https://crad.ict.ac.cn
62(10): 2635−2650, 2025

Ionia: 基于 Coq 的带类型学习框架及代码生成工具

张晓丽[1] 陈 钢[1] 麻莹莹[2] 胡 萍[3]
[1](南京航空航天大学计算机科学与技术学院 南京 211106)
[2](海光信息科技有限公司 江苏苏州 215028)
[3](上海壁仞科技股份有限公司 上海 201114)
(xiaolizhang@nuaa.edu.cn)

Ionia: A Typed Learning Framework and Code Generation Tool Based on Coq

Zhang Xiaoli[1], Chen Gang[1], Ma Yingying[2], and Hu Ping[3]
[1](*College of Computer Science and Technology*, *Nanjing University of Aeronautics and Astronautics*, *Nanjing* 211106)
[2](*Haiguang Information Technology Co. Ltd.*, *Suzhou*, *Jiangsu* 215028)
[3](*Shanghai Biren Technology Co. Ltd.*, *Shanghai* 201114)

Abstract As the scale of AI grows rapidly, errors in deep learning applications are also increasing. Existing popular deep learning frameworks are mostly built on the dynamical-typed language Python, which lacks type checking mechanisms. This leads to many errors that cannot be eliminated through type checking at the compilation stage. We propose a strongly-typed functional programming style deep learning framework based on the theorem prover Coq. The framework features typed tensor structures and a powerful static type checking capability. The experimental results demonstrate that the framework can automatically, quickly, and effectively detect shape mismatch errors in deep learning models, and it has greater advantages in terms of speed and detection capability compared with other checking tools. Furthermore, we design and implement a set of rewriting rules that translate functional programming models into C code, realizing the translation from functional neural network operator expressions to multi-core parallel OpenMP C code. According to the results of multiple sets of experiments, the C code for operators generated by this method is on par with manually written code. Furthermore, the speed of the generated neural network operator C code with multi-core parallel optimization is improved by 4−10 times compared with the sequentially executed operator C code. Additionally, the generated C operators are highly secure and can effectively avoid common issues such as out-of-bounds indexing and memory allocation errors in manually written code.

Key words deep learning (DL) framework; static analysis; type system; shape error checking; Coq; multi-core parallel code generation

摘 要 随着人工智能(artificial intelligence, AI)规模的快速增长,深度学习应用程序中出现的错误也日益增多.现有主流深度学习框架大都建立在动态类型语言 Python 之上,缺乏类型检查机制,导致许多错误无法在编译阶段通过类型检查加以排除.为此,提出了一种基于定理证明器 Coq 的强类型函数式编程风格的深度学习框架.该框架具有带类型的张量结构和强大的静态类型检查能力.实验结果表明,该框架能够自动有效检测到深度学习模型中的形状不匹配错误,相较于其他检查工具,在检测能力和速度方面具有更大优势.进一步地,设计并实现了一套从函数式编程模型到 C 代码的重写转换规则,实现了从函数式神经网络算子表达式到多核并行 OpenMP C 代码的转换.多组实验结果表明,该方法生成的算子 C 代码与手工编写的代码质量相当,且加入多核并行优化后生成的神经网络算子 OpenMP C 代码相较于顺序执行的算子 C 代码,速度提升了 4~10 倍.此外,利用该方法生成的 C 算子具有高安全性,能够有效避免人工编

收稿日期: 2024-08-13; **修回日期**: 2025-01-23
通信作者: 陈钢(gangchensh@nuaa.edu.cn)

写代码时常见的下标越界和存储分配错误等问题.

关键词 深度学习框架;静态分析;类型系统;形状错误检测;Coq;多核并行代码生成

中图法分类号 TP311

DOI: 10.7544/issn1000-1239.202440672 **CSTR:** 32373.14.issn1000-1239.202440672

深度学习模型在图像识别、自然语言处理、机器人技术以及自动驾驶等诸多领域发挥了重要作用. 由此涌现出了一系列主流深度学习(deep learning, DL)框架, 如 PyTorch[1], TensorFlow[2], Caffe[3], Keras[4]. 这些框架使程序员能够高效地开发、训练和评估模型. 尽管这些工具使得将深度学习集成到软件系统中变得容易, 但它们都建立在动态类型语言 Python 之上, 缺乏类型检查机制. 因此, 许多错误无法在编译阶段通过类型检查加以排除. 此外, 基于命令式语言构建的神经网络模型不利于代码优化, 并且不便于进行模型正确性的形式验证.

由于深度学习应用程序独特的操作方式和数据要求, 以及应用和部署的快速增长, 这类程序中出现的错误也日益增多. 其中张量形状错误(shape error)是深度学习代码中的主要错误类型之一, 可能在深度学习任务的各个阶段发生. Wu 等人[5]的研究表明, 形状错误占据了 TensorFlow 程序中 45% 的故障.

在使用深度学习框架(如 PyTorch)构建人工智能(artificial intelligence, AI)模型时可能会出现张量形状不匹配错误. AI 模型通常由多个层组成, 每个层接受输入张量并产生一个输出张量. 每一层的输出张量形状必须与下一层的输入张量形状相匹配, 否则, 在运行时会出现形状错误. 以下是文献 [6] 中的一个示例:

```
① class Net(nn.Module):
②     def __init__(self):
③         super(Net, self). __init__()
④         self.layers = nn.Sequential(
⑤             nn.Linear(28×28, 120),
⑥             nn.ReLU(),
⑦             nn.Linear(80, 10)
⑧         )
```

该例子定义了一个简单的神经网络模型, 该模型由一系列的层组成, 包括一个线性层、一个激活函数 *ReLU* 和另一个线性层. 第 1 个线性层的输入是一个维度大小为 28×28 的 1 维序列, 输出是一个大小为 120 的张量, 最后 1 个线性层的输入是大小为 80 的张量, 输出是一个表示分类的 10 维向量, 其中激活函数不改变张量的形状. 然而, 第 2 个线性层的权重张量尺寸(80×10)与实际传入的张量尺寸(120)不匹配, 从而导致形状错误. 这种错误通常会导致程序崩溃, 从而浪费大量的计算时间. 尤其是当错误出现在复杂模型的后期阶段时, 可能需要运行很长时间才会发现错误, 因为需要处理庞大数据集的冗长预处理过程. 这不仅会导致计算资源和时间的显著浪费, 还会产生错误结果以及耗时的调试工作.

针对这种形状错误, 现有的解决方案是分析学习框架中编写的代码. 已有一些研究尝试解决深度学习模型中的形状错误问题[6-9]. 然而, 随着深度学习模型复杂性的增加, 以及数据集和输入维度规模的扩大, 形状错误的严重性愈发凸显. 这些方法并不能百分之百检查出所有形状错误.

深度学习框架可以被视为用于构建 AI 模型的领域特定编程语言, 用于实现各种算法[10]. 然而, 现有主流学习框架如 PyTorch, TensorFlow 等无法借助类型系统来排除网络模型中的形状错误. 一些强类型语言也曾尝试构建学习框架[11-12], 但是由于这些语言缺乏准确描述张量的类型系统, 也没有实现自动检查形状错误的功能. 因此, 在深度学习框架中加入自动检查形状错误的能力将是一个更好的解决方案.

本文提出了一种新的学习框架 Ionia, 该框架具有以下显著优势. 首先, 该学习框架采用了带类型的网络描述和类型检查, 能够自动解决深度学习模型中的形状错误问题, 这与以往的工作有所不同. Ionia 并不是对使用现有框架编写的代码进行分析, 而是从头开始定义了一个新的强类型学习框架. 与主流的动态类型 Python 语言的学习框架不同, Ionia 嵌入在 Coq 定理证明器中. Coq[13]是一个具有强大类型系统的定理证明助手. 本文充分利用了 Coq 的类型检查机制, 定义了一个专门用于描述张量的类型系统, 并为各种神经网络算子定义了类型以及具体语义. 通过这些类型化的算子, 该学习框架的类型检查机制能够静态检查 AI 模型中是否存在形状错误并精准定位. 换句话说, 能够通过类型检查的网络模型一定不存在形状不匹配错误. 这种方法为形状错误检测提供了更高效、更可靠的解决方案.

更重要的是, 该学习框架为进一步优化和验证

AI 模型奠定了良好基础，这是本文选择在 Coq 定理证明器中构建深度学习框架的主要考量. 自动检查形状不匹配错误只是这一设计的一个有用的副产品和阶段性成果，但 Ionia 的潜力远不止于此. 由于 Coq 是一种函数式编程语言，以其强大的数学理论为基础，提供了几乎无法在传统命令式编程语言中实现的高级优化和形式化验证途径. 凭借 Coq 强大的推理能力，使得对模型进行严格的数学证明成为可能，确保其在逻辑上的一致性和正确性，这为 AI 编译器的优化和正确性验证提供了更为坚实的基础.

最终，高级函数式抽象需要转换为命令式代码，以满足现代并行硬件接口的要求. 本文在文献 [14–16] 的基础上在 Coq 中设计并实现了一系列重写转换规则，用于将带类型的函数式神经网络算子表达式转换为相应的命令式 C 代码. 虽然文献 [16] 在文献 [14–15] 的理论基础上实现了常见函数式矩阵程序，如矩阵加法、乘法到 C 代码的转换工作. 但该技术的表达能力有限，难以在函数式层面表示出如卷积、池化等神经网络算子. 因此，本文将该技术进一步扩展，定义了一个基于有限集的下标索引到元素值映射的张量构造函数，从而能够表示出 100 余个函数式神经网络算子表达式，并设计实现了一系列针对神经网络算子的重写转换规则，用于生成算子 C 代码. 这些重写转换规则便于后续在 Coq 中进行正确性的形式验证，以保证生成的神经网络算子 C 代码的可靠性. 利用该方法，本文成功生成了一批神经网络算子的 C 代码，该代码与手工编写的代码质量相当，且生成的多核并行 OpenMP C 代码相比顺序执行的算子 C 代码速度提升了 4~10 倍. 此外，该方法生成的 C 代码具有高安全性，能够有效避免常见手工编写代码时出现的下标越界和存储分配错误.

本文的主要贡献有 3 点：

1）基于 Coq 建立了一个带类型的函数式深度学习框架 Ionia. 该框架具有带类型的张量结构以及强大的类型检查能力，能够确保构建的神经网络模型无形状错误. 实验结果表明，该框架能够自动静态检测到模型中的形状不匹配错误.

2）设计并实现了一系列重写转换规则，用于将函数式神经网络算子表达式转换为高安全、高可靠的 C 代码. 这项工作实际上具备了一个小型 AI 编译器的功能，不仅在函数式层面构建了深度学习框架，而且实现了函数式表示的神经网络算子到 OpenMP C 代码的翻译.

3）这项工作不仅为形状错误检测提供了方便，更为后续 AI 编译器的优化及正确性验证奠定了基础.

1 相关工作

目前，已有多项工作致力于解决深度学习框架中的形状不匹配问题. Hattori 等人[9] 针对 PyTorch 应用提出了一种半静态分析方法，用于推断动态计算图中的类型和形状. 然而，由于其路径不敏感和半静态的特性，该方法在实际应用的完全静态分析中并不适用. PyTea[6] 是一个自动静态分析器，用于检测 PyTorch 代码中的张量形状不匹配错误. 该分析器基于对所有可能执行路径的静态跟踪，收集张量形状约束，并检查这些约束是否不可满足. 然而，它并不执行通用的值分析，这限制了其在跟踪张量或数组索引非绑定异常时的能力. 相比之下，本文所提出的方法基于 Coq 这种依赖类型编程语言，允许直接通过索引对张量形状进行静态检查. 这使得张量形状的验证更为简单直接，并能够更全面直观地分析张量形状的依赖关系.

