DeepLearning.scala 2.0: Statically Typed Neural Networks

Solving the harder version of expression problem and performing monadic automatic differentiation in parallel

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

The Java and Scala community 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 language do not support the same features. Their users are not able to custom algorithms unless creating plenty of boilerplate code for hard-coded back-propagation.

We solved this problem in DeepLearning.scala 2.0. Our contributions are:

- We discovered a novel approach to perform automatic differentiation in reverse mode for statically typed functions that contain multiple trainable variables.
- 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 keep type safe.

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

1 INTRODUCTION

Backpropagation [Rumelhart et al. 1985] is the key feature in many deep learning frameworks. Combining with other optimization algorithms [Duchi et al. 2011; Kingma and Ba 2014; Zeiler 2012], deep learning frameworks change the values of trainable variables in neural networks during iterations, producing a model of knowledge learnt from training data.

Backpropagation can be seen as a special-purpose 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 calculus 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 the similar expression as creating ordinary non-differentiable functions. Since these frameworks do not require external language, models created by them can be easy integrated into a

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.

2017. XXXX-XXXX/2017/12-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

larger application alone with higher-level configurations [Chollet et al. 2015] or ETL (Extract, Transform and Load) process.

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 support multiple trainable variables.

Other deep learning frameworks in statically typed language (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.

In this paper, we present DeepLearning.scala, which achieves all the above goals, and still get type checked.

2 BASIC CONCEPTS

For example, suppose we are building a robot for answering questions in IQ test like this:

> 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 of the following signature[1]:

```
def guessNextNumber(question: Seq[Double]): DoubleLayer = {
  // Perform a matrix multiplication 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 matrix multiplication between question and weights by invoking higher-order functions `map` and `reduce`.

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

The `weights` and `bias` referenced by `guessNextNumber` must be initialized first:

Then the robot try to answer IQ test questions like calling an ordinary function:

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

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

[1]The code examples from Listing 1 to Listing 7 do not contain necessary **`import`** and configurations. For a runnable IQ test robot example 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 10×20 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.

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

Listing 2. Weight initialization

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

Listing 3. Inference on an untrained model

```scala
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:

```scala
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 squaredLoss 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.

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

When weights and bias have been adjusted to make the loss a very small number, guessNextNumber should return values that are very close to the expected answers.

This time, it will prints a number closed to 45, because the IQ test robot have finally learnt the pattern of arithmetic progression.

The IQ test robot 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 training or predicting.
- 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.

```
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

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

Listing 7. Inference on a trained model

- When `training` 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.

3 DYNAMIC NEURAL NETWORKS

DeepLearning.scala supports dynamic neural network. It means that the control flow of a neural network can differ according to its internal intermediate state when processing a special input. Especially, the ability to conditional enabling a sub-neural network is a crucial feature to build outrageously large neural networks [Shazeer et al. 2017].

Suppose we have two sub-neural networks, `leftSubnet` and `rightSubnet`. 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`.

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

Listing 9. Predefined gate network

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 eagerly executing the `gate`, shown in Listing 10:

```
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 got 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 vectorize `INDArrayLayer`, contain some lazily evaluated differentiable computational graph nodes, which will not be executed until their predict or train methods are invoked.

So, the two calls to the `predict` method 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. But what's even worse is, `input` contains a computational graph, too. Along with `gate`, it will be evaluated three times, which may contain complex future extracting process.

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`.

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 `flatMap`ping 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 (best)

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` 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 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.

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.

```
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

```
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

4 AD HOC POLYMORPHIC DIFFERENTIABLE FUNCTIONS

Neural networks created in DeepLearning.scala are differentiable functions, which contain expressions of differentiable types, which are any types that has their corresponding `DeepLearning` type classes, 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.

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

Ideally, a differentiable function should be an ad hoc polymorphic function that accepts heterogeneous types of input.

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`

Our solution is the dependent-type type class [Gurnell 2017] `DeepLearning` that witnesses any supported expressions including differentiable expressions, trainable variables, or vanilla non-differentiable types. The users can create type aliases to restrict the types of state during forward pass and backward pass as shown in Listing 14.

