ScalaGrad: A Statically Typed Automatic Differentiation Library for Safer Data Science

Benjamin Meyer*†, Thilo Stadelmann*§, Marcel Lüthi‡
mebr@zhaw.ch, stdm@zhaw.ch, marcel.luethi@unibas.ch
*Centre for Artificial Intelligence, ZHAW School of Engineering, Winterthur, Switzerland
†Institute of Neuroinformatics, ETH & University of Zurich, Zurich, Switzerland
‡Departement Mathematik und Informatik, University of Basel, Basel, Switzerland
§European Centre for Living Technology (ECLT), Ca' Bottacin, Venice, Italy
§Fellow, ECLT (European Centre for Living Technology, Venice, Italy) and Senior Member, IEEE

***Abstract*—While the data science ecosystem is dominated by programming languages that do not feature a strong type system, it is widely agreed that using strongly typed programming languages leads to more maintainable and less error-prone code and ultimately more trustworthy results. We believe Scala 3 would be an excellent contender for data science in a strongly typed language, but it lacks a general automatic differentiation library, e.g., for gradient-based learning. We present ScalaGrad, a general and type-safe automatic differentiation library designed for Scala. It builds on and improves a novel approach from the functional programming community using immutable duals, which is conceptually simple, asymptotically optimal and allows differentiation of higher-order code. We demonstrate the ease of use, robust performance, and versatility of ScalaGrad through its applications to deep learning, higher-order optimization, and gradient-based sampling. Specifically, we show an execution speed comparable to PyTorch for a simple deep learning use case, capabilities for higher-order differentiation, and opportunities to design more specialized libraries decoupled from ScalaGrad. As data science challenges evolve in complexity, ScalaGrad provides a pathway to harness the inherent advantages of strongly typed languages, ensuring both robustness and maintainability.**

***Index Terms*—Automatic Differentiation, Scala 3, ScalaGrad**

I. Introduction

The Python programming language dominates the field of data science [1]. Python is easy to learn, fast to experiment with, and has a vast ecosystem of libraries like scikit-learn [2], TensorFlow [3], PyTorch [4] and others [5]–[7]. Despite Python's strengths, it has a dynamic type system that hinders maintainability and trustworthiness in complex applications. With the growing complexity of data science challenges, the case for statically typed languages becomes compelling.

Scala 3 has a static, powerful, and sound type system [8]. It propagates a functional programming paradigm without losing the flexibility for side effects if needed, for example, mutable state, which can be necessary for performance optimization. Scala 3's concise and braceless syntax allows for fast experimentation and eases the switch from Python. Scala has broad tool support, as it is an established language for distributed systems and big data applications. However, Scala 3 lacks a general automatic differentiation library, e.g., to facilitate ubiquitous gradient-based learning [9].

In this paper, we introduce ScalaGrad, a general and type-safe automatic differentiation library written in and designed for Scala. ScalaGrad builds on a novel approach introduced in the functional programming community by Krawiec et al. [10], using immutable duals, resulting in easy-to-reason code while being asymptotically efficient. Moreover, ScalaGrad improves on this approach by removing the necessity of a state monad, making the approach viable for Scala and allowing straightforward higher-order differentiation support. ScalaGrad is easy to use as it can differentiate over general control flow statements like if-else statements, loops, monads, or exceptions. It supports different automatic differentiation modes and has robust performance by hardware-accelerated linear algebra operations using the Breeze library [11]. Demonstrating ScalaGrad's versatility, we applied it to deep learning, higher-order optimization, and gradient-based sampling. Specifically, the deep learning application demonstrates ScalaGrad's performance, calculating 28,630 partial derivatives 938 times in around 5 seconds. The higher-order optimization application demonstrates ScalaGrad's higher-order differentiation capabilities. And the gradient-based sampling application indicates the unique advantage of ScalaGrad's *type-safe* differentiation, as the algorithm was implemented decoupled from ScalaGrad while maintaining type safety. ScalaGrad and all applications have been open-sourced at https://github.com/benikm91/scala-grad.

As automatic differentiation is pivotal for gradient-based applications, ScalaGrad represents a significant step towards realizing a vision of type-safe data science in the Scala ecosystem.

