UDC 004.4

Christian Herta[1]**,** Doctor Rerum Naturalium, Professor, Faculty 4 – Computer Science,
E-mail: christian.herta@htw-berlin.de, ORCID: 0000-0003-2519-6794
Klaus Strohmenger[1]**,** Master of Science, Scientific Research Assistant, Faculty 4 – Computer Science,
E-mail: klaus.strohmenger@htw-berlin.de, ORCID: 0000-0002-4534-1306
Oliver Fischer[1]**,** Master of Science, Scientific Research Assistant, Faculty 4 - Computer Science,
E-mail: oliver.fischer@htw-berlin.de, ORCID: 0000-0002-1871-9350
Diyar Oktay[1]**,** Student, Faculty 4 – Computer Science, E-mail: diyar.oktay99@gmail.com,
ORCID: 0000-0003-1483-5837
[1]HTW Berlin – University of Applied Sciences, Wilhelminenhofstr, 75a, Berlin, Germany, 12459

DP: A LIGHTWEIGHT LIBRARY FOR TEACHING DIFFERENTIABLE PROGRAMMING

Abstract. *Deep Learning has recently gained a lot of interest, as nowadays, many practical applications rely on it. Typically, these applications are implemented with the help of special deep learning libraries, which inner implementations are hard to understand. We developed such a library in a lightweight way with a focus on teaching. Our library DP (differentiable programming) has the following properties which fit particular requirements for education: small code base, simple concepts, and stable Application Programming Interface (API). Its core use case is to teach how deep learning libraries work in principle. The library is divided into two layers. The low-level part allows programmatically building a computational graph based on elementary operations. In machine learning, the computational graph is typically the cost function including a machine learning model, e.g. a neural network. Built-in reverse mode automatic differentiation on the computational graph allows the training of machine learning models. This is done by optimization algorithms, such as stochastic gradient descent. These algorithms use the derivatives to minimize the cost by adapting the parameters of the model. In the case of neural networks, the parameters are the neuron weights. The higher-level part of the library eases the implementation of neural networks by providing larger building blocks, such as neuron layers and helper functions, e.g., implementation of the optimization algorithms (optimizers) for training neural networks. Accompanied to the library, we provide exercises to learn the underlying principles of deep learning libraries and fundamentals of neural networks. An additional benefit of the library is that the exercises and corresponding programming assignments based on it do not need to be permanently refactored because of its stable API.*

Keywords: *Differentiable Programming; Deep Learning; Teaching; Automatic Differentiation*

Introduction

Modern deep learning libraries ease the implementation of neural networks for applications and research. In the last few years, different types of such libraries were developed by academic groups and commercial companies. Examples are Theano [1], TensorFlow [2] or PyTorch [3]. Recently, the term “differentiable programming” emerged (see e.g., [5]) which expresses that e.g. (Deep) Neural Networks can be implemented by such libraries by composing building blocks provided by the library. The term differentiable programming also reflects the fact that a much wider spectrum of models is possible by using additional (differentiable) structures (e.g. memory, stacks, queues) [12; 13] as building blocks and control flow statements.

With the DP library, we provide a minimalistic version of such a library for teaching purposes. The library is designed light-weighted, focusing on the principles of differentiable programming: How to build a computational graph and how automatic differentiation can be implemented.

We also developed a high-level neural network API which allows for more convenient implementation of neural network models by providing predefined functional blocks, typically used in neural networks.

The library is accompanied by many Jupyter [25] notebooks, a de facto standard in data science research and education [27], to demonstrate and teach the underlying principles of a deep learning library. We also provide many exercises that allow students to deepen their understanding. The exercises also include concepts of modern neural networks, e.g., activation functions, layer initialization, versions of stochastic gradient descent, dropout, and batch normalization (see e.g. [5]).

Types of deep learning libraries

Different deep learning libraries follow different concepts, and they distinguish further from each other in various aspects. In some libraries, the neural networks must be defined by configuration (e.g. Caffe [4]). Other libraries provide APIs for programming languages, e.g. for Python or R. Some of the APIs resemble languages that are embedded in a host language. Typically, with these domain-specific languages, the computational graphs are defined symbolically. In the next step, the computational graphs (and the corresponding graphs for the derivatives) are translated into code for another programming language, typically C++ or CUDA [12]. Subsequently, the program is compiled and can be executed. Sometimes the term *static* computation graph is used here which reflects the fact that the graph is defined once declaratively and cannot be changed dynamically.

Contrary to this symbolic approach is the imperative approach. Here, the computation graph is built up implicitly by executing the program line by line. The forward computation is done directly, and the computation of the derivatives can be done at the end, e.g., by recursion. With each execution of the program, control structures in the program can change the structure of the computation graph. In this case the term *dynamic* computation graph is used.

Another aspect is the granularity of the computational operations in a deep learning library. With some libraries, the computational graph can be constructed with elementary tensor operations, e.g. matrix multiplication. In other libraries, the operations may correspond to whole layers of a neural network.

Our library DP is a finely granular, imperative deep learning library for Python, based on NumPy [16]. The focus of the library lies in teaching the principles of a deep learning library and the implementation of neural network models and algorithms. Therefore, we designed the library as simple as possible, and we restrict the tensor order to two, i.e. matrices. So, the code base of DP is significantly smaller and easier to understand as of libraries with much more functionality like autograd [8].

Another problem is that most common deep learning libraries are still subject to frequent changes in their API, which is a big drawback when used for exercises. We are developing exercises for advanced deep learning, e.g., Bayesian neural networks [9] or variational autoencoders [10]. For educational reasons (didactic reduction), we provide all boilerplate code so that the students can focus on the learning objective. The boilerplate code includes implementation against a deep learning library. If then a new version of the used library is released and its usage changes, exercises have to be adjusted accordingly to work correctly. Typically, universities do not have the personal resources to keep the teaching materials and exercises permanently up-to-date. The minimalistic approach of our library and the strict focus on teaching allows us to keep its API stable and therefore eliminates the need for permanent maintenance of the exercises.

Overview on the principles

In deep learning libraries, a machine learning model is built up as a computational graph. A computational graph is a directed graph. The structure of the graph encodes the order of the computation steps. At each inner node, an elementary computation is executed. The inner nodes of the graph are elementary mathematical operations (including elementary functions). Examples of elementary operators are +, - or dot-product and elementary functions are e.g., exp, $tanh$ or $ReLU$. A computational graph corresponds to a mathematical expression. The input nodes are the parameters of the model or data values. In machine learning, the output nodes of the graph usually correspond to the prediction values or cost values. Typically, the computational graph is built up in a computer program which allows different programming techniques such as looping, branching, and recursion.

