# DeepLearning.scala 2: Statically Typed Neural Networks

## Performing monadic automatic differentiation in parallel

YANG, BO, ThoughtWorks, Inc
ZHANG, ZHIHAO, ThoughtWorks, Inc
MARISA KIRISAME, University of Washington, USA
SHI, KAI*, ThoughtWorks, Inc

The Java and Scala community had built a very successful big data ecosystem. However, most of neural networks running on it are modeled in dynamically typed programming languages. These dynamically typed deep learning frameworks treat neural networks as differentiable expressions that contain many trainable variables, and perform automatic differentiation on those expressions when training them.

Until 2017, all deep learning frameworks in statically typed languages do not provide the expressive power of traditional frameworks. Their users are not able to use custom algorithms unless creating plenty of boilerplate code for hard-coded back-propagation.

We solved this problem in DeepLearning.scala 2. Our contributions are:

- We discovered a novel approach to perform automatic differentiation in reverse mode for statically typed functions that contain multiple trainable variables, and can interoperate freely with the metalanguage, just like pytorch.
- We designed a set of monads and monad transformers, which allow users to create monadic expressions that represent dynamic neural networks.
- Along with these monads, we provide some applicative functors, to perform multiple calculations in parallel.

Together of these features, users of DeepLearning.scala are able to create complex neural networks in an intuitive and concise way, and still maintain type safety.

Additional Key Words and Phrases: type class, path-dependent type, monad, scala

## 1 INTRODUCTION

Deep Neural Network has become the state of the art on many tasks, such as vision, game playing, voice recognition and natrual language translation.

A neural network is a computation model that transform the input, which is a tensor, into output(also tensor), by repeated application of tensor operation (matrix multiplication, tensor resizing, elementwise operations, such as max, +, etc). A deep neural network just indicate that there are many such transformations taken.

Additionally, a neural network has additional hidden inputs, called weights (also tensors), and deep learning is the task of finding the best weights for a neural network, such that a predefined objective (called loss) is minimized.

Deep learning is mostly done using different variation of Gradient Descend: we calculate the first order derivative of the loss function with respect to the weight, and update the weight accordingly. The processed is called Backpropagation.

---

*The corresponding author

Authors' addresses: Yang, Bo, ThoughtWorks, Inc, atryyang@thoughtworks.com; Zhang, Zhihao, ThoughtWorks, Inc, zhazhang@thoughtworks.com; Marisa Kirisame, University of Washington, Seattle, Washington, USA, lolisa@marisa.moe; Shi, Kai, ThoughtWorks, Inc, kshi@thoughtworks.com.

2018. XXXX-XXXX/2018/12-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Thus, backpropagation [Rumelhart et al. 1985] is the key feature in many deep learning frameworks. Combined with other optimization algorithms [Duchi et al. 2011; Kingma and Ba 2014; Zeiler 2012], deep learning frameworks mutate the values of weights in neural networks during training, producing a model of knowledge learnt from training data.

Backpropagation can be seen as a specialized instance of Automatic Differentiation (AD) [Baydin et al. 2015b]. Many successful Python deep learning frameworks [Google Brain 2017; Neubig et al. 2017; Paszke et al. 2017; Tokui et al. 2015] implement a common set of features of auto differentiation:

**Reverse mode** All these deep learning frameworks perform reverse mode AD instead of forward mode, as forward mode AD does not scale well for deep neural networks.

**Multiple trainable variable** Neural networks are composed of multiple layers. Each layer contains their own trainable variables. All these deep learning frameworks are able to calculate the derivatives of all trainable variables at once for one training data batch.

**Internal DSL [Fowler 2010]** All these deep learning frameworks are libraries that provide an Internal DSL, allowing users to create their differentiable functions in Python or Lua from similar expressions as creating ordinary non-differentiable functions. Since these frameworks do not require external language, models created by them can be easy integrated into a larger application alone with higher-level configurations [Chollet et al. 2015] or ETL (Extract, Transform and Load) process.

Deep learning framework using Dynamic Graph approch allow one to reuse whatever construct already available in the metalanguage (closure, object, list, tree, control flow, excpetion, etc, and preexisting function over them)

Deep Learning framework let you define weight, or trainable variable, and the framework can automatically do gradient descend over them, without users having to manually manage them.

However, most of the deep learning framework are in python, which is dynamically typed. Hence, it is not clear how to design a dl framework in a statically typed language, such that the dsl itself can enjoy the static type checking of the metalanguage, using technique such as final tagless or GADT.

Unfortunately, deep learning frameworks in statically typed languages have not achieved the above goals until 2017. Several AD libraries [Baydin et al. 2015a; Bischof et al. 1992; Griewank et al. 1996; Hascoët and Pascual 2013] written in Fortran, C++ or F# support AD via operator overloading or external preprocessor but do not scale to deep neural networks, either due to using forward mode or lacking the feature of multiple trainable variables.

Other deep learning frameworks in statically typed languages (including Scala binding of C++ frameworks) [Baydin and Pearlmutter 2016; Chen 2017; Intel 2016; Skymind 2017a; Zhao et al. 2017] do not support AD, instead, they only provide their high level computational graph APIs to compose predefined layers into neural networks. As a result, those frameworks do not have the ability to create fine-grained custom algorithms.

Typing deep neural networks with control flow in a static typed language is still an open problem.

In this paper, we present DeepLearning.scala, which is the first static typed implementation that achieves all the above goals for deep neural networks. Our approach is based on functional programming constructs, thus can be ported to Haskell, Idris and other statically typed functional programming languages.

## 2 BASIC CONCEPTS

For example, suppose we are building a model for predicting numbers generated by integer sequence:

What is the next number in sequence:
3, 6, 9, ?
The answer is 12.

In DeepLearning.scala, the robot can be implemented as a `guessNextNumber` function as following[1]:

```
// Weight initialization
val weights: Seq[DoubleWeight] = Stream.continually(DoubleWeight(math.random))
val bias: DoubleWeight = DoubleWeight(math.random)

def guessNextNumber(question: Seq[Double]): DoubleLayer = {
  // Perform a dot product between question and weights
  (question zip weights).map {
    case (element, weight) => element * weight
  }.reduce(_ + _) + bias
}
```

Listing 1. The differentiable matrix multiplication implemented by map/reduce

`guessNextNumber` performs a dot product between `question` and `weights` by invoking higher-order functions `map` and `reduce`.

Unlike [Chen 2017]'s special tensor type, our tensor can be typed simply as `Seq[Double]`, `Seq[DoubleWeight]` or `Seq[DoubleLayer]`. [2]

The return value of `guessNextNumber`, along with temporary variables in `guessNextNumber`, are `DoubleLayer`s, which are differentiable expressions.

The `weights` and `bias` contains some `DoubleWeight` that are referenced by `guessNextNumber`. They must be initialized before executing the model. Those are trainable variables.

From the user's point of view, both `DoubleLayer` and `DoubleWeight` are opaque types similar to ordinary `scala.Double`. Most of operators of `scala.Double` are also available on `DoubleLayer` and `DoubleWeight`, except those operators are differentiable. For now, type signature of multiplication operator as can be seen as if the Listing 2, and we will reveal the real type signature of `*` in Section 4.

Table 1 lists DeepLearning.scala built-in differentiable types other than `DoubleLayer` and `DoubleWeight`.

Table 1. Built-in Differentiable Types

| | non-trainable value | trainable variable | differentiable expression |
|---|---|---|---|
| single-precision scalar | `Double` | `DoubleWeight` | `DoubleLayer` |
| double-precision scalar | `Float` | `FloatWeight` | `FloatLayer` |
| vector | `INDArray` | `INDArrayWeight` | `INDArrayLayer` |

[1]The code examples from Listing 1 to Listing 7 do not contain necessary **`import`** and configurations. For an executable model backed by ND4J [Skymind 2017b], see Getting Started documentation on DeepLearning.scala website.

