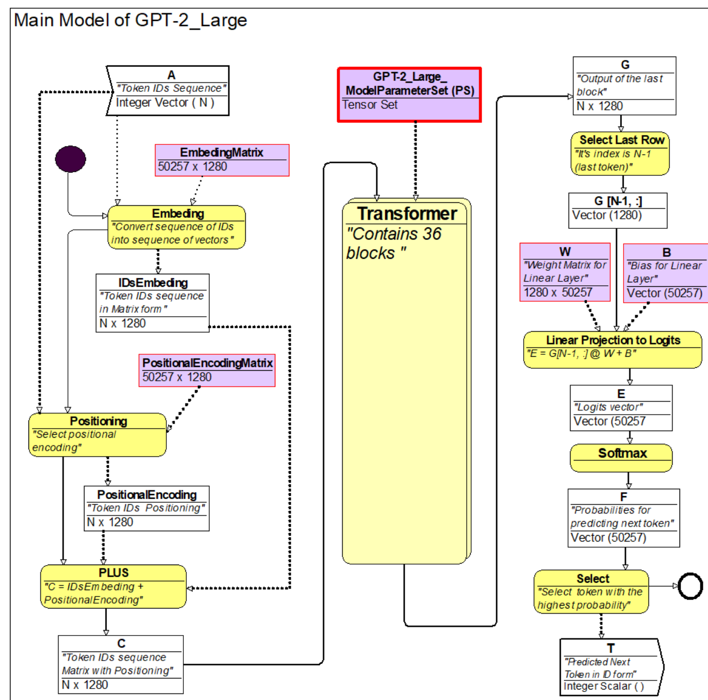

Figure GPT-1.

4.2. Transformer Model (Fig. GPT-2)

The **Transformer Model** (Turner, 2024; Alammar, 2019) is the core component, consisting of 36 sequential blocks. Each block performs:

- **Multi-Head Attention**: Allows the model to focus on different parts of the input.
- **First Add & Normalize**: Stabilizes the attention output for the next step.
- **Feed-Forward Processing**: Transforms the data further.
- **Second Add & Normalize**: Finalizes the block before passing data to the next one.

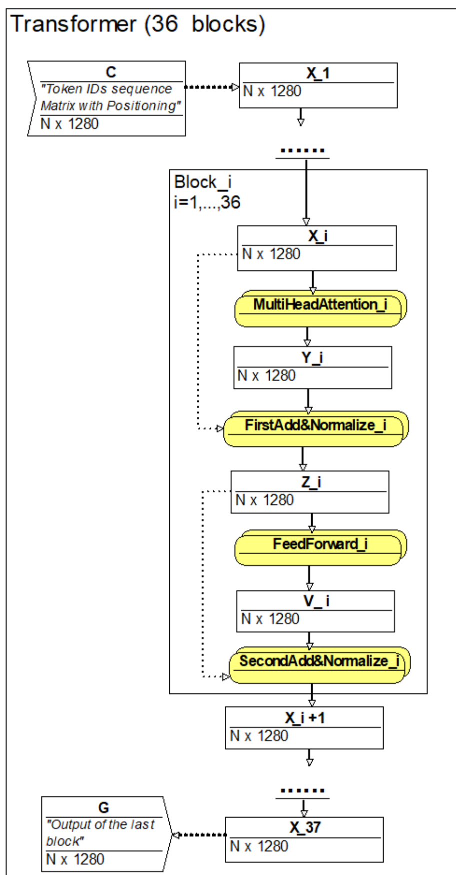

Figure GPT-2.

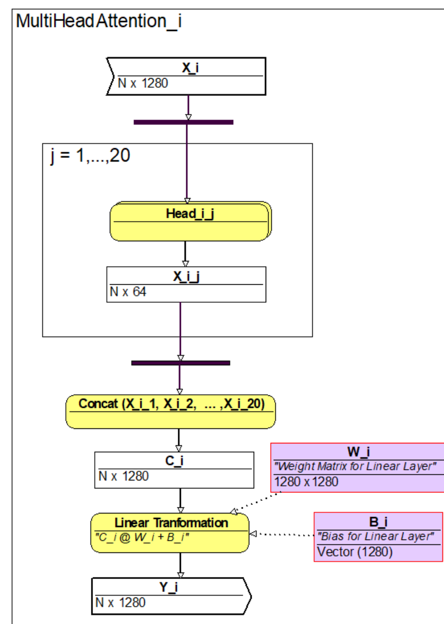

Figure GPT-3.

4.3. Multi-Head Attention Model (Fig. GPT-3)

This model handles the attention mechanism (Vaswany et al., 2023), enabling the Transformer to focus on multiple parts of the input sequence simultaneously. It includes:

- **Individual Head Models**: Each head processes a segment of the input data.