Computational graphs enable automatic differentiation. For each computational node the derivative of the operation must be known. Local derivative computations are combined by the chain rule of calculus to get a numerical value for the derivatives of the whole computational graph for given input values. In deep learning libraries this is typically implemented as reverse-mode automatic differentiation [6].

With *reverse-mode automatic differentiation*, all partial derivatives of the output w.r.t. to all inputs can be calculated efficiently. This feature is very important for machine learning. In the training process of a machine learning model, all partial derivatives of the cost function w.r.t. all parameters of the model must be computed. In neural networks, these parameters are the neuron weights.

The computational graph for the training of a model corresponds to the cost function which should be minimized in the training procedure [19]. The cost $loss(\theta)$ is a function of the parameters θ of the model. During the optimization, the parameters are adapted to minimize the cost value. This optimization is typically realized by variants of stochastic gradient descent (SGD) [11]. In each step of SGD all partial derivatives of the cost w.r.t. the parameters must be computed.

Before the appearance of deep learning libraries, a symbolic expression for the partial derivatives for new models was done by the researcher in a pen-and-paper solution. For an example see e.g. [13]. This manual procedure is error-prone, time consuming and nearly impossible for large complex models.

By building up the model in a deep learning library the build-in feature *reverse automatic differentiation* deliberates the researcher or developer from this work.

Theoretical background of automatic differentiation

In the following we describe the theoretical background of reverse mode automatic differentiation in a semi-formal way. For a more rigorous formal explanation, see e.g. [15].

Notation

In the theoretical description, we use the following mathematical notation. Lower-case Latin letters, e.g. a, denote scalars or vectors. Upper-case Latin letters, e.g. A, denote matrices or more structured objects like graphs. Python variables corresponding to a mathematical object are denoted as lower-case letter in a sans-serif fond, e.g. a, independent of the type.

From the context, it should be clear which objects are referenced by the corresponding letters.

Definition of a computational graph

A computational graph G is a directed acyclic graph. A directed acyclic graph is a set of nodes V (with a node $n^{(i)}$ in V) and a set of edges E, i.e. pairs of nodes $\left(n^{(i)}, n^{(j)}\right) \in E$. i respectively j is the index of the node. Further we assume that the computational graph G is topologically ordered, i.e. for each edge $\left(n^{(i)}, n^{(j)}\right)$ holds $i < j$.

We define the leaves of the graph as the nodes with no incoming edges. Each node $n^{(i)}$ has a corresponding variable $v^{(i)}$. The dimensionality of variable $v^{(i)}$ is $d^{(i)}$. Leaf nodes correspond directly to inputs for the computation and the value of the variable $v^{(i)}$ is directly the input value. Non-leaf nodes $n^{(j)}$ have a corresponding operator $o^{(j)}$. The operator $o^{(j)}$ takes as input the variables $v^{(i)}$ of all nodes with an outgoing edge to the node $n^{(j)}$. For the concatenation of all variables $v^{(i)}$ with an edge to $n^{(j)}$ we write $w^{(j)}$. The concatenation is done in topological order.

For a consistent definition we can define the operator for leaf nodes as the identity which takes as input the (external) input to the (leaf) node.

In summary, a computational graph is a directed acyclic graph where each node has an internal structure. The nodes $n^{(i)}$ consists of a variable $v^{(i)}$ and an operator $o^{(i)}$. The input to the operator is determined by the edge structure of the graph.

Forward propagation algorithm

The forward propagation algorithm computes the values of all non-leaf nodes. The values of the leaf nodes are the input to the algorithm. In topological order all non-leaf nodes $n^{(j)}$ are computed by the corresponding operator $o^{(j)}$ and the variables of the nodes $n^{(i)}$ which have an edge to the node $n^{(j)}$. Note that the variable values of all $n^{(i)}$ are already known. Either because they are leaf nodes or they have a lower order index and are already computed by the algorithm.

Reverse mode automatic differentiation

Reverse-mode automatic differentiation is a two-step procedure. In the first step, the variable values of each inner node of the computational graph are computed by the forward algorithm. The computed values of all variables are stored in an appropriate data structure.

The second step is based on the chain rule of calculus. Here we assume that we have only one node with no outgoing edges. This node has the highest order index m. We call the node the *output node*. In machine learning, the value of the node is typically the cost value and the computational graph computes the cost function. The cost value is a scalar, i.e. the dimensionality of the output variable $v^{(m)}$ is $d^{(m)} = 1$.

In general, the node variables in the computational graph can be tensors of any order. However, for compact indexing we assume that they are flattened to vectors for this theoretical analysis. So, there is only one index for each variable and the variables of the nodes are $d^{(i)}$ dimensional vectors.

We are interested in partial derivatives of the output node variable $v^{(m)}$ with respect to the leaf node variables $v^{(i)}$, i.e. $\frac{\partial v_k^{(m)}}{\partial v_l^{(i)}}$.

On the right side of the equation each summand is a dot-product of Jacobians. j is the index of all nodes which have an edge to node $n^{(m)}$, i.e. $\left(n^{(j)}, n^{(m)}\right) \in E$.

The Jacobian which corresponds to an edge in the computational graph (here from $n^{(j)}$ to $n^{(m)}$) is called a *local* Jacobian (matrix).

For each variable with index the chain rule can be applied again:

$$\left[\frac{\partial v^{(j)}}{\partial v^{(i)}}\right] = \sum_k \left[\frac{\partial v^{(j)}}{\partial v^{(k)}}\right] \cdot \left[\frac{\partial v^{(k)}}{\partial v^{(i)}}\right],$$

k is the index of all nodes which have an edge to node $n^{(j)}$, i.e. $(n^{(k)}, n^{(j)}) \in E$.

Note, that for different nodes j the sum is over different nodes with indices k depending on the graph structure. Repeated application of the chain rule by respecting the graph structure shows that we can compose a global Jacobian from local Jacobians. It can be shown [23] that the dot products of all local Jacobians on all paths from the leaf node $n^{(i)}$ to the output node $n^{(m)}$ must be summed up to get the global Jacobian.