针对 TensorFlow 应用程序，已经提出了不同的方法来检测形状错误. Pythia[7-8] 利用值流分析静态追踪整个程序中的张量形状，并警告可能的形状不匹配. 而 ShapeFlow[17] 则作为动态分析器，通过模拟 TensorFlow 库来监控形状变换. 然而，这些方法都存在局限性. Pythia 对 Python 和 TensorFlow 的覆盖仍不足以满足实际应用的需求，而 ShapeFlow 需要一个缩小的虚拟数据集，并且无法检测由未经测试的输入数据集引起的潜在形状不匹配.

一些工具已经采用了类型系统来对张量形状进行静态检查. Hasktorch[18] 提供了对 libtorch[19] 的 Haskell[20] 绑定，以静态检查张量形状. 然而，使用 Haskell 的类型级编程特性意味着张量不被视为一等对象. Relay[21-22] 是一个深度学习编译器系统，支持丰富的张量形状类型推断，并支持动态和静态形状检查. 与此同时，文献 [23] 提出了一种基于类型的方法来检测张量形状不匹配，首先执行静态类型/形状推断，在静态推断无法保证形状一致性时添加动态检查. 文献 [24–26] 是关于依赖类型系统中的渐进类型研究. 这些工作特别关注于细化类型系统中谓词部分的渐进性，而底层简单类型则保持静态. 与这些方法相比，本文直接利用 Coq 内在的类型检查机制来实现静态类型检查.

以强类型的函数式语言为基础构建深度学习框架的研究也已有探索. DeepDSL[11] 是一种嵌入在强类型语言 Scala 中的领域特定语言，支持直接构造神经

网络模型，并提供早期的静态检查，最终将 DeepDSL 程序编译为 Java 源代码. 另外，文献 [12] 是一个基于 OCaml 编程语言的科学计算系统，包括各种由 OCaml 实现的库，如基于算法微分实现的数学优化、回归和深度神经网络等，专注于从头开始构建神经网络模型. 然而，这些工作由于缺乏对张量的准确定义，因此无法自动检测出张量形状错误.

从函数式编程模型出发，以函数式的方式表示各种操作和优化策略，然后将其转换为硬件接口所需的命令式程序，已成为解决高性能可移植性问题的一种流行方法. 例如，数据并行理想化算法语言(data parallel idealised algol, DPIA)[14-15] 实现了一种策略保留(strategy preserve)的翻译方法，它基于一系列的重写规则将高级函数式代码翻译成低级、无数据竞争的并行命令式代码. 麻莹莹等人[16] 在 Coq 定理证明器中形式化了 DPIA 技术，实现了矩阵代码生成技术，将高级函数式矩阵表达式转换为 C 代码. 然而，这些方法基于元素到元素的映射实现的函数式矩阵表达式虽然能够表示诸如向量/矩阵加法、乘法等常规矩阵操作，但在处理神经网络中的卷积、池化等算子时面临挑战. 例如，表达卷积核在输入张量上的移动过程存在困难. 相比之下，本文所定义的以有限集的下标索引到元素值的映射实现的张量表达式有助于全面表示函数式神经网络算子，从而设计出一系列重写转换规则，用于生成神经网络算子 OpenMP C 代码.

一个张量语言(a tensor language, ATL)[27] 是轻量级的 Coq 框架，旨在优化以纯函数式数组语言编写的张量程序，实现了各种 Halide[28] 优化调度指令，并最终将优化后的张量表达式翻译成 C 代码. 然而，它们的代码生成过程是直接将 ATL 程序降低为 C 代码字符串. 相比之下，本文利用延续传递(continuation passing)风格[29] 的重写转换规则，将函数式张量表达式翻译成命令式原语，提供了更高的灵活性和可扩展性. 这有助于处理复杂的嵌套函数结构，并且便于后续在 Coq 定理证明器中对重写转换规则进行严格的证明，以保证生成代码的可靠性. 此外，ATL 采用基于列表的方式来表示数组/张量，并配有保护表达式以防止数组越界访问. 相比之下，本文使用了带长度的向量类型来表示数组，并利用有限集来确保安全的数组索引访问，更简洁直接. 因此，生成的命令式代码在内存安全性方面更具优势.

本文的显著优势在于，使用本文设计的强类型深度学习框架 Ionia 构建的神经网络模型，能够从根本上避免张量形状不匹配的错误，或者说，能够在模型构建过程中自动有效检测出张量形状相关的错误并精准定位. 这一特性极大地提升了模型开发的效率和可靠性. 此外，将函数式神经网络算子表达式翻译为高安全的多核并行 OpenMP C 代码. 生成的 C 算子不仅能够精确执行预期操作，还能够避免诸如下标越界和存储分配错误等人工编写代码时常见的问题.

2 张量类型

本节将深入探讨张量类型的形式化定义方式. 这种形式化表示不仅便于描述神经网络算子操作，而且能够捕获神经网络模型中固有的形状.

张量类型是神经网络中表示数据的关键概念，它们作为多维数组的抽象表示，用于存储输入、权重和输出数据. 已有一些研究专注于张量类型的形式化定义[30-32]. 本文采用嵌套数组的方式来定义张量类型，并基于一个有限集的下标索引到元素值的映射函数来构造具体的张量.

首先，在 Coq 中定义了一个归纳数据类型 *data*，它作为各种数据结构的基础表示. 其中，构造器 *num* 和 *ary* 分别封装了数值和数组.

```
Inductive data : Set :=
| num : data
| ary : nat -> data -> data.
```

在张量的上下文中，有限集充当张量/数组的索引集，能够避免数组索引访问越界的错误. 在 Coq 中，存在多种定义自然数有限集的方法[13,33]. 本文采用了一种简洁直观的方式，允许有限自然数集与自然数之间的直接转换. 其定义如下所示：

```
Definition fin (n : nat) : Set :=
  match n with
  | O => unit
  | S n' => { i | i < (S n')}
  end.
```

在这个定义中，O 和 S 是自然数的构造函数. O 表示 0，S 表示后继函数. 例如，自然数 3 可以用 $S\,(S\,(S\,O))$ 来构造. 关键字 Definition 用于非递归函数或值的定义，而 Fixpoint 用于递归函数的定义.

fin 的定义接受一个自然数 n，并返回一个集合. 通过对自然数 n 模式匹配. 当 n 为 O 时，返回一个空集合 *unit*，当 n 不为 O 时即为 $S\,n'$，它包含一个自然数 i 以及 $i < S\,n'$的证明. 在 Coq 的构造逻辑中，证明是编码证明过程的显式项，结构 $\{i \mid P\,i\}$表示满足 $P\,i$ 为真的 i 的集合，其包括了 $P\,i$ 为真的证明. 例如，*fin*

5 表示{0, 1, 2, 3, 4}这个集合中的元素，其中每个元素都小于 5. 本文将该有限集作为张量的索引类型，可以确保下标索引保持在边界范围之内.

在定义张量类型时，形式化向量是一个关键步骤. 在 Coq 中，存在多种定义向量类型的方法，包括基于函数的方法[33]和标准库中的依赖类型[12]. 本文采用了 Coquelicot[34]中使用的向量形式化方法，该方法使用元组构造向量.

```
Fixpoint vector (n : nat) : Set :=
  match n with
  | O => unit
  | S n' => ((vector n')×B)%type
  end.
```

本文定义的向量类型是通过元组表示构造的，这种方法的优势在于接近函数式语言中的数据结构，并且便于实现向量的一些基本操作及形式化验证. 然而，这种数据结构不便于根据索引访问数组元素. 因此，当定义涉及数组索引的一些函数时，这种数据类型不方便表示. 另一种建模向量的方式是将其视为从索引集到向量元素的函数映射. 在实际工作中，有时需要在这 2 种表示方法之间进行切换. 下面定义的函数 *mk_vec* 将基于函数的向量表示转换为基于元组的向量表示.

```
Fixpoint mk_vec{n : nat}: (fin n -> B) -> vector n :=
  match n with
  | O => fun ( _:_ ) => tt :vector 0
  | S n' => fun (u : fin (S n') -> B) =>
        (mk_vec (fun i : fin n' => u i), u [n'])
  end.
```

此处，*mk_vec* 是一个递归函数，用于构造包含 *n* 个元素的向量. 其接受一个从有限集 *fin n* 到类型为 *B* 的映射函数作为参数. 当 *n* 为 *O* 时，函数返回空向量 *tt*. 当 *n* 不为 *O* 时，递归调用 *mk_vec* 构建前 *n*–1(此处表示为 *n'*)个元素，然后调用参数函数 *u* 来构建最后一个元素，其中使用函数 *exist* 生成类型为 *fin* (*S n'*)的自然数 *n'*(表示为{*n'* | *n'* < *S n'*}).

接着，引入了函数 *exp*，它在类型 *data* 的元素与本文所实现的 Coq 系统内的类型之间建立了对应关系. 该函数使用模式匹配确定每个数据构造函数的 Coq 类型，将 *num* 映射到实数集$\mathbb{R}$，将 *ary* 映射到具有指定长度和元素类型的向量.

```
Fixpoint exp (d : data) : Set :=
  match d with
  | num => ℝ
  | ary n d => vector (exp d) n
  end.
```

这种形式化为构建张量类型和表示带类型的神经网络算子奠定了基础.

为了简化不同维度张量的表示方式，本文为 Ionia 框架中的张量引入了更简洁的定义. 大小为 *m* 的 1 维张量 *Tensor*1 *m* 即表示为 *exp* (*ary m num*)，大小为 *m*×*n* 的 2 维张量 *Tensor*2 *m n* 表示为 *exp* (*ary m* (*ary n num*)). 通过这种方式，可以方便地表示出不同维度的张量类型.

最后，通过 *mk_vec* 建立基于有限集的下标索引到元素值的映射函数 *mkvSeq*{*d*}: *fin n* -> *exp d* -> *exp* (*ary n d*) 构造出了带有维度大小的张量. 通过灵活应用嵌套的函数 *mkvSeq*，我们能够构造出具有任意维度的张量，并且可以根据索引方便地访问张量中的任意元素. 这极大地简化了诸如 Conv2d 等神经网络算子的表示. 例如，大小为 *m*×*n* 且元素为默认值的 2 维张量构造如下：

```
Definition mkTensor2 (m n : nat) : Tensor2 m n :=
  mkvSeq (fun i : fin m => mkvSeq (fun j : fin n => exp_
    default num)).
```

这种方法不仅便于构造任意维度的张量，而且确保了更高的安全性和可靠性. 通过有限集对元素进行映射，有效地避免了在访问张量元素时可能出现的索引越界错误.