4.4. Individual Head Model (Fig. GPT-4)

Each attention head computes attention scores for a subset of the input sequence:

- **Calculate Head Output**: Produces the attention output for this head.

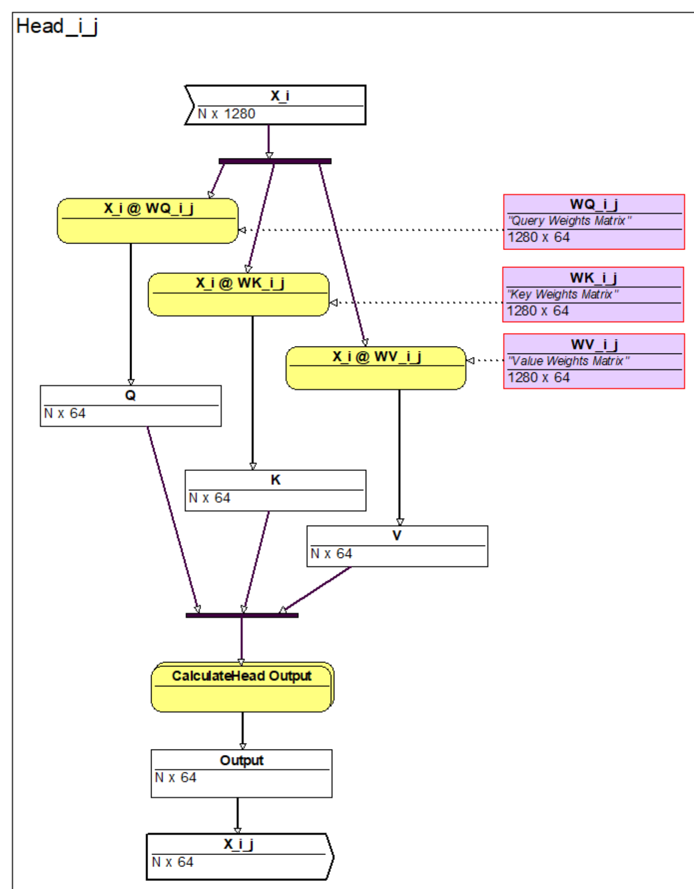

Figure GPT-4.

4.5. Calculate Head Output Model (Fig. GPT-5)

The core mechanism for attention is based on the Scaled Dot-Product Attention introduced by Vaswani et al. (2017; 2023) in their seminal paper *"Attention is All You Need."* It consists of the following steps:

- **Dot-Product Calculation**: Computes the dot product between the query and key matrices to determine the relevance scores for each query-key pair.
- **Softmax Normalization**: The attention scores are normalized across each row (for each query) using the softmax function, converting them into a probability distribution that represents the attention weights.
- **Output Generation**: The normalized attention scores are used to weight the value vectors. This weighted sum of values produces the output for the specific attention head.

The outputs from all heads are then aggregated to form a more comprehensive understanding of the input sequence.

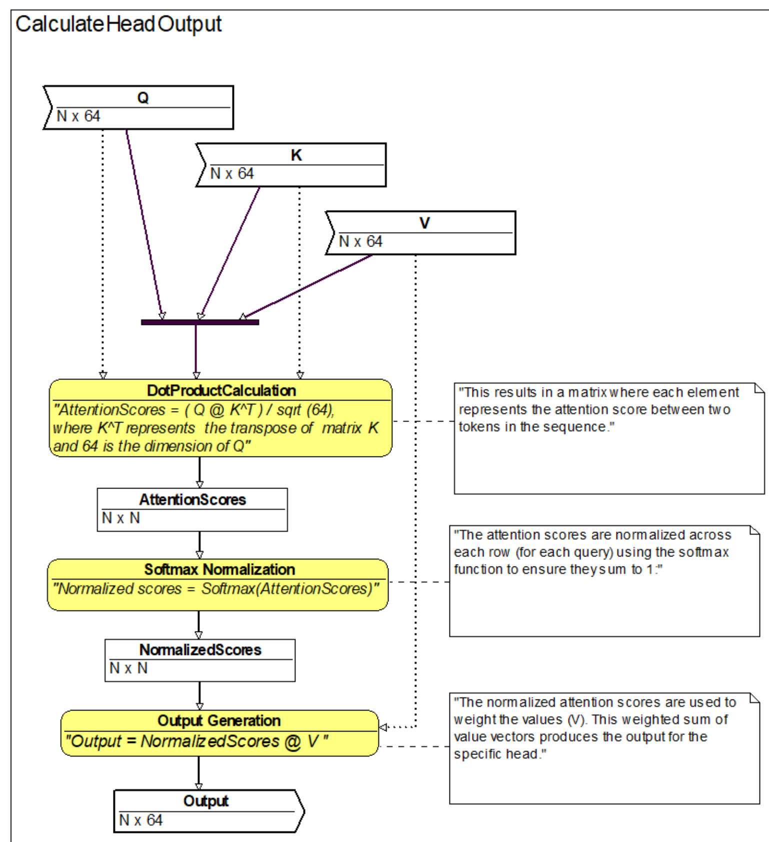

Figure GPT-5.