As already stated, we want to compute (nearly) all global Jacobians, i.e. all global Jacobians w.r.t. (nearly) all leaf variables $v^{(i)}$. The principle idea for an efficient computation is to reuse the partial results $\left[\frac{\partial v^{(m)}}{\partial v^{(p)}}\right]$ for all non-leaf variables $v^{(p)}$. Note that $\left[\frac{\partial v^{(m)}}{\partial v^{(p)}}\right]$ is again the sum of the dot products of all local Jacobians on all paths from the node $^{(p)}$ to the node $n^{(m)}$. So, regrouping of the nested sums is equivalent to send backward signals. A backward signal at a current node is the sum of dot products of the local Jacobians of all paths from the current node to the output node. To compute the backward signal of a new node $v^{(q)}$ it is sufficient to sum up all dot products of the *backward signals* $\left[\frac{\partial v^{(m)}}{\partial v^{(p)}}\right]$ of all nearby upstream nodes $v^{(p)}$ with the local Jacobians $\left[\frac{\partial v^{(p)}}{\partial v^{(q)}}\right]$:

$$\left[\frac{\partial v^{(m)}}{\partial v^{(q)}}\right] = \sum_p \left[\frac{\partial v^{(m)}}{\partial v^{(p)}}\right] \cdot \left[\frac{\partial v^{(p)}}{\partial v^{(q)}}\right],$$

p is the index of all nodes with an edge from node $n^{(q)}$ to $n^{(p)}$, i.e. $(n^{(q)}, n^{(p)}) \in E$.

The algorithm starts at the output node $n^{(m)}$. The initial backward signal is $\left[\frac{\partial v^{(m)}}{\partial v^{(m)}}\right] = I$, i.e. an identity matrix with dimension $d^{(m)} \times d^{(m)}$. Then, the backward signals at the nodes which have an edge to $v^{(m)}$ are computed as described above. This procedure is repeated until all wanted global Jacobians are computed.

In the context of neural networks, reverse mode automatic differentiation is also called *backpropagation*.

Implementation

For the implementation in a computer program we chose as programming language Python, because (scientific) Python is the most common programming language for machine learning. Our library is based mainly on the tensor library NumPy.

Basic (low-level) part

With the basic low-level part of the library the user can build the computational graph (implicitly) imperatively. On such a computational graph the global Jacobians of the output node can be computed efficiently by reverse mode automatic differentiation with the help of the library.

The low-level part consists mainly of the `Node` class. Each instantiation of the `Node` class corresponds to the creation a node for the computational graph. To keep the implementation small and clear, the node variables are restricted to tensors of order 2 and the output node variable $v^{(m)}$ must be a scalar, i.e. $d^{(m)} = 1$. In machine learning, the value of the output node is typically the cost value. So, that is not a severe restriction.

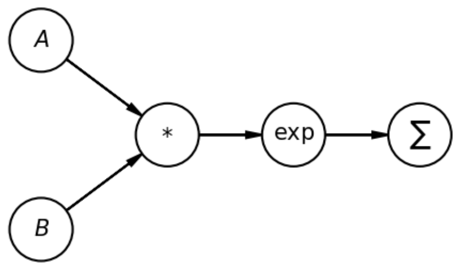

Fig. 1. Example of a computational graph. The leaf nodes are A and B. The output node is the rightmost node (sum over all elements). We denote in topological order, the non-leaf variables C (element-wise product), D (exponentiation) and E (sum of all elements)

In the following, we show how the computational graph of figure1can be build up in the DP-library. Leaf nodes can be instantiated directly by calling the constructor of the `Node` class, e.g. by

```
a = Node(np.array([[1,1,1], [2,2,2]]), "A")
b = Node(np.array([[1,2,3]]), "B").
```

Here, two leaf nodes `a` (with name `A`) and `b` (with name `B`) are generated. Both nodes have got an explicit name given by the optional second argument of the constructor. For all nodes with names the Jacobians (also called gradients) are computed by reverse mode automatic differentiation, see below.

The first node is `a`, i.e. $v^{(1)} = A$ and the second $v^{(2)} = B$. The node variable A is 2x3 matrix. However, note that the node variables described in the theoretical part are formulated as vectors and that the Jacobian indices refer to such vector indices.

As an example, for the correspondence to the matrix A note that the element A_{21} is equivalent to $v_4^{(1)}$, and the total number of elements of the variable $v^{(1)}$ is $d^{(1)} = 6$. For the flattened / vector version of A we write a.

Non-leaf nodes are generated by methods (or overwritten python operators) of the `Node` class. The methods correspond to the mathematical operator, e.g., the element-wise multiplication in figure 1 can be done with the API by

```
c = a * b.
```

Here, a `Node` instance of a non-leaf node is generated by the binary operator "element-wise multiplication" and the instance is assigned to the Python variable `c` (mathematical notation: C).

Note, that the shape of A (2x3 matrix) and B(1x3 matrix) respectively b (vector of dimension 3) are different. The DP-library supports broadcasting [20] for such element-wise operations. As result of broadcasted element-wise multiplication, `c` has the same shape as `a`.

The completion of the computational graph of Fig. 1 is done by the following code,

```
d = c.exp()
e = d.sum() # output e is a scalar.
```

For the variable `d` each element of `c` is exponentiated. For the variable `e` all elements of the variable `d` are summed up to a scalar. `e` is the output variable of the computational graph.

By reverse mode automatic differentiation, the Jacobians of the node `e` w.r.t. node `a` and `b` can be computed. This is done by the method `grad(.)` with argument `1` on the output node,

```
grads = e.grad
```
(1).

The return value is a Python dictionary with an entry for each leaf-variable with a name, here

```
{'A': array([[2.7, 14.78, 60.26],
             [7.39, 109.20, 1210.29]]),
 'B': array([[ 17.50, 116.59, 826.94]])}.
```

Exemplarily, we describe the implementation of the element-wise multiplication operation. The internal implementation is given by the following code:

```
def __mul__(self, other):
  if isinstance(other, numbers.Number) or
       isinstance(other, np.ndarray):
    other = Node(other)
  ret = Node(self.value * other.value)

  def grad(g):
    g_total_self = g * other.value
    g_total_other = g * self.value
    x = Node._set_grad(self, g_total_self,
           other, g_total_other)
    return x

  ret.grad = grad
  return ret.
```