II. Related Work

Automatic Differentiation. Automatic differentiation usually runs in one of two modes, the forward mode or the reverse mode, with the selection dependent on the structure of the function to differentiate. The forward mode was discovered independently multiple times in the 50s, 60s, and 70s [12]–[16], and the reverse mode was discovered independently multiple times in the 70s and 80s [17]. In recent years, researchers have expanded the theoretical groundwork for automatic differentiation. Elliott [18] highlights that derivatives

```
def square(alg: MatrixAlgebraDSL)(x: alg.Scalar): alg.Scalar = x * x // Define function to derive
import ForwardMode.derive as d                                       // Import derive function from ScalaGrad
val dSquare = d(square)                                              // Derive function
assert(dSquare(BreezeDoubleMatrixAlgebraDSL)(3.0) == 2.0 * 3.0)      // Apply derived function and check result
```

Fig. 1: Minimal example of using ScalaGrad. First, the `square` function is defined using `MatrixAlgebraDSL` to abstract over the multiplication operation. Next, the function is differentiated using ScalaGrad's forward mode, resulting in a new function, `dSquare`, the derivative of `square`.

are linear maps and reduces automatic differentiation to two basic operations: the fork and the chain operation. Based on this idea, Krawiec et al. [10] show an elegant and *immutable* implementation for the reverse mode. The crucial new idea is to use a dual, often used for the forward mode, to track the chain and fork operations in the reverse mode. ScalaGrad builds on those concepts and adds *type-safety* to the automatic differentiation operation using Scala 3's match types [19] to compute the function's derivative type at compile-time. Additionally, ScalaGrad simplifies those concepts, making it straightforward to implement the differentiation mechanism in a differentiable way (an idea taken from JAX [20]), enabling higher-order differentiation.

Python ecosystem. In the Python ecosystem, there exist some automatic differentiation libraries. TensorFlow and PyTorch come bundled with a specialized automatic differentiation engine with syntax narrowly tailored to deep learning: One creates a tensor expression, marks the input tensors as such (the weights), and starts to differentiate from an output tensor (the loss). Over the years, TensorFlow and PyTorch added support for general automatic differentiation, including multiple outputs, the forward mode, and higher-order differentiation. However, the afterthought for general automatic differentiation lead to an overly complex syntax for those more advanced automatic differentiation cases. In recent years, libraries like JAX [21] and functorch [22] have provided a cleaner, more functional syntax. Rather than a tensor expression, one defines a function on tensors and differentiates it by a higher-order function. This idea leads to a natural syntax for multiple outputs, different modes, and higher-order differentiation. Different modes are simply different higher-order functions. Higher-order differentiation is just the chaining of such function calls.

Python 3 supports type hints to improve maintainability and safety when programming [23]. Type hints are optional type annotations that can be used for static code analysis. However, there is no compilation phase enforcing the correctness of types, which in practice leads to code that rarely type-checks [24], and many Python libraries do not use type hints. Disregarding those practical concerns, Python 3's type hinting is additionally limited in expressive power compared to Scala 3's. Python does not support, e.g., singleton types, higher-kinded types, path-dependent types, or match types.

Scala ecosystem. In Scala, there exist automatic differentiation engines bundled with deep learning libraries [25], [26] as well as bindings to TensorFlow [27] and LibTorch [28], [29], making automatic differentiation possible in those domains. However, Scala has no general, stand-alone automatic differentiation library; therefore, it lacks a fundamental building block for applications using derivatives, like deep learning and probabilistic programming with gradient-based sampling.

III. ScalaGrad

We present ScalaGrad, a general automatic differentiation library written in Scala for Scala, built on and improving a novel approach from Krawiec et al. [10]. ScalaGrad supports higher-order differentiation, the forward and reverse mode, and can differentiate over any control flow statements. Numerical operations must be polymorphic to add automatic differentiation capabilities, which ScalaGrad achieves using its own type class. When differentiating a function, ScalaGrad tracks the resulting derivative's type, adding type safety to the automatic differentiation operation.

Linear algebra polymorphism. ScalaGrad must abstract over numerical operations like addition and multiplication to track the derivatives while executing those operations. Additionally, for performance reasons, ScalaGrad abstracts over basic linear operations to implement them efficiently by native routines (using Breeze [11]) while tracking the derivatives. For this, ScalaGrad defines a type class `MatrixAlgebra` with four types: a scalar, a column vector, a row vector, and a matrix. The type class defines operations on those types. Using four types adds some type safety, e.g., the undefined inner product between two column vectors does not exist, rather than throwing a runtime error. Finally, for a more concise syntax, ScalaGrad provides the domain-specific language `MatrixAlgebraDSL` using path-dependent types and wrapping the `MatrixAlgebra` type class. Figure 1 shows a minimal ScalaGrad example using the `MatrixAlgebraDSL` abstraction and the `BreezeDoubleMatrixAlgebraDSL` implementation. `BreezeDoubleMatrixAlgebraDSL` implements the `MatrixAlgebra` operations with the Breeze library with double precision. Implementing linear operations by native routines gives ScalaGrad a robust performance, as discussed in the deep learning application.