4.6. First Add & Normalize (Fig. GPT-6)

This model stabilizes the data after the attention step by:

- **Residual Connection**: Adds the original input to the attention output.
- **Normalization (Fig. GPT-9)**: Normalizes the result to improve stability before passing it to the feed-forward layer.

4.7. Feed-Forward Model (Fig. GPT-7)

A fully connected network that further processes the normalized data. It includes:

- **Linear Transformation**: Adds non-linearity with an activation function (GELU).
- **Second Linear Transformation**: Projects the data back to the original dimension, preparing it for the next step.

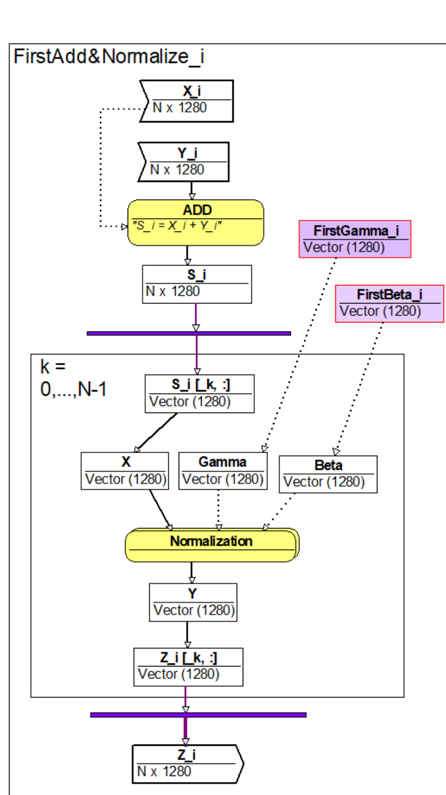

Figure GPT-6.

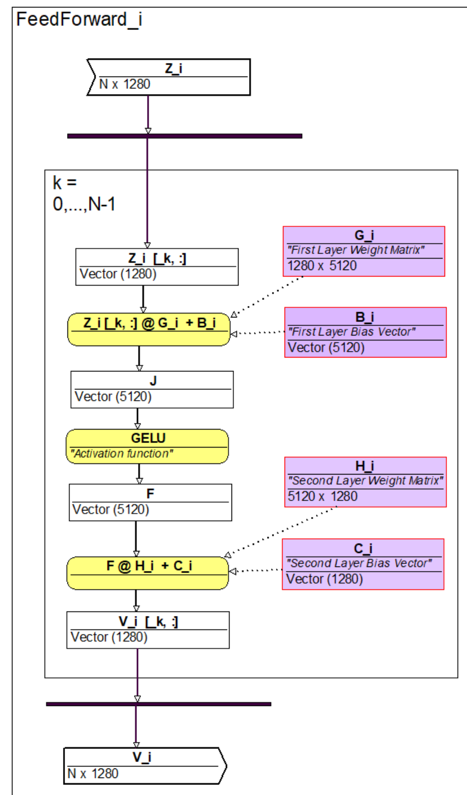

Figure GPT-7.

4.8. Second Add & Normalize (Fig. GPT-8)

Similar to the first, this model stabilizes the output of the feed-forward network. It includes:

- **Residual Connection**: Adds the original input to the feed-forward output to retain information.
- **Normalization (Fig. GPT-9)**: Ensures consistent scaling before passing the data to the next Transformer block or final output.

