ETH ZÜRICH

BACHELOR THESIS

Deep500 – A Deep Learning Meta-Framework and HPC Benchmark

Author:
Simon HUBER

Supervisor:
Dr. Tal BEN-NUN
Examiner:
Prof. Torsten HOEFLER

A thesis submitted in fulfillment of the requirements for the Bachelor degree

in the

ETH Computer Science Department
Scalable Parallel Computing Lab

Zürich, September 13, 2018

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

Titel der Arbeit (in Druckschrift):

Deep500 - A Deep Learning Meta-Framework and HPC Benchmark

Verfasst von (in Druckschrift):

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Huber	Simon

Ich bestätige mit meiner Unterschrift:

- Ich habe keine im Merkblatt „Zitier-Knigge" beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
Sursee, 12.09.18	Simon Huber

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.

Abstract

The rapid grow in machine learning in recent years, especially deep learning, has brought up a lot of different frameworks that help to create and run deep learning models. As a consequence, it makes it really difficult to compare results from one framework with another one. In this work, we split the functionality of deep learning frameworks into four blocks: Graph definition, graph operation implementation, optimizing network parameters and generating metrics. Breaking these blocks into standalone modules gives rise to the possibility to test any block in isolation as well as in connection with other blocks. Furthermore it makes it possible to swap out the same type of blocks of different frameworks against each other. By abstracting away the specifics of the deep learning frameworks, we build general optimizers that work for any framework in a non distributed case as well as in a distributed setting.

Contents

Abstract **iii**

1 Introduction **1**
1.1 Problem description . . . 1
1.2 In this work . . . 1

2 Background **3**
2.1 Neural Networks . . . 3
2.1.1 Problem Description . . . 3
2.1.2 Neural Network . . . 3
2.1.3 Inference . . . 4
2.1.4 Training . . . 4
2.2 Frameworks to build neural networks . . . 4
2.3 Protobuf . . . 5
2.4 ONNX . . . 5

3 Related Work **9**
3.1 High-level frameworks that are Backend independent . . . 9
3.1.1 Keras . . . 9
3.2 Convert ONNX format to a specific Backend . . . 9
3.2.1 Tensorflow-ONNX . . . 9
3.2.2 PyTorch-ONNX . . . 9
3.3 GraphPipe . . . 9
3.4 ONNX alternatives . . . 10
3.4.1 NNEF . . . 10

4 Design **11**
4.1 ONNX to Deep500 . . . 11
4.2 Backend . . . 11
4.2.1 Concrete Backend . . . 12
4.3 Deep500 API . . . 12
4.3.1 Add custom operations to the network . . . 12
4.3.2 Access to the internal state of a network . . . 13
4.3.3 Reference optimizers . . . 13
4.3.4 Training a network with native optimizers . . . 14
4.4 Distributed Training . . . 14
4.5 Metrics and Reports . . . 14
4.5.1 Testing . . . 14
4.5.2 Metrics . . . 15

5 Implementational Details **17**
5.1 Parse ONNX to Deep500 . . . 17
5.2 Backends . . . 18
Network class . . . 19
Visitor . . . 19
5.2.1 Optimization . . . 23
Reference Optimization . . . 23
1 step optimizer . . . 23
3 step optimizer . . . 23
Optimizer . . . 24
Usage . . . 24
Training a network with native optimizer . . . 24
5.2.2 UML Design . . . 26
5.3 Metrics . . . 27
5.3.1 Dataset . . . 27
5.3.2 Runner . . . 27
5.3.3 Tickers . . . 27
5.3.4 SummaryTicker . . . 28
5.3.5 Metrics UML . . . 29
5.3.6 Use Cases . . . 30
Record inference and optimization times . . . 30
A website ticker . . . 30
5.4 Non Runner Metrics . . . 31
Layer 0 Testing specific operations . . . 31
Layer 1: Comparing 2 optimizers step by step . . . 33
5.5 Distributed Training . . . 34
5.5.1 Distributed Reference Implementation . . . 34
Usage in code . . . 36
5.5.2 Distributed Native Optimizers . . . 38
Horovod Distributed Optimizer . . . 38
Caffe2 Distributed Optimizer . . . 38

6 Benchmarking and Experiments **43**
6.1 Benchmarking . . . 43
6.2 Experiments . . . 43
6.2.1 1 GPU . . . 44
6.2.2 3 GPUs . . . 44
6.2.3 Comparisons between vanilla backend and deep500 implementation . . . 47

7 Conclusion **51**
7.1 Accomplishments . . . 51
7.2 Future Work . . . 52
7.2.1 Fix performance problems with reference optimizers . . . 52
7.2.2 Intermediate optimization . . . 52
7.2.3 Distributed Optimization . . . 52
7.2.4 Write optimized code directly . . . 52

List of Figures

1.1 Rough diagram of the abstraction layers interplay 2

2.1 Argmax operation cut out of ONNX catalogue 6

4.1 ONNX Parser 12
4.2 High-level idea of backends 12

5.1 UML Diagram Network 19
5.2 UML Diagram Visitor 22
5.3 UML Diagram Optimization 26
5.4 Metrics Diagram UML 29
5.5 Image of web interface 31
5.6 Consistent Parameter Server 36
5.7 Inconsistent Parameter Server 36
5.8 Inconsistent Decentralized 36

6.1 Comparisons of Momentum optimizers 44
6.2 Only Tensorflow Adam Loss vs Samples 45
6.3 Only Tensorflow Adam Loss vs Time 46
6.4 Only Adam optimizer vs Horovod 46
6.5 Caffe2 consistent parameter server accuracy by epoch 47
6.6 Caffe2 consistent parameter server loss by time 48

List of Tables

5.1 Network class API 20
5.2 Output test accuracy 32
5.3 Time output operator test 33
5.4 Csv file generated after 5 steps 34
5.5 Reference Distributed optimizers in Deep500 35
5.6 Distributed communication API Deep500 37
5.7 Caffe2 distributed operation 39

6.1 Native Tensorflow vs deep500 tensorflow 48

Chapter 1

Introduction

1.1 Problem description

There are currently many Deep Learning frameworks. Most of these frameworks have at least the following 4 things in common:

1. Definition: High level definition of a neural network. This contains the definition of the Graph, the learning method, its loss function and defined output.
2. Implementation: The actual implementation of the specific operations defined in the graph.
3. Training: Run training data through the network and update the weights according to some optimization algorithm and the gradients.
4. Metrics: During training or inference phase, record different metrics as for example accuracy and time used to run an optimization step.

The parts mentioned above are fully integrated in the current frameworks and there is no way to use a part from e.g. Tensorflow in Caffe2 or vice versa. But there had been attempts to develop standalone modules from the list above with a fix defined API. One of these is the ONNX, project which provides an independent format to define neural network graphs. The Keras project [10], on the other hand, tries to facilitate some of the definition part from the actual implementation of the network by allowing either Tensorflow or torch as backends.

There result a multitude of problems from this entanglement: It is difficult to compare the same parts of two different frameworks and not possible to swap the same type of blocks of two different frameworks.

1.2 In this work

We created a framework, called Deep500, that detangles the specific blocks further by providing new abstraction levels. The abstraction layer is a fixed defined API between the different blocks identified above.
By defining a set of all neural network basis functions it is possible to abstract away a specific implementation of any backend.
Every deep learning framework has some kind of mechanism to run and examine a neural network this is hidden behind the network abstraction layer.
Because the access to the specific neural networks is hidden behind the network abstraction layer we can build a general optimization framework on top of it that works for any framework.
Using the non distributed optimization framework we build a distributed optimization framework that can utilizes many GPUs and computers in parallel and works

for any non distributed optimizer.
By implementing the different deep learning frameworks behind the API one gets the possibility to test and report on every block in the framework independently: Single operations, whole networks, optimizers and distributed optimizers. These results then can be compared against other frameworks.
A rough diagram of the parts discussed here can be seen in figure 1.1.

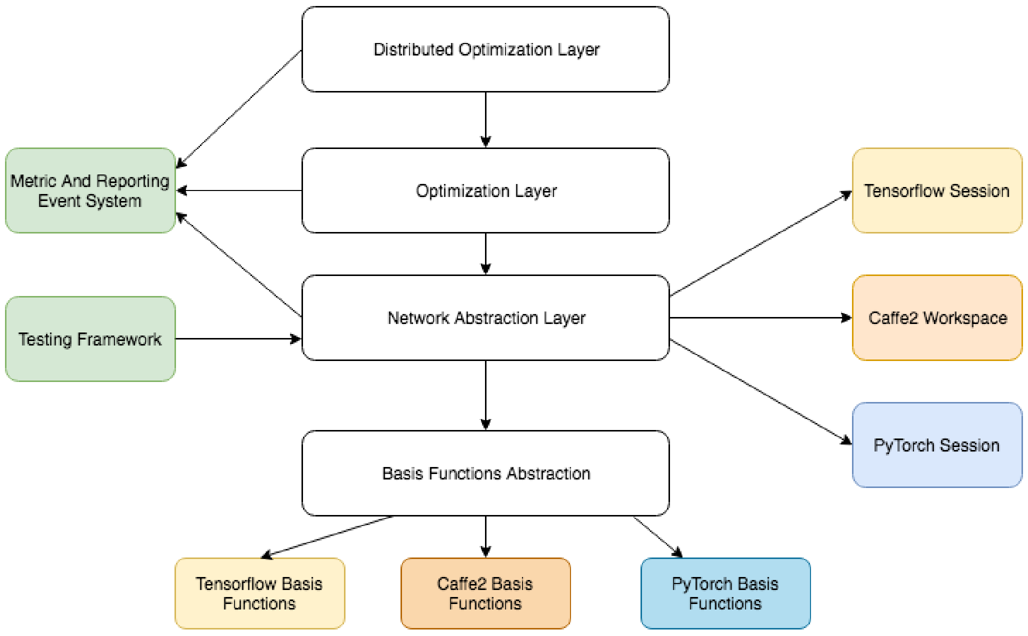

Distributed Optimization Layer
Metric And Reporting Event System
Optimization Layer
Tensorflow Session
Testing Framework
Network Abstraction Layer
Caffe2 Workspace
PyTorch Session
Basis Functions Abstraction
Tensorflow Basis Functions
Caffe2 Basis Functions
PyTorch Basis Functions

FIGURE 1.1: Rough diagram of the abstraction layers interplay

Chapter 2

Background

This chapter gives a formal introduction into the theory, concepts and technologies this work is based upon.

2.1 Neural Networks

2.1.1 Problem Description

The following problem or some kind of variation of it occurs a lot in the real world: Given some set of objects $\{(x_i, y_i) \in \mathcal{X} \times \mathcal{Y}\}_{i \leq n}$, a mapping function $f : \mathcal{X} \to \mathcal{Y}$, and a loss function $\ell : ((\mathcal{X} \times \mathcal{Y})^n \times f) \to \mathbb{R}$ find the function $f^* : \mathcal{X} \to \mathcal{Y} = argmin_f \ell$ which minimizes ℓ given the data $(\mathcal{X} \times \mathcal{Y})^n$.
Example: Given n flowers, let $\mathcal{X}$ be the a common subset of the properties of flowers (e.g. color, size) and $\mathcal{Y}$ its corresponding flower type (e.g. sunflower, orchidaceae) this gives rise to the data $(\mathcal{X} \times \mathcal{Y})^n$. Then the mapping function $f : \mathcal{X} \to \mathcal{Y}$ gives the flower type for given properties. A natural loss function ℓ for this kind of problem is the number of wrong mapping. And then the solution f^* is the mapping that makes the least mistakes.

2.1.2 Neural Network

In broadest terms a neural network is a composition of multiple possibly different functions. This composition of functions is just a new function which could possibly again be combined with other functions. So the author encourages the reader to think about a neural network just in terms of simple computational building blocks (functions) that get stacked on top of each other to create an even bigger building block.
Formally we could describe a network in the following way: We have a set Φ of n possible basis functions with $\phi_i \in \Phi : \mathbb{R}^{d_i} \to \mathbb{R}^{m_i}$ where $n, d_i, m_i, \in \mathbb{N}$. New functions are created by composition: $f := \phi_i \circ \phi_j$ with $i, j \leq n$ ($f := \phi_i \circ id = \phi_i$ possible). It is often the case that a function $\phi_i : \mathbb{R}^{d_i} \to \mathbb{R}^{m_i}$ has d_i amount of elements as input but the function $\phi_j : \mathbb{R}^{d_j} \to \mathbb{R}^{m_j}$ it is composed with provides only $m_j \leq m_i$ dimension of its output. We circumvent this by stacking functions into vectors as follows: $\begin{pmatrix} \phi_j \\ \theta \end{pmatrix}$ where $\theta \in \mathbb{R}^{m_i - mj}$ and is called a parameter. Note that such a parameter could also be just another function. Since ϕ_j itself has also some input the input vector to the whole composition $f := \phi_i \circ \phi_j$ is visualized as follows: $f(\begin{pmatrix} x \\ \theta \end{pmatrix})$.
Since most of the functions work this way a composition $f_l := f_0 \circ f_1 \circ \cdots \circ f_{l-1}$ can

have up to millions of parameters with the final call looking like:

$$f_l(\begin{pmatrix} x \\ \theta_1 \\ \dots \\ \theta_k \end{pmatrix}), k \gg n.$$

The term neural network refers commonly to the composed function f_l.