Efficient, immutable reverse mode implementation. In the reverse mode, the partial derivatives for a function f are calculated from each output to all inputs. So, first, the operations must be tracked while executing f from the inputs to the outputs, called taping, and then the tape must be run back in reverse order from each output. An essential idea of Krawiec et al. [10] is that it is unnecessary to tape all kinds of operations, but only the chain and the fork operations, representing scaling and addition. The idea works because derivatives are linear maps, and non-linear scaling factors can be eagerly calculated while taping [18]. For an efficient reverse

```
def add(alg: MatrixAlgebraDSL)(x1: alg.Scalar, x2: alg.Scalar): alg.Scalar = x1 + x2  // Define function add
import ForwardMode.derive as d                                                        // Import derive function from ScalaGrad
val dAdd = d(add)                                                                     // Derive function add
val (dx1, dx2) = dAdd(BreezeDoubleMatrixAlgebraDSL)(3.0, 4.0)                         // Compiler knows that dAdd returns two scalars
```

Fig. 2: ScalaGrad tracks the type of the derivative of a function during differentiation. Here, the compiler knows that `dAdd` returns two scalars because `add` takes two scalar parameters and returns one scalar, so there are two partial derivatives.

mode implementation, one must run back each operation of the tape exactly once. Therefore, all effects on dependent operations must be accumulated before running back that operation. Common automatic differentiation libraries ensure complete accumulation using global state. They track the operations in the *exact order* of execution using, for example, a global counter as an ordering number while taping and then they run the tape back in the reverse order of execution. The reverse order of execution ensures that all dependent effects are run back and accumulated before running back an operation. ScalaGrad simplifies this approach: it ensures complete accumulation without global state. ScalaGrad tracks the *partial order* of operations' dependencies rather than the order of execution. An ordering number for each operation tracks this partial order; the ordering number is just the maximum of all dependencies' ordering numbers plus one. Running back the tape while respecting the reverse partial order also ensures that all its dependent effects are first run back and accumulated before running back an operation. Figure 3 depicts the simple yet impactful insight behind this simplification.

```
f(x1, x2) =
  a1 = x1 * x2
  a2 = x1 + x2
  b1 = a1 * a2
  b2 = a1 + a2
  y = b1 * b2
return y
```

Exact execution order and run back

$$a_1 \rightarrow a_2 \rightarrow b_1 \rightarrow b_2 \rightarrow y$$
$$a_1 \leftarrow a_2 \leftarrow b_1 \leftarrow b_2 \leftarrow y$$

Partial dependencies order and run back

$$a_1, a_2 \rightarrow b_1, b_2 \rightarrow y$$
$$a_1 \leftarrow a_2 \leftarrow b_2 \leftarrow b_1 \leftarrow y$$

Fig. 3: Showcase the insight driving ScalaGrad's simplification. The left shows the example operations. The top right shows the commonly used exact execution order and how it is run back. The bottom right shows ScalaGrad's partial dependency order. There are multiple ways to run it back, here we just show one example. Both order ensure that all dependent effects have always been run back, before running back an operation.

Krawiec et al. [10] use a state monad to track the exact order of executions. An equivalent implementation in Scala is impractical to use, as every numeric operation would have to run inside this state monad's context[1]. Utilizing mutable global state eliminates the need for a monad, however, it introduces complexities related to state corruption, e.g., re-entrance issues in higher-order differentiation and race conditions in multi-threaded applications. ScalaGrad's simplification allows the best of both worlds, it stays immutable while being ease to use. Additionally, higher-order differentiation, not supported by Krawiec et al. [10] mostly due to the state monad, can naturally be supported. The higher-order optimization application demonstrates higher-order differentiation in ScalaGrad.

[1]Krawiec et al. [10] use a sequence to sequence translation on a custom and limited language not having this issue.

TABLE I: Examples of type-safe differentiation in ScalaGrad

f input type	f output type	Resulting ∇f output type
`(S, S)`	`M`	`(M, M)`
`(S, S)`	`(S, S)`	`((S, S), (S, S))`
`(S, CV)`	`(S, RV)`	`((S, RV), (CV, M))`

`S` stands for `Scalar`, `CV` for `ColumnVector`, `RV` for `RowVector`, `M` for `Matrix`.

