Ideal Languages for Generative AI

Gavin Osborn
Computer Science
Xavier University
Cincinnati, USA
contact@gavosb.com

***Abstract*—*Generative AI suffers from numerous issues, particularly in code assistant tasks regarding accuracy, conciseness, and efficiency. Different language paradigms are examined for their suitability as optimal target languages, with functional languages being found to be most suitable. Machine Learning is examined in a broader sense with domain-specific optimizations at the compiler level, bringing language design for generative AI to a cohesive end-to-end whole.*

Keywords—machine learning, domain specific languages, generative AI, artificial intelligence, deep learning compilers

I. INTRODUCTION

Machine learning (ML) is a subset of AI that fits mathematical models to observed data. Generative AI is a subset of machine learning that creates new data based upon the data which any given model had been trained. Large Language Models (LLMs) are a further subset of Generative AI, and of great importance today. Because every subset carries with it the properties of its supersets, any improvement upon a broader set will lead to improvements in those which are more specific. At the level of Generative AI and their most salient form as LLMs, there is great and wide-ranging opportunity for improvement in many ways. We analyze a collection of possible enhancements in both compilation and generative AI output languages. Principally, speed optimizations by way of structural representations and resource distribution in compilation, and various proper paradigm decisions in a target language that match how the target language is learned by the generative AI. Four languages are considered; OCaml, Python, Relay, and Triton.

A principle way to optimize Machine Learning code is in optimizing the compiler used for ML tasks. A classical compiler takes a source language and outputs executable machine code. Typically, this is done in four stages with a different representation for each stage; lexical analysis and a token stream, a parser and an abstract syntax tree (AST), the semantic analyzer and an annotated AST, then code generation to target code. An intermediate representation, such as an intermediate language, is often used after the semantic analysis stage; this is a representation of the AST that is more suitable for generating the target language code. Proper selection of an intermediate representation is critical to ML applications in particular, due to how certain structures need to be represented, though there are other stages of compilation that can be optimized along the way, and in which the intermediate representation is dependent upon.

Generative AI does not compile its output code for code-assistant tasks, such as when a prompt is given to an LLM model that requests a program written in a particular target language. Rather, the output is derived from the core of machine learning, which is in learned patterns from observed data. Thus, if we want to optimize certain metrics such as feature completeness, accuracy, and conciseness, we must keep in mind how such code is derived and choose a language that is easier for such metrics to be optimized. That is, the language must be within a paradigm that is learned correctly with ease for a machine learning model.

II. PARADIGMS AND LEARNING

Though every programming language can at most be Turing complete, with the possible exception of unknown technologies and of quantum languages, there are still different paradigms that describe what we can do, as a constraint on the style of the Turing-complete expressiveness that the language allows.

There are two main supersets of programming paradigms; the declarative and the imperative. The latter is more traditional, and closely replicates the von Neumann model. Within the imperative, there exist the procedural languages as well as the object-oriented languages, the latter of which can also be concurrent. Within the declarative, there are the functional, logic, and concurrent languages.

The imperative languages operate based around mutable state; every computation is a change in state because variables are abstractions of the von Neumann store itself. Declarative languages strive to be more mathematical; the programmer does not describe exactly the sequence of steps to take in computation, but only describes high-level expressions.

Declarative languages are easier for a machine learning model to learn than imperative languages. The reason for this is because declarative languages hide control and do not obfuscate the underlying ideas behind excessive features that must be learned.

The more features that must be learned by a neural network, the greater the complexity of its architecture. Not only must hyper-parameters be tuned appropriately, but more time must be spent training and more hidden layers must be utilized. Capturing the expressiveness of a programming language is a very complex problem that requires a very complex model. Current generative AI models frequently generate code with errors, such as blindly plagiarizing chunks of code from training data and incorrectly identifying scope [1]. Choosing a language that provides less confusion over code structure is paramount.

When learning a declarative language, all that must be learned is what is described by the language, rather than its control flow or the tracking of how state is managed. This allows for a more simplified model with fewer possibilities to not learn required information, and one which can be trained more easily with greater accuracy.

Functional languages are best among the declarative languages because they cannot only provide the simplicity of the declarative paradigm when compilation is not involved, but can also provide very direct flow of information with stateless functions. Thus there are much fewer features to learn in regard to how different functions communicate with each other; referential transparency provides greater simplicity in how the program is described, which is all the neural network is concerned with.

III. Compiler Optimisation

Generative AI essentially functions as a transpiler from natural language to a given programming language. Ambiguities and discrepancies in expressiveness between classical programming languages and natural language make such translations difficult [2], so optimal choice of target language is vital for proper code generation whenever the language does not matter to the programmer for other reasons.