The method generates and returns a new node `ret` for the element wise multiplication operator. The node instance `ret` has no name. The inner function definition `grad` implements how the backpropagated signal `g` is combined with the local Jacobians for both operands, i.e. in our computational graph `a` and `b`. How this implementation is related to the theory (see above) is not obvious. In the implementation, there is no (explicit) dot-product of Jacobians. In the following this relation is explained for the variable a. We assume in the analysis, that the variable b was internally broadcasted, so that a and b resp. $v^{(1)}$ and $v^{(2)}$ have the same dimension $d^{(1)} = d^{(2)} = 6$:
Here, the output node is `e`, i.e. and the backpropagated signal is at the node c $\left[\frac{\partial v^{(m)}}{\partial v^{(p)}}\right] = \left[\frac{\partial \mathrm{e}}{\partial \mathrm{c}}\right]$ (given to the inner function `grad` as argument `g`. To get the global Jacobian w.r.t. the node a the dot product with the local gradient $\left[\frac{\partial \mathrm{c}}{\partial \mathrm{a}}\right]$ must be calculated and combined with the backpropagated signal:

$$\left[\frac{\partial \mathrm{e}}{\partial \mathrm{a}}\right] = \left[\frac{\partial \mathrm{e}}{\partial \mathrm{c}}\right] \cdot \left[\frac{\partial \mathrm{c}}{\partial \mathrm{a}}\right],$$

or explicitly (with Jacobian) indices:

$$\left[\frac{\partial \mathrm{e}}{\partial \mathrm{a}}\right]_{1j} = \sum_k \left[\frac{\partial \mathrm{e}}{\partial \mathrm{c}}\right]_{1k} \left[\frac{\partial \mathrm{c}}{\partial \mathrm{a}}\right]_{kj}.$$

Note, that the first index of $\left[\frac{\partial \mathrm{e}}{\partial \mathrm{a}}\right]_{1j}$ resp. $\left[\frac{\partial \mathrm{e}}{\partial \mathrm{c}}\right]_{1k}$ is always a 1 because of the scalar output of the computational graph. The local Jacobian for the element-wise multiplication is

$$\left[\frac{\partial \mathrm{c}}{\partial \mathrm{a}}\right]_{kj} = \delta_{kj}\, b_j,$$

δ_{kj} is the Kronecker-Delta, i.e. $\delta_{kj} = 0$ for $k \neq j$ and $\delta_{kj} = 1$ for $\mathrm{k} = j$. So, we have

$$\left[\frac{\partial \mathrm{e}}{\partial \mathrm{a}}\right]_{1j} = \sum_k \left[\frac{\partial \mathrm{e}}{\partial \mathrm{c}}\right]_{1k} \delta_{kj}\, b_j = \left[\frac{\partial \mathrm{e}}{\partial \mathrm{c}}\right]_{1j} b_j.$$

Therefore, the combination of the Jacobians by the dot-product is here equivalent to an element-wise multiplication of the Jacobians. The dimension of the Jacobians (indexed by 1) need not to be considered in the shape of the Jacobian variables in the implementation.

Neural network library (high-level) part

Additionally, to the low-level part, the library includes different building blocks and helper functions which ease the implementation of neural networks.
For teaching purposes, we restrict the provided building blocks to simple fully connected layers (see Fig. 2). With these layers fully connected feed-forward networks can be implemented.
A hidden or output layer consists of an affine transformation given by a weight matrix $W^{(l)}$ and a bias vector $b^{(l)}$ and a (non-linear) activation function *act*( ). Typical activation functions for hidden layers are, e.g. element-wise $ReLU$ or $tanh$. For classification tasks, the activation function of the output (last) layer is typically the logistic (two classes only) or the softmax function.

A layer can be described mathematically by

$$h^{(l+1)} = \text{act}(W^{(l)} \cdot h^{(l)} + b^{(l)}).$$

Here, the superscript is the layer index. The input to the network is therefore $h^{(1)} = x$.
For training of a neural network, a set of training examples must be provided,

$$D_{\{Train\}} = \{ (x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(n)}, y^{(n)}) \}.$$

Each pair $(x^{(i)}, y^{(i)})$ is a training example with an input $x^{(i)}$ and a label (target value) $y^{(i)}$. The superscript is the index of the example. n is the total number of training examples.

On the training data set, the learning corresponds to minimizing a cost function. Here, we neglect for simplification generalization [7] which is very important in practice. The cost (and the prediction) is computed typically on (mini) batches. The inputs of many examples are concatenated in a design matrix X, i.e. each row of the matrix corresponds to an input vector $x^{(i)}$. Each layer of the neural network outputs a matrix H with a hidden representation h for each example as row vectors of the matrix.

$$H^{(l+1)} = \text{act}(W^{(l)} \cdot H^{(l)} + b^{(l)}).$$

The neural network layer building blocks are internally composed from `Node` class objects. In Fig. 2 such a building-block, internally structured by `Node objects`, is shown.

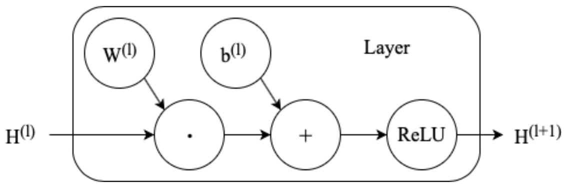

Fig. 2. One neural network layer represented as computational graph with activation function, here $ReLU$. Note, that such a layer is only a part of the full computational graph

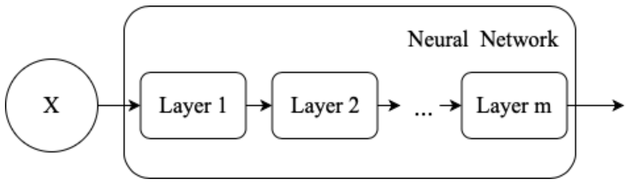

Fig. 3. A complete neural network composed of multiple layers. Each layer is internally composed of `Nodes` objects as shown in Fig. 2

A complete feed forward network is composed of stacked layers, see Fig. 3.

For training, the computational graph of the neural network is augmented with a cost function and an additional node for the provided labels Y of the mini batch. An example of a building block for the cross-entropy cost is show in Fig. 4.
In the next few sections we show how each layer is implemented with our library.

The input layer consists only of input data, also called features, and is represented as a leaf node x in the computational graph. In Python, the input data are typically given as NumPy arrays, so we just need to convert this input array into a node object to enable backpropagation. With the DP-Library the conversation is done via

```
input = Node(X) # X is a NumPy 2d-array.
```

Note, that the optional name argument is omitted as the Jacobian w.r.t. `X` is not needed for the optimization. After converting the data into a `Node` object, we can use all operators and functions implemented in the Node class, including automatic differentiation.