[2]DeepLearning.scala users can use other representations of tensors: (1) For tensors with a statically typed shape, use `shapeless.Sized`. For example a $10 \times 20$ two-dimensional tensor can be typed as `Sized[Sized[Double, _10], _20]`. For differentiable tensors, replace vanilla `Double` to `DoubleWeight` or `DoubleLayer`. (2) For GPU-accelerated tensors, use `INDArray` [Skymind 2017b]. For differentiable tensors, use `INDArrayLayer` or `INDArrayWeight` instead.

```
trait DoubleLayer {
  // Scalar multiplication
  def *(rhs: Double): DoubleLayer
  def *(rhs: DoubleLayer): DoubleLayer
  def *(rhs: DoubleWeight): DoubleLayer

  // Element-wise multiplication
  def *(rhs: INDArray): INDArrayLayer
  def *(rhs: INDArrayLayer): INDArrayLayer
  def *(rhs: INDArrayWeight): INDArrayLayer
}
trait DoubleWeight {
  // Scalar multiplication
  def *(rhs: Double): DoubleLayer
  def *(rhs: DoubleLayer): DoubleLayer
  def *(rhs: DoubleWeight): DoubleLayer

  // Element-wise multiplication
  def *(rhs: INDArray): INDArrayLayer
  def *(rhs: INDArrayLayer): INDArrayLayer
  def *(rhs: INDArrayWeight): INDArrayLayer
}
```

Listing 2. The hypothetical type signature of multiplication operator for `DoubleLayer` and `DoubleWeight`

In addition to differentiable operations, `Layer`s and `Weight`s can be evaluated with a `predict` method, thus, the model can predict next Integer by calling an ordinary function, shown in Listing 3. You may notice the `blockingAwait` suffix appended to `predict`, because `predict` returns a `Future [Double]` [3] , which contains the asynchronous task to compute the result. The actual computation is not performed in `predict` until `blockingAwait` is invoked [4].

```
val question = Seq(42.0, 43.0, 44.0)
println(guessNextNumber(question).predict.blockingAwait)
```

Listing 3. Inference on an untrained model

[3] For readers familiar to Haskell, you can understand `Future` from the corresponding types in Haskell:

- `Future` is an opaque type alias of a `TryT`-transformed `UnitContinuation`, which is used for asynchronous operations with the ability of exception handling.
- "opaque type alias" is similar to the `newtype` feature in Haskell.
- `TryT` provides the ability of exception handling, which is similar to `ExceptT` monad transformer in Haskell.
- `UnitContinuation` is similar to `Cont ()` in Haskell, which is used for asynchronous operations, like an asynchronous version of `IO` in Haskell.

All the above types are general purpose libraries, not parts of DeepLearning.scala. We use those continuation based monadic data types to archive the ability of parallel and asynchronous execution.

[4] When `predict` is used in a real world scenario (e.g. running a neural network in a web service), `blockingAwait` should be replaced to `flatMap`, in stead of blocking the current thread, which is an expensive resource. We use this `blockingAwait` here because it's more straightforward for understanding.

However, guessNextNumber returned an incorrect result because the weights and bias were randomly initialized, and have not been trained.

In order to train them, a loss function is necessary:

```
def squareLoss(robotAnswer: DoubleLayer, expectedAnswer: Double): DoubleLayer =
    {
  val difference: DoubleLayer = robotAnswer - expectedAnswer
  difference * difference
}
```

Listing 4. The differentiable square loss function

The above loss function squareLoss determines the squared error between robot's answer and the correct answer.

Both squareLoss and guessNextNumber are ordinary functions, and can be composed in other functions:

```
def linearRegression(question: Seq[Double], expectedAnswer: Double):
    DoubleLayer = {
  val robotAnswer = guessNextNumber(question)
  squareLoss(robotAnswer, expectedAnswer)
}
```

Listing 5. A differentiable function to train a linear regression model

linearRegression, composed of guessNextNumber and squaredLoss, returns a DoubleLayer of the loss for a specific question and its expected answer. linearRegression is a linear regression model with a square loss, and it can be trained as shown Listing 6. The blockingAwait is invoked because train returns a Future[Double] as well.

```
val question1 = Seq(3.0, 4.0, 5.0)
val expectedAnswer1 = 6.0

val question2 = Seq(13.0, 19.0, 25.0)
val expectedAnswer2 = 31.0

for (iteration <- 0 until 500) {
  linearRegression(question1, expectedAnswer1).train.blockingAwait
  linearRegression(question2, expectedAnswer2).train.blockingAwait
}
```

Listing 6. Training for 500 iterations

The weights and bias referenced by linearRegression are modified during 500 iterations of training, by stochastic gradient descent, to minimize the loss returned from linearRegression.

When weights and bias have been trained to make loss close to zero, guessNextNumber should return values that are very close to the expected answers.

```
val question = Seq(42.0, 43.0, 44.0)
println(guessNextNumber(question).predict.blockingAwait)
```

Listing 7. Inference on a trained model

This time, it will print a number closed to 45, as the model have finally learned the pattern of arithmetic progression.

The above example shows some basic concepts in DeepLearning.scala.

- `guessNextNumber`, `squareLoss` and `linearRegression` are differentiable functions that return differentiable expressions, which are computational graph nodes that can be evaluated when `train`ing or `predict`ing. TODO: TRAINING/PREDICTING SHOULD BE FULLY HYPERLINKED
- Differentiable expressions and trainable variables can be used as if they are ordinary non-differentiable values. For example, as shown in Listing 4, you can perform scalar subtraction and multiplication between `DoubleWeight`, `DoubleLayer` and ordinary `scala.Double`.
- When `train`ing a differentiable expression, it returns a `Future`, which encapsulates the side-effect of adjusting trainable variables referenced by the differentiable function.
- If a differentiable function invokes another differentiable function, then trainable variables trained by one differentiable function affect another one. For example, when training the differentiable function `linearRegression`, The trainable variables `weights` and `bias` are modified, hence `guessNextNumber` automatically gains the ability to `predict` correct answers.

HOW ABOUT DYNAMIC GRAPH/DEFINE-BY-RUN? CLOSER TO THE LITERATURE

## 3 DYNAMIC NEURAL NETWORKS

DeepLearning.scala supports dynamic neural networks. It means that the control flow of a neural network can differ according to values of internal nodes of the computational graph, when processing a special input *what special input?*. This is the key feature of recent deep learning frameworks like PyTorch [Paszke et al. 2017] or Chainer [Tokui et al. 2015]. Especially, dynamic deep neural networks can be more efficient by skipping part of the model [Liu and Deng 2017].

In this section, we will present how to create a simple dynamic neural network in DeepLearning.scala, which can be considered as a simplified version of outrageously large neural networks [Shazeer et al. 2017].

Suppose we have two sub-neural networks, `leftSubnet` and `rightSubnet` (Listing 8) [5]. We want to build a "gated" network, which conditionally runs either `leftSubnet` or `rightSubnet` for a special `input`.

```
def leftSubnet(input: INDArrayLayer): INDArrayLayer
def rightSubnet(input: INDArrayLayer): INDArrayLayer
```

Listing 8. Predefined sub-networks

Which sub-network is selected for the `input` should be determined by the `gate` network, which returns a pair of differentiable double expressions that indicate the preferences between `leftSubnet` and `rightSubnet`, shown in (Listing 9).

[5] For performance purpose, instead of `Seq[DoubleLayer]`, we use `INDArrayLayer` as the type of differentiable expression backed by ND4J. `INDArrayLayer` supports differentiable version of most operations that ND4J's n-dimensional array `INDArray` supports.

```scala
def gate(input: INDArrayLayer): (DoubleLayer, DoubleLayer)
```

Listing 9. Predefined gate network

The differentiable operations on `DoubleLayers` and `INDArrayLayers` form a differentiable embedded DSL inside the meta-language Scala. Thus, the concept of "gated" neural network can be considered as a conditional control flow in the differentiable DSL, and the values of nodes of a gated neural network are simply some let bindings in the DSL.

The control flow of gated network that we want to build is described in Function GatedNetwork.

**Function** GatedNetwork

**Input:** Features extracted by preceding layers
**Output:** Features passed to succeeding layers
scores ← `gate`(*Input*);
**if** *score of left sub-network* > *score of right sub-network* **then**
  **return** *score of left sub-network* × `leftSubnet`(*Input*);
**else**
  **return** *score of right sub-network* × `rightSubnet`(*Input*);
**end**

In DeepLearning.scala, there are three different approaches to implement the gated network. Examples of these approaches are introduced in following Section 3.1, Section 3.2, and Section 3.3.

### 3.1 Eager Execution (bad)

An obvious approach to create the gated network is to eagerly execute the `gate`, shown in Listing 10:

```scala
def naiveGatedNet(input: INDArrayLayer): INDArrayLayer = {
  val scores = gate(input)
  if (scores._1.predict.blockingAwait > scores._2.predict.blockingAwait) {
    scores._1 * leftSubnet(input)
  } else {
    scores._2 * rightSubnet(input)
  }
}
```

Listing 10. The eager execution implementation of gated network

There are three sub-networks in the `naiveGatedNet` function. The `gate` returns a pair of `DoubleLayers`. By blocking await the `prediction` result, we get two `Doubles`, which can be used to determine which sub-network is preferred between `leftSubnet` and `rightSubnet`. The chosen sub-network will multiplies with the value returned by the `gate` in order to enabling backpropagation on the `gate`.

However, there is a performance issue in the `naiveGatedNet`.