Another area of optimization is in optimizing the machine learning framework itself, both in the language that they are programmed in, and how the models are represented and executed. Machine learning utilizes linear algebra algorithms and data structures extensively, thus languages and compilers that handle such structures well are more suitable for their use in machine learning tasks.

Deep Learning frameworks have typically relied on computational graph intermediate representations, to which optimizations were applied at a high level. Deep Learning compilers such as TVM aimed to resolve this issue, opting to provide end-to-end optimization from the lowest level representation to the highest rather than merely graph-level optimization [3].

Relay is a Deep Learning compiler framework and frontend for TVM that takes a high level model definition and converts it into a lower level representation similar to a computational graph through the use of an expressive, functional, and statically typed intermediate representation [4]. This representation is based upon the ML language family, which provides the relay language with a functional core upon which a novel tensor type system is added, and operators are introduced as expressions. These features become the primary value type and language primitives respectively, allowing for a seamless compilation process for deep learning models that operate on tensors. Different aspects of the language are represented at different levels; operators, for instance, are represented by a lower level representation than tensors.

The resulting compiled deep learning model is derived from an enriched representation that allows for an enhanced type system and higher-order functions, whereas most existing representations are merely a computational graph. Compilation significantly improves performance in most areas, though for the Gluon RNN MxNet outperforms Relay, because MxNet uses a hard-coded optimization to unroll the RNN, while Relay expresses it as a loop without unrolling (transforming certain structures). That is, even with compilation, carefully crafted optimizations like loop unrolling in MxNet's Gluon RNN can still lead to better performance in specific cases [4]. Therefore, this framework requires domain-specific optimizations, which can be provided by Relay's extension support.

Functional languages are easily adaptable to machine learning applications [4], though not all compiler frameworks work within a functional, nor even declarative, paradigm. Triton, an intermediate language and compiler for tile-based neural networks, is an imperative C-based language. This framework addresses optimization issues seen in previous frameworks such as TVM by way of automatic tiling, which segments computations of some data structures into smaller components that can be distributed for parallel processing[5].

Despite their differences, both frameworks can benefit from universal optimizations in their underlying structure. MLIR, for instance, provides an extensible compiler infrastructure that can support many paradigms and domain-specific languages. The infrastructure allows for even broader enhancements such as intermediate representations that mix different levels of abstraction and different concepts together to maintain higher-level abstraction whenever possible, only lowering when needed. This enables greater control over the representation and its use, such as custom accelerators that reuse high level structure along with low level primitive vector instructions that the accelerator may use [6].

IV. Explorations in Generative AI

The ideal language in which we may write machine learning applications ought to be chosen the same way in which we may choose the intermediate language for compilation. The language should be suitable for common data structures and easily transferable to an intermediate representation that can be transferred to a machine learning compiler. The natural answer ought to be a language made precisely for use with these compilers. However, such domain-specific languages often lack libraries that make programming harder, and are generally less known to the programming community and with less familiarity. Thus, Python should still be considered, alongside OCaml.

Python, a multi-paradigm language, has extensive support for common machine learning frameworks, such as TensorFlow, MxNet, and PyTorch. However, the language is interpreted and requires extensive optimization of the models involved in comparison to domain-specific languages, though future advancements in deep-learning compilers may be able to circumnavigate this issue by offering greater support for control flow during pre-and-post processing [7]. OCaml does not suffer from this problem to the same extent, and has excellent support for scientific programming tasks, of which there is overlap with machine learning in certain algorithms like gradient descent.

Google Gemini outputted code upon request for simple linear regression in Python with NumPy, OCaml with OWL, and TVM Relay IR. The code was cross-checked for major errors with appropriate documentation. Note that linear regression has a closed form solution:

$$M = \frac{X^T Y}{X^T X}$$

Given the prompt, “Write a closed form solution to linear regression using (the given language)”, Google Gemini provided the following implementations:

```
import numpy as np

def linear_regression(x, y):
"""
Performs linear regression on data poin
ts(x, y).

Args:
x: A list or NumPy array of independent
variable values.
y: A list or NumPy array of dependent
variable values.

Returns:
A tuple containing the slope (m) and
intercept (b) of the fitted line.
"""
m = np.mean((x - np.mean(x))
* (y - np.mean(y))) / np.var(x)

b = np.mean(y) - m * np.mean(x)
return m, b
```

Figure 1: Closed form Linear Regression in Python, generated by Google Gemini. The slope and intercept are calculated using NumPy.

```
open Owl

let linear_regression x y =
  (* Calculate the slope (m) *)
  let m =
    let x_mean = Mat.mean x in
    let y_mean = Mat.mean y in
    let numerator = Mat.(x - x_mean) *
        (y - y_mean) in
    Mat.mean numerator / Mat.var x
  in

  (* Calculate the intercept (b) *)
  let b = Mat.mean y - m * Mat.mean x
  in

  (m, b)
;;
```