2.1.3 Inference

The action of mapping an object $x \in \mathcal{X}$ into a category $y \in \mathcal{Y}$ is called inference in this text. Applied to neural networks this means to call the function composition by supplying the necessary input: $y = f((x, \theta_1, ..., \theta_n)^T)$.

2.1.4 Training

The previous parts raise 2 obvious and important questions:

1. Which and how should one compose the function f_l (the neural network)?
2. Which parameters are best such that the best categories get chosen for some object $x \in \mathcal{X}$?

There are a multitude of methods to solve both of these problems. But since in this work it is assumed that the function f and therefore implicitly the loss function ℓ is already given no solutions to 1 are provided.
Additionally the term deep learning is considered a synonym to neural network in this text.

2.2 Frameworks to build neural networks

There are multiple frameworks that help build a neural network. The most commonly used frameworks are: Tensorflow [1], PyTorch[16] and Caffe2 [3].
These framework commonly provide:

1. **Definition:** These frameworks give simple ways to build the neural network (function f_l) and already provide many building blocks (base functions) so that complicated networks can be written in a concise way. Each of the mentioned frameworks provide this functionality through one or multiple well known programming languages.
2. **Implementation:** Each base function gets called at some point if inference is run. To get as much speed as possible these functions are implemented in high performance code (commonly c++).
3. **Training:** The algorithms that find the parameters mentioned in the training part are provided by the framework.
4. **Metrics:** Most of the time inferring data and training a neural network is just not enough. Reporting the average inference time, losses, etc. are data worth saving and analyzing. In this respect frameworks have a lot to catch up. While there is the possibility to program everything by hand it would certainly make sense to provide this features out of the box.

2.3 Protobuf

Google protobuf is a framework that helps building high performance protocols. With protocol it is meant a way to serialize and dezerialize objects from and to binary code. Tensorflow, Caffe2 and deep500 all incorporate protobuf to communicate between backend and definition part of the frameworks. A protocol can be defined in the following simple way:

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
}
```

LISTING 2.1: Define a simple person protocol

Multiple messages can be reused inside other messages like the concept of classes that is known from object oriented languages.
Is such a protocol defined, protobuf generates endpoints (serializer, dezerializer) for specified programming languages. The usage of which is shown in the two listings below.
Serializing data looks the following way:

```
Person john = Person.newBuilder()
    .setId(1234)
    .setName("John")
    .setEmail("john@doe.com")
    .build();
output = new FileOutputStream("some_file");
john.writeTo(output);
```

LISTING 2.2: Serialize a person

Since this is now a compressed binary file unable to be read by humans the listing 2.3 shows how to work with it in c++:

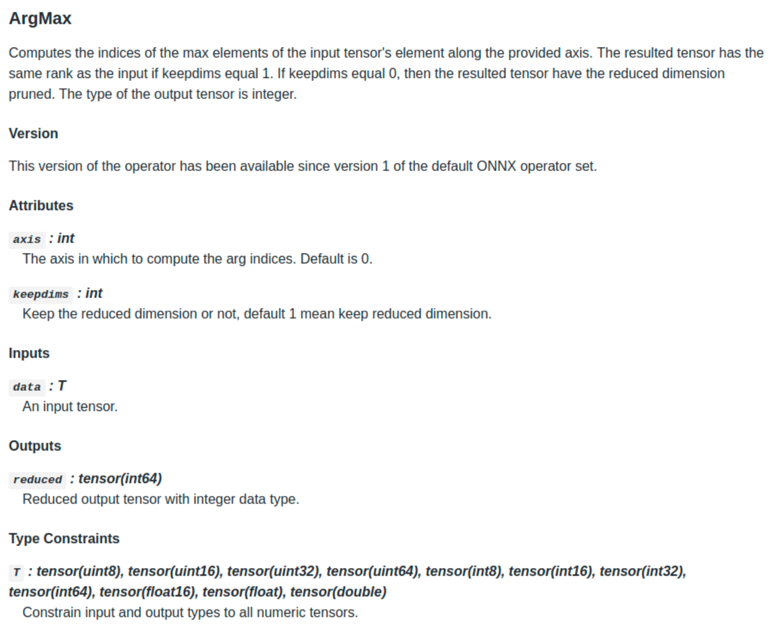
```
Person john;
fstream input("some_file", ios::in | ops::binary);
john.ParseFromIstream(&input);
id = john.id();
name = john.name();
email = john.email();
```

LISTING 2.3: Read the serialized person (deserialize)

Protobuf is used by Caffe2 as well as by Tensorflow to define the network and then send it to its c++ backend.

2.4 ONNX

When one creates a neural network with tensorflow or Caffe2 the framework builds a concise internal representation in the building process. When inference is run this data is sent to the c++ implementation where it is interpreted and executed. This data is built with protobuf.
ONNX now tries to replace this representation in a standardized protocol. Similarity claims between LLVM to ONNX can certainly be made. The frameworks currently fulfill the purpose of a frontend (defining a network) and backend (executing the intermediate language). The building blocks that one can use are defined in the ONNX catalogue. The definition of such a building block in case of argmax can be seen in



ArgMax

Computes the indices of the max elements of the input tensor's element along the provided axis. The resulted tensor has the same rank as the input if keepdims equal 1. If keepdims equal 0, then the resulted tensor have the reduced dimension pruned. The type of the output tensor is integer.

Version

This version of the operator has been available since version 1 of the default ONNX operator set.

Attributes

`axis` *: int*
The axis in which to compute the arg indices. Default is 0.

`keepdims` *: int*
Keep the reduced dimension or not, default 1 mean keep reduced dimension.

Inputs

`data` *: T*
An input tensor.

Outputs

`reduced` *: tensor(int64)*
Reduced output tensor with integer data type.

Type Constraints

`T` ***: tensor(uint8), tensor(uint16), tensor(uint32), tensor(uint64), tensor(int8), tensor(int16), tensor(int32), tensor(int64), tensor(float16), tensor(float), tensor(double)***
Constrain input and output types to all numeric tensors.

FIGURE 2.1: Argmax operation cut out of ONNX catalogue

figure 2.1.
ONNX is defined as a protobuf protocol. The two most important protobuf messages are NodeProto and GraphProto both seen in 2.4 and 2.5 which are copied from the ONNX framework code. NodeProto defines an Operation with inputs, outputs, attributes and most importantly a string op_type. The op_type has to correspond to one of the op_types in the ONNX catalogue. GraphProto on the other hand is just the container that holds all the operations in the graph in topological order.

```
message NodeProto {
  repeated string input = 1;
  repeated string output = 2;

  // An optional identifier for this node in a graph.
  // This field MAY be absent in ths version of the IR.
  string name = 3;

  // The symbolic identifier of the Operator to execute.
  string op_type = 4;
  // The domain of the OperatorSet that specifies the operator
  // named by op_type.
  string domain = 7;

  // Additional named attributes.
  repeated AttributeProto attribute = 5;

  // A human-readable documentation for this node.
  // Markdown is allowed.
  string doc_string = 6;
}
```

LISTING 2.4: ONNX NodeProto definition

```
message GraphProto {
  // The nodes in the graph, sorted topologically.
  repeated NodeProto node = 1;
}
```

LISTING 2.5: ONNX GraphProto definition

Using protobuf yields some disadvantages: Because the python object you get from protobuf is a bunch of C methods you always have to check the proto file to know what methods you can call. Furthermore it would be much more convenient for a user to work with native types like *float* instead of the from protobuf given general python-c-wrapper objects like *FloatAttributeType*.

For this and a multitude of other reasons we first parse the ONNX into a Deep500 pure Python object to further decouple from C.

Chapter 3

Related Work

In this section we cover some related work that follows similar paths to resolve the mentioned problems in the introduction.

3.1 High-level frameworks that are Backend independent

3.1.1 Keras

Keras [4] is a framework that concentrates on making the definition part of a neural network as simple as possible. It offers many compositions already out of the box. It supports different optimizers as well as distributed optimization over multiple GPUs. Some of which only work with a specific backend. At the beginning Keras offered 2 different backends Tensorflow and theano and started to support CNTK recently. It allows to change between the backends via a configuration file.

3.2 Convert ONNX format to a specific Backend

3.2.1 Tensorflow-ONNX

Tensorflow-ONNX [18] is a project that allows you to do inference via Tensorflow given an ONNX file.

3.2.2 PyTorch-ONNX

PyTorch [17] another deep learning framework has a part of ONNX integrated that allows to execute a given ONNX file. Both Caffe2 and PyTorch are from Facebook. They recently merged (beginning 2018) the code into the same repository and allow to switch the executing backend after the ONNX file is loaded. Since Facebook is one of the driving factors behind ONNX and Caffe2 orients itself already strongly on ONNX they will probably replace the internal representation they have with ONNX at some point.

3.3 GraphPipe

At 15. August oracle published a blog post [7] where they introduced GraphPipe. They describe it as a protocol and a collection of software designed to simplify machine learning model deployment and decouple it from framework specific implementations. They use a tool called flatbuffers [5] a similar tool as protobuf to save the serialized models. And also offer with *graphpipe-onnx* and *graphpipe-tf* the interface between the GraphPipe representation, Tensorflow and ONNX. Since they understand ONNX they also can execute Caffe2 and PyTorch. Additionally to these services with GraphPipe you can run a neural network as it would be a server and send tensors to it and get a tensor back in numpy for example.

3.4 ONNX alternatives

3.4.1 NNEF

Press release published 13. August. [8] NNEF from the khronos group [13] is a readable framework independent format to represent neural networks. Very similar to ONNX but readable. NNEF also wants to offer tools that optimize the graph that NNEF represents. Of course there is an NNEF-exporter needed for each of the frameworks as well as a parser that imports NNEF into a framework.

Chapter 4

Design

This chapter gives a detailed overview of the chosen approach to solve the problems already outlined in the previous chapters.

4.1 ONNX to Deep500

Relying on ONNX results in some disadvantages:

1. The high-performance protocol implemented in C++ handles everything over strings. This makes it very hard to write understandable code. E.g. The field `op_type` in NodeProto can be max, min, prod, add etc. Input and Output nodes are defined with string keys and not object references.
2. The NodeProto protobuf message identifies the type of a node (the function it represents, argmax, add, etc.) with the string field op_type. This means that only a cumbersome inspection of this object provides further information about the type, input, output and arguments of it.
3. Given the root node there is no predefined way to access all the nodes expect from looping over it.

The deep500.parser package resolves these problems.
The parser is responsible to parse the provided ONNX model file and encapsulates the model into a deep500 object. It is important that the complicated underlying protocol level is abstracted away.
The problems with the nodes that are only identifiable with the op_string are solved differently. A generator is created that according to the C++ schema defined by ONNX generates a python class for each operation. This class has exactly the inputs, outputs and attributes defined in the ONNX catalogue and not just a list of strings. Additionally to this class files a dictionary variable is initialized that can map op_type to class. When the ONNX file is parsed each corresponding explicitly typed object is created automatically for each node.

The visitor pattern was used to implement the access to the underlying node data structure. The visitor for the nodes is auto-generated based on all possible nodes. Only the visitor for the top-level objects was manually coded.

4.2 Backend

The basis functions $\phi_i \in \Phi$ have to be executed fast because they're called very often. The high-performant implementation and execution of these functions is called the backend.

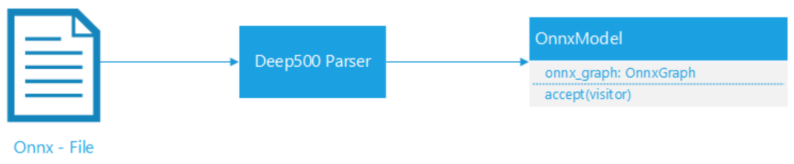


FIGURE 4.1: ONNX Parser

4.2.1 Concrete Backend

Problem: Currently, a framework is tightly coupled to a specific backend. This does not allow to exchange and compare different backends with each other. Deep500 provides a standardized interface that allows to switch to a different backend without changing any code.

The big picture is the following: For each backend there is a different visitor that transforms the ONNX operation to the backend specific implementation. But it isn't as simple as that. Calling, training and generally handling the graphs created by the visitors for the specific backends are different in each framework. Also the internal state of a backend should not be held directly in a visitor. The visitor is thought a stateless actor that transforms ONNX to the language of the backend. Because of this reasons there is an extra `Network` class that encapsulates the internal state of each network after transforming the ONNX model to a backend specific model. This `Network` class encapsulates access to the outer world and abstracts away a specific backend behind common methods (inference, backpropagation, gradients). Therefore each backend is made up of at least one visitor and a network class.

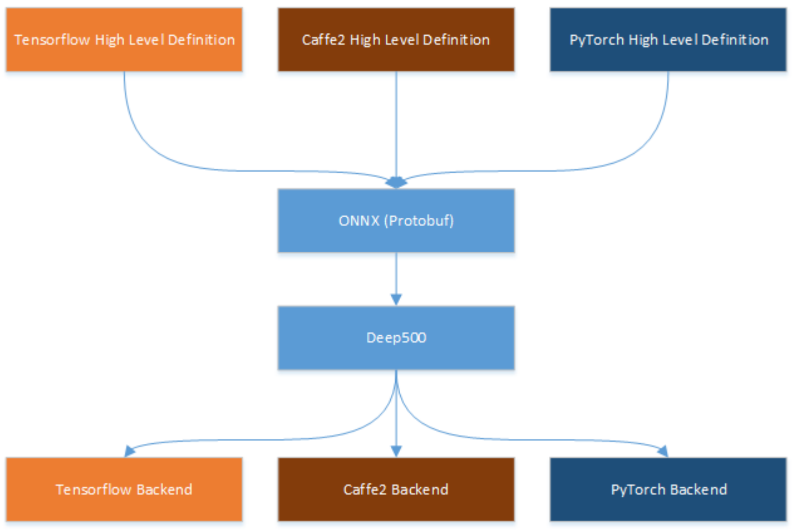

Tensorflow High Level Definition
Caffe2 High Level Definition
PyTorch High Level Definition
ONNX (Protobuf)
Deep500
Tensorflow Backend
Caffe2 Backend
PyTorch Backend

FIGURE 4.2: High-level idea of backends

4.3 Deep500 API

4.3.1 Add custom operations to the network

As soon as the parsing is done we have an internal representation of the network graph. This allows deep500 to provide the functionality to extend the graph by custom operations, additional given basis functions or new output. To implement a custom basis function different from ONNX one has to implement the super class Operation and extend the custom operations visitor. This is helpful because ONNX is still in development and does not provide common loss functions out of the box.

4.3.2 Access to the internal state of a network

Only inference and backpropagation is not enough for a fully functioning neural network. Often one is interested to extract, change and analyse specific values in a neural network. An optimizer must be able to extract and change values while some values are needed to generate specific metrics.
It makes sense to add this additional responsibility to the `Network`. So the `Network` class offers two new methods for fetching values from a `Network` and feeding them back in.

4.3.3 Reference optimizers

The algorithms that find the correct parameters for the neural network during training are called optimizers. The optimizers that work directly on the specific backend graphs are . And the native optimizers are generally faster then optimizers we build with help of the abstraction layer because we can't profit fully from backend specific optimization one could do. But there are cases when one cares less about speed and more about:

1. **Portability:** Same optimizer with same hyperparameter runs on different frameworks.
2. **Correctness:** Optimizer is easy to analyze, debug and to evaluate correctness of an optimizer.

Deep500 has the naming convention of optimizers that follow portability and correctness to call "reference implementation", because they should be considered as a reference for faster implementations. They work on the access methods discussed above provided by the Network class by extracting parameters from the network, change them and put them back in. While optimizers work so similar deep500 offers multiple interfaces to implement optimizers depending on the grade of control one desires:

1. `class Optimizer`: This is a base class and has only the method train in which everything has to be coded by hand.
2. `class ThreeStepOptimizer`: In the three step optimizer one has 3 points of contact to interact with the network:
   (a) New t: This indicates that a new input is happening. Some optimization algorithms have some internal step counter (e.g. Adam) this can be handled here.
   (b) Pre inference and backpropagation: At this point parameters can be changed before forward and backpropagation is happening.
   (c) Update rule: At this point all gradients are calculated and based on this gradients any update of the parameters is happening.
3. `class OneStepOptimizer`: The one step optimizers finally gives the least possible access to underlying mechanism. It only expects some update rule. A common example for this case would be gradient descent containing the update rule: $x_{t+1} = x_t - \lambda * f'(x_t)$.

4.3.4 Training a network with native optimizers

While reference optimizers are useful there are already implemented native optimizers in each of the popular deep learning frameworks. Of course it must be allowed to still use them. Common native optimizer work directly on the graph and update weights in an online fashion by adding operations that directly update the parameters. The `class ViaGraphOptimizer` has a method build, in which all the specific update operations have to be added onto the graph. The build method works directly on the native graph objects and not with the networks possibility to add operations, because the native optimizers expect the specific graph objects common to its backend.

4.4 Distributed Training

One corner stone Deep500 is that it distributes the training of your network with virtually no extra work from the side of an API user. This works based on the two reference implementation classes: One step optimizer and three step optimizer. The distributed optimizer works on these two classes to distribute new gradients and parameters over MPI before or after the update rules are executed.

4.5 Metrics and Reports

Measuring and especially comparing results seems, at least for the author, difficult in the current deep learning frameworks. Deep500 tries to make a step in the right direction by giving the opportunity to test the different building blocks in isolation and by providing extensive possibility to measure results during training and testing.

4.5.1 Testing

Direct testing is used to verify that the specific blocks work isolated or in combination. ONNX offers around 900 tests by providing a model, input and the expected output out of the box. The testing framework is in part built upon this ONNX tests which can be reused with the network abstraction known from Deep500. The following testing possibilities exist:

1. **Layer 0 (Low - level operations) inference:** Measurements of the operations implemented in the visitors for a) Correctness b) Accuracy c) Time. Using the deep500 super classes gives all these measurements for free by reusing the tests from ONNX.

2. **Layer 0 (Low - level operations) backpropagation:** As soon as it is verified that inference works it makes sense to test the gradient calculation. This is done by reusing the ONNX tests. First the gradient is calculated numerically and then compared with the gradients calculated by backpropagation.

3. **Layer 1 (Network inference and backpropagation):** Tests are not limited to models that contain only one operation. There are some ONNX tests that do inference on big networks and therefore both inference and backpropagation can be tested.

4.5.2 Metrics

Testing alone is not enough. Monitoring, data generation and reporting are useful tools that help analyze a network. By encapsulating the process of running and training a network into a specific class called `Runner`, Deep500 introduces an event system that triggers at specific points in time which allows to either generate data or processing it. Some examples of such events are before and after optimizing, or before and after running over a new epoch. With this event system we can test the following two next layers:

1. **Layer 1 (Network optimization and inference):** The event system allows access to all important and interesting events that happen during inference and training in a network.
2. **Layer 2 (Distributed optimization):** The same event system that captures events from local allows to capture events that run globally over distributed optimizers.

If a new method is developed one wants to know how it compares against other implementations. When the Deep500 framework is used one gets automated reports on different metrics. These reports make it easy to compare different implementations against each other. Following this it would be simple to create a website which ranks different implementations against each other on a public ranking and makes it possible to reference the reports.

Chapter 5

Implementational Details

This section provides an in depth view into specific parts of the Deep500 implementation.

5.1 Parse ONNX to Deep500

As explained earlier the ONNX files are parsed into pure deep500 python objects. All the classes and functions concerned with this are stored in *deep500.onnx_parser*.