In DeepLearning.scala, all differentiable expressions, including the scalar `DoubleLayer` and vectorized `INDArrayLayer`, contain some lazily evaluated differentiable computational graph nodes, which will not be executed until their asynchronous task returned from `predict` or `train` is performed in a `blockingAwait` or `onComplete` calls.

So, the two `predict.blockingAwait` calls in the **`if`** will execute the computational graph in `gate` twice. Also the computational graph in `naiveGatedNet` will be executed when users call predict or train call naiveGatedNet in the future. Even worse, input contains a computational graph, too. Along with gate, it will be evaluated three times, which may contain complex future extracting process. *THIS LOOK VERY WRONG. EVERY LAZY SYSTEM WILL INTERNALLY MAKE SURE IT IS ONLY EVALED ONCE*

### 3.2 Monadic Control Flow (good)

Ideally, the calls to predict should be avoided in differentiable functions. The recommended approach to create a dynamic neural network is using forward, which returns a monadic value of Do[Tape[Data, Delta]], which can be used in a monadic control flow via Scalaz [Yoshida 2017]'s type classes [Oliveira et al. 2010] Monad and Applicative. *WHAT IS DO? YOU SHOULD INTRODUCE IT* Listing 11 shows the monadic control flow of gated network.

```
def monadicGatedNet(input: INDArrayLayer): INDArrayLayer = {
  val scores = gate(input)
  val gatedForward: Do[Tape[INDArray, INDArray]] = {
    scores._1.forward.flatMap { tape1: Tape[Double, Double] =>
      scores._2.forward.flatMap { tape2: Tape[Double, Double] =>
        if (tape1.data > tape2.data) {
          (scores._1 * leftSubnet(input)).forward
        } else {
          (scores._2 * rightSubnet(input)).forward
        }
      }
    }
  }
  INDArrayLayer(gatedForward)
}
```

Listing 11. Monadic gated network

This gated network is built from the monadic expression gatedForward, which contains some forward calls, which are asynchronous operations (or Do) that produce Wengert list record (or Tape). The implementation detail of Do and Tape will be discussed in Section 5. For now, we only need to know that Do is a monadic data type that supports flatMap. By flatMapping those forward operations together, we built the entire monadic control flow gatedForward for the gated network.

The monadicGatedNet represents a dynamic neural network, since each forward operation is started after its previous forward done. This behavior allows dynamically determining succeeding operations according to result of previous forward operations, as shown in the **if** clause in Listing 11.

However, flatMap prevents additional optimization, too. scores._2.forward have to wait for scores._1.forward's result, even if the two operations are logically independent.

### 3.3 Parallel Applicative Control Flow + Sequential Monadic Control Flow (better)

Ideally, the independent operations scores._1.forward and scores._2.forward should run in parallel. This can be done by tagging Do as Parallel, and use scalaz.Applicative.tuple2 [McBride and Paterson 2008] instead of flatMap (Listing 12).

This applicativeMonadicGatedNet takes both advantages from applicative functors and monads. The entire control flow is a flatMap sequentially composed of two stages. In stage1, there is

```
def applicativeMonadicGatedNet(input: INDArrayLayer): INDArrayLayer = {
  val scores = gate(input)
  val parallelForward1: ParallelDo[Tape[Double, Double]] = {
    Parallel(scores._1.forward)
  }
  val parallelForward2: ParallelDo[Tape[Double, Double]] = {
    Parallel(scores._2.forward)
  }
  val Parallel(stage1) = {
    parallelForward1.tuple2(parallelForward2)
  }
  def stage2(tapes: (Tape[Double, Double], Tape[Double, Double])) = {
    if (tapes._1.data > tapes._2.data) {
      (scores._1 * leftSubnet(input)).forward
    } else {
      (scores._2 * rightSubnet(input)).forward
    }
  }

  val gatedForward = stage1.flatMap(stage2)
  INDArrayLayer(gatedForward)
}
```

Listing 12. Applicative + monadic gated network

a `tuple2` composed of `scores._1.forward` and `scores._2.forward` in parallel. Then, in `stage2`, the succeeding operation is dynamically determined according to `tapes`, the result of `stage1`.

The parallel applicative operation is also the default behavior for all built-in vector binary operators. Listing 13 shows some simple expressions that will be executed in parallel.

```
def parallelByDefault(a: INDArrayLayer, b: INDArrayLayer, c: INDArrayLayer, d:
    INDArrayLayer): INDArrayLayer = {
  a * b + c * d
}
```

Listing 13. By default, `a * b` and `c * d` will be executed in parallel because they are independent

By combining both applicative functors and monads, DeepLearning.scala supports dynamic neural network and still allows the independent parts of the neural network to run in parallel. In addition, the backward pass of differentiable functions built from parallel applicative functors or built-in vector binary operators will be executed in parallel, too.

### 3.4 Direct style DSL for Applicative and Monadic Control Flow (best)

In the previous section, we had present a dynamic neural network executed in parallel. However, The usage of `flatMap` and `Parallel`-tagged types may scare algorithm authors who are not familiar with monadic programming. Ideally, the code written by those people should look straightforward and has the same structure in pseudo-code GatedNetwork or Listing 10, and still gain benefits of Listing 12.

The goal can be achieved by transforming the direct style code into monadic and applicative code at compile-time. We created an DSL with the help of the !-notation provided by Dsl.scala [Yang 2017], which provides Scala compiler plugins to perform the necessary compiler-time transformation.

As shown in Listing 14, the !-notation "extracts" the `Double` values from a pair of `DoubleLayers` in parallel. Those `Doubles` are ordinary non-differentiable Scala types that will not backpropagate, and can be used in ordinary Scala control flow expression like **`if`**.

```
def dslGatedNet(input: INDArrayLayer): INDArrayLayer = {
  val scoreLayers: (DoubleLayer, DoubleLayer) = gate(input)
  val scores: (Double, Double) = !scoreLayers
  if (scores._1 > scores._2) {
    scoreLayers._1 * leftSubnet(input)
  } else {
    scoreLayers._2 * rightSubnet(input)
  }
}
```

Listing 14. Dsl.scala powered direct style gated network

Generally, a !-notation on a `Layer` will generate a monadic `flatMap` call, to extract the value of forward pass of the `Layer`; a !-notation on a tuple of `Layers` will generate some applicative `<*>` and `map` calls, to extract a tuple of values of forward pass of those `Layers`, in parallel. Thus, the actually code generated by Dsl.scala's compiler plugins for Listing 14 is similar to Listing 12.

## 4 AD HOC POLYMORPHIC DIFFERENTIABLE FUNCTIONS

In section 2, we had present the hypothetical differentiable types of multiplication for `DoubleLayer` and `DoubleWeight`. However, the method overloading approach shown in Listing 2 is too verbose, and requires a lot of boilerplate code. In this section, we will present an approach to create custom differentiable functions that supports without those methods overloading.

A differentiable function is a neural network. Ideally, a differentiable function should be an ad hoc polymorphic function that accepts heterogeneous types of parameters, including:

- A vanilla vector input. i.e. `INDArray`.
- Differentiable expressions of hidden states produced by any previous neural network layers, i.e. any `INDArrayLayers` regardless of the prefixes.
- Trainable variables in the case of activation maximization technique [Erhan et al. 2009].i.e. any `INDArrayWeights` regardless of the prefixes.
- Other user defined differentiable type.

Table 1 shows nine types that have built-in `DeepLearning` type classes.

This can be achieved with the help of [Gurnell 2017]'s Scala encoding of dependent-type type class. We defined a `DeepLearning` (Listing 15 [6]) type class that witnesses any supported expressions including differentiable expressions, trainable variables, or vanilla non-differentiable types. The

[6] For readers who are more familiar with Idris, there is a corresponding notation in Idris:

```
interface DeepLearning Differentiable where
  Data : Type
  Delta : Type
  forward : Differentiable -> Do (Tape Data Delta)
```

users can create type aliases to restrict the types of state during forward pass and backward pass as shown in Listing 16.

```
trait DeepLearning[Differentiable] {
  /** The type of result calculated during forward pass */
  type Data

  /** The type of derivative during backward pass */
  type Delta

  def forward(differentiable: Differentiable): Do[Tape[Data, Delta]]
  // Other auxiliary methods is omitted
}
```

Listing 15. The dependent-type type DeepLearning

```
type INDArrayExpression[Expression] = DeepLearning[Expression] {
  /** The type of result calculated during forward pass */
  type Data = INDArray

  /** The type of derivative during backward pass */
  type Delta = INDArray
}
```

Listing 16. A type class alias that witnesses dense vector expressions

By using `INDArrayExpression` as a context bound, we can create a polymorphic differentiable function that accepts any vector expression.

```
def polymorphicDifferentiableFunction[A: INDArrayExpression, B:
    INDArrayExpression, C: INDArrayExpression, D: INDArrayExpression](a: A, b:
    B, c: C, d: D): INDArrayLayer = {
  a * b + c * d
}
```

Listing 17. A polymorphic differentiable function

Listing 17 is similar to Listing 13, except each argument of `polymorphicDifferentiableFunction` accepts `INDArray`, `INDArrayWeight` or `INDArrayLayer` respectively, not only `INDArrayLayer`.