Type-safe differentiation. The return type of ∇f, the derivative of f, varies depending on f since ∇f returns all the partial derivatives. ScalaGrad captures this mathematical concept at the type level. Given a Scala function `f` of type `(Scalar, Scalar) => Scalar`, its derivative `d(f)` has the type `(Scalar, Scalar) => (Scalar, Scalar)`. ScalaGrad computes the type of `d(f)` at *compile-time* based on `f` utilizing Scala 3's match types [19]. Figure 2 illustrates this concept in code. ScalaGrad can compute the type of `d(f)` for any function `f` of scalars, vectors, and matrices; Table I shows some examples. Keeping track of the type during differentiation adds type safety to the automatic differentiation operation, adding robustness and maintainability to automatic differentiation applications. Additionally, it allows algorithms using derivatives to abstract over their function types, decoupling them from ScalaGrad but still being type-safe, as shown in the gradient-based sampling application.

IV. Applications of ScalaGrad

To highlight ScalaGrad's versatility, we applied it to deep learning, higher-order optimization, and gradient-based sampling. The complete code for those applications can be found in the ScalaGrad repository.

Deep Learning. To showcase the robust performance of ScalaGrad, we trained a fully connected neural network classifying images from the MNIST dataset [9] by implementing the mini-batch gradient descent algorithm. The neural network has one hidden layer with 36 units, resulting in 28,630 parameters to learn, or, put differently, there are 28,630 partial derivatives to calculate each iteration. For one epoch consisting of 938 iterations (60,000 images with 64 as batch size), ScalaGrad took 5.18 $\pm$ 0.98 seconds measured on a MacBook M1 Pro and 100 epochs using OpenJDK 20. A comparable setup in PyTorch, with the same hardware, network architecture, hyperparameters, and floating precision, took 2.08 $\pm$ 0.19 seconds measured on 100 epochs using Python 3.9.

Higher-order optimization. To showcase the higher-order differentiation capabilities of ScalaGrad, we implemented Newton's method to train a linear model where $\nabla^2 f$, the second-order derivative, is needed.

ScalaGrad's differentiation mechanism inside the `derive` function is implemented with its own `MatrixAlgebra` abstraction, making the mechanism itself differentiable. Therefore, to get the second-order derivative of `f`, `derive` (here imported as `d`) is simply applied twice `d(d(f))`.

Gradient-based sampling. This application illustrates a unique design space for gradient-based applications using a *type-safe* automatic differentiation library. We implemented the Hamiltonian Monte Carlo algorithm by taking the posterior's derivative as a parameter. The derivative is just a Scala function and has no dependence on ScalaGrad. When passing the derivative computed by ScalaGrad to the algorithm, the compiler checks that the argument matches the parameter type, so this design is type-safe. However, another automatic differentiation library, or even a manually coded derivative could have been used instead, highlighting that the algorithm's implementation is decoupled from ScalaGrad.

V. Discussion and Conclusion

ScalaGrad is a general automatic differentiation library applicable to various use cases. However, there are certain limitations. Currently, ScalaGrad's `MatrixAlgebra` type class only abstracts operations of tensors with two or fewer dimensions. Additionally, there is no mechanism to extend new operations from the outside, which is necessary to bind other highly optimized native routines like hardware-accelerated activation functions in a deep learning library. Those limitations are not inherent to the method and can be solved by engineering efforts. Regarding future work, an obvious next step is to build a data science library on top of ScalaGrad, e.g., a deep learning or probabilistic programming library. Another future direction is to make ScalaGrad's abstraction more type-safe. One straightforward way is to use Scala's singleton types to track the dimensionality of tensors at compile-time to check for undefined operations. A more ambitious direction is to drop the notation of a tensor from the abstraction, as it is the unit of execution, not abstraction. For example, rather than having a 4-dimensional tensor representing a batch of images, one could have a list of an Image type. An open question is how to map operations on those abstractions back to tensor operations for execution with an acceptable loss of performance.

We presented ScalaGrad, a general automatic differentiation library for Scala. ScalaGrad can differentiate general Scala code, only requiring numerical operations to be implemented by its type class. It provides asymptotically efficient implementations for the forward and reverse mode, supports native linear operations, and tracks types while differentiating, adding type safety to the automatic differentiation operation. Additionally, we discussed our improved automatic differentiation implementation, removing the necessity for global state, which allows for an immutable and ease to use implementation as well as a natural higher-order differentiation support. Finally, we demonstrated ScalaGrad's versatility, robust performance, and type safety by applying it to deep learning, higher-order optimization, and gradient-based sampling.

References

[1] M. Braschler, T. Stadelmann, and K. Stockinger, "Data science," in *Applied Data Science: Lessons Learned for the Data-Driven Business*. Springer, 2019, pp. 17–29.