```
def load_and_parse_model(path: str) -> OnnxModel:
```

LISTING 5.1: Parse onnx file into deep500.

The object `OnnxModel` contains an object of type `OnnxGraph` which holds the for us important list of operations / nodes / base functions Φ in topological order. An operation implements `OnnxNode` which is the deep500 equivalent of the ONNX `NodeProto`. To make information retrieval about ONNX as easy as possible the ONNX operation documentation is available directly in the concrete deep500 python class.

The class Operation is abstract and all the concrete implementations of Operations correspond to all the possible building blocks that ONNX defines in its ONNX operator catalogue. A specific example is the following "MaxPool" Operation. Which is defined in the link: [12].

```
class MaxPool(Operation):
    """
 MaxPool consumes an input tensor X and applies max pooling across the
 the tensor according to kernel sizes, stride sizes, and pad lengths.
 max pooling consisting of computing the max on all values of a
 subset of the input tensor according to the kernel size and downsampling the
 data into the output tensor Y for further processing. The output spatial shape will be following:
 '''
 output_spatial_shape[i] = floor((input_spatial_shape[i] + pad_shape[i] - kernel_spatial_shape[i]) / strides_spatial_shape[i] + 1)

 * pad_shape[i] is sum of pads along axis i
 '''

 'auto_pad' is a DEPRECATED attribute. If you are using them currently, the output spatial shape will be following:
 '''
 VALID: output_spatial_shape[i] = ceil((input_spatial_shape[i] - kernel_spatial_shape[i] + 1) / strides_spatial_shape[i])
 SAME_UPPER or SAME_LOWER: output_spatial_shape[i] = ceil(input_spatial_shape[i] / strides_spatial_shape[i])
 '''
 And pad shape will be following if 'SAME_UPPER' or 'SAME_LOWER':
```

```
'''
pad_shape[i] = (output_spatial_shape[i] - 1) * strides_spatial_shape[i] +
    kernel_spatial_shape[i] - input_spatial_shape[i]
'''
The output of each pooling window is maximum number of elements exclude
   pad.
    """

    def __init__(self, input, output, name, op_type, domain, attributes,
    doc_string):
        super(MaxPool, self).__init__(input, output, name, op_type, domain
    , attributes, doc_string)
        self.auto_pad = self.attributes.get('auto_pad')
        self.kernel_shape = self.attributes.get('kernel_shape')
        self.pads = self.attributes.get('pads')
        self.strides = self.attributes.get('strides')
        # Input data tensor from the previous operator; dimensions for
    image case are (N x C x H x W), where N is the batch size, C is the
    number of channels, and H and W are the height and the width of the
    data. For non image case, the dimensions are in the form of (N x C x D1
     x D2 ... Dn), where N is the batch size. Optionally, if dimension
    denotation is in effect, the operation expects the input data tensor to
     arrive with the dimension denotation of [DATA_BATCH, DATA_CHANNEL,
    DATA_FEATURE, DATA_FEATURE ...].
        self.i_X = self.input[0]
        # Output data tensor from average or max pooling across the input
    tensor. Dimensions will vary based on various kernel, stride, and pad
    sizes. Floor value of the dimension is used
        self.o_Y = self.output[0]

    def accept(self, visitor, network):
        super(MaxPool, self).accept(visitor, network)
        visitor.visit_maxpool(self, network)

    @classmethod
    def create_op(cls, i_X: str, o_Y: str, auto_pad: OnnxAttribute,
    kernel_shape: OnnxAttribute, pads: OnnxAttribute, strides:
    OnnxAttribute):
        attributes = {
            'auto_pad': auto_pad,
            'kernel_shape': kernel_shape,
            'pads': pads,
            'strides': strides,
        }
        return cls([i_X], [o_Y], None, None, None, attributes, None)
```

LISTING 5.2: deep500.onnx_parser.generated_operators.py

What generated_operators.py already indicates is worth noting again. The operators are directly created from the ONNX schema definitions and the attribute names are the same as in the documentation. Only the names for the specific input and output fields are indicated by the prefix i_ for input resp. o_ for output.

5.2 Backends

If the reader is only concerned with the implementation of a backend one can check out our jupyter tutorial in the deep500/tutorial folder.

The heart of a neural network framework is the implementation of the specific basis functions. For example in the call in PyTorch `F.relu(x)`, where `x` is transformed with a PyTorch specific implementation of relu. But it could also be the case that when a convolution operation is called in either PyTorch or Tensorflow the same cuDNN [14] implementation is used.

Deep500 provides a way of switching out specific implementations. A use case may be that first the network is described in PyTorch because it is simple to debug and later on in a productive environment should be executed with Tensorflow.

Behind the scenes this is done by 2 classes. The first one is the `Network` class. Its purpose is to abstract away a common access to inference and backpropagation. The second one is the `OnnxBaseVisitor` that decouples the access from the actual data of the graph.

Network class

To get a first idea of the network class and see the UML diagram in figure 5.1.

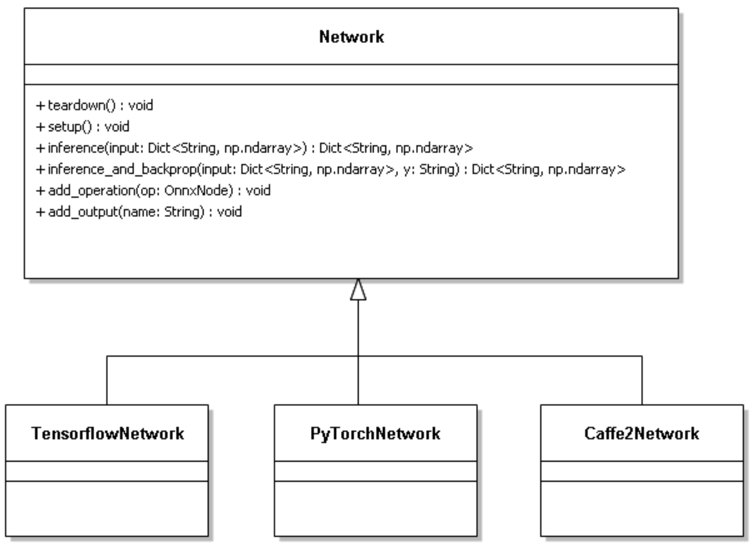

Network
+ teardown() : void
+ setup() : void
+ inference(input: Dict<String, np.ndarray>) : Dict<String, np.ndarray>
+ inference_and_backprop(input: Dict<String, np.ndarray>, y: String) : Dict<String, np.ndarray>
+ add_operation(op: OnnxNode) : void
+ add_output(name: String) : void
TensorflowNetwork
PyTorchNetwork
Caffe2Network

FIGURE 5.1: UML Network

Visitor

The network class alone is not of much use. To do meaningful work the network needs a graph over which it can run inference or backpropagation. In this regard the visitor comes into play. The visitors will transform the ONNX file into the backend specific graph. In the eager executed context the visitor will execute the corresponding functions directly.
In the code the Network superclass provides a parse method which expects a visitor. This visitor then runs over all the nodes and executes the corresponding visit methods of the visitor. If a maxpool node is given in the nodes of the graph the corresponding *visit_maxpool(op: MaxPool, network: Network)* is called in the visitor and this is the moment where the corresponding backend maxpool operation is added or executed.

The listing 5.3 shows a simple implementation of the add method in a possible backend. This is an eager executed backend. First the variables A and B are loaded from the current state added and the result saved back into the network state.

TABLE 5.1: Network class API

Network class API description implement this methods to get own backend	
API	Documentation
setup(self)	Setup the last point where initialization can be made before inference or backpropagation. Normally sessions, devices, input and output dictionaries get setup.
teardown(self)	Any open files and devices are closed here.
inference(self, input: Dict[str, np.ndarray]) −> Dict[str, np.ndarray]:	Inference runs forward on the network for a given input. Normally this function consists of filling the session with the current input and then running the built network with *session.run(output, input)* or something equivalent. In eager execution the execution is directly started at this point.
gradient(self) −> List[Tuple[str, str]]	This returns a list of tuples with the field 1 as name of the parameter and field 2 as the name of the corresponding gradient. Normally this is translated to a call like "'tf.gradients"' from Tensorflow.
inference_and_backprop(self, input: Dict[str, np.ndarray]) −> Dict[str, np.ndarray]:	First this method runs inference. Then if no graph optimizer is directly optimizing over graph operations this method fills the gradients that are available over the above gradient method.