Built-in operations including arithmetic operations, `max`, and `dot` are polymorphic differentiable functions defined in this approach, too, which can be used in user-defined polymorphic differentiable functions.

## 5 IMPLEMENTATION

In this section, we will introduce the internal data structure used in DeepLearning.scala to perform AD.

- For ease of understanding, Section 5.1 starts from a simple dual number implementation `DualNumber`, which was known as an approach to perform forward mode AD for scalar values.
- Section 5.2 introduces our variation of dual number `ClosureBasedDualNumber`, which supports tree-structured reverse mode AD (aka backpropagation) for multiple trainable variables.
- Section 5.3 shows the actual data type `Tape` in DeepLearning.scala, which is generalized to not only scalar types, but also vector types and any other differentiable types.
- Section 5.4 discovered the monadic control flow `Do`, which manages the life circle of `Tape`s, sharing `Tape`s for common computational graph nodes, allowing arbitrary DAG(Directed Acyclic Graph)-structured computational graph.
- Section 5.5 summarizes the entire execution process during a training iteration, showing how the user-defined differentiable functions get executed through internal mechanisms `Do` and `Tape`.

### 5.1 Ordinary Dual Number

Our approach for reverse mode AD uses a data structure similar to traditional forward mode AD, with only a few changes.

Forward mode AD can be viewed as computation on dual number. For example, dual number for scalar types can be implemented as Listing 18:

```scala
type Data = Double
type PartialDelta = Double
case class DualNumber(data: Data, delta: PartialDelta)
```

Listing 18. Dual number for forward mode AD

Arithmetic operations on those dual number can be implemented as Listing 19:

```scala
object DualNumber {
  def plus(left: DualNumber, right: DualNumber): DualNumber = {
    DualNumber(left.data + right.data, left.delta + right.delta)
  }
  def multiply(left: DualNumber, right: DualNumber): DualNumber = {
    DualNumber(left.data * right.data, left.data * right.delta + right.data *
        left.delta)
  }
}
```

Listing 19. Arithmetic operations on dual number

### 5.2 Monadic Closure-based Dual Number

However, it is hard to type this approach if we want to support multiple trainable variables, with the number unknown before runtime. `PartialDelta` in Listing 18 represents the partial derivative of trainable variables. In AD tools that support only one trainable variable, the trainable variable is usually forced to be the input. Hence `PartialDelta` is the input type for those AD tools. This assumption is broken for our case, since our delta type of a specific `DualNumber` must contain derivatives for all trainable variables that were used to produce the `DualNumber`, not only the partial derivative of input. As a result, the type of delta varies when the number of trainable variables grows.

To type-check the delta, considering the only usage of the delta in a neural network is in updating trainable variables in a gradient descent based optimization algorithm. We can replace `PartialDelta` to a `UpdateWeights` closure Listing 20. In order to implement arithmetic operations for the new dual number, the operations on `PartialDelta` should be replaced to custom functions for `UpdateWeights` (Listing 21).

*THIS IS NOT CUSTOM FUNCTION - AT THIS POINT WE ONLY KNOW THERE IS SOME UNKNOWN TYPE OF UPDATEWEIGHTS, A BETTER TERM WILL BE UNKNOWN TYPE*

```scala
type Data = Double
case class ClosureBasedDualNumber(data: Data, backward: UpdateWeights)
```

Listing 20. Replacing `PartialDelta` to a closure

```scala
object ClosureBasedDualNumber {
  def plus(left: ClosureBasedDualNumber, right: ClosureBasedDualNumber):
      ClosureBasedDualNumber = {
    ClosureBasedDualNumber(left.data + right.data, UpdateWeights.plus(left.
        backward(), right.backward()))
  }
  def multiply(left: ClosureBasedDualNumber, right: ClosureBasedDualNumber):
      ClosureBasedDualNumber = {
    ClosureBasedDualNumber(
      left.data * right.data,
      UpdateWeights.multiply(left.data, right.backward()) + UpdateWeights.
          multiply(right.data, left.backward()))
  }
}
```

Listing 21. Replacing operations on `PartialDelta` to custom functions for `UpdateWeights`

The only question remaining is to implement the `UpdateWeights` to make its behavior be equivalent to the original dual number implementation.

Mathematically, the `UpdateWeights` type in a dual number should form any vector space, i.e. the `UpdateWeights` closure itself must support addition and scalar multiplication operations.

Our approach is making `UpdateWeights` be a function type that contains side-effects to update trainable variables. Thus the addition operation for closures can be defined as (1).

$$(f_0 + f_1)(x) = f_0(x) + f_1(x) \tag{1}$$

And the scalar multiplication operation for closures is defined as (2):

$$(x_0 f)(x_1) = f(x_0 x_1) \tag{2}$$

The above definition of arithmetic operations can be implemented in monadic data types as shown in Listing 22.

```scala
type UpdateWeights = Do[Double] => SideEffects
object UpdateWeights {
  /** (f0 + f1)(x) = f0(x) + f1(x) */
  def plus(f0: UpdateWeights, f1: UpdateWeights) = { doX: Do[Double] =>
    f0(doX) |+| f1(doX)
  }

  /** (x0 f)(x1) = f(x0 x1) */
  def multiply(x0: Double, f: UpdateWeights) = { doX1: Do[Double] =>
    f(doX1.map(x0 * _))
  }
}
```

Listing 22. Arithmetic operations for the closure that contains side-effects

`UpdateWeights`, as a replacement to original `PartialDelta`, is a closure able to update derivatives for all weights with a coefficient (the `Double` parameter). `|+|` is the append operation of `scala.Semigroup`, which could be any cumulative data type.

Also note that the parameter is a monadic data type `Do` that encapsulates the computation of derivative. Unlike strictly evaluated values, `Do` is an operation evaluated in need.

In DeepLearning.scala, our `SideEffects` is based on the asynchronous operation type `UnitContinuation`. Our built-in differentiable operations execute the independent parts of backward pass in parallel with the help of `Applicative` type class instances of `UnitContinuation`.

```scala
type SideEffects = UnitContinuation[Unit]
```

Listing 23. Monadic side-effects

`UnitContinuation[A]` is an opaque alias [Osheim and Cantero 2017] of `(A => Trampoline [Unit]) => Trampoline[Unit]`, implemented in a separate library at future.scala. It is used in DeepLearning.scala as a monadic data type for encapsulating side effects in stack-safe asynchronous programming.

The `SideEffects` for neural networks conform associative law because the only side effects is updating trainable variables. Thus, our `UpdateWeights.plus` and `UpdateWeights.multiply` are equivalent to the operations on strictly evaluated scalar value `PartialDelta` in forward mode AD.

Since `UpdateWeights` is a closure with side effects, a trainable variable can be represented as a tuple of a mutable value and the action to modify the mutable value.

In Listing 24, the trainable variable is trained by a fixed learning rate to simplify the hyperparameters of optimization algorithms. The actual DeepLearning.scala implementation uses a more sophisticated approach to configure the hyperparameters

*WHAT ABOUT USING CUSTOM OPTIMIZER LIKE ADAGRAD? IT SEEMS THAT THIS CAN SCALE TO MOMENTUM BUT IS WORTH MENTIONING EXPLICITLY*

Similar to trainable variables, a non-trainable value can be represented as a tuple of the value and a no-op closure shown in Listing 25.

Because `delta` is an action instead of pre-evaluated value, the implementation of `backward` for non-trainable value can entirely avoid executing unnecessary computation in `doDelta`.

```
def createTrainableVariable(initialValue: Double, learningRate: Double):
    ClosureBasedDualNumber = {
  var data = initialValue
  val backward: UpdateWeights = { doDelta: Do[Double] =>
    val sideEffects: Do[Unit] = doDelta.map { delta =>
      value -= learningRate * delta
    }
    convertDoToUnitContinuation(sideEffects)
  }
  ClosureBasedDualNumber(data, backward)
}
```

Listing 24. Create a dual number for a trainable variable

```
def createLiteral(data: Double): ClosureBasedDualNumber = {
  val backward = { doDelta: Do[Double] =>
    UnitContinuation.now(())
  }
  ClosureBasedDualNumber(data, backward)
}
```

Listing 25. Create a dual number for a non-trainable value

Finally, we can create a differentiable function as shown in Listing 26, whose leaf nodes are `createTrainableVariable` and `createLiteral`, and internal nodes are arithmetic operations in Listing 21.

```
val w0 = createTrainableVariable(math.random, 0.001)
val w1 = createTrainableVariable(math.random, 0.001)

def computationalTree(x: ClosureBasedDualNumber) = {
  val y0 = ClosureBasedDualNumber.multiply(x, w0)
  val y1 = ClosureBasedDualNumber.multiply(y0, w1)
  y1
}
```

Listing 26. A tree-structured differentiable function

The computational graph of `computationalTree` is shown in Figure 1. Note that the arrow direction denotes the dependency between expressions, from arrow tail to arrow head, which is the reverse of the direction of data flow.