For the hidden layers, our library contains a class called `NeuralNode`, which initializes a weight matrix $W^{(l)}$ and a bias vector $b^{(l)}$. Both are leaf-nodes (see Fig. 2) with unique names given to the `Node` constructor. Since the most common used activation function is $ReLU$ we implemented also a $ReLU$ layer besides a pure linear layer. The pure linear layer can be used together with any activation functions specified by the user with the `Node` class, e.g. $leakyReLU$, $tanh$, $logistic$ etc.

Stacking many of these layers results in a fully connected neural network, see Fig. 3. We call the output of the last layer O. O is automatically produced by sending the input X forward through the network (forward propagation).

The training of the neural network is done by minimization of the cost. The cost is a function of the parameters θ of the neural network. The parameters θ are the weight matrices and bias vectors:

$$\theta = \{W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}, \dots, \; W^{(m)}, b^{(m)}\},$$

m is the number of layers in the network.

The cost function is implemented as part of the computational graph. Therefore, is consists of structured `Node` objects, see Fig. 4.

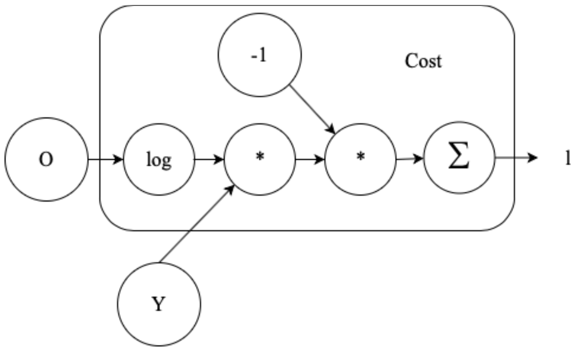

Fig. 4. Calculation of the loss value l using a cost function, here cross entropy represented as computational graph. The labels Y must be provided in one-hot encoding. O is the output of the neural network (last `Node` object of the last layer)

The final output from the cost (sub-)graph will be a scalar l. So, the gradient of the cost (loss) with respect to all model parameters θ can be calculated by the DP-library. This gradient is then used to train the network via an update rule, to tune the network parameters to lower the loss l. The full calculation pipeline of l is shown in Fig. 5.

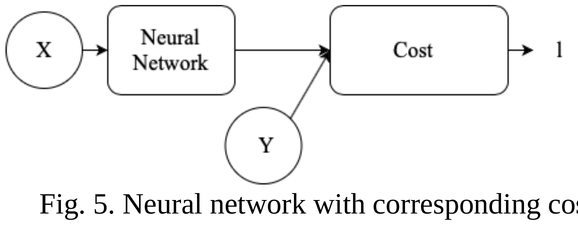

Fig. 5. Neural network with corresponding cost function. The X input is mapped to the output via the neural network (see Fig. 3). The output of the neural network and the labels Y are mapped to the cost value via cost block

To ease the implementation of a neural network, we provide a `Model` class. The user has to derive from the `Model` class a concrete model. The layers must be defined as instance variables. Additionally, the user has to define a loss method and a forward pass method.

The following code shows an example of a neural network for MNIST classification:

```
class Network(Model):
  def __init__(self):
    super(Net, self).__init__()
    self.h1 = self.ReLu_Layer(784,500,"h1")
    self.h2 = self.ReLu_Layer(500,200,"h2")
    self.h3 = self
          .Linear_Layer(200,10,"h3")

  def loss(self, x, y):
    if not type(y) == Node:
      y = Node(y)
    out = self.forward(x)
    loss = -1 * (y * out.log())
    return loss.sum()

  def forward(self, x):
    if not type(x) == Node:
      x = Node(x)
    out= self.h3(self.h2(self.h1(x)))
          .softmax()
    return out
```

In the constructor code two $ReLU$ layers and a linear layer are defined as instance variables. The linear layer is later complemented with a softmax activation function, since this network deals with multiclass classification (10 disjunct classes).

The constructor signature of a layer instantiation is:

```
def ReLu_Layer(number_of_inputs,
      number_of_outputs, name_of_layer").
```

The forward pass to generate the output O is defined in `def forward(self, x)` simply by stacking all defined layers plus an additional `softmax()` as explained above.

The loss function which outputs l is defined in `def loss(self, x, y)` where `self.forward(x)` is used to calculate the network output O. Y represents our target values, here fixed class labels (one hot encoded) for classification. Notice, that each time we start a calculation it is checked whether the input is a `Node` object or not, and if not, the data is converted into one.

After that, the user-defined network can be instantiated by calling the constructor:

```
net = Network().
```

For training, we also provide different optimizers which inherit from the basic (abstract) `Optimizer` class. The optimizer updates the model parameter according to special update rules. The optimizer we provide are SGD, SGD Momentum, RMSProp and Adam [22]. An instance of an optimizer can be initialized, e.g. by

```
optimizer = SGD(net,x_train,y_train).
```

The first parameter, `net`, is the network (see above). `x_train` and `y_train` are the training data, equivalent to X and Y. Training can be started with

```
loss = optimizer.train(steps=1000,
              print_each=100),
```

`steps` is the number of total training loops to adjust the model parameters. `print_each` is the number of steps after which we want to receive a feedback about the current training error, basically the loss value, which should decrease if training succeeds. Per default the `train` function will return the final loss value which we saved into `loss` in our example above. For a more detailed analysis of the training it is also possible to call

```
loss, loss_hist, para_hist = optimizer
      .train(steps=1000, print_each=100,
             err_hist=True).
```

With the parameter `err_hist=True` a complete history of the loss value the model parameters will be returned. These can be used for further analytics, e.g. to visualize the training process.

After the network is trained, it is quite common to test how well the network learned its task by testing its prediction using a set `x_test`. Using the network prediction from the forward pass

```
y_pred = net.forward(x_test),
```

the test accuracy of the network can be calculated. For classification for example this means how many labels the network predicted correctly.

For a deeper understanding on neural networks and optimizers or for special purposes it is possible to implement the training process from scratch. The `Model` class provides the functions `get_grad()`, `get_param()` and `set_param()`. These are also used internally called by the `Optimizer` class. A manually implemented training loop, using basic gradient descent, could look like the following

```
net= Network()
for epoch in range(100):
  # compute the loss and gradients
  grad,loss = net.get_grad(x,y)

  # get the current parameters
  param_current = net.get_param()

  # calc new parameters, actual learning
  param_new = { name : param_current[name]
        - 0.001 * grad[name]
          for name in param_current.keys()}

  # set new parameters
  net.set_param(param_new).
```

Accompanying exercises

To make the entry into the topic of differentiable programming as easy as possible, the DP library is part of a differentiable programming course and can be found, together with accompanying exercises, on the deep-teaching website [187] or directly at the GitLab repository [18]. The exercises are divided into three groups.