```python
class MinimalisticVisitor(OnnxBaseVisitor):

    def visit_add(self, op: Add, network: MinimalisticBackend):
        A = network.variables[op.i_A]
        B = network.variables[op.i_B]

        C = A + B
        network.variables[op.o_C] = C
```
LISTING 5.3: deep500.backend.minimalistic.mini_backend.py

The listing 5.4 shows a simple implementation of inference in a network. First update the internal variable state with the new input then run the visitor which in eager execution directly executes the corresponding operations. Lastly return the wanted output.

```python
class MinimalisticBackend(NetworkBackend):

  def inference(self, input: Dict[str, np.ndarray]) -> Dict[str, np.ndarray]:
      self.setup()
      self.variables.update(input)
      self._parse(visitor=MinimalisticVisitor())

      output_dict = {}
      for name in self.net_output:
          output_dict[name] = self.variables[name]
            return output_dict
```

LISTING 5.4: deep500.backend.minimalistic.mini_backend.py

If all the methods in the visitor are implemented from the `OnnxBaseVisitor` super class one would have a fully functioning deep learning framework since all the optimization, reporting and distributed training is provided by deep500.
The interplay between the different components are displayed in figure 5.2.

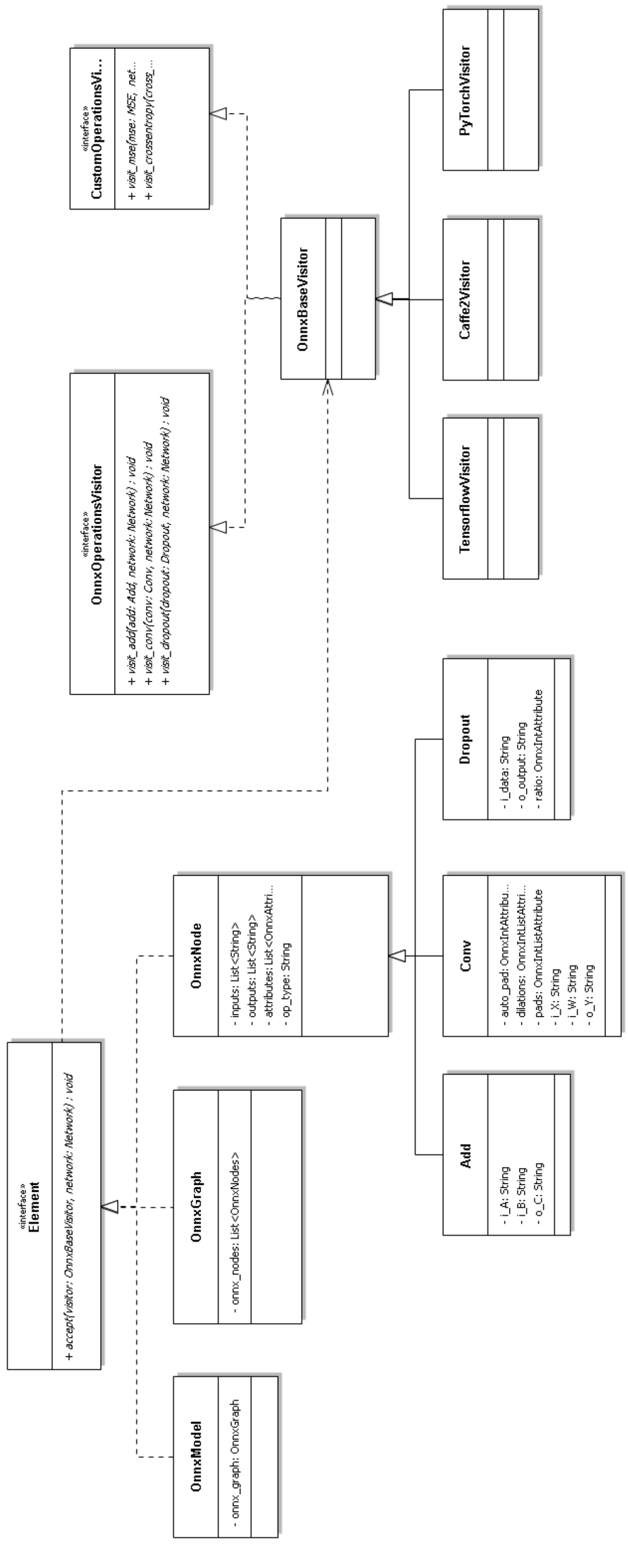

«interface»
CustomOperationsVi...
+ visit_mse(mse: MSE, net...
+ visit_crossentropy(cross_...
«interface»
OnnxOperationsVisitor
+ visit_add(add: Add, network: Network) : void
+ visit_conv(conv: Conv, network: Network) : void
+ visit_dropout(dropout: Dropout, network: Network) : void
OnnxBaseVisitor
PyTorchVisitor
Caffe2Visitor
TensorflowVisitor
«interface»
Element
+ accept(visitor: OnnxBaseVisitor, network: Network) : void
OnnxNode
- inputs: List<String>
- outputs: List<String>
- attributes: List<OnnxAttri...
- op_type: String
OnnxGraph
- onnx_nodes: List<OnnxNodes>
OnnxModel
- onnx_graph: OnnxGraph
Dropout
- i_data: String
- o_output: String
- ratio: OnnxIntAttribute
Conv
- auto_pad: OnnxIntAttribu...
- dilations: OnnxIntListAttri...
- pads: OnnxIntListAttribute
- i_X: String
- i_W: String
- o_Y: String
Add
- i_A: String
- i_B: String
- o_C: String

FIGURE 5.2: UML Visitor

5.2.1 Optimization

Reference Optimization

To guarantee the correctness of an optimizer it is advantageous going from a simple and more slow implementation to a more sophisticated and fast one. The simple and slow ones can be implemented and therefore debugged whereas the fast ones are implemented directly with basis operations in the graph and are (depending on the backend) harder to debug and to get right. Contrary to the design section here the optimizers are presented from a bottom up approach. First the one step optimizer then the three step optimizer and at last the base class optimizer which gives full control over every aspect of the optimizer.

1 step optimizer

Many of the current established gradient optimizers update the weights according to a specific rule after forward & backpropagation has been run. Such an optimizer has the interface in 5.5

```
class UpdateRuleOptimizer(ThreeStepOptimizer,
                          metaclass=abc.ABCMeta):
  def update_rule(self, grad, old_param, param_name):
    pass
```

LISTING 5.5: deep500.backend.optimizer.py

And gradient descent with its simple update rule: $x_{t+1} := x_t - \gamma \nabla f(x_t), \gamma > 0$ is then implemented as simple in the update rule optimizer seen in 5.6.

```
class GradientDescentReferenceImplementation(UpdateRuleOptimizer):

    def __init__(self, network: NetworkBackend, lr=0.1):
        super(GradientDescentReferenceImplementation, self)
                                             .__init__(network)
        self.lr = lr

    def update_rule(self, grad, old_param, param_name):
        return old_param - self.lr * grad
```

LISTING 5.6: deep500.backend.optimizer.py

3 step optimizer

Some of the more elaborate optimizers work in a 3 step fashion.

1. New input callback which is used to updating some internal tracking information e.g. update steps.
2. Prepare parameters for the network before running it.
3. Update parameters after running forward & backpropagation on the network.

The presented superclass in listing 5.7 executes the callbacks exactly as described in the enumeration.

```
class ThreeStepOptimizer(Optimizer, metaclass=abc.ABCMeta):

  def new_input(self):
      pass

  def prepare_param(self, param_name):
      pass


  def update_rule(self, grad, old_param, param_name):
      pass
```

LISTING 5.7: deep500.backend.optimizer.py

Optimizer

If this granularity is not enough there is always the possibility to implement the optimizer superclass which both the one step optimizer and the three step optimizer implement directly. This gives the opportunity to implement a fully customized optimizer by hand.
The listing 5.8 shows the base optimizer class. It expects only to implement the train method which should optimize the parameters according to some input: $(x_i, y_i) \in (\mathcal{X} \times \mathcal{Y})$.

```
class Optimizer(metaclass=abc.ABCMeta):
    def __init__(self, network: Network, loss: str = 'loss'):
        self.network = network
        self.loss = loss

    def train(self, inputs):
        pass
```

LISTING 5.8: deep500.backend.optimizer.py

Usage

Of course the usage should be as simple as possible shown in listing 5.9.

```
optimizer = GradientDescentReferenceImplementation(network)

for inp in inputs:
  loss = optimizer.train(inp)
```

LISTING 5.9: deep500.examples.train_mnist.py

Training a network with native optimizer

Instead of reading values out of the network and applying a simple optimizer, there are faster ways of doing optimization. One option is to implement the used optimization algorithms directly with basis functions.
This graph optimizer implements the base optimizer interface and extends it with the build method which gives the possibility to add methods directly to the graph. In listing 5.10 the graph optimizer is displayed and in listing 5.11 a native Caffe2 momentum optimizer is shown.

```python
class GraphOptimizer(Optimizer, metaclass=abc.ABCMeta):

    def __init__(self, network):
        super(GraphOptimizer, self).__init__(network)
        self.optimizer_built = False

    @abc.abstractmethod
    def build(self):
        pass
```

LISTING 5.10: deep500.backend.via_graph_optimizers.py

```python
class Caffe2MomentumOptimizer(Momentum):

    def build(self):
        with core.DeviceScope(self.network.device_option):
          # optimizer here is just the caffe internal library
          # that provides the
          # optimization operations added to the graph
            optimizer.build_sgd(self.network.train_model,
                                base_learning_rate=self.lr,
                                momentum=self.momentum
                                )
```

LISTING 5.11: deep500.backend.caffe2_graph_optimizers.py

5.2.2 UML Design

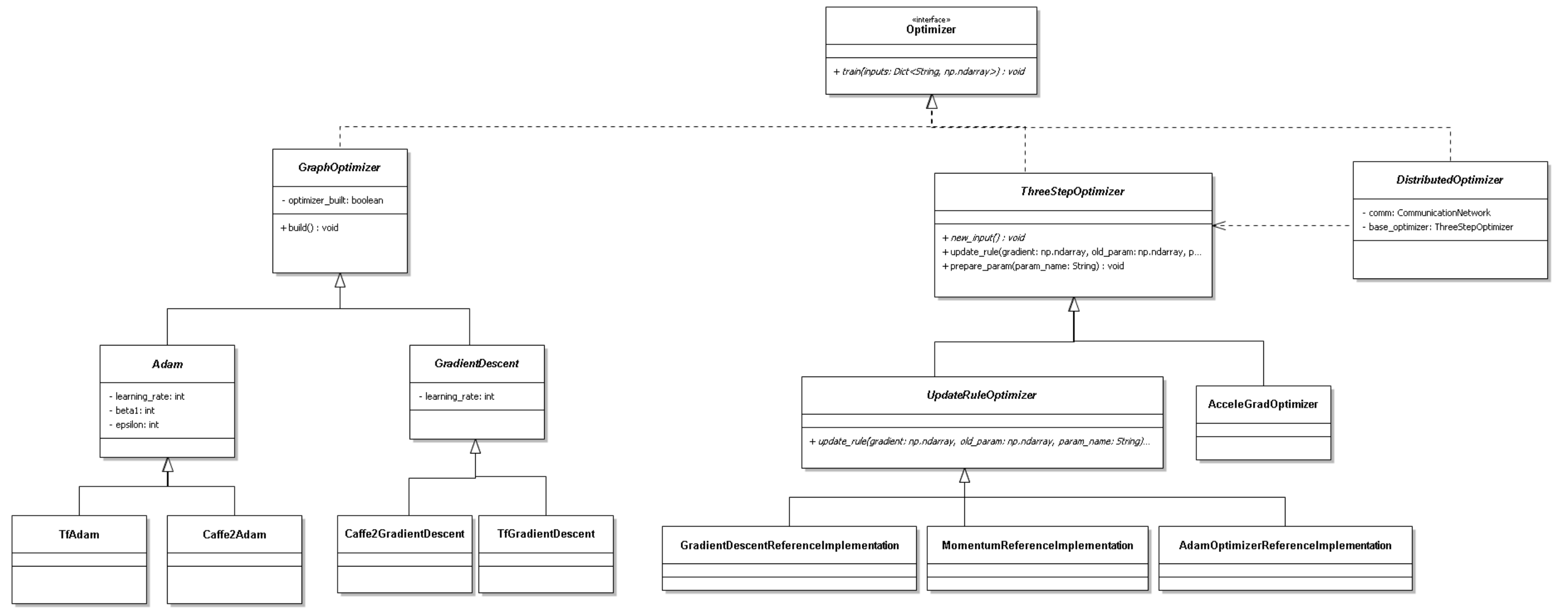

«interface»
Optimizer
+ train(inputs: Dict<String, np.ndarray>) : void
GraphOptimizer
- optimizer_built: boolean
+ build() : void
ThreeStepOptimizer
+ new_input() : void
+ update_rule(gradient: np.ndarray, old_param: np.ndarray, p...
+ prepare_param(param_name: String) : void
DistributedOptimizer
- comm: CommunicationNetwork
- base_optimizer: ThreeStepOptimizer
Adam
- learning_rate: int
- beta1: int
- epsilon: int
GradientDescent
- learning_rate: int
UpdateRuleOptimizer
+ update_rule(gradient: np.ndarray, old_param: np.ndarray, param_name: String)...
AcceleGradOptimizer
TfAdam
Caffe2Adam
Caffe2GradientDescent
TfGradientDescent
GradientDescentReferenceImplementation
MomentumReferenceImplementation
AdamOptimizerReferenceImplementation

FIGURE 5.3: UML Optimization

5.3 Metrics

There is need for some kind of feedback and reporting from the training and testing loop. This can get messy when everything is packed into this loop construct. The next sections explain how the Deep500 metrics abstraction works in detail and how to use it.

5.3.1 Dataset

Before diving into the metrics directly it is reasonable to start with the dataset abstraction. A `Dataset` expects an object of type `Input`. An `Input` is just a class built from a node name and its corresponding data. `Data` demands only that *__len__* and *__next__* is overridden. Commonly lists or numpy [9] arrays are used as data.