The closure-based dual number `y1` has a closure `backward`, which returns `SideEffects` that recursively change all trainable variables referenced by the closure.

Note that `backward` itself does not perform any side effects. It just collects all side effects into a `UnitContinuation[Unit]`. Figure 2 shows how the side effects of updating trainable variables are collected.

Finally, the collected side effects of `UnitContinuation[Unit]` returned from `y1.backward` can be performed by a `blockingAwait` or `onComplete` call.

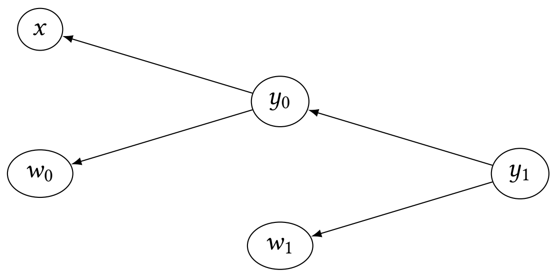


Fig. 1. A tree-structured computational graph

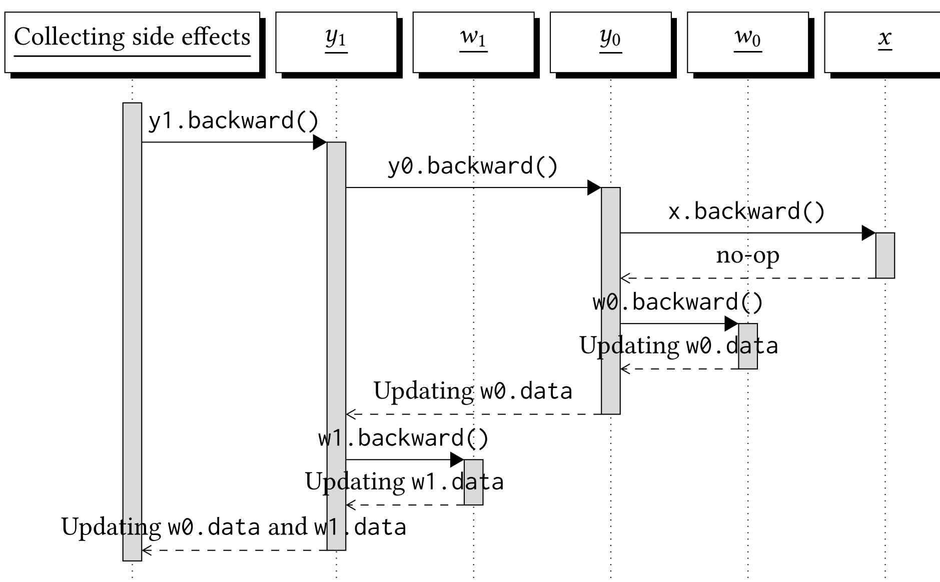


Fig. 2. Backpropagation for a tree-structured computational graph

### 5.3 Generic Tape

This closured-based monadic dual number can be generalized to any linear spaces, not only scalar types such as `Double`, but also n-dimensional arrays.

The dual number type that we actually defined in DeepLearning.scala is `Tape`, a generic version of `ClosureBasedDualNumber` in Listing 20. We replaced `ClosureBasedDualNumber`'s hard-coded `Double` to type parameters `Data` and `Delta`, as shown in Listing 27.

```
final case class Tape[+Data, -Delta](
  data: Data,
  backward: Do[Delta] => UnitContinuation[Unit]
)
```

Listing 27. Generic closured-based monadic dual number

`Data` and `Delta` are usually the same, but they can also be different types. For example, you can create a type whose `Data` is a dense n-dimensional array and whose Delta is a pair of index and scalar, representing a dense tensor that sparsely updates.

This data structure is similar to Wengert list in traditional reverse mode AD, except our tape is a tree of closures instead of a list.

### 5.4 Reference Counted Tape

Although the closured-based dual number approach from Listing 20 to Listing 27 supports multiple trainable variables, the closure-based computation has a performance issue in the case of diamond dependencies.

Listing 28 shows a differentiable function `diamondDependentComputationalGraph` that contains diamond dependencies to some differentiable expressions or trainable variables.

```
val w = createTrainableVariable(math.random, 0.001)
def diamondDependentComputationalGraph(x: ClosureBasedDualNumber) = {
  val y0 = ClosureBasedDualNumber.multiply(x, w)
  val y1 = ClosureBasedDualNumber.multiply(y0, y0)1
  val y2 = ClosureBasedDualNumber.multiply(y1, y1)
  y2
}
```

Listing 28. A diamond dependent differentiable function

The computational graph of `diamondDependentComputationalGraph` are shown in Figure 3.

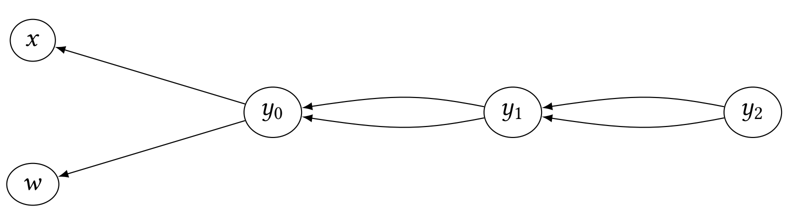


Fig. 3. A diamond dependent computational graph

When `y2.backward` is invoked, in order to collect side effects of `y2`'s dependencies, `y1.backward` will be invoked, twice, and each `y1.backward` call will triggers two `y0.backward` calls. As a result, for each iteration of backpropagation, `y0.backward`, `w.backward` and `x.backward` are invoked four times, respectively.

The process in `y2.backward` is shown in Figure 4.

Generally, given $n$ levels of nested diamond dependencies, the computational complexity is $O(2^n)$, which is unacceptable for neural networks that may share common differentiable expressions.

We introduced a reference counting algorithm for dual numbers, to avoid the exponential time complexity, by only calling backward once.

The reference counting is managed in a wrapper of `Tape`, which has additional `acquire` and `release` functions.

Each wrapper has two internal states: (1) reference counter, (2) accumulator of delta. Respectively, `acquire` and `release` calls will increase and decrease the reference counter, and `backward` calls will cumulate the `delta` to the accumulator.