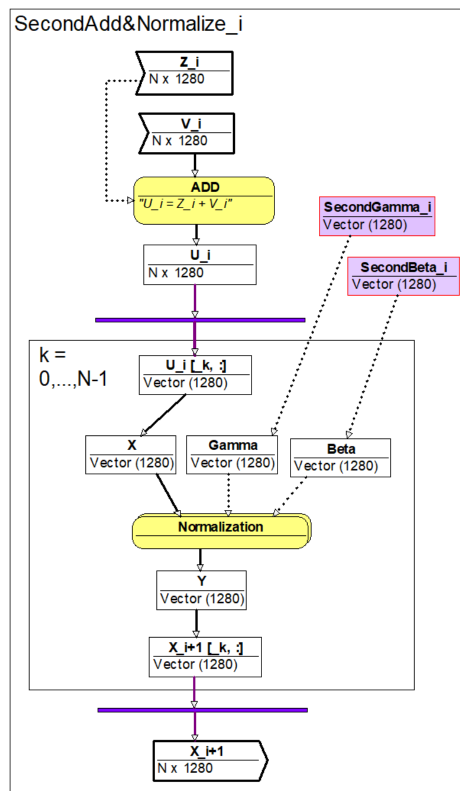

Figure GPT-8

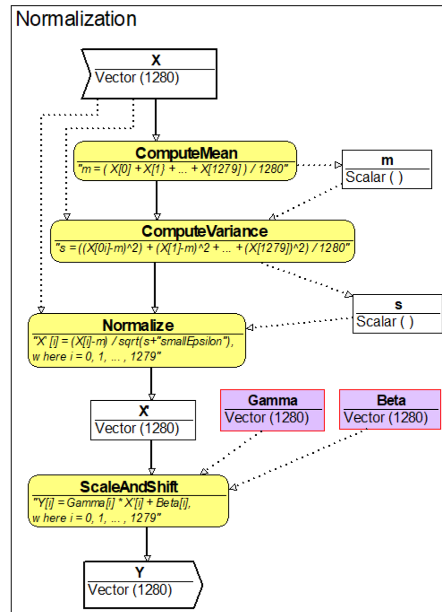

Figure GPT-9

5. Use_Case 2: Fine-Tuning GPT-2 for Question-Answering

Fine-tuning GPT-2 for question-answering tasks (Radford et al., 2019) involves adapting the pre-trained GPT-2 model to generate accurate answers for a specific dataset or task. This fine-tuning process includes several stages: loading the model, processing data in batches, calculating gradients, updating model parameters, and generating answers. This process is represented in the UM1NN modeling language in **Figures FT-1 through FT-4**, where each diagram (model) is responsible for a distinct phase of the workflow. Below is a brief explanation of these models in natural language.

5.1. Main Model of Fine-Tuning GPT-2 for Question-Answering (Fig. FT-1)

The **Main Model of Fine-Tuning GPT-2 for Question-Answering** governs the entire fine-tuning process. This high-level model manages the key stages, ensuring efficient data flow and invoking other submodels to handle specific tasks. It includes the following stages:

- **Load GPT-2 Model**: Initializes and loads the pre-trained GPT-2 model.
- **Collect Data**: Gathers task-specific question-answer data for fine-tuning.
- **Tokenize Data**: Converts the text data (questions and answers) into tokens that GPT-2 can process.
- **Batch Processing**: Invokes the **Batch Processing Model** to handle individual batches of tokenized data.

- **Evaluation**: Evaluates the model's performance using a validation dataset after each epoch to assess improvements or determine when to stop the fine-tuning process.