```python
class Input:
    def __init__(self, node_name: str, data: Data):
```

LISTING 5.12: deep500.utils.dataset.py

5.3.2 Runner

Having a dataset you can use the runner class with which the whole power of the metrics framework comes for free. First create the runner by one of the provided factory methods. The base constructor of the runner class is shown in listing 5.13.

```python
class Runner:
    def __init__(self,
                 train_set: ADataset,
                 test_set: ADataset,
                 network: Network,
                 optimizer: Optimizer,
                 network_output: str,
                 tickers: List[BaseTicker],
                 summary_ticker: RunnerTicker):
```

LISTING 5.13: deep500.metrics.runner.py

5.3.3 Tickers

In the design section it is claimed that there is an event system with which it should be possible to catch all the relevant events that are of interest for any use case. The following events are catched in this order:

1. Before run over epochs: Initialize indices, files, etc.
2. Before run over epoch: Increment epoch counter.
3. Before run over training set: Shuffle training data.
4. Before optimizer step: Start stop watch.
5. After optimizer step: record optimization time, save loss value.
6. Abort condition after optimizer, if it returns true this event runner will abort: Is loss nan abort gracefully with records of last gradients so is is easier debugging is possible.

7. after run over training set: reset training set. Write plots specific to this training epoch.
8. Before run over test set: shuffle test data, reset counters.
9. Before inference step: start time watch.
10. After inference step: record inference time, record MSE with true value or accuracy in classification case.
11. After run over test set: reset test set. Write plots specific to this run over test set.
12. After run over epoch: Report average inference time / optimization time for this epoch.
13. Abort condition after test, stops training/testing gracefully if condition is true: After some
14. After run over epochs: Write CSV file with all the possible data that has been gathered.

All these methods are then reflected into the *BaseTicker* class. Some natural use case that occurs almost always in practice is the one of calculating the accuracy of a model while training. In some non deep500 framework one would do the following as seen in listing 5.14. This is okay as long as only accuracy as data and printing to stdout is needed. But often there are a lot of interesting statistics to be measured: avg. loss, avg. time inference / optimizing, current samples (progress bar), etc. as well as multiple out sources, e.g. print to stdout, write into a CSV file.

```
corrects = 0
for input, y in dataset:
  y_pred, loss = network.inference(input)
  corrects += np.sum(np.argmax(y_pred, axis=0) == y)
print('accuracy: {}'.format(100/len(dataset)*(corrects)))
```

LISTING 5.14: non deep500 accuracy example

With deep500 it is a lot simpler to handle many use cases. This is achieved by separating data generation and usage into separate classes that can be enabled if needed. For a deep500 specific example see listing **??**.

5.3.4 SummaryTicker

SummaryTicker is the base class used for data generation. Its event should change the *JobSummary* object that gets passed from event to event. Deep500 has two summary tickers. One for the default case in which common data is recorded and one which does time recordings which could possibly slow down a network. The default summary ticker that measures among other data, the accuracy and saves it on the current job object, is shown in listing 5.15.

```python
class SummaryTicker(RunnerTicker):

    def _after_inference_step(self, job_summary: JobSummary, runner, inp,
   output):
        y_corr = inp[runner.test_set.label_node]
        y_network = output[runner.network_output]

        job_summary.current_summary.wrong += np.sum(y_corr != np.argmax(
   y_network, axis=1))
```

LISTING 5.15: deep500.metrics.summary_ticker.py

Reporting to the console is then done in a data processing ticker. The accuracy reporting case can be seen in listing 5.16.

```python
class AccuracyTicker(BaseTicker):
    def after_run_over_epoch(self, job_summary: JobSummary, epoch):
        print('epoch: {} | accuracy: {}%'
              .format(epoch, job_summary.current_summary.accuracy))
```

LISTING 5.16: non deep500 accuracy example

5.3.5 Metrics UML

Before looking at some use cases a part of the UML Design is show in figure 5.4 to get some idea of how everything is connected. It is concentrated on the relationship between the classes. Some less important fields and connections were intentionally left out.

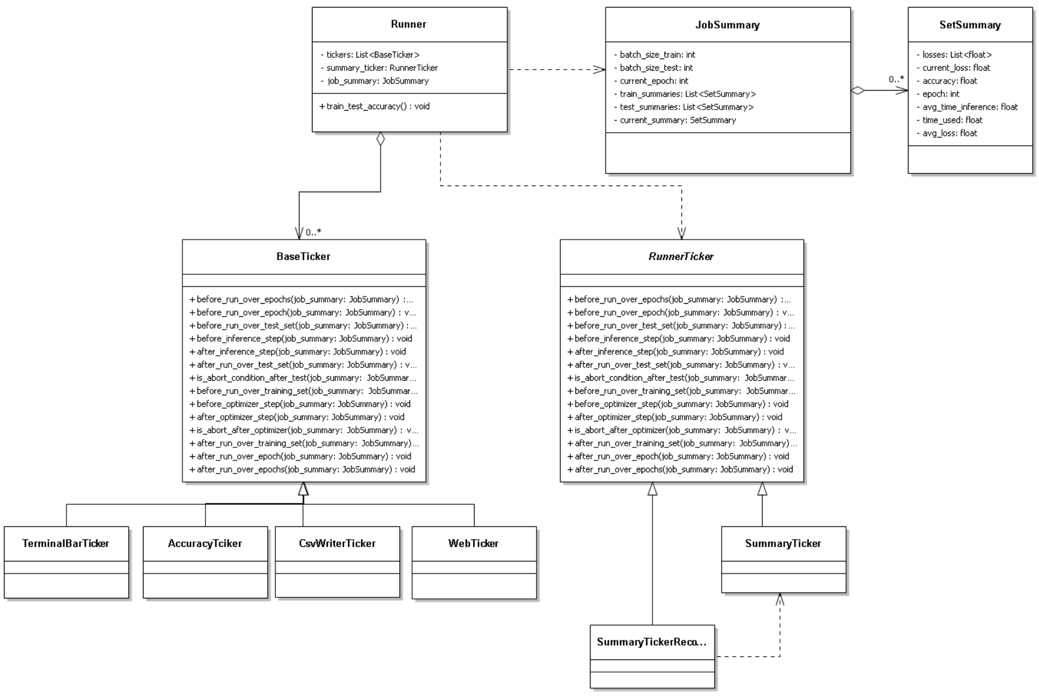

Runner
- tickers: List<BaseTicker>
- summary_ticker: RunnerTicker
- job_summary: JobSummary
+ train_test_accuracy() : void
JobSummary
- batch_size_train: int
- batch_size_test: int
- current_epoch: int
- train_summaries: List<SetSummary>
- test_summaries: List<SetSummary>
- current_summary: SetSummary
SetSummary
- losses: List<float>
- current_loss: float
- accuracy: float
- epoch: int
- avg_time_inference: float
- time_used: float
- avg_loss: float
0..*
0..*
BaseTicker
+ before_run_over_epochs(job_summary: JobSummary) :...
+ before_run_over_epoch(job_summary: JobSummary) : v...
+ before_run_over_test_set(job_summary: JobSummary) :...
+ before_inference_step(job_summary: JobSummary) : void
+ after_inference_step(job_summary: JobSummary) : void
+ after_run_over_test_set(job_summary: JobSummary) : v...
+ is_abort_condition_after_test(job_summary: JobSummar...
+ before_run_over_training_set(job_summary: JobSummar...
+ before_optimizer_step(job_summary: JobSummary) : void
+ after_optimizer_step(job_summary: JobSummary) : void
+ is_abort_after_optimizer(job_summary: JobSummary) : v...
+ after_run_over_training_set(job_summary: JobSummary)...
+ after_run_over_epoch(job_summary: JobSummary) : void
+ after_run_over_epochs(job_summary: JobSummary) : void
RunnerTicker
+ before_run_over_epochs(job_summary: JobSummary) :...
+ before_run_over_epoch(job_summary: JobSummary) : v...
+ before_run_over_test_set(job_summary: JobSummary) :...
+ before_inference_step(job_summary: JobSummary) : void
+ after_inference_step(job_summary: JobSummary) : void
+ after_run_over_test_set(job_summary: JobSummary) : v...
+ is_abort_condition_after_test(job_summary: JobSummar...
+ before_run_over_training_set(job_summary: JobSummar...
+ before_optimizer_step(job_summary: JobSummary) : void
+ after_optimizer_step(job_summary: JobSummary) : void
+ is_abort_after_optimizer(job_summary: JobSummary) : v...
+ after_run_over_training_set(job_summary: JobSummary)...
+ after_run_over_epoch(job_summary: JobSummary) : void
+ after_run_over_epochs(job_summary: JobSummary) : void
TerminalBarTicker
AccuracyTciker
CsvWriterTicker
WebTicker
SummaryTicker
SummaryTickerReco...

FIGURE 5.4: Metrics UML

5.3.6 Use Cases

In this section the flexibility of the ticker system is shown by displaying two practical use cases.

Record inference and optimization times

Recording times can be slow especially if it is done for each inference and optimization step and when all of it is added to one big list. This will only be necessary in special cases. Such a time tracking ticker is shown in listing 5.17.

```
class SummaryTickerRecordInferenceTime(RunnerTicker):
  def __init__(self):
     super(SummaryTickerRecordInferenceTime, self)
          .__init__(parent_ticker=SummaryTicker())


    def _before_inference_step(self, job_summary: JobSummary):
        self.time_before_inference = time.time()

    def _after_inference_step(self,
                              job_summary: JobSummary,
                              runner,
                              inp,
                              output):
        used_time = time.time() - self.time_before_inference
        job_summary.current_summary.time_used_inference.append(used_time)
```

LISTING 5.17: deep500.metrics.summary_ticker.py

A website ticker

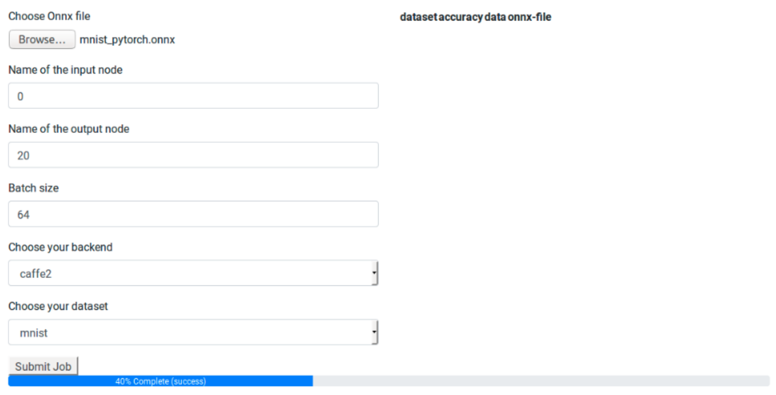
This use case shows a website that offers the possibility to upload an ONNX file choose a specific dataset and a button that starts training the network on the server. The ticker system is then used to get information back from the server to the client that started the training. The data generating tickers need no change. A new ticker is created for the processing side where data gets transformed from python objects into JSON strings and sent back to the client. Therefore the website needs absolutely no domain knowledge to process the information. In listing 5.18 the after optimizer step event is shown as an example on how the data gets transformed into JSON and sent back to the client.