3 带类型的 AI 模型描述

本节首先在第 2 节张量类型的形式化定义基础上表示出一组函数式神经网络算子表达式，这些表达式不仅足以构建大多数神经网络模型，还具备检查形状错误的能力. 多组实验验证，使用带类型的算子并结合 Coq 的类型检查机制，能够确保所构建的模型不存在形状错误. 这些算子涵盖了多种神经网络操作，包括卷积、反卷积、池化、全连接、归一化等. 最终，这些函数式表示的神经网络算子表达式将作为第 5 节中代码生成工具的输入层，根据重写转换规则将其翻译为命令式 C 代码.

3.1 基本操作函数

在形式化表示神经网络算子之前，介绍 5 个针对不同数据类型的基本操作函数. 其中结构化数组的基本操作函数如表 1 所示.

表 1 中的 *let_binding* 是在第 5 节生成命令式 C 代码的上下文中，用于存储代码生成的中间结果. 虽

Table 1 Basic Operation Functions of Array

表 1 数组的基本操作函数

函数	功能	符号表示
mkvSeq	构造数组/张量	⊞
let_binding	中间变量值绑定	*tlet x* := *e* in *y*
idx	根据下标索引访问数组第 *i* 个元素	*a* ![i]
max	求数组元素最大值	
sum	求数组元素的和	∑

然在 Coq 的语法结构中有类似的 let 绑定, 但是这会忽略函数式表达式中的临时变量, 导致该临时变量在最终的表达式中不会显式出现. 这样在后续命令式代码生成过程中, 翻译器无法利用延续传递规则翻译中间变量的存储结构, 最终使函数式表达式到命令式代码的翻译不完整. 因此, 本文在 Coq 中形式化了中间变量值的绑定.

数组/张量构造函数 *mkvSeq* 和元素访问函数 *idx* 满足 *idx_mkvSeq* 性质.

定理 1. 对于任意的 *d n* (*f* : *fin n* -> *exp d*) *i*, 满足 *n* > 0 -> (*mkvSeq f*)![*i*] = *f i*.

证明. 该定理的证明采用了对数组元素数量 *n* 的归纳法. 当 *n* 为 *O* 时, 直接根据定义展开即可验证该定理成立; 对于非基础情况, 即 *n* 为非 *O* 时(即元素个数为 *S n*), 将包含 *S n* 个元素的构造函数转换为基于前 *n* 个元素的向量构造和最后 1 个元素的对偶形式, 即 (*mkvSeq f*, *e*), 然后通过索引访问新构造的向量元素, 并利用 *n* 的归纳假设条件证明包含 *S n* 个元素的构造数组函数也满足该定理. 证毕.

针对数值型的基本操作函数如表 2 所示.

Table 2 Basic Operation Functions of Numerical Type

表 2 数值型的基本操作函数

函数	功能
zero, *one*	分别代表实数 R_0, $R1$
negate	数值型元素取反的操作
add/*sub*/*mul*/*div*	2 个数值型元素的加/减/乘/除操作
scal	实数与数值型元素的乘积操作
div_n	数值型元素除以自然数的操作
pow/log/*pExp*	数值型元素的幂次方/对数/指数操作
sqrt	数值型元素的平方根操作
leExp/*gtExp*	2 个数值型元素的小于等于/大于关系

以上数值型的基本操作函数是基于 Coq 库中实数集$\mathbb{R}$上的基本操作. 基于上述数值与数组的基本操作函数, 我们可以方便地表示出各类函数式神经网络算子表达式.

3.2 函数式神经网络算子表达式

本节将以 2 维张量的卷积算子为例, 详细阐述如何在函数式层面精确地表示神经网络算子表达式. 对于其他神经网络算子, 可以采用类似的方式表示, 以确保每种操作都能得到准确的描述.

2 维卷积操作的示例如图 1 所示.

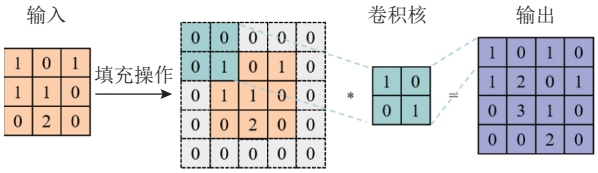


Fig. 1 Example of 2D convolution operation

图 1 2 维卷积操作示例

在 Coq 中实现的 2 维卷积算法如算法 1 所示.

算法 1. Coq 中实现的 2 维卷积算法.

输入: 填充因子 *p*, 步长 *s*, 大小为 $n_1 \times n_2$ 的卷积核张量 ***w***, 以及大小为 $m_1 \times m_2$ 的输入张量 ***input***;

输出: 大小为 $((m_1-n_1+2p)/s+1)\times((m_2-n_2+2p)/s+1)$ 的 2 维张量.

① 填充操作. 根据输入张量 ***input*** 以及填充因子 *p* 计算出填充后的张量, 该 2 维张量的维度大小分别为 (m_1+2p) 和 (m_2+2p). 具体地, 采用了嵌套的张量构造方式, 即外层构建行且内层构建列. 通过 if 条件判断当前索引是否在输入张量的范围内. 若在范围内, 则将输入张量 ***input*** 中对应的元素赋给当前索引处的元素. 如果当前索引超出了输入张量的索引范围, 则使用默认元素 *zero* 在该特定索引处进行填充.

② 引入一个中间张量 ***pad*** 用来存储①计算出的结果. 接着, 构建嵌套的 2 维张量结构, 其中外层结构表示输出张量的行 (i_0), 内层结构表示输出张量的列 (j_0). 当前索引处的输出元素通过嵌套的求和操作 (*sum*) 计算得出, 该操作遍历 2 维张量结构, 以迭代卷积核中的每个元素. 卷积核 ***w*** 的对应元素与 ***pad*** 在指定索引处的元素相乘, 并将结果相加, 以获得最终的输出元素.

2 维张量的函数式卷积表达式如下所示:

$$tlet\ \boldsymbol{pad} := \square_{i,j=0}^{m_1+2p,\ m_2+2p} \left[\begin{array}{l} [(p \leqslant i < p+m_1)\ \&\&\ (p \leqslant j < (p+m_2))] \\ input[i-p][j-p] \mid zero \end{array} \right] \text{in}$$

$$\square_{i_0,j_0=0}^{(m_1-n_1+2p)/s+1,\ (m_2-n_2+2p)/s+1} \left[\sum\sum_{i_1,j_1=0}^{n_1,n_2} \square \left[\begin{array}{l} pad[i_0 \times s + i_1][j_0 \times s + j_1] \\ \times w[i_1][j_1] \end{array} \right] \right].$$

在该表达式中，“⊞”表示所构造的张量结构，紧接着的 i, j 等符号表示张量的下标索引，张量的维数决定了下标索引的个数，“[ ]”括号内的内容对应下标索引处的张量元素值. 符号“$[\,b\,|\,a1\,|\,a2\,]$”表示 if 条件判断，即当 b 为 True 时，选择 $a1$；当 b 为 False 时，选择 $a2$. 利用这种符号表示便于直观简洁地表示出函数式神经网络算子表达式.

图 1 中的 2 维卷积示例在 Coq 中的计算结果可进行正确性验证，如图 2 所示. 其中填充因子 p 和步长 s 均为 1.

```
Definition input: Tensor2 3 3 :=
  [[1, 0, 1], [1, 1, 0], [0, 2, 0]].
Definition w: Tensor2 2 2 := [[1, 0], [0, 1]].
Example conv2d: Conv2d 1 1 w input =
                [[1, 0, 1, 0],
                 [1, 2, 0, 1],
                 [0, 3, 1, 0],
                 [0, 0, 2, 0]].
Proof. compute. auto. Qed.
```

Fig. 2 Coq example of 2D convolution operation

图 2 2 维卷积操作的 Coq 示例

以这种方式定义的卷积算子融入了类型检查机制，增强了张量表达式的健壮性和可靠性，有效解决了传统深度学习框架中常见的类型不匹配问题.

本文在 Coq 中实现了 100 余个函数式神经网络算子表达式，涵盖了多种操作类型，包括元素级操作、元素复合操作以及张量重塑等. 这些算子的实现旨在丰富 Ionia 框架的表达能力和应用场景. 通过引入这些类型化算子，Ionia 能够有效地支持各种神经网络模型的描述和类型检查. 部分的函数式神经网络算子表达式如表 3 所示.

3.3 构造带类型的神经网络模型

本节将重点展示如何使用 3.2 节中定义的函数式神经网络算子构建 AI 模型，重点关注其类型正确性的自动验证. 借助带类型的网络描述以及 Coq 的类型检查机制，可以确保模型形状的可靠性. 通过一系列 AI 模型的构建，证实了本文提出的类型化学习框架能够自动验证 AI 模型中的类型正确性.

Table 3 Functional Neural Network Operator Expression

表 3 函数式神经网络算子表达式

分类	函数	功能
元素操作	*ReLU1d/ReLU2d/ ReLU3d/ReLU4d*	对不同维度的张量元素取激活函数 *ReLU*
	Softmax1d/Softmax2d/ Softmax3d/ Softmax4d	对不同维度的张量元素取激活函数 *Softmax*
	Tanh2*d*/Tanh4*d*	对不同维度的张量元素取激活函数 Tanh
	Tadd2d/Tadd4d	对相同维度张量元素相加
	Tmul2d/Tmul4d	对相同维度张量元素相乘
复合操作	*Conv2d/Conv4d*	不同维度张量的卷积操作
	Avgpool2d/Avgpool3d/ Avgpool4d	不同维度张量的平均池化操作
	Maxpool2d/Maxpool3d/ Maxpool4d	不同维度张量的最大池化操作
	Linear/Linear2d	不同维度张量全连接操作
	NLLLoss2d	损失函数 *NLLLoss*
	ConvTranspose2d/ ConvTranspose4d	反卷积操作，扩大张量尺寸
重塑操作	*Flatten3d/Flatten4d*	平铺操作
	Truncl1d/Truncl2d	向左/右截断张量 k 个元素
	Padl/Padr	向左/向右填充 k 个元素
	Concat	连接 2 个张量

本文通过经典卷积神经网络（convolutional neural network, CNN）模型 LeNet-5[35] 来展示如何在 Coq 中构建 AI 模型，并确保所构建的 AI 模型不存在张量形状错误. LeNet-5 模型的结构图如图 3 所示.

通过带类型的神经网络算子表达式构建网络时，每一层的参数都受到严格的约束，确保在各个阶段输入和输出张量形状相匹配. LeNet-5 模型的类型化设计以及 Ionia 框架的静态类型检查机制确保了该模型中不存在张量形状错误.