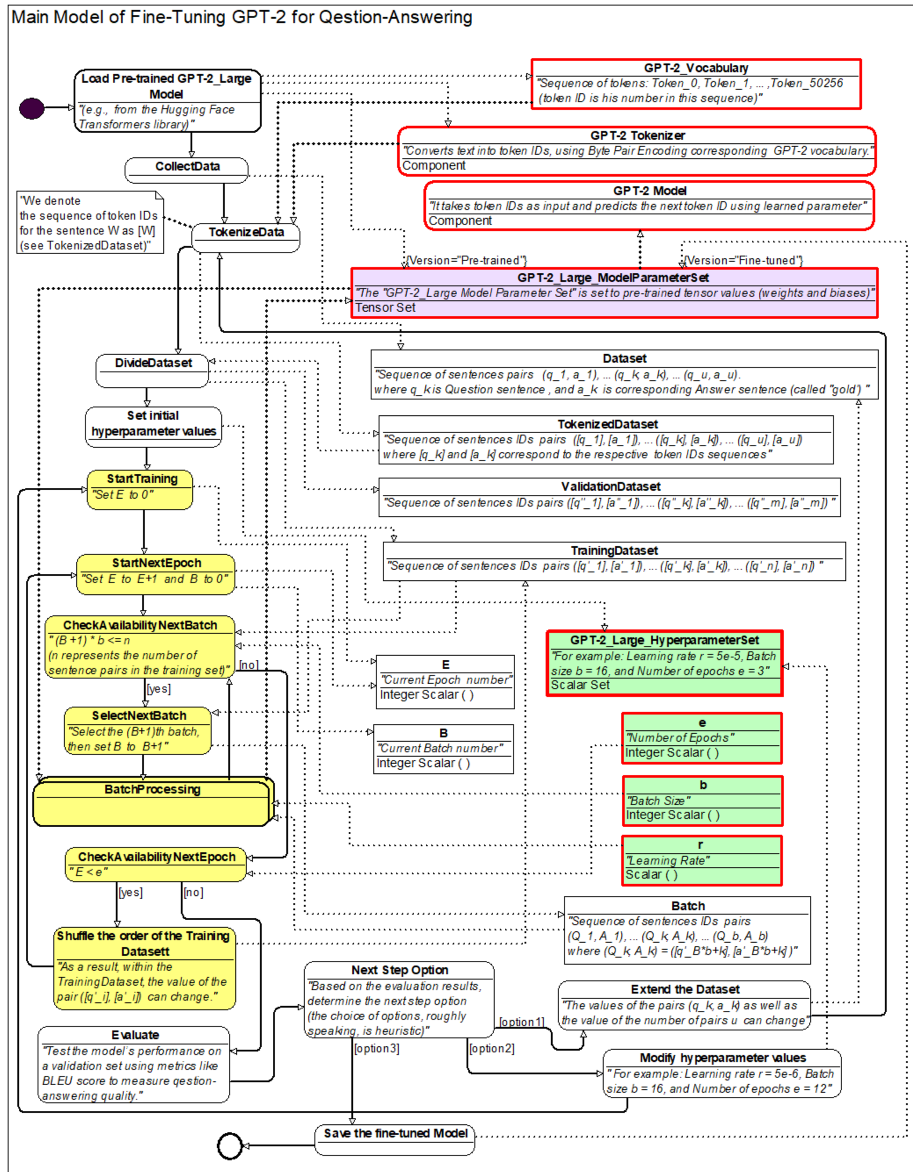

Figure FT-1

5.2. Batch Processing Model (Fig. FT-2)

The **Batch Processing Model** focuses on processing each batch of tokenized data. It includes the following key actions:

- **Process Current Batch**: Handles the batch of tokenized data.
- **Compute Gradients**: Invokes the **Gradient Calculation Model** to compute gradients for the current batch.
- **Parameter Update**: Updates the GPT-2 model's weights and biases based on the gradients received from the **Gradient Calculation Model**.

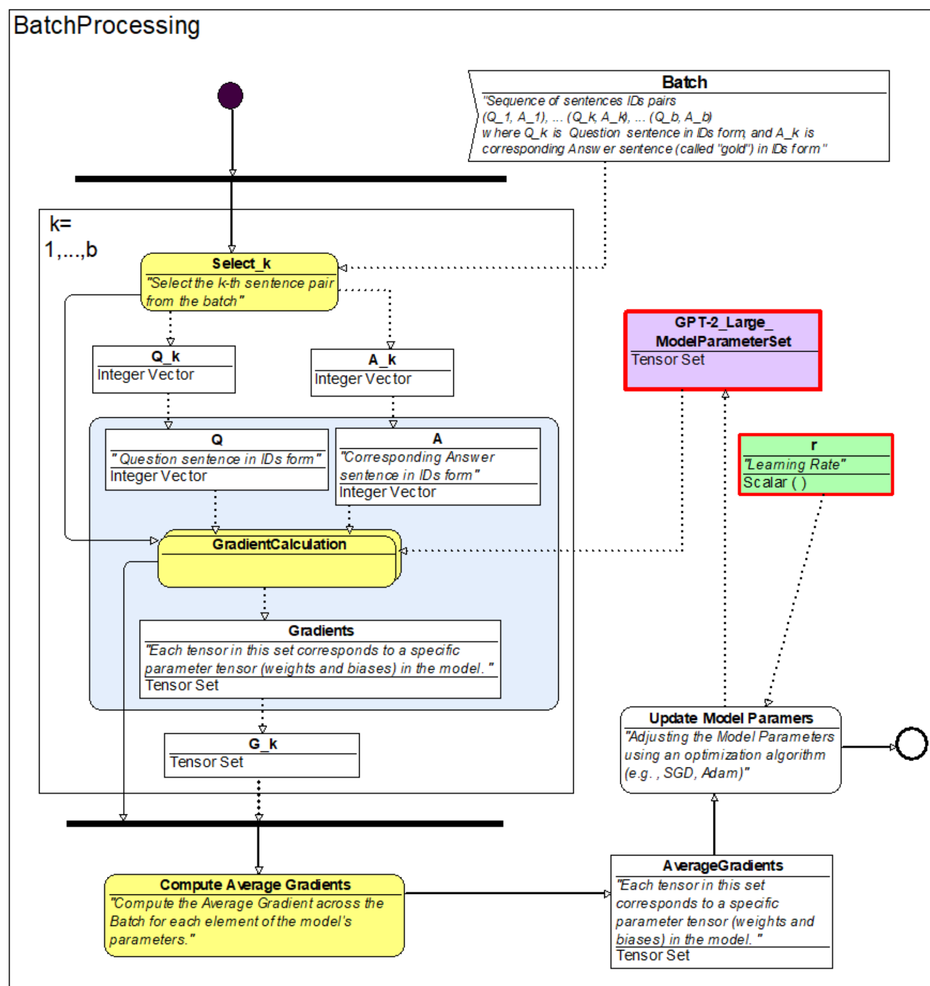

Figure FT-2

5.3. Gradient Calculation Model (Fig. FT-3)

The **Gradient Calculation Model** is responsible for calculating the gradients for the current batch. It includes the following key actions:

- **Generate Predictions**: Invokes the **Answer Generation Model** to generate predictions for the current question and calculate the loss, which is used for gradient calculation.