```
class WebRunnerTicker(BaseTicker):

    def after_optimizer_step(self,job_summary: JobSummary, runner, loss):
        self.current_msg['type'] = 1
        self.current_msg['current_bar'] += 1
        self.current_msg['loss'] = float(loss)
      self.queue.put(json.dumps(self.current_msg))
```

LISTING 5.18: deep500.web.web_runner_ticker.py

And in figure 5.5 a picture from the web site can be seen. At the top of the image is a form where an ONNX model can be uploaded to be run on the server and at the bottom a progress bar that gets updated according to data from the server.


Choose Onnx file

Browse... mnist_pytorch.onnx

Name of the input node: 0

Name of the output node: 20

Batch size: 64

Choose your backend: caffe2

Choose your dataset: mnist

Submit Job

40% Complete (success)

dataset accuracy data onnx-file

FIGURE 5.5: Image of web interface

5.4 Non Runner Metrics

Not everything interesting and not every metric can be generated by the runner. The metrics of the framework can roughly be splitted into 3 parts:

1. Layer 0: The specific operations: E.g. texttMin, `Max`, `Sum`, `Relu`.
2. Layer 1: The whole network specifically inference and training of a network.
3. Layer 2: Distributed training and inference for a network over multiple nodes.

The runner can test really thorough layer 1 and layer 2. Especially testing the whole inference & backpropagation process. But isolating the specific operations is not possible. Also the case of comparing two optimizers step by step is difficult. Deep500's way of solving these problems is discussed in the next two subsections.

Layer 0 Testing specific operations

ONNX offers the option of testing specific operations in a simple way. It provides python data or protobuf data as input and then python data as given output.
To test specific operations one can use one of four metrics:

1. Pass / Fail test
The ONNX testing framework is encapsulated into the class: *deep500.onnx_parser.TestBackend* but this does concern an API user only via a call to this test backend.

1. Initialize the test backend with the backend of choice.
2. Include the tests to run.
3. Update the global available tests
4. Run main

This then looks as simple as in listing 5.19 and is commonly copied from another backend. And only the name of the backend class is replaced.

```
backend_test = onnx.backend.test.BackendTest(
                        TestBackend(TensorflowBackend),
                        __name__)
backend_test.include('_gemm_')

globals().update(backend_test.enable_report().test_cases)

if __name__ == '__main__':
    unittest.main()
```

LISTING 5.19: deep500.tests.onnx.test_onnx_full_backend_tf.py

The generated output is shown in listing 5.20.

```
ssssss....ssssss
----------------------------------------------------------------------
Ran 862 tests in 0.830s

OK (skipped=846)
```

LISTING 5.20: pass/fail test output

where **s** stands for skipped and **.** stands for correct and **E** for error.

2. Accuracy
It is possible that accuracy is sacrificed for performance which means that a pass / fail test is not enough. A simple call to *deep500.metrics.layer_level_metrics .generate_-accuracy_metrics.run_metrics()* generates a CSV file with min and max differences in a number, vector, matrix or tensor. A sample call that tests the operations in the Caffe2 network is shown in listing 5.21.

```
if __name__ == '__main__':
  run_metrics(Caffe2Network, categories_excluded=[],
            specific_test_excluded=[
                          'vgg19',
                          'zfnet512'],
            prefix_name="caffe2_backend_big")
```

LISTING 5.21: deep500.metrics.layer_level_metrics.generate_accuracy_metrics.py

The upper part of the generated CSV file is show in table 5.2.

Upper part of test accuracy generated csv file		
Test name	min_diff	max_diff
test_abs_cpu	0.0	0.0
test_abs_cuda	0.0	0.0
test_add_bcast_cpu	-3.9671757221221924	1.8805294036865234
test_add_bcast_cuda	-3.9671757221221924	1.8805294036865234
test_add_cpu	-3.9671757221221924	3.5368399620056152
test_add_cuda	-3.9671757221221924	3.5368399620056152

TABLE 5.2: Output test accuracy

3. Time
Isolated time for an operation is an important measurement. This is implemented

in the same manner as in the accuracy case. But instead of calling `generate_accuracy_metrics`, `deep500.metrics.layer_level_metrics .generate_time_metrics` is called. Also to get a more accurate measurement, multiple runs can be made and are averaged. The generated CSV file is shown in table 5.3.

Upper part of test time generated CSV file				
Test name	run_1	run_2	...	avg
test_abs_cpu	0.0006086826324462891	0.00036430358886718875	...	0.00040569305419921873
test_abs_cuda	0.12066340446472168	0.0005109310150146484	...	0.024487924575805665
test_add_bcast_cpu	0.0023500919342041016	0.00046539306640625	...	0.000833892822265625

TABLE 5.3: Time output operator test

4. Gradient
Checking if the backpropagation algorithm calculates the gradients correctly is possible by calling `deep500.metrics.layer_level_metrics.gradient_checking.py`. Where a call possibly can look like in listing 5.22 and the top five rows of the output is seen in listing 5.23.

```
if __name__ == '__main__':
    # filter function returns true if test should be executed
    def test_file(file):
        return 'abs' in file or 'pow' in file

    run_metrics(PyTorchNetwork, test_file=test_file , categories_excluded
    =[], prefix_name="pytorch")
```

LISTING 5.22: deep500.metrics.layer_level_metrics.generate_checking.py

```
testname : test_pow_bcast_array | param gradient diff : no params
testname : test_pow_bcast_array | input gradient diff :
    0.0004947619240716238
testname : test_pow_bcast_scalar | input gradient diff :
    0.00025031224312801727
testname : test_pow_bcast_scalar | param gradient diff : no params
testname : test_pow | input gradient diff : nan
testname : test_pow | param gradient diff: no params
```

LISTING 5.23: Gradient checking output

Layer 1: Comparing 2 optimizers step by step

When building a new optimizer it could be interesting to compare it with another one step by step. There is a method call that provides exactly that. Given two optimizers and a dataset they are compared step by step and the norm of the difference of the parameters is written to a CSV file. While the call and an example can be seen in *deep500.metrics.optimizer.bench_optimizer.py* the CSV output for a simple linear model with two parameters is shown in table 5.4.

The output of 5.4 is generated by comparing gradient descent and adam on a linear regression model. Interestingly the difference between the first parameter which is the matrix parameter and the second one, the vector parameter, seems to increase much slower.

Csv file generated after 5 steps, the numbers 1 and 2 represent parameters in the network	
1	2
0.5360693	0.02644828
0.92403895	0.047031283
1.2599208	0.0665448
1.305804	0.06864658
1.3557968	0.07171327

TABLE 5.4: Csv file generated after 5 steps

5.5 Distributed Training

A feature we're proud of is the simple to use distributed training of neural networks in a multitude of different ways.

5.5.1 Distributed Reference Implementation

First again the slow but simple implementation that builds upon Deep500's optimizer reference implementation.
The common idea is the following: The same neural network is running on multiple isolated processes (nodes) with each having a unique id. At certain points in time, e.g. before or after running the local optimizer, the nodes communicate with each other to either get in agreement over the current gradient or parameter. The whole communication network is built with MPI [15]. This framework allows for simple communication of data with other groups of nodes or a specific one, at any point. Commonly one transmits weights or gradients, but is not limited to that.
This simple concept gives rise to a multitude of different algorithms. Deep500 implements several of them as seen in table 5.5. These are all based upon the underlying class textitCommunicationNetwork, which can be used to implement own distributed algorithms. A more in depth analysis of the implemented distributed optimizers can be found in [2].

If one is interested in implementing its own distributed optimizer it is certainly worth building upon the *CommunicationNetwork* class, which abstracts away MPI. The API of this class is shown in 5.6.

Equipped with the knowledge of the API it is simple to build a custom distributed optimizer. All that has to be done, is to implement the train method which takes some input object $input : (x_i, y_i) \in (\mathcal{X} \times \mathcal{Y})$. Commonly this method executes inference and backpropagation locally in order to then update the resulting gradient with all other calculating nodes, before applying it to the local non distributed optimizer. An example of this is shown in listing 5.24.

Distributed optimizers implemented in deep500	
Classname	**Functioning**
`ConsistentParameterServer`	In a consistent parameter server all gradients are gathered at the parameter node and when every gradient is gathered it calculates the mean of it runs an optimization step and distributes the new parameters back. Seen in figure 5.6.
`InconsistentParameterServer`	In the inconsistent case as soon as a gradient is calculated on one of the nodes it gets sent to the parameter server where according to the optimizer the new param gets calculated and sent back. The server does not wait for any other gradients. It immediately calculates new gradients. Seen in figure 5.7.
`ConsistentDecentralized`	In the consistent decentralized case there is synchronous communication between all active nodes. It uses the common known allreduce operation from distributed computing to calculate the mean over all gradients. The optimizer update rule is calculated for each node by itself. Seen in figure 5.8.
`ConsistentNeighbors`	It works in the same way as consistent decentralized with one important difference: Instead of calling the allreduce operation across all nodes it restricts itself on its immediate neighbors. In this method there is less communication than in the others.

TABLE 5.5: Reference Distributed optimizers in Deep500

```python
class ConsistentDecentralized(DistributedOptimizer):

    def train(self, inputs):
        self.base_optimizer.new_input()
        [self.base_optimizer.prepare_param(param)
                            for param in self.network.get_params()]
        output = self.network.inference_and_backprop(inputs)
        gradients = self.network.gradient()
        for param_name, grad_name in gradients:
            param, grad = self.network
                        .fetch_tensors([param_name, grad_name])
            # Here the network is called and does an all reduce
            grad = self.communication.sync_all(grad)

            param = self.base_optimizer
                        .update_rule(grad, param, param_name)
            self.network.feed_tensor(param_name, param)

        return output
```

LISTING 5.24: deep500.distributed.distributed_optimizer.py

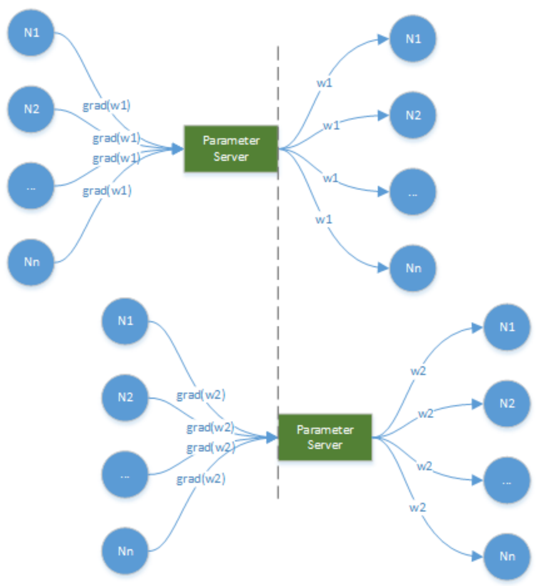


FIGURE 5.6: Consistent Parameter Server

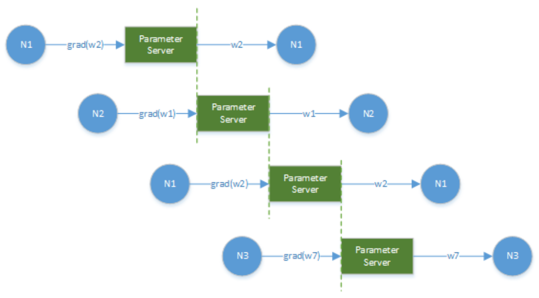


FIGURE 5.7: Inconsistent Parameter Server

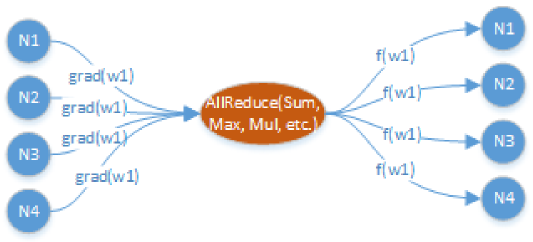


FIGURE 5.8: Inconsistent Decentralized

Usage in code

Assuming MPI and multiple GPUs are available on a computer then Deep500 has an easy way to start n MPI available nodes as seen in listing 5.25.

distributed communication API in Deep500	
Classname	Functioning
gather_at_ps(self, tensor)	Gathers all the arrays at node with rank 0.
sync_all_with_root(self, tensor)	Rank 0 forces its tensor to all the nodes.
send_to_root(self, tensor, tag)	Send from rank != 0 to rank 0.
wait_for_any_slave(self)	Waits for any slave to send some tensor to the rank 0. Returns a tensor and a status (tag, slave id).
send_to_specific_slave(self, tensor, dest, tag)	Sends the specific numpy tensor to the node with id "dest" and some tag. Most of the time the tag is to identify the parameter to update.
sync_all(self, tensor, reduce_func=MPI.SUM)	Allreduce over all nodes according to specific function.
wait_for_root(self)	Wait for a response from node with rank 0.
reduce_from_neighbors(self, tensor, reduce_func=MPI.SUM)	Only make the allreduce over the immediate neighbors.

TABLE 5.6: Distributed communication API Deep500

```python
if __name__ == "__main__":
  n = 3
  if mpi_fork(n) == "child":
      print("child process one of the nodes")
      runner()
  else:
      print("main process that called mpi_fork")
```

LISTING 5.25: deep500.utils.mpi_helper.py

The main process itself is no MPI process and gets control back as soon as all the n MPI child processes are finished.
Another way to start the MPI process would be normally via the `mpiexec -np 3 ...` command.
Then in the child processes sticking the base optimizer into a distributed optimizer distributes the optimization process as seen in listing 5.26.