Coq 中实现的带类型 LeNet-5 模型如下所示：

Definition *LeNet_5* (***input*** : *Tensor*3 1 32 32)

(***w1*** : *Tensor*4 6 1 5 5) (***b1*** : *Tensor*1 6)

(***w2*** : *Tensor*4 16 6 5 5) (***b2*** : *Tensor*1 16)

(***fw1*** : *Tensor*2 120 (16×5×5)) (***fb1*** : *Tensor*1 120)

(***fw2*** : *Tensor*2 84 120) (***fb2*** : *Tensor*1 84)

(***fw3*** : *Tensor*2 10 84) (***fb3*** : *Tensor*1 10)

:=

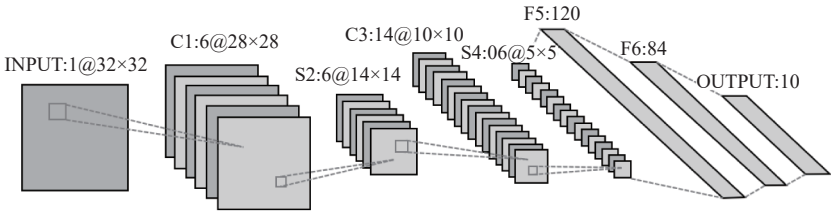


Fig. 3 Architecture diagram of LeNet-5 model

图 3 LeNet-5 模型结构图

tlet ***fm1*** := *Conv3d* 0 1 ***b1*** ***w1*** ***input*** in

tlet ***rm1*** := *ReLU3d* ***fm1*** in

tlet ***pm2*** := *Maxpool3d* 2 ***rm1*** in

tlet ***fm3*** := *Conv3d* 0 1 ***b2*** ***w2*** ***pm2*** in

tlet ***rm3*** := *ReLU3d* ***fm3*** in

tlet ***pm4*** := *Maxpool3d* 2 ***rm3*** in

tlet ***fm5*** := *Linear* ***fw1*** ***fb1*** (*Flatten3d* ***pm4***) in

tlet ***rm5*** := *ReLU*1*d* ***fm5*** in

tlet ***fm6***:= *Linear* ***fw2*** ***fb2*** ***rm5*** in

tlet ***rm6*** := *ReLU*1*d* ***fm6*** in

tlet ***fm7***:= *Linear* ***fw3*** ***fb3*** ***rm6*** in *Softmax*1*d* ***fm7***.

LeNet-5 神经网络模型通过一系列算子构建而成, 每个算子操作都经过 Coq 的静态类型检查, 以确保张量形状的正确性. 该模型的输入张量为 1×32×32, 首先经过一个操作 *Conv3d* 和激活函数 *ReLU3d*, 然后是 *Maxpool3d* 操作. 这一过程在第 2 个卷积块中重复执行. 之后, 输出被展平并通过 3 个 *Linear* 层, 中间使用激活函数 *ReLU*1*d*. 最后一层采用函数 *Softmax*1*d* 产生模型的输出. 在每个操作中, Coq 的类型检查机制均严格确保了张量维度的精确匹配. 一旦检测到张量形状不匹配的问题, Coq 将自动定位并报告问题所在, 从而确保了经过 Coq 类型检查的 AI 模型能够完全避免形状相关的错误.

为进一步展示该框架的表达能力, 本文实现了更复杂的 AI 模型, 包括 GoogLeNet[36], ResNet[37], VGG[38], LSTM[39] 等神经网络模型. 在模型构建过程中, 充分利用了 Ionia 对张量的静态类型检查能力, 确保每个模型在构建时都能避免形状错误. 表 4 列出了这一学习框架中已实现的 AI 模型.

Table 4 Typed Neural Network Model

表 4 带类型的神经网络模型

分类	模型	介绍
CNN	LeNet	一个经典神经网络模型, 由 2 个卷积层和 3 个全连接层组成
	AlexNet	一种经典的深度 CNN 模型, 由 5 个卷积层和 3 个全连接层组成
	ResNet-34	一种经典的残差 CNN 模型
	GoogLeNet	一种网中网的深度 CNN 模型, 引入了 Inception 模块
	VGG-16	深度 CNN 模型, 适用于迁移学习
	DenseNet-121	一种稠密 CNN 模型, 引入了 DenseBlock 块与前面所有层连接
GM	DCGAN	一种深度卷积生成对抗网络
	VAE	一种深度生成模型, 由 1 个编码器和 1 个解码器组成
RNN	LSTM	一种特殊的递归神经网络, 引入了 3 个门和 1 个细胞状态以处理序列中的长期依赖关系

4 形状错误检测

在本节中, 通过一系列实际的应用程序验证了该类型化学习框架可以自动、精准、有效地检测出张量形状不匹配的错误, 这对于确保 AI 模型的可靠性至关重要.

针对引言中 AI 模型形状错误的示例, 可以利用 Ionia 框架在 Coq 中构建出相同的模型, 示例如下:

① Definition *example_mismatch* (***input*** :

② *Tensor*1 (28×28)) (***fw1*** : *Tensor*2 120 (28×28))

③ (***fb1*** : *Tensor*1 120) (***fw2*** : *Tensor*2 10 80)

④ (***fb2*** : *Tensor*1 10)

⑤ :=

⑥ *tlet* ***fmap1*** := *Linear* ***fw1*** ***fb1*** ***input*** in

⑦ *tlet* ***rmap2*** := *ReLU*1*d* ***fmap1*** in

⑧ *Linear* ***fw2*** ***fb2*** ***rmap2.***

Coq 的类型系统使得在定义中可以自动发现类型不匹配错误, 并在行⑧报告错误信息: “The term “*rmap*2” has type “*Tensor*1 120” while it is expected to have type “*Tensor*1 80”. 此错误表明最后一个算子 *Linear* 的实际输入张量 ***rmap2*** 的类型是 *Tensor*1 120, 而参数权重是 *Tensor*2 10 80, 导致维度大小为 120 的 1 维张量与 10×80 的 2 维张量相乘时尺寸大小不匹配, 导致形状错误.

尽管运行相应的 PyTorch 应用程序也会出现错误信息: “RuntimeError: mat1 and mat2 shapes cannot be multiplied(1×120 and 80×10)”. 但是该错误信息仅能提供整个层的提示, 而无法精确指向具体某层中的某个操作. 此外, 获取此类错误信息需要完整运行整个应用程序. 然而, 由于复杂的深度学习模型可能需要较长时间的训练, 因此发现错误可能需要花费大量时间, 导致显著的开销. 相比之下, 本文利用 Coq 的静态类型检查机制, 能够直接将错误精准定位到具体操作上. 更重要的是, 由于这种检查方式是静态的, 因此能够立即发现问题, 而无需等待整个应用程序运行完成, 从而大大缩短了响应时间. 这一优势不仅提高了错误检测的效率, 也降低了调试的复杂性.

具体解决方法为调整第 2 个全连接层 ***fw2*** 权重的尺寸, 将其更改为 *Tensor*2 10 120 以匹配上一层的输出维度. 在做出相应更改后, 得到了修正后的代码:

① Definition *example_mismatch* (***input*** :

② *Tensor*1 (28×28)) (***fw1*** : *Tensor*2 120 (28×28))

③ (***fb1*** : *Tensor*1 120) (***fw2*** : *Tensor*2 10 120)

④ (***fb2*** : *Tensor*1 10)

⑤ :=

⑥ *tlet* ***fmap1*** := *Linear* ***fw1*** ***fb1*** ***input*** in

⑦ *tlet* ***rmap2*** := *ReLU*1*d* ***fmap1*** in

⑧ *Linear* ***fw2*** ***fb2*** ***rmap2.***

该代码通过了 Coq 的类型检查，表明此时该模型描述不包含任何形状错误.

为了评估该学习框架在形状错误检测方面的实际应用能力，本文研究了 PyTorch 应用程序，并从 StackOverflow 平台收集了与形状错误相关的典型案例. 随后，展示了 Ionia 凭借其强大的静态类型检查能力，能够精准有效检测出张量形状不匹配的问题.

首先，在 Coq 中实现了基于 GitHub 上 pytorch/examples[40] 官方 PyTorch 示例项目的 AI 模型. 为了验证 Ionia 框架在检测和纠正张量形状不匹配方面的有效性，我们在模型中主动引入了形状错误. 同时，利用该框架复现了从 StackOverflow 平台中收集到的错误案例[41]，以测试 Ionia 框架的形状检查能力. 为验证 Ionia 的检测能力，本文将检测结果与 PyTea[6] 和 Hattori 等人[9] 的检测工具对比，结果如表 5 所示.

结果表明，Ionia 均成功快速地检测到了每个形状不匹配的情况，并且能够精准定位问题所在. 与此同时，尽管 PyTea[6] 能够检测到大部分 PyTorch 应用程序的形状错误，但无法分析出 TensorFlow 中的错误，并且其需要一定的响应时间，相比之下效率较低. 而 Hattori 等人[9] 的分析器在测试中均未能成功检测出形状错误. 值得注意的是，本文提出的方法不仅能够精准定位到形状不一致的具体位置，而且几乎无响应时间，足够快速有效. 这种能力潜在地节省了调试时间，因为它可以快速准确地定位并解决形状不匹配的问题，而无需执行整个程序.

Table 5 Shape Error Detection Results

表 5 形状错误检测结果

模型/问题	检测工具		
	Ionia（本文）	PyTea[6]	Hattori 等人[9]
mnist	○	○	×
dcgan	○	○	×
vae	○	○	△
alexnet	○	○	△
resnet	○	○	×
case 1(76938435)	○	○	×
case 2(66995380)	○	○	×
case 3(60121107)	○	○	×
case 4(43067338)	○	×	×
case 5(63049638)	○	×	×

注：“○”代表检测成功并发现了注入错误；“△”代表检测成功但需要修改主代码，如提供明确的输入张量；“×”表示检测失败.

通过多个模型构建和形状错误检测的实例后，可以得出：学习框架 Ionia 不仅具有足够大的表达能力，而且具备卓越的类型检查功能. 这一能力能够极大地协助用户进行高效的模型调试，帮助用户快速发现并精准定位模型中存在的形状不匹配错误. 此外，该框架提供了丰富的 API 和库函数，用户可以轻松构建各种类型的神经网络模型，而无需担心形状错误.

5 代码生成工具

在本节，提出了一种神经网络算子代码生成工具，用于将 3.2 节中定义的函数式神经网络算子表达式翻译为相应的命令式 C 代码. 该代码生成方法受 DPIA[14-16] 的启发. 其中文献 [16] 是 DPIA 的 Coq 实现，通过重写机制将函数式矩阵程序转换为 C 代码. 然而，该方法针对的是常规矩阵操作，如矩阵加法、乘法，并不支持神经网络算子如卷积、池化的代码生成. 因此，本文在此基础上，进一步扩展了该技术，设计并实现了一系列重写转换规则，用于将函数式神经网络算子表达式转换为 OpenMP C 代码.

代码生成方法主要由 5 部分组成. 第 1 部分是将 Ionia 框架所描述的神经网络模型及函数式算子表达式作为代码生成的输入层. 第 2 部分是基本的函数式原语，包括 3.1 节中定义的数值与数组的基本函数. 第 3 部分是将中间命令式组合子作为函数式表示的命令式中间结构. 第 4 部分是低级命令式原语对应于命令式 C 代码中的语句. 第 5 部分是生成的命令式程序. 其中第 1 部分的函数式算子由第 2 部分的基本函数构造，第 3 部分的中间命令式组合子由第 4 部分命令式原语组成. 因此，这 2 部分的转换只要展开相应的定义即可. 基本的函数式原语通过翻译器 A 和翻译器 C 的重写转换规则转换为中间命令式组合子，而命令式原语通过显示翻译规则（explicit translation rules）直接转换为 C 语句字符串. 通过这一过程，以函数式形式表示的算子表达式最终翻译成对应的 C 程序. 图 4 是整个代码的生成过程.

图 4 中的函数式神经网络算子表达式和基本的函数式原语在第 3 节中已定义，接下来将分别介绍命令式原语、中间命令式组合子、重写转换规则以及显示翻译规则.

