**MacroNet: A Python toolkit and a novel method for formalizing and constructing computational graphs such as AI networks**

Yongfan Ming, Danyang Xiong, KeXin Chen, Yifan Wu, Xiao He\*, Min Li\*

**Abstract:** The essence of an artificial intelligence model is to construct a network computing flow from the initial input data. However, whether it is through code construction, text description, or graphical representation, details cannot be expressed concisely and clearly. Our goal is to build a Python extension that allows for the expression and programming of AI network flows like formulas. For this purpose, the tool implements the following functions: (1) function names with variables (m.ddf, m.get); (2) Formulated computational flow (m.f); (3) Integrate computational diagrams (m.net) and standardize commonly used artificial intelligence structures into PyTorch based ddf format. Then, using the above functions, this article demonstrated the construction of classic networks such as XXX, XXX, and XXX. This tool enhances the scalability of functions, provides formulaic computing power based on function objects, compresses network structure code from tens of lines to a few lines, greatly enhancing the flexibility and simplicity of artificial intelligence model construction. This tool can be accessed through ` ` pip install macronet ` ` or ` ` pip install - q - e``` install

**Keywords:** Python；PyTorch；Artificial intelligence; Calculation networks

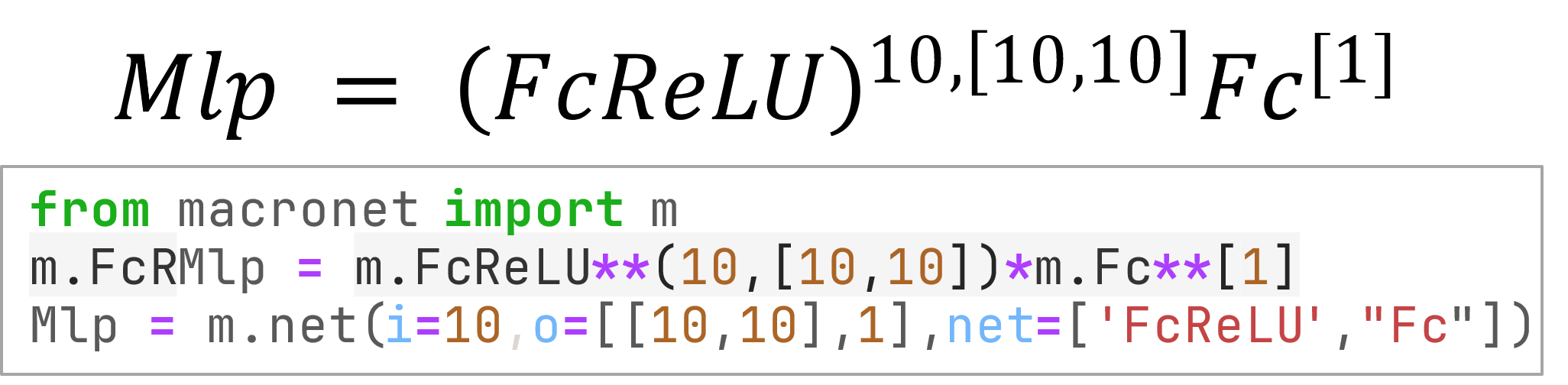
1. **Introduction**

Modern artificial intelligence (AI) leverages machine learning and its derivatives to accomplish predictive and interactive tasks such as matching (supervised learning), decision-making (reinforcement learning), and generation (unsupervised learning) through advancements in computer vision (CV), natural language processing (NLP), and speech recognition technology (SRT). From a mathematical perspective, this can be conceptualized as utilizing a parameterized computational graph (AI model) trained through feedback mechanisms to address complex, high-dimensional problems with combinatorial structures.

The current methodologies for describing and programming such computational graphs appear to lack intuitiveness and conciseness. Commonly adopted approaches in existing literature often involve circuit-like network diagrams, which, while intuitive, lack standardization and succinctness. In conventional programming paradigms, tools like PyTorch, TensorFlow, and Keras facilitate the rapid construction of network structures; however, they tend to be verbose, lacking in intuition, and inconvenient when adjusting network input/output dimensions. The inconsistent manner in which AI models are expressed and programmed poses challenges for understanding, implementation, and adjustment.

In fields such as mathematics, physics, and chemistry, concise formulas are often employed to express principles. However, current mathematical formulations for representing AI models also suffer from verbosity and lack of intuitiveness, failing to conveniently convey information regarding network structure, parameters, and data shapes. To address this issue, drawing from experience in AI model development, this paper introduces a novel approach for formalizing and constructing AI networks and computational graphs, MacroNet (https://github.com/sumowi/MacroNet.git, https://pypi.org/project/MacroNet/).