```python
# from
optimizer = MomentumReferenceImplementation(network)

# to
optimizer = ConsistentParameterServer(
              MomentumReferenceImplementation(network))
```

LISTING 5.26: deep500.examples.distributed.distributed_train_mnist_reference.py

The simplicity that we get from this approach has its cost in its slow performance compared with the native optimizers.

5.5.2 Distributed Native Optimizers

One way to support faster communication is to implement operations directly in some fast native code of the specific frameworks that are used.
These native distributed optimizers still use *GraphOptimizer* as the same superclass as in the non distributed case. This can be seen in listing 5.10.
It is therefore the same as in the non distributed case where the right operations have to be added to the graph and these operations have to have some native implementation backing them up when called from python.
Deep500 implements two native distributed optimizers presented in the next two subsections.

Horovod Distributed Optimizer

Horovod [6] is a distributed optimizer developed by Uber for Tensorflow & PyTorch. With our optimization layer it is simple to add this specific native optimizer to Deep500. The listing 5.27 shows the full code needed to add it to Deep500. After adding its support to Deep500 it can be used as any other distributed optimizer just only for Tensorflow and PyTorch.

```
class HorovodDistributedOptimizer(GraphOptimizer):
    def __init__(self, graph_optimizer: GraphOptimizer,
                 comm=CommunicationNetwork()):
        super(HorovodDistributedOptimizer, self)
                          .__init__(graph_optimizer.network)
        self.communication = comm
        self.parameter_optimizer = graph_optimizer

    def build(self):
        opt: tf.train.Optimizer = self.parameter_optimizer.build()
        hvd.init()
        self.network.session_config.gpu_options.visible_device_list
                                        = str(hvd.local_rank())
        opt = hvd.DistributedOptimizer(opt)
        hooks = [hvd.BroadcastGlobalVariablesHook(0)]
        self.network.add_hooks(hooks)
        return opt
```

LISTING 5.27: deep500.examples.distributed.tf.tf_distributed_optimizer.py

First the local optimizer is called to build and then Horovod adds its distribution operations on top of it.

Caffe2 Distributed Optimizer

For demonstration purposes and because the Caffe2 distributed framework is implemented very complicatedly Deep500 has built its own Caffe2 native distributed optimizer. The operations are implemented in native Caffe2 code and shown in table 5.7. They orient themselves at the `CommunicationNetwork` class from the reference implementation.
The operations are implemented in *cpp_operators/caffe2/mpi_operations/mpi_operations.cc*.

With this operations there are 3 native caffe2 optimizers provided:

1. Caffe2ConsistentParameterServer

2. Caffe2ConsistentDecentralized

Deep500 native Caffe2 distributed operations	
Classname	Functioning
DMpiSend(dest, tag)	Same as send_to_specific_slave in numpy case.
DMpiRecv (source, tag)	If no source and no tag is given it waits for any source and any tag. Same as in wait_for_any_-slave in numpy case.
DMpiBroadcast	Root synchronizes its tensor with all the others. Sync_all_with_root is the equivalent case in Deep500 communications class.
DMpiGather	Gathers all the tensors of the nodes at the root. Same as gather_at_ps for the numpy case.
DMpiAllReduce	Normal allreduce operation known from mpi. Same as allreduce in the numpy case. As reduce function MPI.Sum is used.
DMpiAllReduceMean	Allgather returns a matrix of dimensions n x d. Reduce additionally applies a reduce function. Some possible reduce functions are: Sum, Product, Max, Min, Or, And. Many of the algorithms use only need the reduced mean of this matrix dimension d. This function makes this extra additional step compared to the normal DMpiAllReduce operation and takes the mean after the allreduce operation by dividing through the world size.(sum_allreduce_-vector/num_of_nodes). The numpy equivalent is the sync_all operation from Deep500 communication network.
MpiReduceMean	This operations reduces to a specific node e.g. root node.

TABLE 5.7: Caffe2 distributed operation

3. Caffe2ConsistentNeighbors

These three optimizers also implement the *GraphOptimizer* interface and just add the operations to the graph. Most of the code needs to be written to ensure that data is on the cpu before it gets transmitted. The implementation of Caffe2ConsistentParameterServer is seen in 5.28.

```
class Caffe2ConsistentParameterServer(GraphOptimizer):
    def __init__(self,
                 graph_optimizer: GraphOptimizer,
                 comm=CommunicationNetwork()):
        super(Caffe2ConsistentParameterServer, self)
                            .__init__(graph_optimizer.network)
        self.communication = comm
        self.parameter_optimizer = graph_optimizer

    def build(self):
        self.parameter_optimizer.build()
        self.build_consistent_parameter_server_gradients(
                                self.network,
                                self.communication)

    def build_consistent_parameter_server_gradients(
                            self,
                            network: Caffe2Network,
                            comm_network: CommunicationNetwork):
        _load_custom_dll()

        gradients = network.gradient()

        ptr = comm_network.get_comm_numpy_ptr()
        network.feed_tensor(
                    "mpi_comm",
                    ptr,
                    device_option=core.DeviceOption(caffe2_pb2.CPU))


        # copy gpu data to cpu to make mpi
        if network.is_cuda:
            for (param_name, grad_name) in gradients:
                grad_name_from = grad_name + "_cpu"
                                if network.is_cuda else grad_name
                with core.DeviceScope(network.device_option):
                # we copy on the same device on where mpi_comm is
                    network.train_model.EnsureCPUOutput(
                                    [grad_name], grad_name_from)

        # make mpi
        for (param_name, grad_name) in gradients:
            grad_name_from = grad_name + "_cpu"
                                if network.is_cuda else grad_name
            with core.DeviceScope(core.DeviceOption(caffe2_pb2.CPU)):
                # now we use the copied tensor as input
                network.train_model.
                DMpiReduceMean(
                        [grad_name_from, "mpi_comm"],
                        grad_name + "_buffer")
                network.train_model.DMpiBroadcast(
                        [grad_name + "_buffer", "mpi_comm"],
                        grad_name_from)

        # we have to copy back the mpi'd tensor if we're cuda
        if network.is_cuda:
            for (param_name, grad_name) in gradients:
                with core.DeviceScope(network.device_option):
                # we copy on the same device on where mpi_comm is
                    network.train_model.CopyFromCPUInput(
                                [grad_name + "_cpu"],
                                        grad_name)
```

LISTING 5.28: deep500.distributed.cf2.caffe2_distributed_optimizer.py

Chapter 6

Benchmarking and Experiments

Deep500 provides a lot of tools to generate metrics automatically. Data and plots alone are only the first step to a thorough investigation of an algorithm. To assist with next steps, deep500 offers some tools and guidance. The tools are presented and explained according to tests that we have conducted ourselves.

6.1 Benchmarking

It was discussed in the metrics section how to generate plots and data with already provided metrics or implement new ones. But it gets really interesting when one compares different optimizers, backends and distributed optimization algorithms. Deep500 provides methods that can read the generated CSV files and generates new plots out of it.
The plots can be generated very easily by using the code seen in listing 6.1.