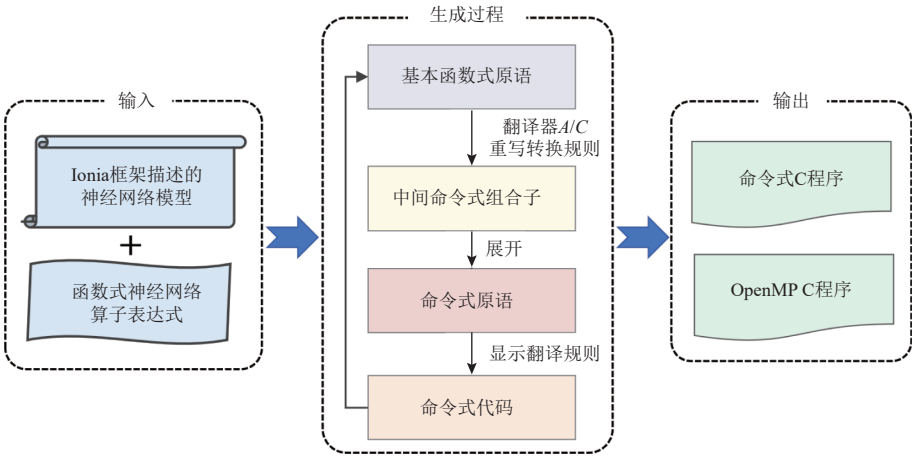


Fig. 4 Code generation process

图 4 代码生成过程

5.1 命令式原语

在命令式 C 代码中，常将表达式的值赋给某个变量，该变量充当被赋值的对象. 在 DPIA[14-16] 中表示为接收器类型，该类型是一组由输入数据类型确定的数据访问路径，定义如下：

```
Fixpoint acc (d : data) : Set :=
  match d with
  | num => path
  | ary n d => vector (acc d) n
  end.
```

其中 *path* 是归纳定义的路径，由字符串组成.

DPIA 提出了一组用于描述 C 语句的命令式原语. 为生成带有选择语句的 C 程序，本文增加了 *If* 条件命令，现有的命令式原语包括：

comm ::= *skip* | *seq* | *If* | *assign* | *new* | *for* | *parfor*.

本文在 Coq 中用归纳定义的数据结构以实现 *comm*，它包括 7 个构造子. 其中，*skip* 表示空命令或无操作命令. *seq* 表示顺序组合，即 2 个命令按顺序执行，用符号“*p*1; *p*2”表示. *If* 表示条件命令，根据布尔条件决定命令的执行. *assign* 表示赋值命令，将表达式的值 *e* 赋给接收器变量 *a*，用符号“*a* |:= *e*”表示. *new* 表示声明新变量的命令，该命令由一个函数生成，该函数接受一个接收器和表达式的对偶以及另一个命令作为参数，并返回一个新的命令. *for* 表示循环，根据有限集合中的元素，重复执行一个命令. *parfor* 表示并行循环，其接受一个变量接收器，用于存储并行操作时的临时变量.

5.2 中间命令式组合子

本节将带有高阶的基本函数式原语定义为由命令式原语组成的中间命令式组合子. 包括张量构造函数 *mkvSeq* 的中间命令式组合子 *mkvSeql*、并行操作张量的中间命令式组合子 *mkvParl*、中间变量绑定值 *let_binding* 的中间命令式组合子 *letl*，以及对数组元素求最大值、规约求和的中间命令式组合子 *maxl* 和 *suml*. 具体定义如表 6 所示.

表 6 中的 *mkvSeql* 接受一个函数 *f*，该函数将有限集映射到具有输出访问路径的命令上. 通过 *for* 命令对有限集中的每个元素 *i* 执行 *f* 操作，其中 *out*|{*i*} 表示输出张量接收器的第 *i* 个元素. 该定义以函数式

Table 6 Intermediate Imperative Combinators

表 6 中间命令式组合子

函数	具体实现
mkvSeql	Definition *mkvSeql* {*n*:*nat*} {*d*:*data*} (*f*:*fin n -> acc d -> comm*) (*out*:*acc*(*ary n d*)) := *for* (*fun i*:*fin n => f i out*|{*i*}).
mkvParl	Definition *mkvParl* {*n*:*nat*} {*d*:*data*} (*f*:*fin n -> acc d -> comm*) (*out*:*acc*(*ary n d*)):= *parfor out* (*fun i o => f i o*).
letl	Definition *letl*{*s t*}(*e*:*exp s*)(*f*:*exp s -> acc t -> comm*)(*out*:*acc t*) := *f e out*.
maxl	Definition *maxl* {*n*:*nat*} (***xs***:*exp* (*ary n num*)) (*f*:*exp num -> acc num -> comm*) (*c* : *exp num -> comm*) := *new* (*fun tmp =>* let (*w*,*v*) := *tmp* in (*w* |:= *xs* |[0]) ; (*for* (*fun i => If* (*gtExp xs*|[*i*] *v*) (*f xs*|[*i*] *w*,*f v w*)) ; *c v*)).
suml	Definition *suml* {*n*:*nat*} (***xs*** : *exp* (*ary n num*)) (*f*:*exp num -> acc num -> comm*) (*c* : *exp num -> comm*) := *new* (*fun tmp =>* let (*w*,*v*) := *tmp* in (*w* |:= *zero*) ; (*for* (*fun i => f* (*add xs*|[*i*] *v*) *w*)) ; *c v*).

的方式表示出命令式的数组循环操作. 在 5.3 节中，利用翻译器 *A* 和翻译器 *C* 的重写转换规则将张量构造函数 *mkvSeq* 转换为命令式表示的 *mkvSeql*，从而实现从函数式原语到命令式原语的转换. *mkvParl* 是针对有限集到元素映射的并行操作，具体地，通过 *parfor* 命令并行执行数组的循环操作，其中添加了临时变量. *letl*, *maxl*, *suml* 是中间变量绑定值、数组最大值、数组元素和的命令式表示.

5.3 重写转换规则

重写转换规则是将函数式表达式转换为命令式原语的核心机制. 这些转换规则通过翻译器 *A* 和翻译器 *C* 互相定义. 翻译器 *A* 的具体定义为 *A*(*a*, *e*)，表示为命令的赋值操作，具体地，将表达式的值 *e* 赋给接收器变量 *a*. 而翻译器 *C* 的具体定义为 *C*(*e*, *c*)，其中 *e* 代表给定的表达式，*c* 是表达式到命令的映射函数，具体形式为 *exp d -> comm*，该函数将表达式域中的元素映射到命令域中的元素. 通过将表达式 *e* 作用于映射函数 *c* 上，可以继续根据映射结果翻译后续的命令，最终返回一个新命令.

根据本文所实现的函数式神经网络算子表达式，我们设计并实现了翻译器 *A* 的重写转换规则，这些重写转换规则旨在将基本的函数式原语有效转换为低级命令式原语的组合形式. 具体地，翻译器 *A* 的重写转换规则如图 5 所示.

A1~A8 是针对数值型的重写转换规则. 例如，A2 将 2 个数值型的表达式求和结果赋给接收器 *a*，相当于依次对 2 个表达式翻译后再求和的赋值操作. A9~A14 是高阶数组的重写转换规则，该重写转换规则将函数式原语转换为由命令式原语组成的中间命令式组合子. 例如，A9 中对于给定的函数 *f* 和表达式 *e* 将中间变量绑定值 *let_binding e f* 赋给接收器 *a*，相当于 *letl* 中将变量绑定值赋给变量 *a*. A10 表明将整个张量赋给接收器等价于逐个对张量中的元素进行赋值

A(*a*, *negate e*)	=	*C*(*e*, *fun x* => *a* |:= *negate x*)	(A1)
A(*a*, *add e1 e2*)	=	*C*(*e1*, *fun x* => *C*(*e2* , *fun y* => *a* |:= *add x y*))	(A2)
A(*a*, *sub e1 e2*)	=	*C*(*e1*, *fun x* => *C*(*e2* , *fun y* => *a* |:= *sub x y*))	(A3)
A(*a*, *mul e1 e2*)	=	*C*(*e1*, *fun x* => *C*(*e2* , *fun y* => *a* |:= *mul x y*))	(A4)
A(*a*, *div e1 e2*)	=	*C*(*e1*, *fun x* => *C*(*e2* , *fun y* => *a* |:= *div x y*))	(A5)
A(*a*, *scal c e*)	=	*C*(*e*, *fun x* => *a* |:= *scal c x*)	(A6)
A(*a*, *div_n e n*)	=	*C*(*e*, *fun x* => *a* |:= *div_n x n*)	(A7)
A(*a*, log *e*)	=	*C*(*e*, *fun x* => *a* |:= log *x*)	(A8)
A(*a*, *let_binding e f*)	=	*C*(***e***, *fun* ***x*** => *letl* ***x*** (*fun x*1 => (*fun o* => *A*(o, *f x*1))) *a*)	(A9)
A(*a*, *mkvSeq f*)	=	*mkvSeql* (*fun i o* => *A*(*o*, *f i*)) *a*	(A10)
A(*a*, *mkvPar f*)	=	*mkvParl* (*fun i o* => *A*(*o*, *f i*)) *a*	(A11)
A(*a*, max ***e***)	=	*C*(***e***, *fun* ***x*** => *maxl* ***x*** (*fun x*1 *tmp* => *A*(*tmp*, *x*1)) (*fun r* =>*A*(*a*, *r*)))	(A12)
A(*a*, *sum* ***e***)	=	*C*(***e***, *fun* ***x*** => *suml* ***x*** (*fun x*1 *tmp* => *A*(*tmp*, *x*1)) (*fun r* =>*A*(*a*, *r*)))	(A13)
A(*a*|{*i*}, if *b* then ***e1*** else ***e2***)	=	*If b* (*A*(*a*|{*i*}, ***e1***), *A*(*a*|{*i*}, ***e2***))	(A14)

Fig. 5 Rewriting translation rules of translator *A*

图 5 翻译器 *A* 的重写转换规则

操作. A11 是张量赋值的并行循环实现，其中 *mkvPar* 的实现与 *mkvSeq* 相同，该规则表示对张量整体的赋值等价于并行地对张量中的每个元素进行赋值(相当于 *mkvParl*). 重写转换规则 A12 和 A13 是将张量元素的最大值以及张量元素规约求和转换为中间命令式组合子 *maxl* 和 *suml*. A14 用于将 if 判断的函数式表示等价地转换为命令式条件原语 *If* 分别进行赋值.

相应地，翻译器 *C* 的重写转换规则如图 6 所示.