[2] F. Pedregosa, G. Varoquaux *et al.*, "Scikit-learn: Machine Learning in Python," *J Mach Learn Res*, vol. 12, pp. 2825–2830, 2011.

[3] M. Abadi, A. Agarwal *et al.*, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2015.

[4] A. Paszke, S. Gross *et al.*, "Automatic differentiation in PyTorch," in *NIPS 2017 Workshop on Autodiff*, 2017. URL: https://openreview.net/forum?id=BJJsrmfCZ

[5] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. ACM, 2016, pp. 785–794. URL: https://doi.acm.org/10.1145/2939672.2939785

[6] G. Ke, Q. Meng *et al.*, "LightGBM: A Highly Efficient Gradient Boosting Decision Tree," *Advances in NIPS*, vol. 30, pp. 3146–3154, 2017.

[7] E. Bingham, J. P. Chen *et al.*, "Pyro: Deep Universal Probabilistic Programming," *J. Mach. Learn. Res.*, vol. 20, pp. 28:1–28:6, 2019.

[8] N. Amin, S. Grütter *et al.*, *"The Essence of Dependent Object Types"*. Springer International Publishing, 2016, pp. 249–272.

[9] Y. Lecun, L. Bottou *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[10] F. Krawiec, S. Peyton Jones *et al.*, "Provably Correct, Asymptotically Efficient, Higher-Order Reverse-Mode Automatic Differentiation," *Proc. ACM Program. Lang.*, vol. 6, Jan. 2022. URL: https://doi.org/10.1145/3498710

[11] D. Hall, "Breeze, Scala library for numerical processing, machine learning, and natural language processing," 2023. URL: https://github.com/scalanlp/breeze

[12] L. Beda, L. Korolev *et al.*, "Programs for automatic differentiation for the machine BESM," Technical Report, Institute for Precise Mechanics and Computation Techniques, Tech. Rep., 1959.

[13] R. E. Wengert, "A simple automatic derivative evaluation program," *Communications of the ACM*, vol. 7, pp. 463–464, 1964.

[14] J. Joss, "Algorithmisches Differenzieren," Ph.D. dissertation, ETH Zurich, 1976.

[15] G. Kedem, "Automatic Differentiation of Computer Programs," *ACM Trans. Math. Softw.*, vol. 6, no. 2, p. 150–165, Jun. 1980. URL: https://doi.org/10.1145/355887.355890

[16] D. D. Warner, "A partial derivative generator." Bell Telephone Laboratories, Computing Science Technical Report No. 28, 1975.

[17] A. Griewank, "Who invented the reverse mode of differentiation?" *Documenta Mathematica*, Jan. 2012.

[18] C. Elliott, "The simple essence of automatic differentiation," Apr. 2018.

[19] O. Blanvillain, J. I. Brachthäuser *et al.*, "Type-Level Programming with Match Types," *Proc. ACM Program. Lang.*, vol. 6, jan 2022. URL: https://doi.org/10.1145/3498698

[20] JAX authors. Jax from scratch. Accessed: 18 August 2023. URL: https://jax.readthedocs.io/en/latest/autodidax.html

[21] J. Bradbury, R. Frostig *et al.*, "JAX: composable transformations of Python+NumPy programs," 2018. URL: https://github.com/google/jax

[22] R. Z. Horace He, "functorch: JAX-like composable function transforms for PyTorch," 2021. URL: https://github.com/pytorch/functorch

[23] G. Van Rossum, J. Lehtosalo, and L. Langa, "PEP 484 - Type Hints." URL: https://peps.python.org/pep-0484/

[24] I. Rak-amnouykit, D. McCrevan *et al.*, "Python 3 Types in the Wild: A Tale of Two Type Systems," in *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*, ser. DLS 2020. Association for Computing Machinery, 2020, p. 57–70.

[25] ThoughtWorks, "DeepLearning.scala: A simple library for creating complex neural networks," 2023. URL: https://github.com/ThoughtWorksInc/DeepLearning.scala

[26] G. Modena, "Scalagrad: A tiny reverse-mode autodiff and neural network library," https://github.com/gmodena/scalagrad, 2020.

[27] E. A. Platanios, "TensorFlow Scala," 2018. URL: https://github.com/eaplatanios/tensorflow_scala

[28] Microsoft, "Scala bindings for LibTorch," 2023. URL: https://github.com/microsoft/scala_torch

[29] S. Brunk, "Storch: Gpu accelerated deep learning and numeric computing for scala 3," https://github.com/sbrunk/storch, 2023.