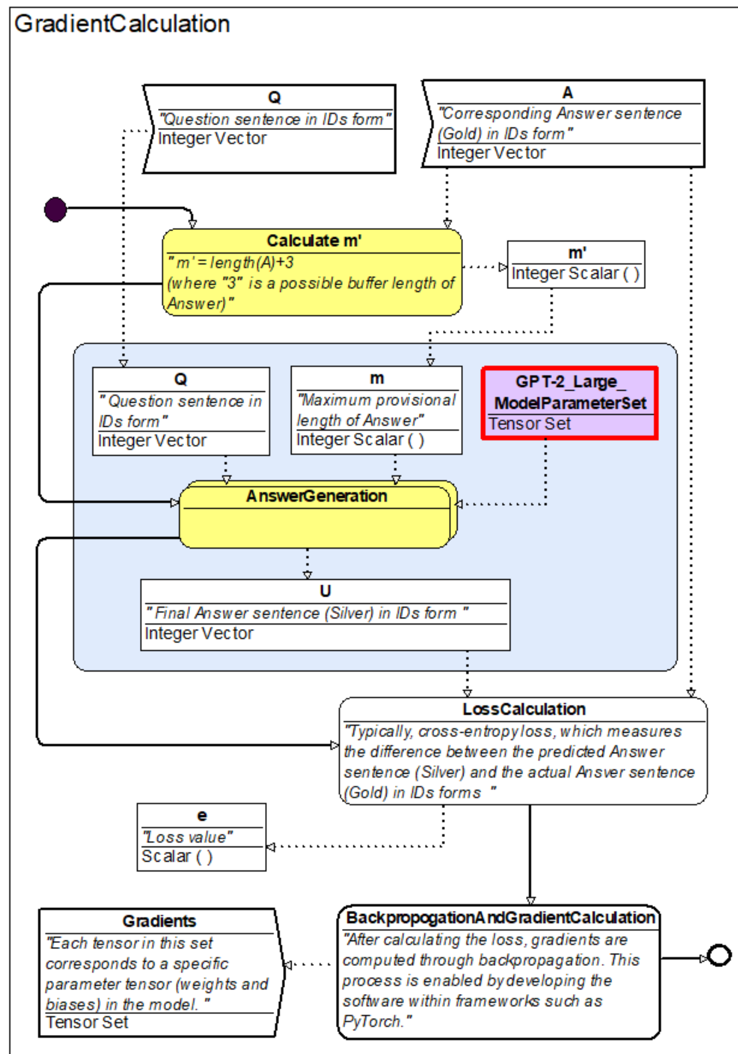

Figure FT-3

5.4. Answer Generation Model (Fig. FT-4)

The Answer **Generation Model** is responsible for processing the current query and generating a predicted answer. It includes the following key action:

- **Process Current Query**: For each question in the batch, generate the model's predicted answer.

Within the **Answer Generation Model**, during training, formatting is introduced to the sequences by adding a delimiter token, such as <|endoftext|>, to clearly mark the end of both the question and the answer. This ensures that the model knows when to stop generating responses.

- **If the model does not generate the <|endoftext|> token,** limit the generated sequence to a maximum length of n+3 tokens (where n is the number of tokens in the expected answer, with a buffer of 3).
- **Generate the answer**: The model generates an answer, stopping either when the <|endoftext|> token is produced or when the sequence reaches the n+3 limit.
- **Loss Calculation** (in Parent Gradient Calculation Model): The loss is calculated based on the tokens generated up to either the <|endoftext|> token or the maximum sequence length, whichever occurs first.

This approach ensures that the sequence generation is controlled and aligned with the expected output format.