C(*negate e*, *c*)	=	*C*(*e*, *fun x* => *c* (*negate x*))	(C1)
C(*add e1 e2*, *c*)	=	*C*(*e1*, *fun x* => *C*(*e2* , *fun y* => *c* (*add x y*)))	(C2)
C(*sub e1 e2*, *c*)	=	*C*(*e1*, *fun x* => *C*(*e2* , *fun y* => *c* (*sub x y*)))	(C3)
C(*mul e1 e2*, *c*)	=	*C*(*e1*, *fun x* => *C*(*e2* , *fun y* => *c* (*mul x y*)))	(C4)
C(*div e1 e2*, *c*)	=	*C*(*e1*, *fun x* => *C*(*e2* , *fun y* => *c* (*div x y*)))	(C5)
C(*scal m e*, *c*)	=	*C*(*e*, *fun x* => *c* (*scal m x*))	(C6)
C(*div_n e n*, *c*)	=	*C*(*e*, *fun x* => *c* (*div_n x n*))	(C7)
C(log *e*, *c*)	=	*C*(*e*, *fun x* => *c* (log *x*))	(C8)
C(*let_binding* ***e*** *f*, *c*)	=	*new* (*fun tmp* => let (***w***, ***v***) := *tmp* in *A*(***w***, *let_binding* ***e*** *f*) ; *c* ***v***)	(C9)
C(*mkvSeq f*, *c*)	=	*new* (*fun tmp* => let (***w***, ***v***) := *tmp* in *A*(***w***, *mkvSeq f*) ; *c* ***v***)	(C10)
C(*mkvPar f*, *c*)	=	*new* (*fun tmp* => let (***w***,***v***) := *tmp* in *A*(***w***, *mkvPar f*) ; *c* ***v***)	(C11)
C(max ***e***, *c*)	=	*C*(***e***, *fun* ***x*** => *maxl* ***x*** (*fun x' o'* => *A*(*o'*, *x'*)) *c*)	(C12)
C(*sum* ***e***, *c*)	=	*C*(***e***, *fun* ***x*** => *suml* ***x*** (*fun x' o'* => *A*(*o'*, *x'*)) *c*)	(C13)
C(if *b* then *e1* else *e2*, *c*)	=	*If b* (*C*(*e1*, *c*), *C*(*e2*, *c*))	(C14)

Fig. 6 Rewriting translation rules of translator *C*

图 6 翻译器 *C* 的重写转换规则

C1~C8 是翻译器 *C* 针对数值型的重写转换规则. C9~C14 是高阶数组的重写转换规则，其中 C9 表示将中间变量绑定值 *let_binding* ***e*** *f* 传给映射函数 *c*(表达式到命令的映射函数)，等价于命令式语句中声明一个临时变量，将中间变量绑定值赋给该临时变量，并在后续命令中继续使用该变量的值，这是生成命令式 C 代码中临时变量存储结构的核心. 通过存储中间变量，有效解决了代码生成中部分代码翻译不完整的情况. C10 和 C11 分别是张量构造函数的顺序和并行重写转换规则. C12 和 C13 分别是针对数组的最大值和元素求和操作的重写转换规则，分别将函数式表示的 max 和 *sum* 转换为中间命令式组合子 *maxl* 和 *suml*，并对数组表达式继续应用翻译 *C* 的重写转换规则. C14 是将函数式表示的 if 判断操作转换为低级命令式原语 *If*，并对每个判断条件的分量继续翻译.

5.4 显示翻译规则

本节将命令式原语、数值型的表达式以及数值型的接收器显示翻译为 C 语句字符串. 显示翻译规则如图 7 所示.

规则(1)~(7)是针对 5.1 节中的命令式原语的显示翻译，将命令式原语显示翻译为 C 语句字符串. 规则(8)~(13)是数值型表达式的显示翻译，规则(14)(15)是数值型接收器的显示翻译.

由于本文采用的是基于有限集的下标索引到元素值的映射函数所构造的张量表达式，因此生成的

CG_comm skip	= "/* skip */"	(1)
CG_comm (*If b* (*c*1, *c*2))	= "if (" ++ *b* ++") { " ++ *CG_comm c*1++ "} else { " ++ *CG_comm c*2++ "}"	(2)
CG_comm (*p*1 ; *p*2)	= (*CG_comm p*1) ++ "" ++ (*CG_comm p*2)	(3)
CG_comm (*a* |:= *e*)	= (*CG_acc* [] *a*) ++ " = " ++ (*CG_exp* [] *e*) ++ " ;"	(4)
CG_comm (*new p*)	= float *d* ++";" ++ (*CG_comm* (*p cnt*, *vrd cnt*))	(5)
CG_comm (*for p*)	= "for(int "++(*i cnt*)++"=0; "++(*i cnt*)++"<" ++*n*++"; "++(*i cnt*)++" += 1) { " ++ (*CG_comm* (*p cnt*)) ++ "}"	(6)
CG_comm (*parfor a p*)	= "#pragma omp parallel for for(int "++(*i cnt*)++"=0; "++(*i cnt*)++"<" ++ *n* ++"; "++(*i cnt*)++" += 1) { " ++ (*CG_comm* (*p cnt*) *a*) ++ "}"	(7)
CG_exp [] (*add e*1 *e*2)	= *CG_exp* [] *e*1++" + "++*CG_exp* [] *e*2	(8)
CG_exp [] (*mul e*1 *e*2)	= *CG_exp* [] *e*1++" * "++*CG_exp* [] *e*2	(9)
CG_exp [] (log *e*)	= "(log("++(*CG_exp* [] *e*)++"))"	(10)
CG_exp ps (*idx* ***e*** *i*)	= *CG_exp* (*Aidx i*::*ps*) ***e***	(11)
writeBool (*leExp e*1 *e*2)	= (*CG_exp* [] *e*1)++" <= "++(*CG_exp* [] *e*2)	(12)
writeBool (*gtExp e*1 *e*2)	= (*CG_exp* [] *e*1)++" > "++(*CG_exp* [] *e*2)	(13)
CG_acc ps (*vwt a*)	= "a"	(14)
CG_acc ps (*a* |{*i*})	= "a[*i*]"	(15)

Fig. 7 Explicit translation rules

图 7 显示翻译规则

命令式 C 代码不存在下标越界的错误. 下面是基于上述重写转换规则和显示翻译规则生成的 2 维张量卷积算子 *Conv*2*d* 的命令式 OpenMP C 代码.

① float $v_0[2\times p + m_1][2\times p + m_2]$;
for (int $i_1 = 0$; $i_1 < 2\times p + m_1$; i_1 += 1)
  for (int $i_2 = 0$; $i_2 < 2\times p + m_2$; i_2 += 1)
    if ($i_1 < p$ && $i_2 < p$)
      $v_0[i_1][i_2] = 0$;
    else if ($p \leqslant i_1$ && $i_1 < p + m_1$ && $p \leqslant i_2$ && $i_2 < p + m$))
      $v_0[i_1][i_2] = input[i_1 - p][i_2 - p]$;
      else
      $v_0[i_1][i_2] = 0$;
      end if
    end if
  end for
end for
# pragma omp parallel for
for (int $i_2 = 0$; $i_2 < (m_1 - n_1 + 2\times p) / s + 1$; i_2 += 1)
    # pragma omp parallel for
    for (int $i_3 = 0$; $i_3 < (m_2 - n_2 + 2\times p) / s + 1$; i_3 += 1)
      float v_4;
      $v_4 = 0$;
      for (int $i_5 = 0$; $i_5 < n_1$; i_5 += 1)
        float v_6;
        $v_6 = 0$;
        for (int $i_7 = 0$; $i_7 < n_2$; i_7 += 1)
          $v_6 = ((v_0[i_2\times s+i_5][i_3\times s + i_7]\times kernel[i_5][i_7])+ v_6)$;
        end for
        $v_4 = (v_6 + v_4)$;
      end for
    $out[i_2][i_3] = v_4$;
  end for
end for

具体的翻译步骤如下.

输入: 变量接收器 *out*, 带并行策略的函数式 2 维张量卷积算子表达式 *Conv*2*d*, 填充因子 *p*, 步长 *s*, 卷积核 ***kernel2d***, 输入张量 ***input2d***;

输出: 命令式 2 维卷积算子 OpenMP C 代码.

①展开函数式卷积算子 *Conv*2*d* 得到:

A(*vwt* "*out*", *tlet* ***pad*** := *Padding*2*d p* ***input2d*** in *mkvPar* (λ_{i0}. *mkvPar* (λ_{j0}. *sum* (*mkvSeq* (λ_{i1}. *sum* (*mkvSeq* (λ_{j1}. *mul* (*pad* |[*i*0 × *s* + *i*1]) |[*j*0 × *s* + *j*1] (*kernel2d* |[*i*1]) |[*j*1])))))).

②应用翻译器 *A* 的重写转换规则 A9(Alet), 将中间变量绑定值 *tlet* 转换为中间命令式组合子 *letl*, 得到如下表达式:

C(*Padding*2*d p* ***input2d***, λ_x. *letl x* (λ_{x1} *o*. *A*(*o*, *mkvPar* (λ_{i0}. *mkvPar* (λ_{j0}. *sum* (*mkvSeq* (λ_{i1}. *sum* (*mkvSeq* (λ_{j1}. *mul* (*x*1|[*i*0 × *s* + *i*1]) |[*j*0 × *s* + *j*1] (*kernel2d* |[*i*1]) |[*j*1])))))))) (*vwt* "*out*")).

③展开中间命令式组合子 *letl* 得到:

C(*Padding*2*d p* ***input2d***, λ_x. *A*(*vwt* "*out*", *mkvPar* (λ_{i0}. *mkvPar* (λ_{j0}. *sum* (*mkvSeq* (λ_{i1}. *sum* (*mkvSeq* (λ_{j1}. *mul* (*x*|[*i*0 × *s* + *i*1]) |[*j*0 × *s* + *j*1] (*kernel2d* |[*i*1]) |[*j*1])))))).

④展开 *Padding*2*d* 的定义得到:

C(*mkvSeq* (λ_i. *mkvSeq* (λ_j. if (*i* < *p*) && (*j* <*p*) then *zero* else if (*p* ⩽ *i*) && (*i* < *p* + *m*1) && (*p* ⩽ *j*) && (*j* < *p* + *m*2) then (*input2d* |[*i* − *p*]) |[*j* − *p*] else *zero*)), λ_x. *A*(*vwt* "*out*", *mkvPar* (λ_{i0}. *mkvPar* (λ_{j0}. *sum* (*mkvSeq* (λ_{i1}. *sum* (*mkvSeq* (λ_{j1}. *mul* (*x*|[*i*0 × *s* + *i*1]) |[*j*0 × *s* + *j*1] (*kernel2d* |[*i*1]) |[*j*1])))))).

⑤应用翻译器 *C* 的重写规则 C10(CmkvSeq), 将整个表达式转换为 2 部分. 首先声明一个类型为 2 维张量的临时变量 *tmp*(*tmp* 为接收器变量 *w* 和表达式 *v* 的对偶 (*w*, *v*)), 并将表达式的值赋给该变量 *w*. 接着, 在后续命令中继续使用该变量的值 *v*:

CG_comm (*new* (λ_{tmp}. *A*(*w*, *mkvSeq* (λ_i. *mkvSeq* (λ_j. if (*i* < *p*) && (*j* < *p*) then *zero* else if (*p* ⩽ *i*) && (*i* < *p* + *m*1) && (*p* ⩽ *j*) && (*j* < *p* + *m*2) then (*input2d* |[*i* − *p*]) |[*j* − *p*] else *zero*)))) ;

A(*vwt* "*out*", *mkvPar* (λ_{i0}. *mkvPar* (λ_{j0}. *sum* (*mkvSeq* (λ_{i1}. *sum* (*mkvSeq* (λ_{j1}. *mul* (*v*|[*i*0 × *s* + *i*1]) |[*j*0 × *s* + *j*1]

(*kernel2d* |[*i*1]) |[*j*1])))))))).

