Записки научных
семинаров ПОМИ
Том 546, 2025 г.

A. Zabashta, A. Filchenkov

USING ALGEBRAIC DATA TYPES TO BUILD DEEP LEARNING ARCHITECTURES

Abstract. This paper proposes to use algebraic data types (ADT) to describe the types of objects to be processed in deep learning (DL). A method for constructing DL architectures for basic ADT constructors such as the sum and product of types was described. It has been shown that this approach allows to generalize existing approaches to constructing DL architectures for processing sets and sequences, categories and missing values. Conducted practical experiments showed that in some scenarios, the DL architectures obtained for the task of filling missing values surpassed standard methods.

§1. Introduction

Algebraic data types are actively used for data typing in functional programming languages, analytic combinatorics [5] and databases [8]. The algebraic type is constructed from basic constructors: empty type, sum type and product type. Additional constructors are also used: Set, MSet, Seq, Syc. These constructors allow to abstractly describe data invariants.

The empty type, or type ε, is the simplest type. The product type is used to combine types, similar to tuples, records or structures in programming languages. In machine learning, the product type can be used to type multimodal data, for example, Input = Text × Image. The product type can also be used for typification of vector data, for example, Iris = SepalLength × SepalWidth × PetalLength × PetalWidth. The sum type unites the types, similar to unions, enums or interfaces in programming languages. For example, Species = Iris-setosa + Iris-versicolor + Iris-virginica.

Both Set and Seq typify a collection of elements. But for a sequence the order of elements is important, while for a set it is not. For example, Text = Seq(Word), but if Seq will be replaced with Set, a bag of words approach

Key words and phrases: artificial intelligence and machine learning and deep learning and algebraic data type.

This paper was carried out as part of ITMO University project No. 624125 "Development of advanced machine learning methods and algorithms".

will be obtained. MSet and Syc are mostly used in combinatorics and are not very important for deep learning. But another new type will be useful, a vector of numbers or just $\mathbb{R}^n$. Because deep learning architectures work with vectors. And even if more complex data is used, it is first transformed into vectors. The empty type can also be thought of as a zero size vector.

There are different approaches to formalizing deep learning architectures, for example based on graphs [3] or category theory [7]. Graphs can be used to encode collections, in particular sets (see Figure 1). But the graph itself is defined by a set of vertices, and a vertex by a set of edges. It turns out that the problem of processing a set remains, it is just hidden inside the processing of a graph. Also, graphs cannot encode simple types. However, graphs are very useful for handling fully recursive structures.

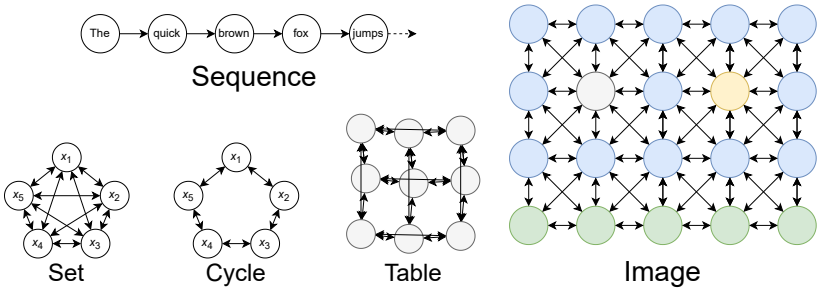

Figure 1. Encoding collections with graphs.

The purpose of this study is to generalize data typing for deep learning using an ADT description. This approach will allow building a large architecture for complex data types from basic functions for data with simple types.

§2. DL Architecture Constructions by ADT

2.1. Key Ideas and Vec Type. The vector type is a universal type. On the one hand, it is a universal intermediate representation, on the other hand, in ADT construction, it will be a terminal type. It is assumed that the ADT is given by an acyclic graph or a rooted tree, in which the vertices are constructors and the leaves are a vector type. If the type is primitive recursive, for example: $\mathrm{Seq}(X) = \varepsilon + X \times \mathrm{Seq}(X)$, then such a type must be deconstructed dynamically for a particular object. If the type is fully

recursive, then it is necessary to use approaches that are used to build DL architectures for graph networks.