The first group of exercises teaches the principles of reverse mode automatic differentiation. It is explained how the DP library itself is implemented, i.e. how to implement the operator methods for instantiation of a computational graph, consisting of scalars, matrices, elementary operators (+, -, dot-product) and functions ($tanh$, exp, etc.) and how to implement automatic differentiation. Finally, everything is combined in an object-oriented architecture forming the DP library and therefore enabling easy use of the low level and high-level functionalities mentioned.

The second group of exercises is about using the DP library to build neural networks, train them and using them for inference. At the same time each of these exercises is about best practices and findings of neural network research of the last couple of years, including batch-norm [21], dropout [1422], optimizers (improvements of SGD, e.g. Adam [22]), weight-initialization methods (e.g. Xavier [24]) and activation functions.

The last-mentioned exercise, at which we will have a look at for illustration purposes, teaches about different activation functions and the so-called vanishing gradient problem [26].

We consider a simple deep neural network, i.e. one that consists of many layers, e.g. 10 linear layers. The output of the first linear layer is computed with $H^{(2)} = act^{(1)}\left(W^{(1)}H^{(1)} + \mathrm{b}^{(1)}\right)$, with $H^{(1)} = X$ the input, $W^{(1)}$ the first weight matrix, $\mathrm{b}^{(1)}$ the corresponding bias vector and $act^{(1)}$ the activation function. The output of the second linear layer then is computed with $H^{(3)} = act^{(2)}\left(W^{(2)}H^{(2)} + \mathrm{b}^{(2)}\right)$ and so on, until the last layer $O = act^{(10)}\left(W^{(10)}H^{(10)} + \mathrm{b}^{(10)}\right)$. Training the network, we first calculate the loss l, i.e. the difference of the output of our last layer O (our predictions) and the true labels Y. This is a binary classification tasks, i.e. there are two possible labels (0 and 1). The output O for an example input x is the predicted probability for the positive class, i.e. $p_\theta(y = 1|x)$. For such problems the binary cross-entropy as cost function is typically used:

$$loss(\theta) = -\big(Y\ log\ (O) + (1 - Y)\ log\ (1 - O)\big).$$

Second, we adjust the weight matrices for all layers i by the update rule of gradient descent:

$$W^{(l)^{NEW}} \leftarrow W^{(l)^{OLD}} - \alpha \cdot \frac{\partial loss(\theta)}{\partial W^{(l)^{OLD}}}.$$

Using the chain rule to calculate $\frac{\partial loss(\theta)}{\partial W^{(1)}}$ for example, we get:

$$\frac{\partial loss(\theta)}{\partial W^{(1)}} = \frac{\partial loss(\theta)}{\partial O} \cdot \frac{\partial O}{\partial H^{(10)}} \cdot \frac{\partial H^{(10)}}{\partial H^{(9)}} \cdot \frac{\partial H^{(9)}}{\partial H^{(8)}} \cdot \ldots \cdot \frac{\partial H^{(2)}}{\partial W^{(1)}}.$$

For binary classification, the typical activation function $act^{(10)}$ of the output layer is the logistic function $\sigma(z) = \frac{1}{1+exp^{-z}}$ which has the range]0,1[. However, a problem arises, if the logistic function is further used as activation function $act^{(1)}$ to $act^{(9)}$ in intermediate layers, because the absolute value of its derivative is at most $\frac{1}{4}$, which in turn leads to the partial derivative $\frac{\partial loss(\theta)}{\partial W^{(1)}}$ becoming smaller and smaller the more layers the network has in between, as $\lim_{l\to\infty}\left(\frac{1}{4}\right)^l = 0$.

The derivative of the $tanh$ or the $ReLU$ function on the other hand is defined in the range of $]0, 1]$, resp. 0,1.

The task of this sample exercise consists of (a) building the neural network model for the computational graph using the DP library, (b) train and validate the network with different activation functions while (c) visualizing the vanishing gradient problem by plotting the sum of the absolute values of the partial derivates $\frac{\partial \mathrm{loss}(\theta)}{\partial \mathrm{W}^{(l)}}$ for all weights of each layer $l \in \{1,2,\ldots,10\}$.

The third group of exercises is on using more common, but also more complex deep learning libraries, like PyTorch and TensorFlow. This kind of exercises is not directly related to our DP library, but still should be mentioned here because they are the last step of our educational path for students on differentiable programming, that is: (1) Learn the principles of differentiable programming and how to build a framework for it at the example of our lightweight DP library, (2) learn how to use this library to build models, train them, validate them and use them for inference and (3) make a transition to using well-known but more complex frameworks.

After that, the students should then have a good starting point for understanding the inner implementation and software-architecture of libraries, like PyTorch and TensorFlow.

Conclusion

The use of machine learning, especially of artificial neural networks, in practical applications has increased tremendously over the last years and most likely will keep increasing in the near and far future. Yet already today research and industry suffer from a lack of specialists in this field. Unfortunately, becoming an AI specialist has a very flat learning curve and requires knowledge in the fields of mathematics, computer science, statistics and ideally in the domain, which you want to provide with AI driven applications.

With our library for educational purpose, teaching the fundamentals of differentiable programming can be improved significantly by opening the black box of deep learning libraries.

With less than 1.000 lines of code, including about 400 lines of comments, in contrast to 3.5 million lines for TensorFlow [28], the goal of a lightweight, clear and easy understandable library was achieved. Following the concept of didactic reduction [29], its use and architecture have a lot in common with TensorFlow and PyTorch, but with a focus on the core principles of differentiable programming.

Lastly the stable API does not force teachers to re-adjust their exercises and educational material over and over again to keep them up-to-date.

Acknowledgment

The project Deep.Teaching is funded by the German National Ministry of Education and Research (BMBF), project number 01IS17056.

References

1. Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G. & Bengio, Y. (2010). “Theano: a CPU and GPU math Expression Compiler”. *In Proceedings of the Python for scientific computing conference (SciPy)* Vol. 4, No. 3, pp 3-10.

2. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C. & Ghemawat, S. (2016). “Tensorflow: Large-scale Machine Learning on Heterogeneous Distributed Systems”. arXiv preprint arXiv:1603.04467.

3. Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L. & Lerer, A. (2017). “Automatic Differentiation in PyTorch”. *NIPS 2017 Workshop on Autodiff.*

4. Maclaurin, D., Duvenaud, D. & Adams, R. P. (2015). “Autograd: Effortless Gradients in numpy. *In ICML 2015. AutoML Workshop,* Vol. 238.

5. Baydin, A. G., Pearlmutter, B. A., Radul, A. A. & Siskind, J. M. (2018). “Automatic Differentiation in Machine Learning: a Survey”. *In Journal of Machine Learning Research*, 18(153), pp. 1-43. arXiv preprint arXiv:1502.05767.

6. “Official Caffe Website”. [Electronic resource]. – Access mode https://caffe.berkeleyvision.org/ – Active link: – August 2019.

7. Nickolls, J., Buck, I. & Garland, M. (2008, August). „Scalable Parallel Programming”. *In 2008 IEEE Hot Chips 20 Symposiums (HCS),* pp. 40-53. IEEE.

8. Goodfellow, I., Bengio, Y. & Courville, A. (2016). “Deep Learning”. *MIT press*, pp. 271-273. DOI: 10.1007/s10710-017-9314-z.

9. (2015). Blundell, Charles, et al. “Weight Uncertainty in Neural Networks”. *Proceedings of the 32nd International Conference on International Conference on Machine Learning*, Vol. 37, pp. 1613-1622, arXiv preprint arXiv: 1505.05424.

10. Kingma, D. P. & Welling, M. (2013). “Auto-encoding Variational Bayes”. *In 2nd International Conference on Learning Representations*, {ICLR} 2014. arXiv preprint arXiv:1312.6114.

11. Bottou, L., Curtis, F. E. & Nocedal, J. (2018). “Optimization Methods for large-scale Machine Learning”. *SIAM Review,* Vol. 60(2), pp. 223-311. DOI: 10.1137/16M1080173.

12. Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A. & Badia, A. P. (2016). “Hybrid Computing using a Neural Network with Dynamic External Memory”. *Nature,* 538(7626), pp. 471–476. DOI: 10.1038/nature20101.

13. Grefenstette, E., Hermann, K. M., Suleyman, M. & Blunsom, P. (2015). „Learning to Transducer with Unbounded Memory”. *In Advances in neural information processing systems (NIPS*), pp. 1828-1836. arXiv: 1506.02516.

14. Gers, F. (2001). “Long Short-Term Memory in Recurrent Neural Networks”, PhD Thesis, Lausanne, EPF, Switzerland, pp 17-19.

15. M. Collins. (2018). “Computational Graphs, and Backpropagation”, Lecture Notes, Columbia University, pp 11-23. [Electronic resource]. – Access mode http://www.cs.columbia.edu/~mcollins/ff2.pdf. – Active link: – August 2019.

16. Travis E. Oliphant (2006). “A Guide to NumPy”, *Trelgol Publishing*, USA: pp. 13-17.

17. (2016). Thomas Kluyver et al. “Jupyter Notebooks – a Publishing Format for Reproducible Computational Workflows”, *In Positioning and Power in Academic Publishing: Players*, Agents and Agendas. *IOS Press*. pp. 87-90. DOI:10.3233/978-1-61499-649-1-87.

18. Herta, Christian et al. “deep.TEACHING.org – Website for Educational Material on Machine Learning”. [Electronic resource]. – Access mode https://www.deep-teaching.org/courses/differential-programming. – Active link: – August 2019.

19. Herta, Christian et al. “deep.TEACHING.org – “Repository of “deep.TEACHING.org”. [Electronic resource]. – Access mode: – https://gitlab.com/deep.TEACHING/educational-materials/blob/master/notebooks/differentiable-programming/dp.py – Active link: – August 2019.

20. “Array Broadcasting in Numpy”. [Electronic resource]. – Access mode https://www.numpy.org/devdocs/user/theory.broadcasting.html. – Active link – August 2019.

21. Ioffe, S. & Szegedy, C. (2015). “Batch Normalization: Accelerating deep Network Training by Reducing Internal Covariate Shift. ICML'15” *Proceedings of the 32nd International Conference on International Conference on Machine Learning* – Vol. 37, pp. 448-456. arXiv preprint arXiv:1502.03167.

22. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. & Salakhutdinov, R. (2014). “Dropout: a Simple way to Prevent Neural Networks from Overfitting”. *The Journal of Machine Learning Research,* 15(1), pp. 1929-1958.

23. Kingma, D. P. & Ba, J. (2014). “Adam: A Method for Stochastic Optimization”. *3rd International Conference on Learning Representations*, ICLR 2015. arXiv preprint arXiv:1412.6980.

24. Glorot, X. & Bengio, Y. (2010, March). “Understanding the Difficulty of Training deep feed Forward Neural Networks”. *In Proceedings of the thirteenth international conference on artificial intelligence and statistics,* pp. 249-256. PMLR 9:249-256, 2010.

25. Hochreiter, S. (1991). „Untersuchungen zu Dynamischen Neuronalen Netzen“. Diploma thesis. TU Munich (in German).

26. Jupyter Homepage. [Electronic resource]. – Access mode: https://jupyter.org/ – Active link: – August 2019.

27. Jeffrey M. Perkel. (2018). “Why Jupyter is data Scientists' Computational Notebook of Choice”. *Nature* 563.7729., pp. 145-146. DOI: 10.1038/d41586-018-07196-1.

28. -OpenHub – Projects – TensorFlow [Electronic resource]. – Access mode: https://www.openhub.net/p/tensorflow/analyses/latest/languages_summary – Active link – August 2019.

29. Herta, C., Voigt, B., Baumann, P., Strohmenger, K., Jansen, C., Fischer, O. & Hufnagel, P. (2019). “Deep Teaching: Materials for Teaching Machine and Deep Learning. In HEAD'19”. *5th International Conference on Higher Education Advances*, pp. 1153-1131. DOI: http://dx.doi.org/10.4995/HEAd19.2019.9177.