MacroNet is a Python toolkit that treats fundamental units for data processing, function methods, as primary objects. It overloads operators such as multiplication (\*) and addition (+) to organize and construct the most basic structures—sequential and parallel computation methods (flowfunc)—within the computational graph. Furthermore, it implements function name passing for function parameters (ddf), network objects with adaptive data dimensions (macro), and construction methods matching the formulaic representation (macronet). The representation and programming of an AI model with MacroNet are illustrated in Figure 1.



**Figure 1 MacroNet example for building MLP models in formula and code forms**

1. **Function and Use**

**2.1 Basic Usage and Examples**

The tool can be installed remotely via PyPI with “pip install macronet” or locally from the code repository with “pip install -q -e .”. MacroNet aims to simplify the construction and expression of computational flows such as AI neural networks, akin to formulas. To achieve this, MacroNet standardizes the format for individual functions within the computational flow in terms of dimensions of data processing (dim), data input (i, input), and output size (o, output), function labeling (l, label), and parameters (args) in both formulaic and code-based representations. Additionally, MacroNet adopts multiplication (\*) to denote sequential concatenation (seq, sequence, where the output of the previous layer serves as the input for the next layer in a sequential iteration), and addition (+) to denote layout concatenation (lay, layout, where the outputs of each layer are merged together), aiming to simplify continuous nesting of functions and merging of outputs. Moreover, for neural networks, MacroNet further simplifies the specification of input and output sizes, refering to Table 1 for details.

:

**Table 1 Summary of Basic Features and Usage of MacroNet**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Content | Function | Sequential | Layout | Net IO |
| Formula |  |  |  |  |
| Code | F = Func(i,o, N, dim, args) | F = (F1\*F2)\*\*(i,[o1,o2]) | F = (F1+F2)\*\*(i,(o1,o2)) | F = m.net(i, [o1,o2,(o3,o4)],[F1,F2,(F3,F4)]) |
| Output | y = F(x) | y1 = F1(i,o1)(x)  y = F2(o1,o2)(y1) | y1 = F1(i,o1)(x)  y2 = F2(i,o2)(x)  y=(y1, y2) | y1 = F1(i,o1)(x)  y2 = F2(o1,o2)(y1)  y3 = F1(o2,o3)(y2)  y4 = F2(o2,o4)(y2)  y=(y3, y4) |

N: data dim; i: input size; o1~on: output size; l: lable; a1~an: args;

***Basic Usage Rules:***

1. Use **“\*”** to construct sequential concatenation structure (seq) and **“+”** to construct layout concatenation structure (lay). In the code, initiate a blank computational flow using m.f.
2. Use **“@m.ddf”** to convert a regular def function into a ddf object with variable function names (where variable names in the function name must match parameter variable names). Then, call it with a matching pattern using **“m.get(‘name’)”**, or simply call it using **“m.ddf\_name”** / **“ m.f\*‘ddf\_name’ ”**.
3. Use **m.net(i, o, net)** for concise parameter i, o, where o, net can be either a **list (seq structure)** or a **tuple (lay structure)**, supporting nesting. Additionally, in **formulas** and **m.net**, only the input shape **i** of the first layer of the computational flow needs to be defined.\

***Basic Usage Example:***

**Code1 The Basic Usage Exanple of MacroNet**

|  |
| --- |
| from macronet import m,nn  import torch  *# @m.ddf Define the ddf object, where Act is a variable parameter for the function name*  @m.ddf  def FcAct(i,o,Act=""):      Fc = nn.Linear(i,o)      if Act!="":          Act = eval(f"nn.{Act}()")  *# m.fStart the computational flow and use \*/+to construct the seq/lay computational flow*      return m.f\*Fc\*Act  *# m.name Get the ddf object and pass parameters through name*  Input = m.FcReLU(10,10)  Hidden = m.FcReLU(10,10)  Out = m.Fc(10,1)  Mlp = Input\*Hidden\*Out  *# m.net Integrate i and o streams, use []/() to build seq/lay computing streams*  Mlp = m.FcReLU\*\*(10,[10,10])\*m.Fc\*\*[1]  Mlp = m.net(i=10,o=[[10,10],1],net=['FcReLU',"Fc"])  Mlp |

**2.2 Basic Usage and Examples**

**Table2 Predefined content based on pytorch:**

|  |  |  |
| --- | --- | --- |
| Formula | Code | **ddf funcspace dict** |
|  | FcAct\_bias | **{“**FcAct\_bias**“: lamda i,o,Act=””,bias=true: nn.Linear(i,o,bias)}** |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

1. **Method**

funcspace

namespace

1. **Example**
   1. **Pla & MLP for XOR**
   2. **CNN: AlexNet**
   3. **GNN**
   4. **RNN**
   5. **LSTM**
   6. **Transformer**
2. **Conclusion**