```
is_gradient_descent = lambda f: 'gradientdescent' in f.lower()
generate_plots(folder, is_gradient_descent, 'only_graddesc')
```

LISTING 6.1: deep500.examples.benchmarking.generate_vs_plots.py

The method call in listing 6.1 generates 4 plots. It takes the path to the CSV file folder as input and also a lambda function to filter out the CSV files The last argument is just the title of the plots.
The plots generated are the following:

1. Samples vs Accuracy plot.
2. Time in seconds vs Accuracy plot.
3. Samples vs Loss plot.
4. Time in seconds vs Loss plot.

6.2 Experiments

Experiment setting:

1. Dataset: Cifar10
2. Model: Resnet50
3. Backends: Tensorflow, Caffe2
4. Optimizers: Data generated to every possible optimizer

5. Distributed: Run on 1 GPU non distributed case and 3 GPUs in distributed setting
6. The whole benchmarking code is ready to be run in:
   `deep500.examples.benchmark.non_distributed` resp.
   `deep500.examples.benchmark.distributed`

The experiments are given to showcase the possibilities of deep500 and for interested people to investigate further.

6.2.1 1 GPU

To showcase the possibilities in figure 6.1 one can definitely see how equal both the reference and the corresponding native optimizers are. The really important question emerges that why do both of the reference optimizers not converge equally since they make exactly the same calculations. Since the operations are checked by ONNX and therefore correctly implemented by the backends, the operations can be assumed to be correct. Therefore one could take a look at the gradient calculation as a first step in finding the reason for this difference in convergence.

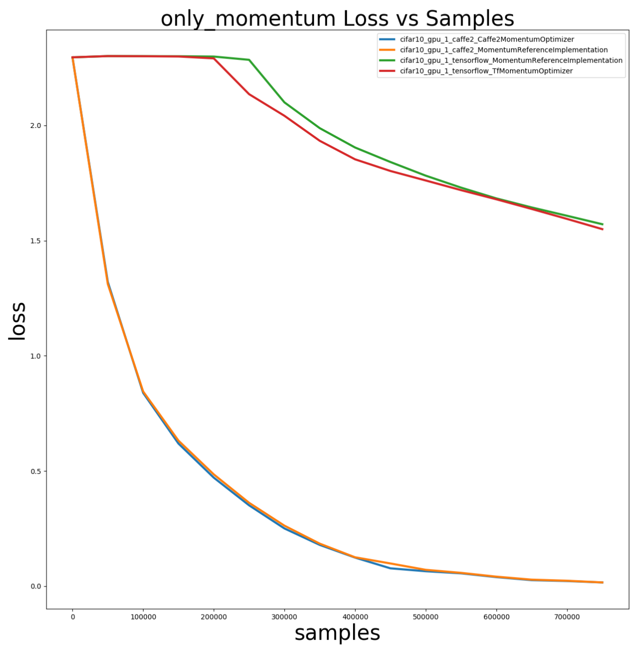


FIGURE 6.1: Comparisons of Momentum optimizers

6.2.2 3 GPUs

With testing on three GPUs there are three interesting things to check:

1. Is the accuracy per sample of our distributed optimizers comparable with Horovod?
2. Is the time used for our native Caffe2 optimizer comparable with Horovod?

3. How do the distributed optimizers compare against the non distributed ones ?

We can read off the answer to the first question out of the figure 6.2 and conclude that Horovod has definitely a slower convergence rate per sample than our reference implementations. That the overall convergence seems to go in the same direction can be seen in figure 6.3.

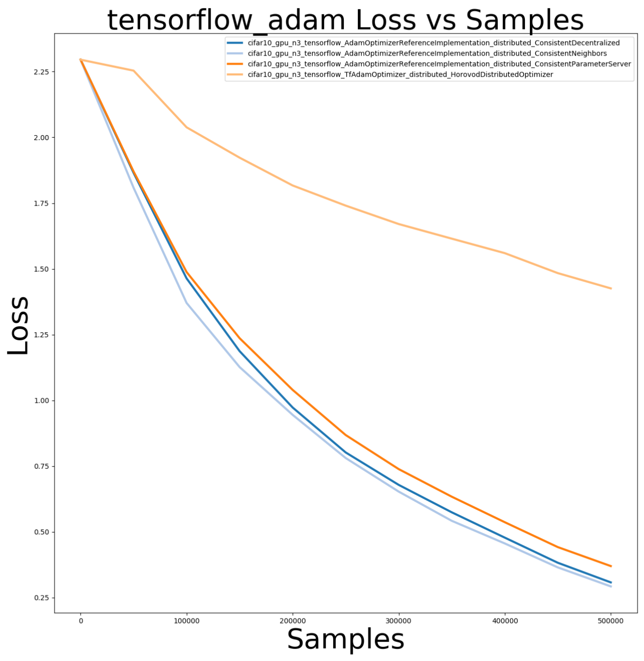


FIGURE 6.2: Only Tensorflow Adam Loss vs Samples

For the second question one can look at the figure 6.4 where the time achieved with our own simple implementation for distributed computing with no optimization at all is comparable with the one achieved by Horovod. The difference in loss could be attributed to the overall slower convergence of Tensorflow compared to Caffe2 but it had to be investigated further obviously.

Considering the last question our intuition tells us the following: In the distributed case the time used should be a third of the time needed in the non distributed case neglecting the communication overhead since each of the nodes has to work through a third of the data only. And for the efficiency per sample one can consider the distributed case as just using a bigger batch size. And since we always correct only a small step into the right direction a lot of smaller update steps are better then one big step and therefore a bigger batch size means that we should converge a little bit slower. Our intuition is supported by the plot in 6.5 where one can see that the accuracy rises slower for the distributed ones despite working through the same dataset with the same algorithms.

In the second figure we can see that the distributed reference implementation (light orange) needs less then half the time that the non distributed reference implementation (light blue) needs to comb through the whole dataset. For the two native optimizers one can also see that the non distributed (dark blue) needs about double the time that our own distributed fast optimization (dark orange) needs. Which is

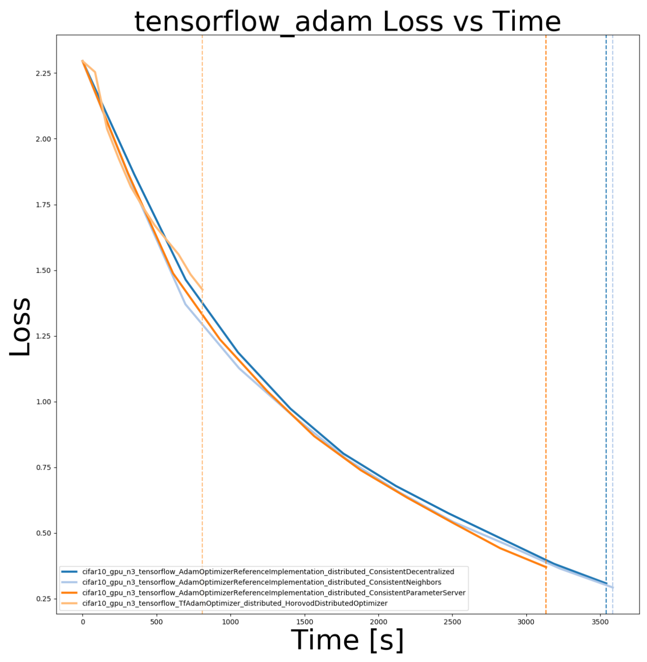


FIGURE 6.3: Only Tensorflow Adam Loss vs Time

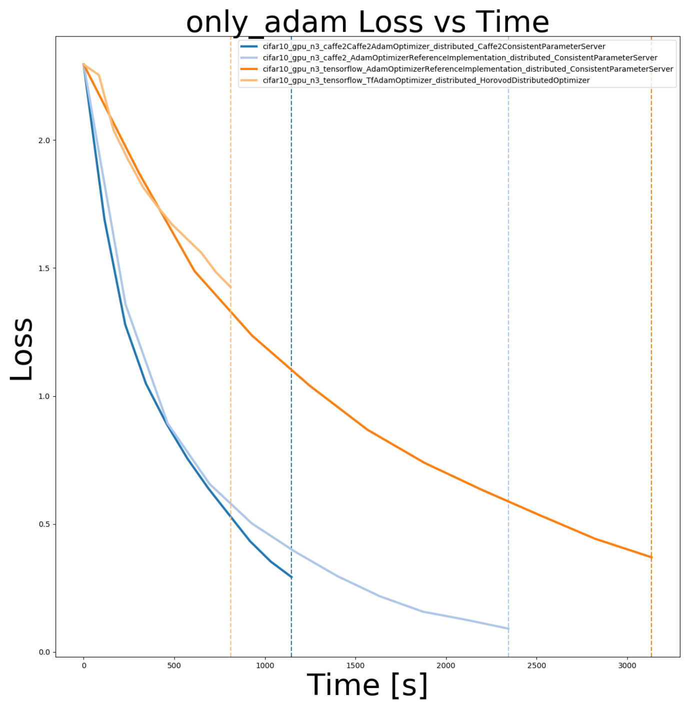


FIGURE 6.4: Only Adam optimizer vs Horovod

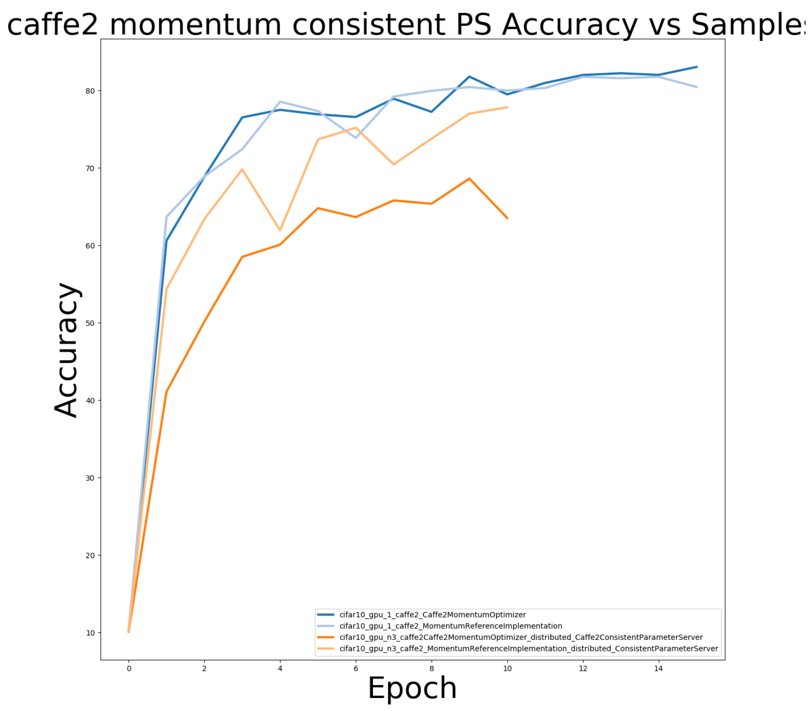


FIGURE 6.5: Caffe2 consistent parameter server accuracy by epoch

not to bad considering all the overhead we get from copying from GPU to CPU and sending over the wire. 6.6

6.2.3 Comparisons between vanilla backend and deep500 implementation

Since deep500 offers so much flexibility one could think that it has to take significant performance hits because of that. But seeing deep500 as any other compiler should switch this view from how slow is deep500 to how slow the backend is. Maybe the backend implementation from one method is just wrongly implemented. Depending on the backend deep500 works more like an interpreter as in the pytorch case or "machine code" / static graph as in the Caffe2/Tensorflow case.
To be sure that the argumentation is correct it has to be tested, too. The experiment was conducted in three steps:

1. The ONNX graph to test is imported into the backend.
2. After the visitor parsed the graph it is exported into the native graph representation of the framework to test.
3. Removed from deep500 the graph is run natively and the execution time is measured.
4. Compare the times with the not exported times from native deep500.

All the code with the exporting of the graphs etc. is kept in the python file and package *deep500.examples.benchmark.export_model_native*.

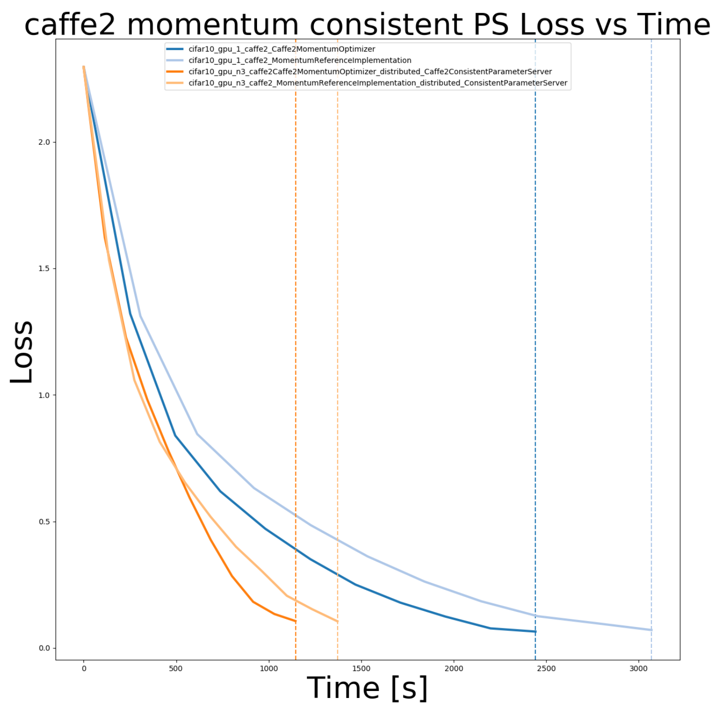


FIGURE 6.6: Caffe2 consistent parameter server loss by time

The data generated seen in the table 6.1 is from comparing adagrad with Horovod run on 3 GPUs and the backend is Tensorflow deep500 vs Tensorflow native both running Horovod optimizer.

native adagrad Horovod	with vs deep500 adagrad	with Horovod on 3 GPUs
epoch	time used training native	time used training deep500
0	247.2994971275	254.2741508484
1	242.9404921532	242.9453964233
2	242.8943395615	242.9795174599
3	242.9469845295	242.9351446629
4	242.9235265255	242.9188194275
5	243.021951437	242.9739191532
6	242.9277544022	242.9587442875
7	242.9519767761	242.9454965591
8	243.0073399544	242.92359519
9	242.9535527229	243.0242242813

TABLE 6.1: Native Tensorflow vs deep500 tensorflow

As one can see there is no significant difference. In the repository that is linked above there are three more tests and the corresponding data committed where multiple combinations of optimizers can be seen and that there is no significant difference in time.

Chapter 7

Conclusion

7.1 Accomplishments

We started by adding a parser that parses ONNX and creates a internal python representation of the model, graph and node structure that is simple to access with the visitor pattern. This internal representation allows to be extended by any operation that ONNX offers. Furthermore there is a mechanism so that custom operations can be added. Deep500 uses this to add the yet from ONNX missing loss functions. Accessing this internal structure is possible via an auto generated visitor structure based on the C++ schemas that ONNX provides. The visitor structure allows simple translation from ONNX to pytorch (20% operator coverage), Tensorflow (50% operator coverage) or Caffe2 (50% operator coverage) specific backend code. (The percentage in the brackets are the support that is already implemented in Deep500)

By abstracting away the backends by a so called network layer we could add a very general optimization framework. To show case the optimization layer we built the following reference implementation of optimizers: gradient descent optimizer, momentum optimizer, adam optimizer, accelegrad and adagrad. The optimization layer still supports native optimizers with the following ones already implemented: For Tensorflow, momentum, adam, and adagrad and for Caffe2, gradient descent, momentum, adam, and adagrad.

Based on MPI we added a communication network. This communication network and the optimization layer we added is used to implement the following distributed optimization algorithms that work with any non distributed reference optimizer as base optimizer: Consistent parameter server, inconsistent parameter server, consistent decentralized and consistent neighbors decentralized.

It is not only possible to use native non distributed optimization but also native distributed optimization. As a use case Deep500 has implemented its own version of native distributed optimization for Caffe2 with the native operations implemented in C++ where consistent parameter server, consistent decentralized, and consistent neighbors decentralized can be used. To show the flexibility of the framework we also added the Horovod distributed optimizer which uses native Tensorflow operations.

For testing purposes Deep500 can test a single operation for correctness, accuracy and time by reusing all the tests that ONNX offers. Reusing all the ONNX tests again it can test gradients by calculating gradients numerically and comparing it versus gradients calculated by backpropagation. There are ONNX tests that run over a whole network with many operations and so more complicated gradient can be tested. By running two networks parallel optimizers can be compared step by step.

Besides specific testing, measuring and reporting is also important. For this purpose Deep500 has a built in metric framework. It works as an event system where by implementing the specific super class one gets all the interesting events (e.g. before epoch, before inference, after inference, before optimizing step, after optimizing step). Based on the metric framework some subset of use cases added are: Progress bar, CSV generator, loss plot generator and a callback ticker to a specific website.

7.2 Future Work

Some ideas that came up while working on this project.

7.2.1 Fix performance problems with reference optimizers

It is definitely a problem that the reference optimizers are really slow. This is mainly due to the extraction of the weights out of the network optimizing externally and put them back in. One of the possible solutions would be to add a tensor abstraction layer which works then directly on the specific native tensors and are only extracted outside of the network if needed in a different format.

7.2.2 Intermediate optimization

On could create a backend that takes ONNX and outputs optimized ONNX again.

7.2.3 Distributed Optimization

Since it is so easy to implement distributed training one could test many different models.

7.2.4 Write optimized code directly

A backend basically calls already finalized operations in the backend. These operations get executed in isolation of the other operations in the graph. This can be a performance hit since every operation has to be started and handled in isolation, too. So one way around this would be to instead of directly executing the corresponding operations, comparable with an interpreted programming language, to write the code for the operation into a single file and then compile this single file into a single operation, comparable with a compiled programming language. This would be simple with Deep500's visitor pattern.

Bibliography

[1] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[2] Tal Ben-Nun and Torsten Hoefler. "Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis". In: *CoRR* abs/1802.09941 (2018). arXiv: 1802.09941. URL: http://arxiv.org/abs/1802.09941.

[3] *Caffe2 page*. Accessed: 2018-09-10. URL: https://caffe2.ai/.

[4] François Chollet et al. *Keras*. https://keras.io. 2015.

[5] *Flatbuffers page*. Accessed: 2018-09-10. URL: https://google.github.io/flatbuffers/.

[6] *Horovod github page*. Accessed: 2018-09-10. URL: https://github.com/uber/horovod.

[7] *Introducing GraphPipe blog post*. Accessed: 2018-09-10. URL: https://blogs.oracle.com/developers/introducing-graphpipe.

[8] *Introducing NNEF blog post*. Accessed: 2018-09-10. URL: https://www.khronos.org/news/press/khronos-releases-nnef-1.0-standard.

[9] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *A guide to NumPy*. Accessed: 2018-09-12. 2006–. URL: http://www.numpy.org/.

[10] *Keras page*. Accessed: 2018-09-10. URL: https://keras.io/.

[11] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980 (2014). arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980.

[12] *Maxpool operation in Onnx catalogue*. Accessed: 2018-08-10. URL: https://github.com/onnx/onnx/blob/master/docs/Operators.md#MaxPool.

[13] *NNEF Khronos page*. Accessed: 2018-09-10. URL: https://www.khronos.org/nnef.

[14] *NVIDIA cuDNN*. Accessed: 2018-09-12. URL: https://developer.nvidia.com/cudnn.

[15] *OpenMPI online documentation*. Accessed: 2018-09-10. URL: https://www.open-mpi.org/doc/.

[16] Adam Paszke et al. "Automatic differentiation in PyTorch". In: (2017).

[17] *PyTorch github repository*. Accessed: 2018-09-10. URL: https://github.com/pytorch/pytorch/.

[18] *Tensorflow-onnx github repository*. Accessed: 2018-09-10. URL: https://github.com/onnx/onnx-tensorflow.