Received 01.09.2019
Received after revision 20.11.2019
Accepted 23.11.2019

УДК 004.4

[1]**Херста, Крістіан**, доктор природних наук, професор факультету 4 – Комп'ютерні науки
E-mail: christian.herta@htw-berlin.de, ORCID: 0000-0003-2519-6794
[1]**Штоменгер, Клаус**, магістр наук, науковий співробітник факультету 4 – Комп'ютерні науки
E-mail: klaus.strohmenger@htw-berlin.de, ORCID: 0000-0002-4534-1306
[1]**Фішер, Олівер**, магістр наук, науковий співробітник факультету 4 – Комп'ютерні науки
E-mail: oliver.fischer@htw-berlin.de, ORCID: 0000-0002-1871-9350
[1]**Октай, Діварі,** студент факультету 4 – Комп'ютерні науки,
E-mail: diyar.oktay99@gmail.com, ORCID: 0000-0003-1483-5837
[1]HTW Берлін – Університет прикладних наук, Вільгельменхофштр, 75а, Берлін, Німеччина,12459

DP: ПОЛЕГШЕНА БІБЛІОТЕКА ДЛЯ НАВЧАННЯ ДІФФЕРЕНЦІЙНОМУ ПРОГРАМУВАННЮ

Анотація: *Технології глибокого навчання викликають великий інтерес, так як в даний час на ньому базується велика кількість прикладних додатків. Як правило, ці програми реалізуються за допомогою спеціальних бібліотек глибокого навчання, внутрішню реалізацію яких важко зрозуміти. Ми розробили таку бібліотеку в полегшеному вигляді з упором на викладання відповідних дисциплін. Наша бібліотека має наступні характеристики, які відповідають певним вимогам з урахуванням специфіки навчального процесу: невелика кодова база, прості концепції і стабільний інтерфейс прикладного програмування (API). Основне призначення бібліотеки - допомога у володінні принципами роботи з бібліотеками глибокого навчання. Бібліотека розділена на два шари. Низькорівнева частина дозволяє програмно побудувати обчислювальний графік на основі елементарних операцій. У машинному навчанні обчислювальний графік зазвичай є функцією вартості, що включає в себе модель машинного навчання, наприклад, нейронну мережу. Вбудований зворотний режим автоматичного диференціювання на обчислювальному графіку дозволяє навчати моделі машинного навчання. Це робиться за допомогою алгоритмів оптимізації, таких як стохастичний градієнтний спуск. Ці алгоритми використовують похідні, щоб мінімізувати вартість шляхом адаптації параметрів моделі. У разі нейронних мереж параметри є вагами нейронних мереж. Частина бібліотеки вищого рівня полегшує реалізацію нейронних мереж, надаючи більші будівельні блоки, такі як нейронні шари і допоміжні функції, наприклад, реалізацію алгоритмів оптимізації (оптимізаторів) для навчання нейронних мереж. Також до бібліотеки ми додаємо вправи для вивчення основних принципів роботи бібліотеки глибокого навчання і основ нейронних мереж. Додатковою перевагою бібліотеки є те, що вправи і відповідні програмні завдання на її основі не потребують постійного рефакторінгу через її стабільного API.*

Ключові слова: *диференційоване програмування; глибоке навчання; навчання; автоматичне диференціювання*

УДК 004.4

[1]**Херста, Кристиан**, доктор естественных наук, профессор факультета 4 – Компьютерные науки
E-mail: christian.herta@htw-berlin.de, ORCID: 0000-0003-2519-6794
[1]**Штоменгер, Клаус,** магистр наук, научный сотрудник факультета 4 – Компьютерные науки,
E-mail: klaus.strohmenger@htw-berlin.de, ORCID: 0000-0002-4534-1306
[1]**Фишер, Оливер,** магистр наук, научный сотрудник факультета 4 – Компьютерные науки,
E-mail: oliver.fischer@htw-berlin.de, ORCID: 0000-0002-1871-9350

[1]**Октай, Дивар**, студент факультета 4 – Компьютерные науки, E-mail: diyar.oktay99@gmail.com, ORCID: 0000-0003-1483-5837
[1]HTW Берлин – Университет прикладных наук, Вильгельменхофштр, 75а, Берлин, Германия, 12459

DP: ОБЛЕГЧЕННАЯ БИБЛИОТЕКА ДЛЯ ОБУЧЕНИЯ ДИФФЕРЕНЦИРУЕМОМУ ПРОГРАММИРОВАНИЮ

***Аннотация**: Технологии Глубокого обучения вызывают большой интерес, так как в настоящее время на нем базируются многие практические приложения. Как правило, эти приложения реализуются с помощью специальных библиотек глубокого обучения, внутреннюю реализацию которых трудно понять. Мы разработали библиотеку глубокого обучения в облегченном виде с упором на преподавание. Наша библиотека имеет следующие характеристики, которые соответствуют определенным требованиям с учетом специфики учебного процесса: небольшая кодовая база, простые концепции и стабильный интерфейс прикладного программирования (API). Основное назначение этой библиотеки - обучение принципам глубокого обучения. Библиотека разделена на два слоя. Низкоуровневая часть позволяет программно построить вычислительный график на основе элементарных операций. В машинном обучении вычислительный график обычно является функцией стоимости, включающей в себя модель машинного обучения, например, нейронную сеть. Встроенный обратный режим автоматического дифференцирования на вычислительном графике позволяет обучать модели машинного обучения. Это делается с помощью алгоритмов оптимизации, таких как стохастический градиентный спуск. Эти алгоритмы используют производные, чтобы минимизировать стоимость путем адаптации параметров модели. В случае нейронных сетей параметры являются нейронными весами. Часть библиотеки более высокого уровня облегчает реализацию нейронных сетей, предоставляя более крупные строительные блоки, такие как нейронные слои и вспомогательные функции, например, реализацию алгоритмов оптимизации (оптимизаторов) для обучения нейронных сетей. В дополнение к библиотеке мы предоставляем упражнения для изучения основополагающих принципов работы библиотеки глубокого обучения и основ нейронных сетей. Дополнительным преимуществом библиотеки является то, что упражнения и соответствующие программные задания на ее основе не нуждаются в постоянном рефакторинге из-за ее стабильного API.*

Ключевые слова: *дифференцируемое программирование; глубокое обучение; автоматическое дифференцирование*

Christian Herta, Dr. rer. nat**.**, Regular Professor at the HTW Berlin
Research field: applications and theoretical aspects of machine learning and deep learning

Klaus Strohmenger, Scientific Research Assistant at the HTW Berlin
Research field: machine learning, practical applications of neural networks, computer vision

Oliver Fischer, Scientific Research Assistant at the HTW Berlin
Research field: theoretical background of neural networks, building and explicability of neural networks

Diyar Oktay, Student of the HTW Berlin
Research field: image processing, application of AI in urban planning and