⑥接着，应用命令的显示翻译规则(5)(CGnew)和规则(3)(CGseq)，显示翻译出接收器变量 *v*0，以及命令的顺序执行：

“float *v*0[((2)×(*p*)) + (*m*1)][((2)×(*p*)) + (*m*2)]; ” ++ *A*(*v*0, *mkvSeq* (λ*i*. *mkvSeq* (λ*j*. if (*i* < *p*) && (*j* < *p*) then *zero* else if (*p* ⩽ *i*) && (*i* < *p* + *m*1) && (*p* ⩽ *j*) && (*j* < *p* + *m*2) then (*input2d* |[*i* − *p*]) |[*j* − *p*] else *zero*))) ++ *A*(*vwt* “*out*”, *mkvPar* (λ*i*0. *mkvPar* (λ*j*0. *sum* (*mkvSeq* (λ*i*1. *sum* (*mkvSeq* (λ*j*1. *mul* (*v*0|[*i*0 × *s* + *i*1]) |[*j*0 × *s* + *j*1] (*kernel2d* |[*i*1]) |[*j*1])))))))).

其中“++”是字符串的连接. 此时整个表达式的翻译分为 2 部分：第 1 部分是针对填充操作的翻译，第 2 部分是针对卷积的翻译.

⑦第 1 部分针对填充操作的翻译，分别应用 2 次 A10(AmkvSeq)和 1 次 A14(Aif)，并展开中间命令式组合子 *mkvSeql*，以及 2 次显示翻译规则(6)(CGfor)、1 次显示翻译规则(2)(CGif)和规则(6)(CGfor)即可得到相应命令式 C 代码.

⑧第 2 部分卷积的翻译. 首先，应用翻译器 *A* 的重写转换规则 A11(AmkvPar)，并展开 *mkvParl*，将 *mkvPar* 转换为并行执行的命令式循环操作：

CG_comm (*parfor* (*vwt* “*out*”) (λ*i* *o*. *A*(*o*, *mkvPar* (λ*j*0. *sum* (*mkvSeq* (λ*i*1. *sum* (*mkvSeq* (λ*j*1. *mul* (*v*0|[*i* × *s* + *i*1]) |[*j*0 × *s* + *j*1] (*kernel2d* |[*i*1]) |[*j*1])))))))).

⑨接着，应用显示翻译规则(7)(CGparfor)将命令式原语 *parfor* 显示翻译为字符串：

“#pragma omp parallel for

for (int *i*2=0; *i*2 < ((((*m*1) − (*n*1)) + ((2)×(*p*))) / (*s*)) + (1); *i*2 += 1) {” ++ *A*((*vwt* “*out*”) |{*i*2}, *mkvPar* (λ*j*0. *sum* (*mkvSeq* (λ*i*1. *sum* (*mkvSeq* (λ*j*1. *mul* (*v*0|[*i*2 × *s* + *i*1]) |[*j*0 × *s* + *j*1] (*kernel2d* |[*i*1]) |[*j*1])))))).

⑩继续翻译后面的表达式. 应用翻译器 *A* 的重写转换规则 A11(AmkvPar)，展开 *mkvParl*，并应用显示翻译规则(7)(CGparfor)得到：

“#pragma omp parallel for

for (int *i*3=0; *i*3 < ((((*m*2) − (*n*2)) + ((2)×(*p*))) / (*s*)) + (1); *i*3 += 1) {” ++ *A*(((*vwt* “*out*”) |{*i*2}) |{*i*3}, *sum* (*mkvSeq* (λ*i*1. *sum* (*mkvSeq* (λ*j*1. *mul* (*v*0|[*i*2 × *s* + *i*1]) |[*i*3 × *s* + *j*1] (*kernel2d* |[*i*1]) |[*j*1])))))).

⑪应用翻译器 *A* 的重写转换规则 A13(Asum)，并展开命令式组合子 *suml* 和翻译器 *C*：

CG_comm (*new* (λ*tmp*. *w* |:= *zero*; *for* (λ*i*. *A*(*w*, *add* (*mkvSeq* (λ*i*1. *sum* (*mkvSeq* (λ*j*1. *mul* (*v*0|[*i*2 × *s* + *i*1]) |[*i*3 × *s* + *j*1] (*kernel2d* |[*i*1]) |[*j*1])))) |[*i*] *v*)) ; *A*(((*vwt* “*out*”) |{*i*2}) |{*i*3}, *v*))).

⑫应用命令的显示翻译规则(5)(CGnew)，声明 1 个数值型的临时变量. 此时，剩余的表达式包含 3 个命令的顺序执行，重复应用显示翻译规则(3)(CGseq)得到：

“float *v*4; ” ++ *CG_comm* (*v*4 |:= *zero*) ++ *CG_comm* (*for* (λ*i*. *A*(*v*4, *add* (*mkvSeq* (λ*i*1. *sum* (*mkvSeq* (λ*j*1. *mul* (*v*0|[*i*2 × *s* + *i*1]) |[*i*3 × *s* + *j*1] (*kernel2d* |[*i*1]) |[*j*1])))) |[*i*] *v*4))) ++ *CG_comm* (((*vwt* “*out*”) |{*i*2}) |{*i*3} |:= *v*4).

⑬剩余的命令中，第 1 个和第 3 个是简单的赋值命令，应用显示翻译规则(4)(CGassign)即可完成翻译. 针对第 2 个命令的翻译，首先应用显示翻译规则(6)(CGfor)得到：

“for(int *i*5 = 0; *i*5 < *n*1; *i*5 += 1) {” ++ *A*(*v*4, *add* (*mkvSeq* (λ*i*1. *sum* (*mkvSeq* (λ*j*1. *mul* (*v*0|[*i*2 × *s* + *i*1]) |[*i*3 × *s* + *j*1] (*kernel2d* |[*i*1]) |[*j*1])))) |[*i*5] *v*4).

⑭接着，应用 3.1 节中 *mkvSeq* 与 *idx* 之间的引理 *idx_mkvSeq* 得到：

A(*v*4, *add* (*sum* (*mkvSeq* (λ*j*1. *mul* (*v*0|[*i*2 × *s* + *i*5]) |[*i*3 × *s* + *j*1] (*kernel2d* |[*i*5]) |[*j*1]))) *v*4).

⑮应用翻译器 *A* 的重写转换规则 A2(Aadd)得到：

C(*sum* (*mkvSeq* (λ*j*1. *mul* (*v*0|[*i*2 × *s* + *i*5]) |[*i*3 × *s* + *j*1] (*kernel2d* |[*i*5]) |[*j*1])), λ*x*. *C*(*v*4, λ*y*. *v*4 |:= *add x y*)).

⑯应用翻译器 *C* 的重写转换规则 C13(Csum)并展开 *suml* 和翻译器 *C* 的定义得到：

CG_comm (*new* (λ*tmp*. *w* |:= *zero*; *for* (λ*i*. *A*(*w*, *add* (*mkvSeq* (λ*j*1. *mul* (*v*0|[*i*2 × *s* + *i*5]) |[*i*3 × *s* + *j*1] (*kernel2d* |[*i*5]) |[*j*1])))|[*i*] *v*) ; (*v*4 |:= *add v v*4))).

⑰接下来，重复步骤⑪~⑭即可将表达式中剩余的所有函数式原语转换为命令式原语，最后应用显示翻译规则翻译为 OpenMP C 代码的字符串.

从函数式卷积表达式到 OpenMP C 代码的翻译在 Coq 中的验证过程如图 8 所示.

本文成功从函数式神经网络算子表达式出发，生成了一批神经网络算子的 OpenMP C 代码.

我们将生成的神经网络算子 C 代码与手工编写的算子 C 代码，以及顺序执行的算子 C 代码与带 OpenMP 并行执行的算子 C 代码进行性能比较. 本次测试使用的处理器为 Intel® i7-12700H @ 2.30GHz(6 核+8 核). 主要测试结果如表 7 所示.

实验结果表明，使用代码生成工具生成的神经网络算子 C 代码与手工编写的算子 C 代码在质量上可以相媲美，执行结果一致且性能相当. 同时，该工

```
Goal CG_comm (A(vwt "out",Conv2d p s kernel2d input2d)) 0 =
  "float v0[((2) * (p)) + (m1)][((2) * (p)) + (m2)];
for(int i1=0; i1<((2) * (p)) + (m1); i1 += 1)
  { for(int i2=0; i2<((2) * (p)) + (m2); i2 += 1)
  { if (i1 < p && i2 < p)
  { v0[i1][i2] = 0 ;}
else
  { if (p <= i1 && i1 < (p) + (m1) && p <= i2 && i2 < (p) + (m2))
  { v0[i1][i2] = input[(i1) - (p)][(i2) - (p)] ;}
else
  { v0[i1][i2] = 0 ;}}}}
#pragma omp parallel for
for (int i2=0; i2 < ((((m1) - (n1)) + ((2) * (p))) / (s)) + (1); i2 += 1)
  { #pragma omp parallel for
for (int i3=0; i3 < ((((m2) - (n2)) + ((2) * (p))) / (s)) + (1); i3 += 1)
  { float v4;
v4 = 0 ;
for(int i5=0; i5<n1; i5 += 1)
  { float v6;
v6 = 0 ;
for(int i7=0; i7<n2; i7 += 1)
  { v6 = ((v0[((i2) * (s)) + (i5)][((i3) * (s)) + (i7)] * kernel[i5][i7]) + v6) ;}
v4 = (v6 + v4) ;}
out[i2][i3] = v4 ;}}".
Proof.
  unfold Conv2d. rewrite Alet. unfold letl. unfold Padding2d.
  rewrite CmkvSeq. rewrite CGnew. repeat rewrite CGseq. rewrite AmkvSeq.
  unfold mkvSeql. rewrite CGfor. rewrite AmkvSeq. unfold mkvSeql. rewrite CGfor.
  repeat rewrite Aif. repeat rewrite CGif. unfold A. repeat rewrite CGassign.
  repeat rewrite AmkvPar. unfold mkvParl. rewrite CGparfor. rewrite AmkvPar.
  unfold mkvParl. rewrite CGparfor. rewrite Asum. unfold suml. unfold C.
  rewrite CGnew. repeat rewrite CGseq. rewrite CGfor. rewrite Aadd. rewrite idx_mkvSeq.
  rewrite Csum. unfold suml. unfold C. rewrite CGnew. repeat rewrite CGseq.
  rewrite CGfor. rewrite Aadd. unfold C. rewrite idx_mkvSeq. repeat rewrite CGassign.
  repeat rewrite CGadd. repeat rewrite CGmul. str_simpl.
  auto. auto. auto.
Qed.
```

Fig. 8 Translation process of Conv2d in Coq

图 8 Coq 中的 Conv2d 翻译过程

Table 7 Test Results

表 7 测试结果

算子操作	张量规模	顺序执行时长/s		并行执行时长/s		加速比	
		生成代码	手工编写	生成代码	手工编写	生成代码	手工编写
Conv2d (p=1, s = 1)	700×700 300×300	5.910 935	6.978 041	0.720 788	0.823 746	8.20	8.47
Avgpool2d (k = 2)	8 000×8 000	0.130 755	0.142 020	0.016 803	0.059 553	7.78	2.38
Maxpool2d (k = 2)	8 000×8 000	0.141 180	0.065 849	0.022 787	0.047 422	6.20	1.39
ReLU2d	8 000×8 000	0.081 935	0.082 872	0.022 103	0.023 369	3.71	3.55
Linear2d	2 000×3 000 3 000×4 000	13.760 259	13.848 600	1.355 773	1.418 254	10.15	9.76

具生成的多核并行 OpenMP C 算子相比于顺序执行的神经网络算子 C 代码表现出了显著的性能提升，速度提高了 4~10 倍. 值得注意的是，本文生成的神经网络算子 C 代码具有高安全性，能够有效避免人工编写代码时常见的下标越界和存储分配错误.