To build an architecture for $X \to Y$ transformation, it is necessary to build auxiliary architectures for $X \to \text{Vec}$ and $\text{Vec} \to Y$ transformations. Of course, with some transformations, for example: $\text{Set}(X) \to \text{Set}(Y)$, it can be encounter a bottleneck problem. In this case, it can be used pre-prepared special architectures, but this will be some optimization to the described universal algorithm.

Thus, it is necessary to be able to build architectures only for $X \to \text{Vec}$ and $\text{Vec} \to Y$ transformations. Architectures for $\text{Vec} \to \text{Vec}$ transformation have been widely studied in deep learning. For an Eps type $f : \varepsilon \to \mathbb{R}^n$ is a function that does not contain arguments. Therefore, it can only be a constant trainable vector of size n.

2.2. Prod $\to$ Vec. There are two approaches that can be used to construct the trainable function $f : P \to \mathbb{R}^n$ where $P = \prod_{i=1}^{k} T_i$. In both case, auxiliary trainable functions $g_i : T_i \to \mathbb{R}^{m_i}$ are needed. In the first approach, $f(x_1, x_2, \ldots, x_k) = \text{concat}_i(g_i(x_i))$ and $\sum_{i=1}^{k} m_i = n$. For example, such an approach can be found in RNN [12], where the state vector is concatenated to the input vector.

In the second approach, $f(x_1, x_2, \ldots, x_k) = \sum_i (g_i(x_i))$ and $\forall_i : m_i = n$. For example, such approach can be found in transformers [15], where the positional encoding vector is added to the corresponding input vector.

Let g_i be the auxiliary functions for the first approach and $\sum_{i=1}^{k} m_i = n$. If g'_i are selected as the auxiliary functions for the second approach, then they will be equivalent if

- $g'_1(x) = (g_1(x)_1, g_1(x)_2 \ldots g_1(x)_{m_1}, 0 \ldots\ldots 0)$,
- $g'_i(x) = (0 \ldots 0, g_i(x)_1, g_i(x)_2 \ldots g_i(x)_{m_i}, 0 \ldots 0)$,
- $g'_k(x) = (0 \ldots\ldots 0, g_k(x)_1, g_k(x)_2 \ldots g_k(x)_{m_k})$.

2.3. Vec $\to$ Prod. Let $P = \prod_{i=1}^{k} T_i$, construct $f : \mathbb{R}^n \to P$. Let $g_i : \mathbb{R}^{m_i} \to T_i$ be auxiliary functions. Both $\text{Prod} \to \text{Vec}$ approaches can be inverted. The first approach will be as follows:

(1) Apply another auxiliary function $h : \mathbb{R}^n \to \mathbb{R}^m$, where $m = \sum_i m_i$.
(2) Split the result into k subarrays.
(3) Apply the functions g_i to corresponding subarrays.
(4) Combine the results into a tuple.

The second approach will be as follows:

(1) Directly apply g_i to input. $\forall_i : m_i = n$ required.
(2) Combine the results into a tuple.

2.4. Sum $\to$ Vec. To construct the trainable function $f : S \to \mathbb{R}^n$ where $S = \sum_{i=1}^{k} T_i$ auxiliary trainable functions $g_i : T_i \to \mathbb{R}^n$ are also needed. In this approach, pattern matching must be used to calculate $f(x : T_i) = g_i(x)$.

2.5. Vec $\to$ Sum. Let $S = \sum_{i=1}^{k} T_i$, construct $f : \mathbb{R}^n \to S$. Let $c : \mathbb{R}^n \to \mathbb{R}^k$ and $g_i : \mathbb{R}^n \to T_i$ be auxiliary functions. Then $f(x) = g_{\text{argmax}(c(x))}(x)$. Since argmax makes a strict decision, it will not be differentiable. Therefore, c cannot be trained as usual. But if it is trained, inference will work fine. There are three approaches that can be used for training.

2.5.1. *Supervised Approach.* If S is the output of the function (the correct choice of i is known), then $c(x)$ can be trained directly as for classification.