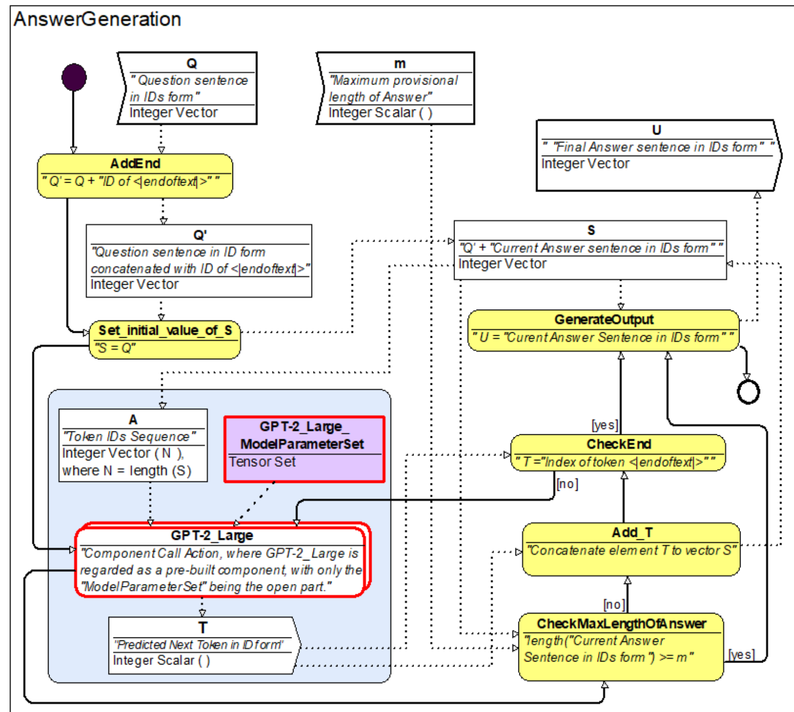

Figure FT-4

6. Conclusion

In this paper, we introduced UM1NN, a universal modeling language specifically designed for neural networks. We demonstrated its utility through two key use cases: a detailed description of the GPT-2_Large model and the fine-tuning process for question-answering tasks. These examples highlight UM1NN's capability to model complex neural network structures and operations while also showcasing its broader applicability to Machine Learning-Enabled Systems.

UM1NN goes beyond neural network modeling by providing a structured and user-friendly approach for modeling complex workflows that integrate machine learning components into larger systems. It effectively represents not only the neural networks themselves but also the surrounding systems and processes, making it a valuable tool for managing systems where machine learning plays a significant role. Its graphical representations enable clear communication among stakeholders from diverse backgrounds.

The graphical representations of UM1NN diagrams in this paper were created using the ontology graphical editor OWLGRED (Barzdins et al., 2010; WEB (g)), which supports the symbol styling mechanism needed to imitate our graphical language. Similar results can be achieved using freely available tools like Dravio (WEB (h)) and others.

Looking ahead, a potential direction for future research is to explore whether a detailed system described using UM1NN can be automatically translated into executable code with the assistance of advanced language models like ChatGPT-4 or its successors. Developing a robust serialization method for UM1NN would be crucial for this endeavor. Such a method would convert graphical representations into a format that is easily interpretable by both humans and language models, potentially enabling these models to generate implementation code based on the comprehensive system descriptions provided in UM1NN. Promising insights for this line of research are offered in recent papers (Combemale B., 2023; Petrovic N., 2023; WEB (i)), which discuss ChatGPT in software modeling.

This advancement could simplify the implementation of complex machine learning systems, making it easier for stakeholders to move from high-level design to executable code. Such automation would streamline the development process and enhance collaboration between domain experts and developers.

In conclusion, UM1NN offers a promising framework for modeling neural networks and Machine Learning-Enabled Systems. Further refinement and extension of this language could support more complex architectures and workflows, making it a practical tool for both system design and implementation in various applications.

Acknowledgments

The research was partially supported by the EU Recovery and Resilience Facility project Language technology Initiative (No 2.3.1.1.i.0/1/22/I/CFLA/002), the EU Recovery and Resilience Facility project Latvian Quantum Initiative (No 2.3.1.1.i.0/1/22/I/CFLA/001), and by research organization base financing at the IMCS UL.

References

Abadi M., Agarwal A., Barham P., Brevdo E., Chen Z., Citro C., Corrado GS. et al. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv:1603.04467v2, 2016

Alammar J. (2019). The Illustrated Transformer. https://jalammar.github.io/

Barzdins J., Barzdins G., Cerans K., Liepins R., Sprogis A. (2010). UML style graphical notation and editor for OWL 2. *Lecture Notes in Business Information Processing, vol. 64*, Springer, 2010, pp. 102-114

Brambilla M., Cabot J., Wimmer M. (2012). *Model-Driven_Software_Engineering_in_Practice*. Morgan & Claypool Publishers

Chollet F., Watson M. (2024). *Deep Learning with Python*, 3rd ed., Manning Publications, MEAP

Combemale B., Gray J., Rumpe B. (2023). ChatGPT in software modeling. *Software and Systems Modeling, vol.22,* 2023. https://link.springer.com/article/10.1007/s10270-023-01106-4

Farris D., Raff E., Biderman S. (2024). How GPT Works. Manning, MEAP Edition, version 7

Friedenthal S., Moore A., Steiner R. (2014). *A Practical Guide to SysML,* Third Edition: The Systems Modeling Language, Morgan Kaufmann Publishers

Friese P., Efftinge S., Köhnlein J. (2008). Build your own textual DSL with Tools from the Eclipse Modeling Project. https://www.eclipse.org/articles/Article-BuildYourOwnDSL/

Horvath A., Rath I., Varro D. (2015). Introducing EMF-IncQuery: super-fast incremental query evaluation over EMF models. https://www.eclipse.org/community/eclipse_newsletter/2015/november/article4.php

Kirchhof J.C., Moin A., Badii A., Guennemann S., Challenger M. (2022). MDE for Machine Learning-Enabled Software Systems: A Case Study and Comparison of MontiAnna & ML-Quadrat. arXiv:2209.07282v1