```
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 14. 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 15. A polymorphic differentiable function

Listing 15 is similar to Listing 13, except each argument of `polymorphicDifferentiableFunction` accepts `INDArray`, `INDArrayWeight` or `INDArrayLayer` respectively, not only `INDArrayLayer`.

Note that built-in operations including arithmetic operations, `max`, and `dot` are polymorphic differentiable functions, 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 use 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 16:

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

Listing 16. Dual number for forward mode AD

Arithmetic operations on those dual number can be implemented as Listing 17:

```
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 17. Arithmetic operations on dual number

5.2 Monadic Closure-based Dual Number

However, this approach is hard to type-check if we want to support multiple trainable variables. `PartialDelta` in Listing 16 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 updating trainable variables in a gradient descent based optimization algorithm. We can replace `PartialDelta` to a `UpdateWeights` closure.

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

Listing 18. Replacing PartialDelta to a closure

UpdateWeights in Listing 18 is a function type that contains side-effects to update trainable variables. In order to implement arithmetic operations for the new dual number, the operations on PartialDelta should be replaced to custom functions for UpdateWeights (Listing 19):

```
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 19. Replacing operations on PartialDelta to custom functions for UpdateWeights

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

The addition operation for closures is 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):

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

These arithmetic operations can be implemented in monadic data types as shown in Listing 20.

UpdateWeights, as a replacement to original PartialDelta, is a closure able to update derivatives for all weight 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 UnitContinuation
.

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.

```
type UpdateWeights = Do[Double] => SideEffects
object UpdateWeights {
  /** (f_0 + f_1)(x) = f_0(x) + f_1(x) */
  def plus(f0: UpdateWeights, f1: UpdateWeights) = { doX: Do[Double] =>
    f0(doX) |+| f1(doX)
  }

  /** (x_0 f)(x_1) = f(x_0 x_1) */
  def multiply(x0: Double, f: UpdateWeights) = { doX1: Do[Double] =>
    f(doX1.map(x0 * _))
  }
}
```

Listing 20. Arithmetic operations for the closure that contains side-effects

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

Listing 21. Monadic side-effects

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.

```
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 22. Create a dual number for a trainable variable

In Listing 22, 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 sophisticate approach to configure the hyperparameters

.

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 23.

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

Listing 23. Create a dual number for a non-trainable value

Finally, we can create a differentiable function as shown in Listing 24, whose leaf nodes are `createTrainableVariable` and `createLiteral`, and internal nodes are arithmetic operations in Listing 19.

```
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 24. 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.

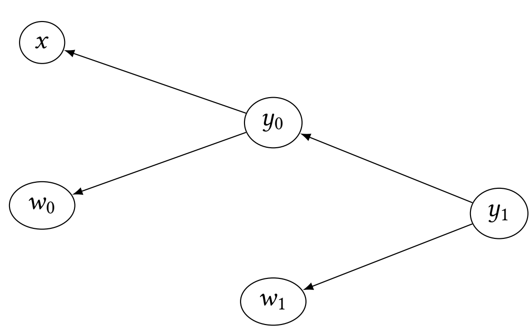

Fig. 1. A tree-structured computational graph

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 collecting 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.