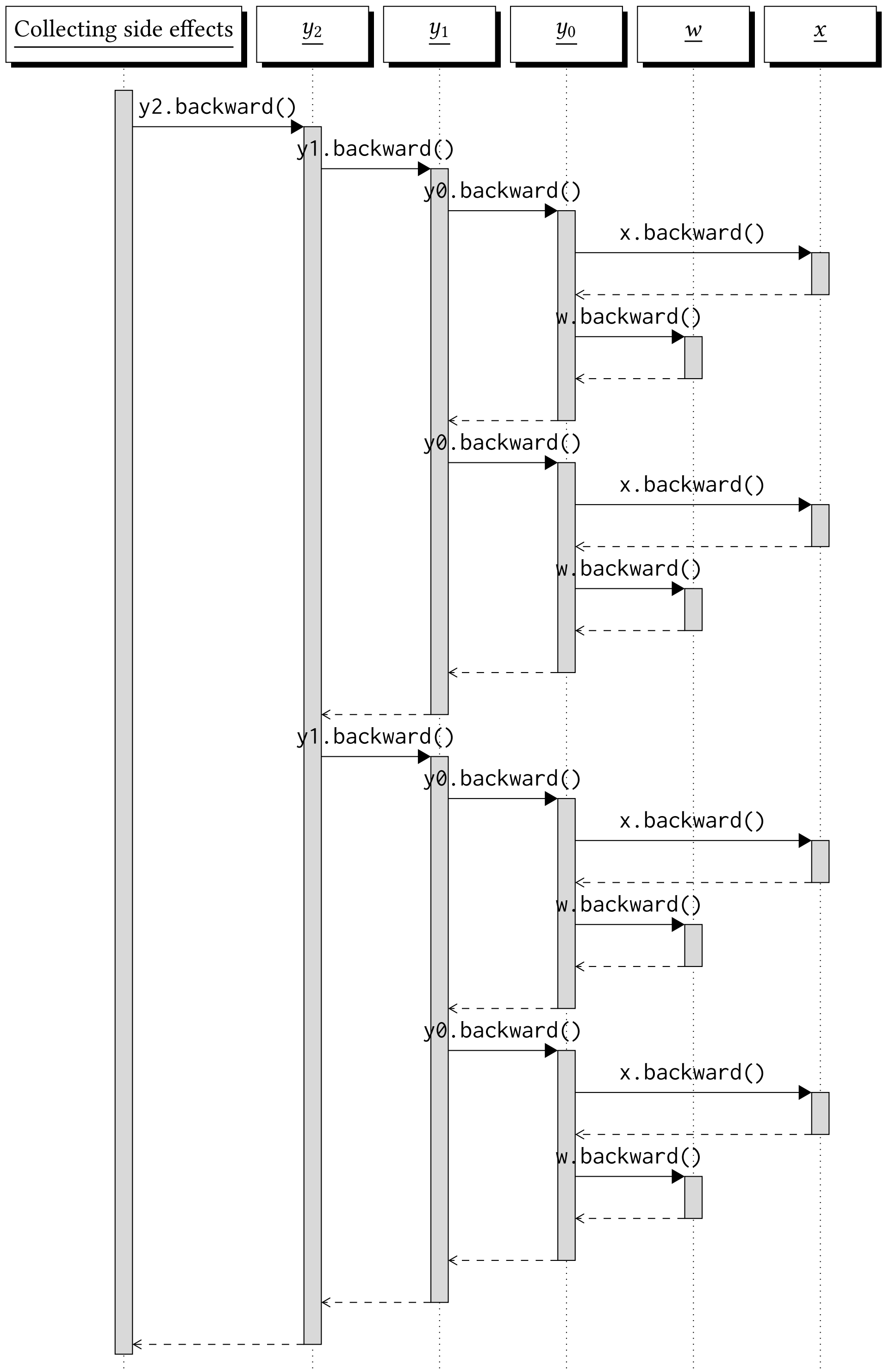


Fig. 4. Backpropagation for a diamond dependent computational graph

When reference counting algorithm is enabled, `backward` is not recursive any more. Instead, a wrapper only call `backward` of its dependencies when the reference counter is decreased to zero. The entire process of backpropagation is shown in Figure 5.

This wrapper is implemented as the monadic data type `Do`, in which the side effects of updating counters and accumulators are monadic control flows. With the help of `Do`, now our computational graphs are modeled in `Do[Tape[Data, Delta]]`, which can be created by `forward` methods described in Section 3.2. As mentioned in Section 3.3, computational graph node of binary operations are evaluated in parallel.

In traditional backpropagation implementation, tape is a list, hence both the execution order of backward pass and forward pass must be sequentially reverse to each other. Even previous attempt of closure-based tape[Pearlmutter and Siskind 2008] still requires conversion to sequential expressions of A-normal form[Sabry and Felleisen 1993].

By introducing reference counting, the execution order of our backward pass and forward pass do not have to be exactly reverse, hence the conversion to A-normal form becomes unnecessary. As a result, DeepLearning.scala supports out-of-order execution in both forward pass and backward pass, in which the independent sub-graph can be even executed in parallel.

### 5.5 The Overview of a Training Iteration

In brief, in each iteration, a differentiable function that contains multiple trainable variables can be trained in the following steps:

(1) Executing the user-defined differentiable function with a batch of input, to obey a differentiable expression (i.e. a subtype of `Layer`).
(2) Calling `forward` on differentiable expression to build a computational graph (i.e. a `Do[Tape[Data, Delta]]`). The reference counter to the computational graph is zero at the point.
(3) Performing the `forward` pass of differentiable expression to build a tape (i.e. a `Tape[Data, Delta]`), which contains a pair of the result of forward pass and a `backward` closure. The reference counter of each node in computational graph is increased during this step.
(4) Performing `backward` closure of the root node of the computational graph. The accumulator of delta on the root node is updated.
(5) Releasing of the root node of the computational graph. The reference counter of each node in computational graph is decreased to zero and `backward` closure of each node is performed during this step, thus all referenced trainable variables are updated.

Note that step 1 and step 2 are pure function calls, with no side effects. Step 3 to step 5 are monadic control flows, which encapsulate some side effects that will be performed only when an unsafe method `blockingAwait` or `onComplete` is eventually called.

## 6 EVALUATION

We created some benchmarks to evaluate the computational performance of DeepLearning.scala, especially, we want to measure:

(1) How parallel execution affect the time cost of a training iteration of different structured neural networks;
(2) the computational performance impact when part of a neural network is disabled.

Those benchmarks are built with Jmh [Shipilev 2018], running on CPU, measuring the number of mini-batches per second, for training neural networks that contain variant number of branches of sub-networks, as classifiers for CIFAR-100 [Krizhevsky and Hinton 2009], which is a dataset

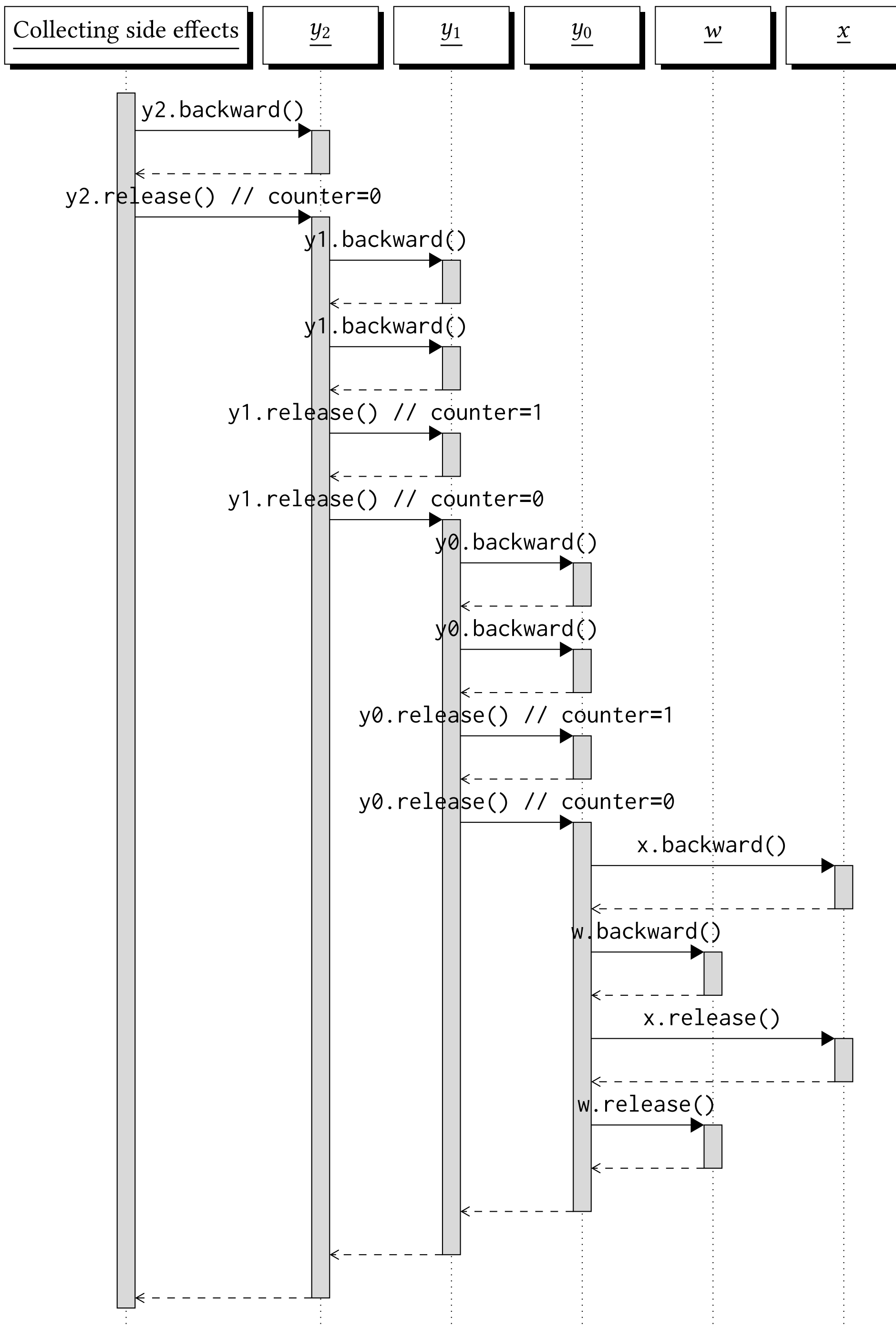


Fig. 5. Backpropagation for a diamond dependent computational graph (with reference counting)

for image recognition, which has 100 fine-grained classes containing 600 images each. These fine-grained classes are grouped into 20 coarse-grained classes.