Figure 2: Closed form Linear Regression in OCaml, using the OWL library. Generated by Google Gemini.

```
fn@(x: Tensor[(i32, d)], y: Tensor[i32])
-> (Tensor[i32], Tensor[i32]) {
  # Get the shapes of x and y
  let x_shape = tvm.ir.shape_of(x)
  let y_shape = tvm.ir.shape_of(y)

  # Ensure x and y have compatible shapes
  (same number of elements in the first
  dimension)
  assert(tvm.ir.Equal(tvm.ir.Nth(x_shape,
  0), tvm.ir.Nth(y_shape, 0)))

  # Define weights (slope) and bias (inte
  rcept) as Relay tensors
  let w = tvm.ir.Var("w", tvm.ir.TensorTy
  pe(tvm.ir.Kind.Float32, (d,)))  # Weigh
  ts with dimension d (number of features)
  let b = tvm.ir.Var("b", tvm.ir.TensorTy
  pe(tvm.ir.Kind.Float32, (1,)))

  # Calculate the linear prediction
  let prediction = tvm.ir.Call(tvm.ir.Op.
  nn.dot, [x, tvm.ir.Reshape(w, (tvm.ir.T
  ensorExpr(tvm.ir.PrimExpr.shape_
  of(x)[1]), d))], tvm.ir.Kind.Float32)
  let prediction = tvm.ir.Add(prediction,
         b)

  # Calculate mean squared error (MSE) as
         a loss metric
  let mse = tvm.ir.Mean(tvm.ir.Square(tvm
        .ir.Subtract(prediction, y)))

  # Return both prediction and MSE
  (prediction, mse)
}
```

Figure 3: Closed form linear regression in a language similar to TVM Relay, but not actually any existent language and likely an amalgamation of learned patterns from limited data. Generated by Google Gemini.

The TVM Relay code is incorrect and largely a hallucination of a correct intermediate representation function signature and the Python frontend.[1] The TensorType representation is interesting, however, because it alludes to Relay's optimization of lower level types while being described in a high level language. It is likely that the model was simply not trained on enough data for the relatively new and niche language.

The Python code is the shortest, and offloaded to the highly optimized NumPy library, but even at its level of functional abstraction pales in comparison to the readability of OCaml's declarative style. However, being short, the code could still be learned quite easily, though possibly recombined with other code poorly because of the abstraction that is not necessarily described in a declarative manner. For end-user code, it may be best to choose OCaml when possible because of the relative lack of performance overhead. Performance increases through deep learning compilers can still be a factor that weighs heavily in favor of languages such as Python, particularly when existing libraries are taken into account.

V. CONCLUSION

Generative AI for code generation must produce code that is correct and optimal, in terms of performance, conciseness, accuracy, and readability. The pathway to achieving this is in choosing an optimal target language, along with efficient and suitable intermediate representations. Research on the correct paradigms to choose for these languages is inconclusive, but suggestive of declarative languages, particularly functional ones, being the most suitable as both target languages for code generation and intermediate representations in compiler frameworks. Though the different aspects of any given language must be evaluated in choosing a language for any given task, domain-specific optimizations and suitable language paradigms can aid in making a sound decision in choosing the ideal language for generative AI problems and use cases.

REFERENCES

[1] A. Sarkar, et al. "What is it like to program with artificial intelligence?." arXiv preprint arXiv:2208.06213, 2022.

[2] A. Zúñiga, et al., “Towards a natural language compiler,” in *Advances in Computational Intelligence: 17th Mexican International Conference on Artificial Intelligence*, MICAI 2018, Guadalajara, Mexico, October 22–27, 2018, Proceedings, Part II, vol. 17, Springer International Publishing, 2018.

[3] T. Chen, et al., “TVM: An automated End-to-End optimizing compiler for deep learning,” in *13th USENIX Symposium on Operating Systems Design and Implementation* (OSDI 18), 2018.

[4] J. Roesch, et al., “Relay: A high-level compiler for deep learning,” arXiv preprint arXiv:1904.08368, April 2019.

[5] P. Tillet, H.-T. Kung, and D. Cox, “Triton: an intermediate language and compiler for tiled neural network computations,” in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2019.

[6] C. Lattner, et al., “MLIR: A compiler infrastructure for the end of Moore’s law,” arXiv preprint arXiv:2002.11054, February 2020.

[7] X. Li, et al., “Mlog: Towards declarative in-database machine learning,” *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1933-1936, 2017.

[1] https://tvm.apache.org/docs/reference/api/python/ir.html