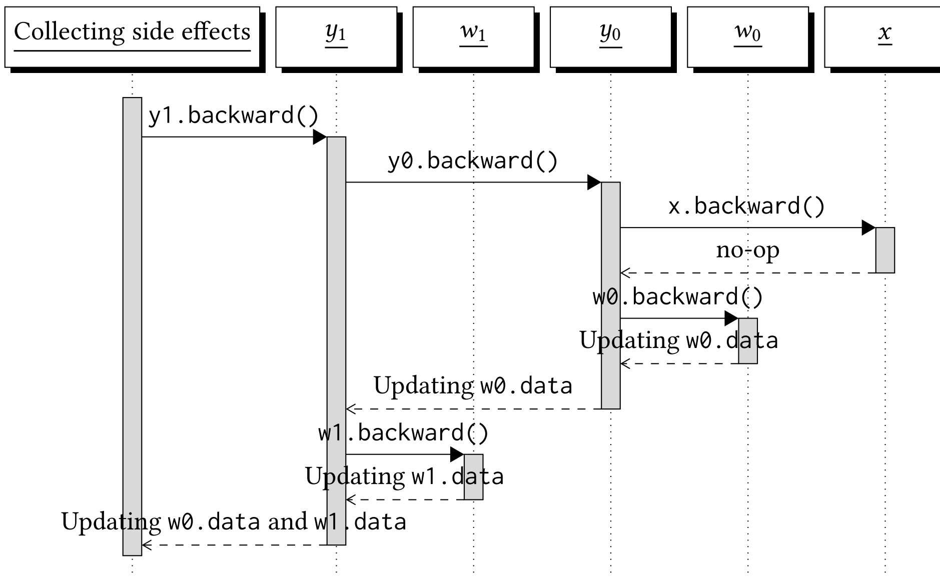

Fig. 2. Backpropagation for a tree-structured computational graph

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 18. We replaced `ClosureBasedDualNumber`'s hard-coded `Double` to type parameters `Data` and `Delta`, as shown in Listing 25.

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

Listing 25. 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 18 to Listing 25 supports multiple trainable variables, the closure-based computation has a performance issue in the case of diamond dependencies.

Listing 26 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 26. A diamond dependent differentiable function

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

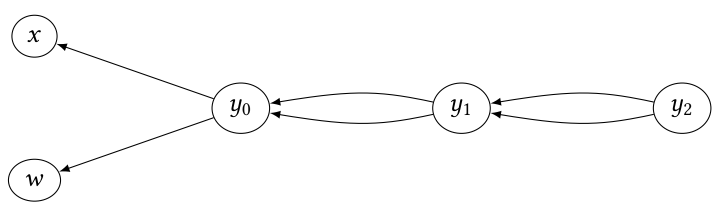

Fig. 3. A diamond dependent computational graph

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 reference counting algorithm for dual numbers, to avoid the exponential time complexity.

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.

When reference counting algorithm is enabled, `backward` is recursive any more. Instead, a wrapper only call `backward` of its dependencies when 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 sequential 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].