The architecture of the network used for benchmarks summarized in Figure 6.

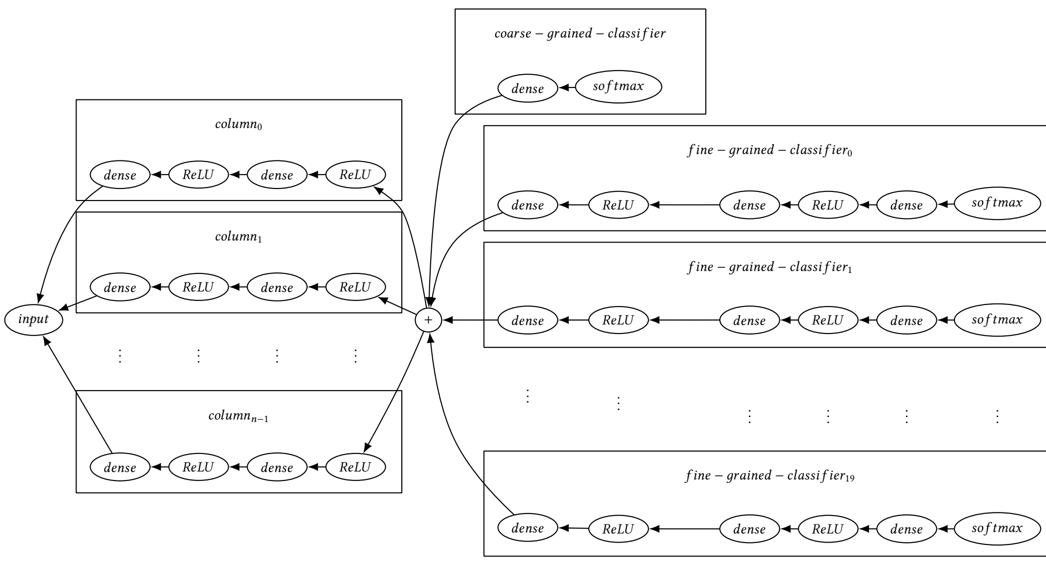


Fig. 6. The models used for benchmarks

We created $n$ columns [Ciregan et al. 2012] of experts sub-networks for feature extracting. The output from those experts are then summed for further layers and classifiers. Each column contains two dense layers [7] followed by ReLU activation layers. Those columns are independent, and we expect our framework can execute them in parallel when training or inference.

Since CIFAR-100 dataset has both coarse-grained and fine-grained labels, we created a coarse-grained classifier and 20 fine-grained classifiers. The coarse-grained classifier contains a dense layer to convert features to scores of 20 coarse-grained classes, followed by a softmax layer. Each fine-grained classifier corresponding to a coarse-grained class contains three dense layers, two ReLU activation layers, and a softmax layer, classifying 5 fine-grained classes.

We constructed mini-batches by coarse-grained class when training. All samples in a mini-batch belongs to one coarse-grained class. With the help of the feature of dynamic neural network in DeepLearning.scala, for each training iteration, only one fine-grained classifier is used, other 19 fine-grained classifiers are skipped. When inference, the fine-grained classifier is chosen dynamically according to the prediction by coarse-grained classifier. For comparison, we also created another set of benchmarks that do not skip unmatched fine-grained classifiers.

The benchmark result when training these models is shown in Table **??** and Table 2 (larger score is better). The sizes of those dense layers previous to softmax layers in these models are the number of classes (20 for coarse-grained classifier, 5 for fine-grained classifiers). Other dense layers output 64 features. These models are trained in a SGD optimizer with batch size 16. Both the number of columns in each model and the number of threads used in training vary among benchmarks.

[7] We use dense layers instead of convolution layers because the underlying N-dimensional array library ND4J is not able to efficiently perform immutable operation of convolution. See section 7.1 for discussion.

| number of columns | thread pool size | Score, ops/s | |
|---|---|---|---|
| 4 | 1 | 2.550 | ± 0.043 |
| 4 | 2 | 3.233 | ± 0.276 |
| 4 | 4 | 3.338 | ± 0.217 |
| 2 | 1 | 3.345 | ± 0.095 |
| 2 | 2 | 3.987 | ± 0.091 |
| 2 | 4 | 4.828 | ± 0.200 |
| 1 | 1 | 3.854 | ± 0.046 |
| 1 | 2 | 5.239 | ± 0.288 |
| 1 | 4 | 6.328 | ± 0.058 |

Table 2. The benchmark result when no fine-grained classifier is skipped

| number of columns | thread pool size | Score, ops/s | |
|---|---|---|---|
| 4 | 1 | 5.314 | ± 0.304 |
| 4 | 2 | 5.484 | ± 0.362 |
| 4 | 4 | 5.607 | ± 0.159 |
| 2 | 1 | 8.702 | ± 0.430 |
| 2 | 2 | 9.527 | ± 0.259 |
| 2 | 4 | 8.831 | ± 0.268 |
| 1 | 1 | 12.430 | ± 1.514 |
| 1 | 2 | 13.531 | ± 0.618 |
| 1 | 4 | 14.513 | ± 0.554 |

Table 3. The benchmark result when unmatched fine-grained classifiers are skipped

The benchmark result verified the performance improvement when increasing thread pool size, since DeepLearning.scala execute independent sub-networks in parallel. This benchmark also shows performance improvement when unmatched fine-grained classifiers are skipped.

## 7 FUTURE WORK

### 7.1 New Back-end

Currently, DeepLearning.scala 2's built-in differentiable vector expression type `INDArrayLayer` is based on nd4j's `INDArray` [Skymind 2017b]. As described in Section 5.5, in each training iteration, for each computational graph node, `forward` and `backward` operations are performed, which internally call some methods on `INDArray`, resulting in GPU kernel executions for nd4j's CUDA runtime. These ND4J operations have bad computational performance because: (1) ND4J is not designed (2) some operations [8] are extremely slow; (3) enqueuing a kernel is relatively expensive.

We are developing a new back-end as an alternative to nd4j. The new back-end will be able to merge multiple primitive operations into one larger kernel by dynamically generating OpenCL code. The new back-end will support more optimized operations on the GPU and reduce the number of kernel executions. We expect our new version will achieve better computational efficiency.

[8] `INDArray.broadcast` for example

### 7.2 Distributed Model

Current DeepLearning.scala is only able to run on a standalone JVM, not a distributed cluster, thus it does not support "outrageously large neural networks" [Shazeer et al. 2017] that does not fit into memory of a single node.

Since our computational graph is made of monadic expressions that consist of closures, they can be serialized and executed remotely in theory. We are investigating how to build a distributed machine learning system based on remotely executed monadic expression. We will find out if this suggested approach can support more complex model than the parameter server approach can.

## 8 DISCUSSION

DeepLearning.scala is an unique library among all deep learning frameworks. Our approach of AD has some attributes that never appears in previous frameworks.

### 8.1 Interoperable Differentiable Computational Graph

There were two different mechanisms in state-of-the-art deep learning frameworks: Define-and-Run v.s. Define-by-Run.

State-of-the-art Define-and-Run frameworks [Abadi et al. 2016; Bergstra et al. 2010; Chen et al. 2015; Collobert et al. 2008; Intel 2016; Jia et al. 2014; Skymind 2017a] allows users to create computational graphs, which are immutable Abstract Syntax Trees (ASTs) of some object languages which can be evaluated by the framework runtime. Define-and-Run frameworks can schedule computational graphs to multiple CPUs, GPUs or other devices. However, the object languages have bad interoperability with the metalanguage. For example, a DeepLearning4j user cannot use Java control flows nor call Java native methods in neural networks.

State-of-the-art Define-by-Run frameworks [Google Brain 2017; Neubig et al. 2017; Paszke et al. 2017; Tokui et al. 2015] can eagerly execute actual forward pass calculation in user written code, and, at the same time, generate the internal states for running backward pass. Unlike Define-and-Run frameworks, Define-by-Run frameworks have good interoperability with the hosting language. Control flows and native function calls can be easily used during the execution of neural networks. However, Define-and-Run frameworks tend to store states and perform side effects when defining neural network structures *THE DEFINITION IS LITERALLY A PYTHON DEF IN PYTORCH, SIDE EFFECT IS PERFORM ONLY WHEN YOU CALL BACKWARD, AND THERE ARE PURE INTERFACE*, which makes this mechanism unsuitable for implementation in a purely functional flavor.