6 总结与展望

本文实现了一个基于 Coq 的带类型函数式深度学习框架 Ionia. 该框架具有足够大的表达能力，所构建的模型涵盖了生成对抗网络、卷积神经网络及循环神经网络等. 同时，实验验证了该学习框架能够自动静态检测到神经网络模型中的张量形状不匹配错误，且无任何时间开销. 此外，本文设计并实现了一系列重写转换规则，用于将函数式神经网络算子表达式翻译为命令式 C 代码. 最终成功生成了一批高可信、高安全的神经网络算子 OpenMP C 代码，相比顺序执行的算子 C 代码，速度有了大幅提升.

后续计划包含以下 3 部分：1)在 Coq 定理证明器中严格证明从函数式神经网络算子表达式到命令式原语的重写转换规则的正确性，以此严格证明生成的神经网络算子 C 代码的可靠性；2)在函数式层面对神经网络算子进行优化及正确性验证，包括各种优化调度策略，如循环融合、循环拆分、循环重组等 Halide 调度指令，并将优化后的算子翻译为相应的 C 代码，以保证生成代码的高效可靠性；3)将代码生成扩展到 GPU 和其他 AI 加速器上，使得生成的代码能够充分利用 AI 硬件的并行计算能力. 本文所做的工作为构建高可靠、高安全、高性能的 AI 编译器提供了一种新的方案.

作者贡献声明：张晓丽负责完成实验并撰写论文；陈钢提出实验方案设计思路并修改论文；麻莹莹提出基础方案设计；胡萍提供方案思路.

参 考 文 献

[1] Paszke A, Gross S, Massa F, et al. PyTorch: An imperative style, high-performance deep learning library[J]. Advances in Neural Information Processing Systems, 2019, 32: 8024−8035

[2] Abadi M, Agarwal A, Barham P, et al. TensorFlow: Large-scale machine learning on heterogeneous distributed systems[J]. arXiv preprint, arXiv: 1603.04467, 2016

[3] Jia Yangqing, Shelhamer E, Donahue J, et al. Caffe: Convolutional architecture for fast feature embedding[C]//Proc of the 22nd ACM Int Conf on Multimedia. New York: ACM, 2014: 675−678

[4] Gulli A, Pal S. Deep Learning with Keras[M]. Birmingham: Packt Publishing Ltd, 2017

[5] Wu Dangwei, Shen Beijun, Chen Yuting. An empirical study on tensor shape faults in deep learning systems[J]. arXiv preprint, arXiv: 2106.02887, 2021

[6] Jhoo H Y, Kim S, Song W, et al. A static analyzer for detecting tensor shape errors in deep neural network training code[C]//Proc of the 44th ACM/IEEE Int Conf on Software Engineering. New York: ACM, 2022: 337−338

[7] Lagouvardos S, Dolby J, Grech N, et al. Static analysis of shape in TensorFlow programs[C/OL]//Proc of the 34th European Conf on Object-Oriented Programming (ECOOP 2020). Dagstuhl, Germany: Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2020[2024-03-10]. https://drops.dagstuhl.de/storage/00lipics/lipics-vol166-ecoop2020/LIPIcs.ECOOP.2020.15/LIPIcs.ECOOP.2020.15.pdf

[8] Dolby J, Shinnar A, Allain A, et al. Ariadne: Analysis for machine learning programs[C/OL]//Proc of the 2nd ACM SIGPLAN Int Workshop on Machine Learning and Programming Languages. New York: ACM, 2018[2024-03-10]. https://arxiv.org/pdf/1805.04058

[9] Hattori M, Sawada S, Hamaji S, et al. Semi-static type, shape, and symbolic shape inference for dynamic computation graphs[C]//Proc of the 4th ACM SIGPLAN Int Workshop on Machine Learning and Programming Languages. New York: ACM, 2020: 11−19

[10] Ji Zeyu, Zhang Xingjun, Fu Zhe, et al. Performance-aware based dynamic batch size SGD for distributed deep learning framework[J]. Journal of Computer Research and Development, 2019, 56(11): 2396−2409 (in Chinese)
(纪泽宇, 张兴军, 付哲, 等. 分布式深度学习框架下基于性能感知的 DBS-SGD 算法[J]. 计算机研究与发展, 2019, 56(11): 2396−2409)

[11] Zhao Tian, Huang Xiaobing, Cao Yu. DeepDSL: A compilation-based domain-specific language for deep learning[J]. arXiv preprint, arXiv: 1701.02284, 2017

[12] Wang Liang, Zhao Jianxin. Architecture of Advanced Numerical Analysis Systems: Designing a Scientific Computing System Using OCaml[M]. Berlin: Springer, 2023

[13] INRIA. The Coq proof assistant[EB/OL]. 1997[2022-09-15]. https://coq.inria.fr/

[14] Steuwer M, Fensch C, Lindley S, et al. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance OpenCL code[J]. ACM SIGPLAN Notices, 2015, 50(9): 205−217

[15] Atkey R, Steuwer M, Lindley S, et al. Strategy preserving compilation for parallel functional code[J]. arXiv preprint, arXiv: 1710.08332, 2017

[16] Ma Yingying, Chen Gang. Matrix code generation technology based on Coq[J]. Journal of Software, 2022, 33(6): 2224−2245 (in Chinese)
(麻莹莹, 陈钢. 基于 Coq 的矩阵代码生成技术[J]. 软件学报, 2022, 33(6): 2224−2245)

[17] Verma S, Su Zhendong. ShapeFlow: Dynamic shape interpreter for tensorflow[J]. arXiv preprint, arXiv: 2011.13452, 2020

[18] Huang A, Hashimoto J, Paszke A, et al. Hasktorch[EB/OL]. 2019[2024-04-13]. https://github.com/hasktorch/hasktorch

[19] Paszke A, Gross S, Chintala S, et al. Automatic differentiation in PyTorch[J/OL]. 2017[2024-05-15]. https://api.semanticscholar.org/CorpusID:40027675

[20] Thompson S. Haskell: The Craft of Functional Programming[M]. Boston: Addison-Wesley, 2011

[21] Roesch J, Lyubomirsky S, Weber L, et al. Relay: A new IR for machine learning frameworks[C]//Proc of the 2nd ACM SIGPLAN Int Workshop on Machine Learning and Programming Languages. New York: ACM, 2018: 58−68

[22] Roesch J, Lyubomirsky S, Kirisame M, et al. Relay: A high-level compiler for deep learning[J]. arXiv preprint, arXiv: 1904.08368, 2019

[23] Hattori M, Kobayashi N, Sato R. Gradual tensor shape checking[C]//Proc of the 32nd European Symp on Programming. Berlin: Springer, 2023: 197−224

[24] Eremondi J, Tanter É, Garcia R. Approximate normalization for gradual dependent types[J]. Proceedings of the ACM on Programming Languages, 2019, 3(88): 1−30

[25] Lehmann N, Tanter É. Gradual refinement types[J]. ACM SIGPLAN Notices, 2017, 52(1): 775−788

[26] Vazou N, Tanter É, Van Horn D. Gradual liquid type inference[J]. Proceedings of the ACM on Programming Languages, 2018, 2(132): 1−25

[27] Liu A, Bernstein G L, Chlipala A, et al. Verified tensor-program optimization via high-level scheduling rewrites[J]. Proceedings of the ACM on Programming Languages, 2022, 6(55): 1−28

[28] Ragan-Kelley J, Barnes C, Adams A, et al. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines[J]. ACM SIGPLAN Notices, 2013, 48(6): 519−530

[29] Danvy O, Millikin K, Nielsen L R. On one-pass CPS transformations[J]. Journal of Functional Programming, 2007, 17(6): 793−812

[30] Ma Zhenwei, Chen Gang. Matrix formalization based on Coq record[J]. Computer Science, 2019, 46(7): 139−145 (in Chinese)
(马振威, 陈钢. 基于 Coq 记录的矩阵形式化方法[J]. 计算机科学, 2019, 46(7): 139−145)

[31] Ma Yingying, Ma Zhenwei, Chen Gang. Formalization of block matrix operations based on Coq[J]. Journal of Software, 2021, 32(6):

1882−1909 (in Chinese)
(麻莹莹, 马振威, 陈钢. 基于 Coq 的分块矩阵运算的形式化[J]. 软件学报, 2021, 32(6): 1882−1909)
[32] Shi Zhengpu, Xie Guojun, Chen Gang. CoqMatrix: Formal matrix library with multiple models in Coq[J]. Journal of Systems Architecture, 2023, 143: 102986
[33] Mahboubi A, Tassi E. Mathematical components[EB/OL]. 2021 [2024-05-15]. https://math-comp.github.io/
[34] Boldo S, Lelay C, Melquiond G. Coquelicot: A user-friendly library of real analysis for Coq[J]. Mathematics in Computer Science, 2015, 9: 41−62
[35] LeCun Y, Bottou L, Bengio Y, et al. Gradient-based learning applied to document recognition[J]. Proceedings of the IEEE, 1998, 86(11): 2278−2324
[36] Tai Cheng, Xiao Tong, Zhang Yi, et al. Convolutional neural networks with low-rank regularization[J]. arXiv preprint, arXiv: 1511.06067, 2015
[37] He Kaiming, Zhang Xiangyu, Ren Shaoqing, et al. Deep residual learning for image recognition[C]//Proc of the 29th IEEE Conf on Computer Vision and Pattern Recognition. Piscataway, NJ: IEEE, 2016: 770−778
[38] Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition[J]. arXiv preprint, arXiv: 1409.1556, 2014
[39] DiPietro R, Hager G D. Deep learning: RNNs and LSTM[M]//Handbook of Medical Image Computing and Computer Assisted Intervention. San Diego: Elsevier, 2020: 503−519
[40] McCann B. github[EB/OL]. [2023-09-05]. https://github.com/pytorch/examples
[41] Spolsky J. stackoverflow [EB/OL]. [2024-02-22]. https://stackoverflow.com/questions



Zhang Xiaoli, born in 1998. Master. Student member of CCF. Her main research interests include formal methods, Coq, and AI compiler.
张晓丽, 1998 年生. 硕士. CCF 学生会员. 主要研究方向为形式化方法、Coq、AI 编译器.



Chen Gang, born in 1958. PhD, professor, PhD supervisor. His main research interests include programming language design, higher order theorem proving, and AI compiler.
陈 钢, 1958 年生. 博士, 教授, 博士生导师. 主要研究方向为程序语言设计、高阶定理证明、AI 编译器.



Ma Yingying, born in 1997. Master. Her main research interests include formal engineering mathematics, functional language, and Coq.
麻莹莹, 1997 年生. 硕士. 主要研究方向为形式化工程数学、函数式语言、Coq.



Hu Ping, born in 1965. PhD, Chief Software Engineer. Her main research interests include type theory, compiler optimization, and high performance computing.
胡 萍, 1965 年生. 博士, 首席软件工程师. 主要研究方向为类型理论、编译器优化、高性能计算.