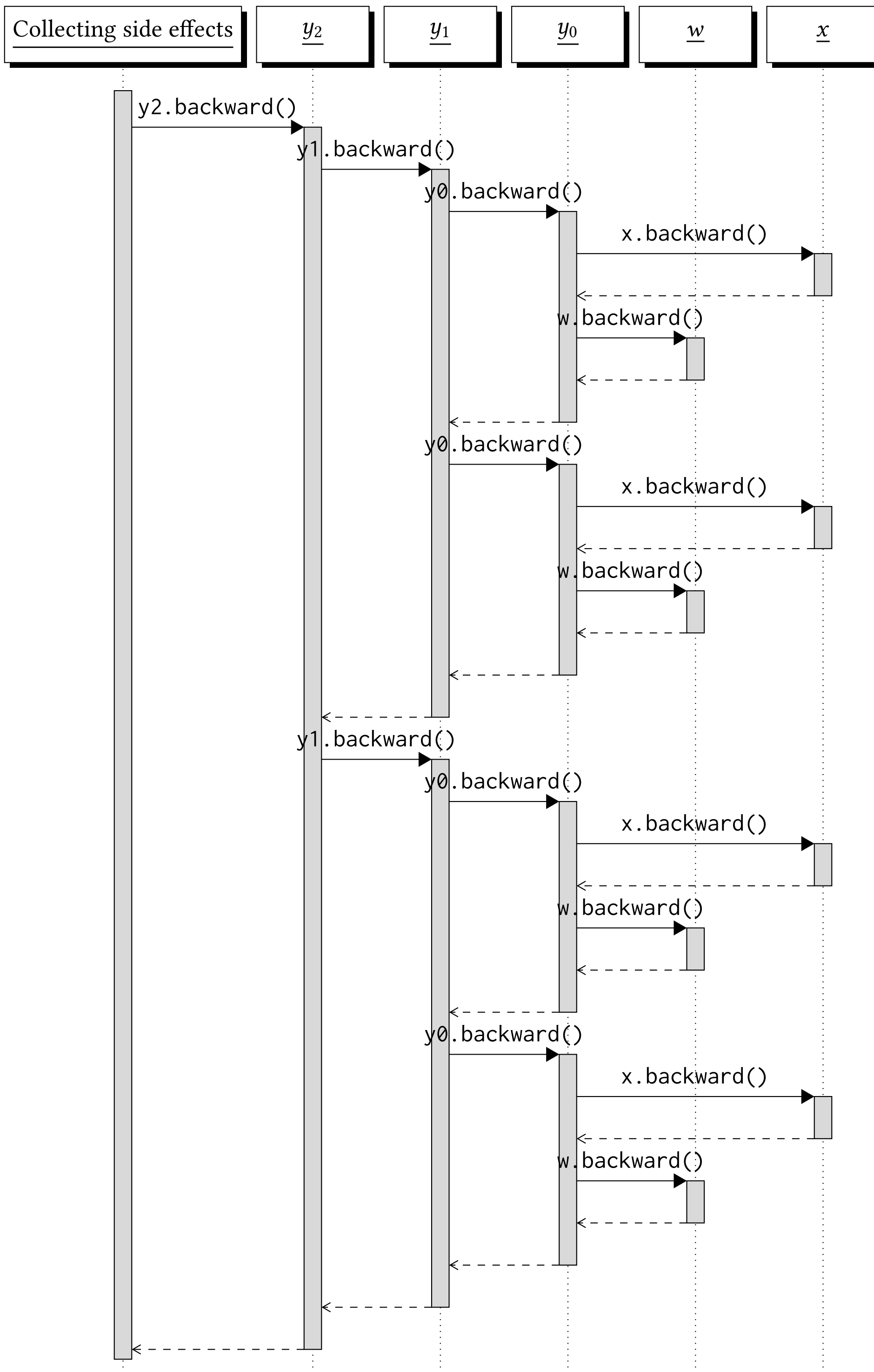

Fig. 4. Backpropagation for a diamond dependent computational graph

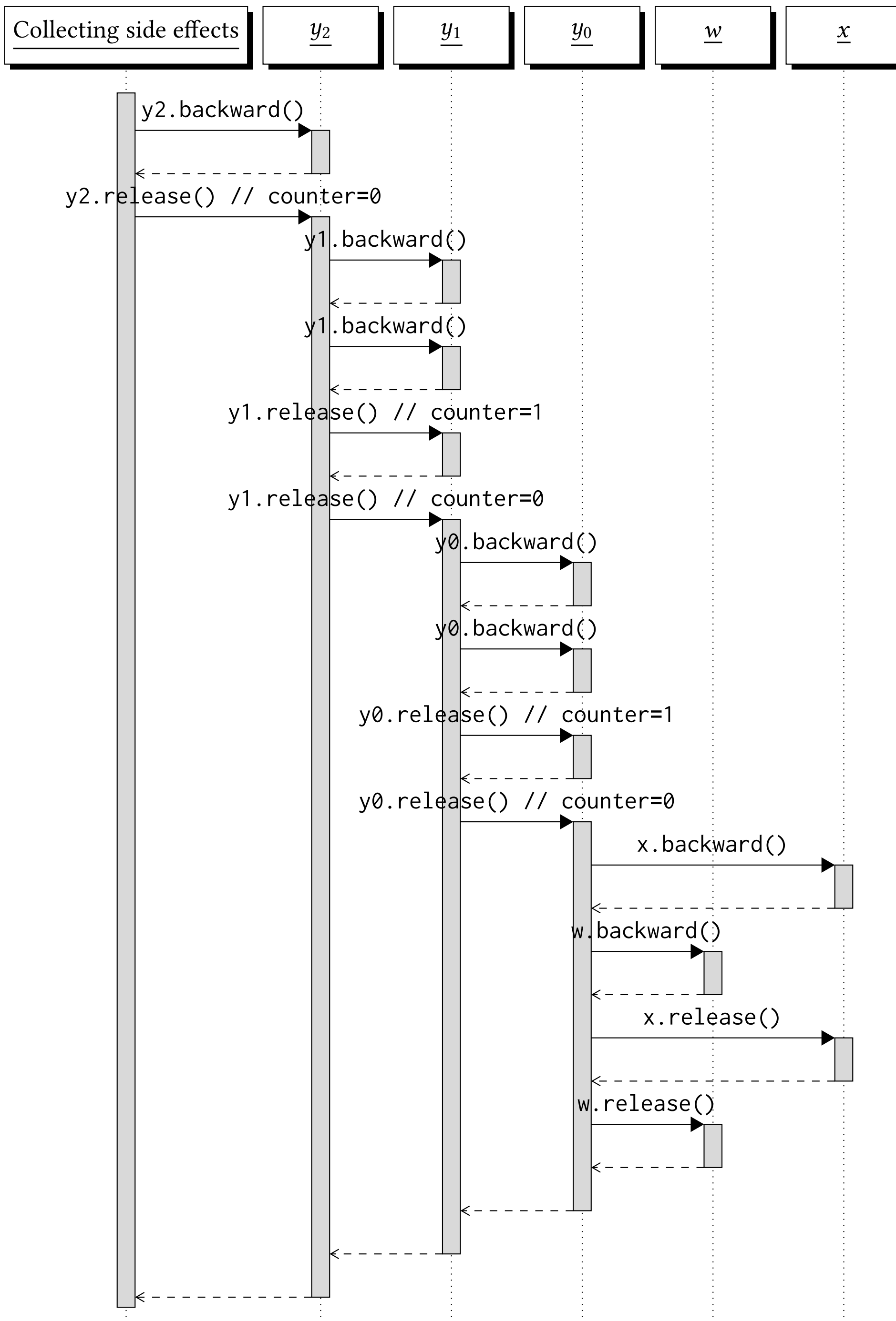

Fig. 5. Backpropagation for a diamond dependent computational graph (with reference counting)

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 the to build a computational graph (i.e. a `Do[ Tape[Data, Delta]]`), though 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 are increased during this step.
(4) Performing `backward` closure of the root node of the computational graph. The accumulator of delta of the root node is stored.
(5) Releasing of the root node of the computational graph. The reference counter of each node in computational graph are decreased to zero and `backward` closure of each node are 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 FUTURE WORK

6.1 New Back-end

Currently, DeepLearning.scala 2.0'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 GPU kernel executions if nd4j's CUDA runtime is used. These nd4j operations have a bad computational performance because: (1) Some operations [3] are extremely slow; (2) 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 GPU and reduce the number of kernel executions. We expect our new version will achieve better computational efficient.

6.2 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 are 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.

[3] `INDArray.broadcast` for example

7 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.

7.1 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 graph to multiple CPU, GPU 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, generates 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, which makes this mechanism unable to implement in a pure 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.

7.2 AD in a Functional Library

Reverse mode AD in a functional library was previously considered impossible to be implemented 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`:

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

Listing 27. 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 more powerful type `DoubleLayer` instead of `Double`. It can be written as Listing 28:

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.

8 CONCLUSION

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

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

Listing 28. Type safe transform function for AD

- type safe
- pure functional
- reverse mode AD
- multiple trainable variables
- interoperable internal DSL
- dynamic 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 knowledges.

GLOSSARY

computational graph is an asynchronous monadic data type that manages the life cycle of tapes, whose type is `Do[Tape[Data, Delta]]` . 2, 3, 5, 9, 12–18, 20

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, 4, 7, 8, 13, 14, 17, 20

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]) . 3, 4, 7–9, 12–14, 17

trainable variable is a scalar or vector weight in a model, whose type is `DoubleWeight`, `FloatWeight`, `INDArrayWeight`, or other subtypes of `Weight` . 1, 3, 4, 7–13, 17, 19

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).

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).

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).

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.

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).