We discovered the third mechanism of monadic deep learning. Neural networks in DeepLearning.scala are immutable like in Define-and-Run frameworks, and interoperable with Scala like in Define-by-Run frameworks. *see above comment on purity*

### 8.2 AD in a Functional Library

Reverse mode AD as a functional library was previously impossible to implement without the ability to reflectively access and transform expressions associated with closures [Pearlmutter and Siskind 2008]. For example, if you want to create a `transform` function that returns the derivative for given function `f`:

```scala
def transform(f: Double => Double): Double => Double
```

Listing 29. Impossible transform function for AD

Obviously this `transform` function is impossible without the knowledge of the implementation of `f`.

Fortunately, in a statically typed language, the differentiable types and non-differentiable types should differ for type safety. The type signature of our AD function can use the differentiable type `DoubleLayer` instead of `Double`. It can be written as Listing 30:

```
def typeSafeTransform(f: Double => DoubleLayer): Double => SideEffects = {
    input: Double =>
  val tape = f(input).forward
  tape.backward(Do.now(1.0))
}
```

Listing 30. Type safe transform function for AD

Unlike [Pearlmutter and Siskind 2008]'s compiler primitive $\overleftarrow{\mathcal{J}}$, our `typeSafeTransform` can use the additional methods on `DoubleLayer`. As a result, our `typeSafeTransform` can be implemented without reflection, as an ordinary Scala function, instead of a compiler primitive or a macro.

We also change the derivative type to an opaque monadic type `SideEffects`. Unlike `Double` derivative, `SideEffects` can contain derivatives for more than one trainable variables, although this change exclude the ability of higher order numerical differentiation.

There are other approach for statically typed deep learning framework *citation*, however they all has different property, and we believe our approach is interesting in it's own right.

Haskell AD - do not mention tensor in anyway, thus hard to run on GPU Grenade - only allow you to define NN in traditional layer-wise sense, so no differentiable programming Lantern - it look really good to me. it's only problem AFAICS is not encapsualting param. Relay - is a seprate IR, and has almost no interoperability with python left

## 9 CONCLUSION

DeepLearning.scala is the first framework that achieves all the following goals:

- static type safety
- purely functional interface
- reverse mode AD
- multiple trainable variables
- interoperable internal DSL
- dynamic neural network
- statically type check the generated neural network

With the help of DeepLearning.scala, a normal programmer is able to build complex neural networks from simple code. He still writes code as usual, and the only difference is that the code written in DeepLearning.scala are differentiable, which contains trainable variables that learn knowledge.

## GLOSSARY

**computational graph** is an asynchronous monadic data type that manages the life cycle of tapes, whose type is `Do[Tape[Data, Delta]]` . 2, 6–8, 12, 15–20, 22, 23, 25

**differentiable expression** is a composable expression that supports operator overloading, whose type is `DoubleLayer`, `FloatLayer`, `INDArrayLayer`, or other subtypes of `Layer`. After a differentiable expression is built, it can perform `forward` pass to create a differentiable computational graphs. . 3, 6, 10, 17, 19, 25

**differentiable function** is a Scala function that returns a differentiable expression. It may represent a loss functions, a neural network or a subset of a neural network (e.g. a dense block in DenseNet[Iandola et al. 2014]) . 6, 10–12, 15, 17, 19

**trainable variable** is a scalar or vector weight in a model, whose type is `DoubleWeight`, `FloatWeight` , `INDArrayWeight`, or other subtypes of `Weight` . 1, 3, 6, 10, 12–15, 17, 19, 24

## ACKNOWLEDGMENTS

## REFERENCES


Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning.. In *OSDI*, Vol. 16. 265–283.

Atilim Gunes Baydin and Barak A Pearlmutter. 2016. Hype: Compositional Machine Learning and Hyperparameter Optimization. (2016). https://hypelib.github.io/Hype/

Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2015b. Automatic differentiation in machine learning: a survey. *arXiv preprint arXiv:1502.05767* (2015).

Atilim Gunes Baydin, Barak A Pearlmutter, and Jeffrey Mark Siskind. 2015a. Diffsharp: Automatic differentiation library. *arXiv preprint arXiv:1511.07727* (2015).

James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. 2010. Theano: A CPU and GPU math compiler in Python. In *Proc. 9th Python in Science Conf.* 1–7.

Christian Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. 1992. ADIFOR–generating derivative codes from Fortran programs. *Scientific Programming* 1, 1 (1992), 11–29.

Tongfei Chen. 2017. Typesafe Abstractions for Tensor Operations. *arXiv preprint arXiv:1710.06892* (2017).

Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

François Chollet et al. 2015. Keras. (2015).

Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. 2012. Multi-column deep neural networks for image classification. In *Computer vision and pattern recognition (CVPR), 2012 IEEE conference on.* IEEE, 3642–3649.

Ronan Collobert, K Kavukcuoglu, and C Farabet. 2008. Torch. In *Workshop on Machine Learning Open Source Software, NIPS*, Vol. 76.

John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research* 12, Jul (2011), 2121–2159.

Dumitru Erhan, Y Bengio, Aaron Courville, and Pascal Vincent. 2009. Visualizing Higher-Layer Features of a Deep Network. (01 2009).

Martin Fowler. 2010. *Domain-specific languages.* Pearson Education.

Google Brain 2017. *Eager Execution: An imperative, define-by-run interface to TensorFlow.* Google Brain. https://research.googleblog.com/2017/10/eager-execution-imperative-define-by.html

Andreas Griewank, David Juedes, and Jean Utke. 1996. Algorithm 755: ADOL-C: a package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software (TOMS)* 22, 2 (1996), 131–167.

Dave Gurnell. 2017. *The Type Astronaut's Guide to Shapeless.* Underscore Consulting LLP. https://underscore.io/books/shapeless-guide/

L. Hascoët and V. Pascual. 2013. The Tapenade Automatic Differentiation tool: Principles, Model, and Specification. *ACM Transactions On Mathematical Software* 39, 3 (2013). http://dx.doi.org/10.1145/2450153.2450158

Forrest Iandola, Matt Moskewicz, Sergey Karayev, Ross Girshick, Trevor Darrell, and Kurt Keutzer. 2014. Densenet: Implementing efficient convnet descriptor pyramids. *arXiv preprint arXiv:1404.1869* (2014).

Intel. 2016. BigDL. (2016). https://github.com/intel-analytics/BigDL

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 675–678.

Diederik Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

Alex Krizhevsky and Geoffrey Hinton. 2009. Learning multiple layers of features from tiny images. (2009).

Lanlan Liu and Jia Deng. 2017. Dynamic deep neural networks: Optimizing accuracy-efficiency trade-offs by selective execution. *arXiv preprint arXiv:1701.00299* (2017).

Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *Journal of functional programming* 18, 1 (2008), 1–13.

Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, et al. 2017. DyNet: The Dynamic Neural Network Toolkit. *arXiv preprint arXiv:1701.03980* (2017).

BCDS Oliveira, A Moors, and M Odersky. 2010. Type classes as objects and implicits. *ACM SIGPLAN Notices* (2010).

Erik Osheim and Jorge Vicente Cantero. 2017. *SIP-35 - Opaque types*. https://docs.scala-lang.org/sips/opaque-types.html

Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. 2017. *PyTorch: Tensors and Dynamic neural networks in Python with strong GPU acceleration*. http://pytorch.org/

Barak A Pearlmutter and Jeffrey Mark Siskind. 2008. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30, 2 (2008), 7.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1985. *Learning internal representations by error propagation.* Technical Report. California Univ San Diego La Jolla Inst for Cognitive Science.

Amr Sabry and Matthias Felleisen. 1993. Reasoning about programs in continuation-passing style. *Lisp and symbolic computation* 6, 3-4 (1993), 289–360.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538* (2017).

Aleksey Shipilev. 2018. JMH: Java Microbenchmark Harness. (2018). http://openjdk.java.net/projects/code-tools/jmh/

Skymind 2017a. *Deeplearning4j: Open-Source, Distributed, Deep Learning Library for the JVM.* Skymind. https://deeplearning4j.org/

Skymind 2017b. *ND4J: N-Dimensional Arrays for Java.* Skymind. https://nd4j.org/

Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. 2015. Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, Vol. 5.

Bo Yang. 2017. Dsl.scala: a framework to create DSL in Scala. (2017). https://github.com/ThoughtWorksInc/Dsl.scala/

Kenji Yoshida. 2017. *Scalaz: An extension to the core scala library.* https://scalaz.github.io/scalaz/

Matthew D Zeiler. 2012. ADADELTA: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701* (2012).

Tian Zhao, Xiaobing Huang, and Yu Cao. 2017. DeepDSL: A Compilation-based Domain-Specific Language for Deep Learning. *arXiv preprint arXiv:1701.02284* (2017).