Kusmenko E., Nickels S., Pavlitskaya S., Rumpe B., Timmermanns T. (2019). Modeling and Training of Neural Processing Systems. In MODELS'19 (Munich). IEEE, pp.283–293

Lukyanenko R., Samuel B., Parsons J., Storey V., Pastor O., Jabbari A. (2024). Universal conceptual modeling: principles, benefits, and an agenda for conceptual modeling research. *Software and System Modeling, vol.* ***23***. https://link.springer.com/article/10.1007/s10270-024-01207-8

Michael J., Bork D, Wiimmer M., Mayr H. (2023). Quo Vadis Modeling? *Software and System Modeling, vol.**23**,* 2023. https://link.springer.com/article/10.1007/s10270-023-01128-y

Naveed H., Arora C., Khalajzadeh H., Grundy J., Haggag O. (2024). Model driven engineering for machine learning components: A systematic literature review. *Information and Software Technology*, **169** (2024). https://arxiv.org/abs/2311.00284

OMG (2017). Unified Modeling Language. Standard, Version 2.5.1. Object Management Group (OMG). https://www.omg.org/spec/UML/2.5.1/

OMG (2024). Kernel Modeling Language (KerML), Version 1/0 Beta 2, Release 2024-02. https://www.omg.org/spec/KerML/1.0/Beta2/PDF

Paszke A., Gross S., Massa F., Lerer A., Bradbury J., Chanan G., Killeen T., Lin Z., Gimelshein N. et al. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. arXiv:1912.01703v1

Petrovic N., Al-Azzoni I. (2023). Automated approach to model-driven engineering leveraging ChatGPT and Ecore. In: 16th International Conference on Applied Electromagnetics, 2023, Serbia. https://www.researchgate.net/publication/373439135

Pires L., Guizzardi G., Wagner G., Almeida J. (2024). An Analysis of the Semantic Foundation of KerML and SysML v2. The 43rd International Conference on Conceptual Modeling (ER 2024), Carnegie Mellon University, Pittsburgh, USA. https://www.researchgate.net/publication/383738728_An_Analysis_of_the_Semantic_Foundation_of_KerML_and_SysML_v2

Radford A., Wu J., Child R., Luan D., Amodel D., Sutskever I. (2019). Language Models are Unsupervised Multitask Learners. https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

Rädler S., Berardinelli L., Winter K., Rahimi A., Rinderle-Ma S. (2014). Bridging MDE and AI: a systematic review of domain-specific languages and model-driven practices in AI software systems engineering. *Software and System Modeling*. Published online: 28 September 2024. https://arxiv.org/abs/2307.04599v2

Rumbaugh J., Jacobson, I., Booch G. (2005). *The Unified Modeling Language Reference Manua*l, Second ed., Addison-Wesley

Shapiro R., White S.A., Bock C., Muehlen M., Brambilla M., Gagne D et al. (2012). *BPMN Handbook*, Second ed., Future Strategies Inc.

Steinberg D., Budinsky F., Paternostro M., Merks E. (2009). *EMF: Eclipse Modeling Framework,* 2nd ed. Addison-Wesley Professional

Van der Aalst W., van Hee K. (2002). *Workflow Management: Models, Methods, and Systems,* MIT Press

Vaswany A., Shazeer N., Parmar N., Uszkoreit J., Jones L., Gomez A., Kaiser L., Polosukhin I. (2017). Attention is all you need. Advances in Neural Information Processing Systems, volume 30. Curran Associates, Inc., 2017.

Vaswany A., Shazeer N., Parmar N., Uszkoreit J., Jones L., Gomez A., Kaiser L., Polosukhin I. (2023). Attention is all you need. arXiv:1706.03762v7

WEB (a) What is BPMN? Business Process Model and Notation. CAMUNDA. https://camunda.com/bpmn/

WEB (b). SysML Diagram Tutorial. https://sysml.org/tutorials/sysml-diagram-tutorial/

WEB (c). Keras home page. https://keras.io/

WEB (d). Netron: Visualizer for neural network, deep learning and machine learning models. https://github.com/lutzroeder/netron

WEB (e). Deep Learning Studio – ArcGIS. https://doc.arcgis.com/en/deep-learning-studio/latest/get-started/about-deep-learning-studio.htm

WEB (f). Deep Learning Studio homepage by Deep Cognition. https://deepcognition.ai/

WEB (g). OWLGrEd home page. http://owlgred.lumii.lv/

WEB (h). Draw.io tool home page. https://www.drawio.com/

WEB (i). Top 20 ChatGPT Prompts For Machine Learning. https://www.geeksforgeeks.org/top-chatgpt-prompts-for-machine-learning/

WEB (j). Loop in an activity diagram. https://arxiv.org/abs/2311.00284

Received October 27, 2024, accepted December 6, 2024