2.5.2. *Branching approach.*

(1) Let $w = \text{SoftArgMax}(c(x))$.
(2) Then calculate several branches for all i.
(3) Consider the result of i-th branch in loss function with the weight w_i.

This approach can lead to exponential growth of branches.

2.5.3. *RL* [16] *approach.* Let $c(x)$ predict the value of the quality function. It can be trained using reinforcement learning, where i is chosen as in contextual multi-armed bandit problem [18]. Note that training Eps $\to$ Sum architecture would be equivalent to solving the classic multi-armed bandit problem [11].

2.6. Collections. Img and Seq are very popular and well-studied data types. There are many architectures for Vec $\to$ Seq, Seq $\to$ Seq, Vec $\to$ Seq transformations and for Img type too. It seems that Set is not such a popular type, but a very popular transformer block can be used to handle Set, because, in fact, transformers works with sets, not sequences. Also a special DL architecture [17] was developed for it. Table is a less popular type in deep learning, but special architectures [6, 9] were developed in meta-learning. A table can be represented as $\text{Table}(X) = \text{Set}_{\text{row}}(\text{Set}_{\text{col}}(X)) \times \text{Set}_{\text{col}}(\text{Set}_{\text{row}}(X))$. Note that $\text{Table}(X) = \text{Set}_{\text{row}}(\text{Set}_{\text{col}}(X))$ is not enough,

because if values in the rows are swapped not synchronously, the table will be corrupted, although the invariant will be saved.

Note that all of these architectures for collections work with vectors. For example, a transformer is of the Set(Vec) → Set(Vec) type. Therefore, to use it for Set(X) → Set(Y) transformation, two auxiliary architectures of types $X \to \mathrm{Vec}$ and $\mathrm{Vec} \to Y$ are needed.

§3. Analysis of Existing Architectures

3.1. Seq ↔ Vec. A sequence can be defined recursively as $\mathrm{Seq}(T) = \varepsilon + T \times \mathrm{Seq}(T)$. If the previously described approach applied to construct the Seq → Vec transformation, the result will be an architecture similar to the classical recurrent network. But this architecture will process the elements of the sequence from right to left and the initial state vector (memory) will be trainable.

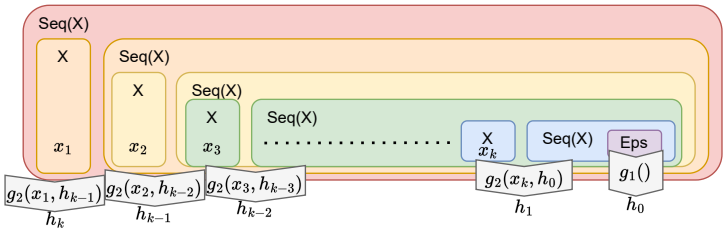

Figure 2. Seq → Vec architecture. h_k is a result of transformation.

If the Vec → Seq transformation will be constructed, the resulting architecture will generate elements correctly from left to right. But instead of using a special stop token, an auxiliary decision function from the sum type will be used.

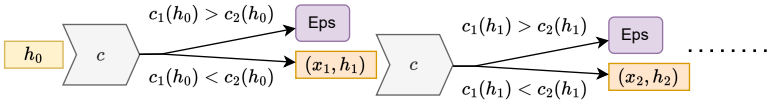

Figure 3. Vec → Seq architecture. Eps encode stop of sequence generation.

3.2. Differentiable Programming. Vec $\to$ Sum architectures is a step towards differentiable programming. As in the previous example Vec $\to$ Seq is based on Vec $\to$ Sum architecture. It turns out that Vec $\to$ Seq(Eps) architecture can be used to obtain a natural number in a differentiable way. Of course, it may be pointless to use such number to set the size of an collection, since it can usually be set directly during its generation.

Also Vec $\to$ Sum architecture can be used to control the flow of program execution. Consider the logical category $B = \{\text{True}, \text{False}\} \overset{\text{ADT}}{=} \text{True} + \text{False}$ and some function $f : \mathbb{R}^n \to B$. Now this can now be used for branching, for example: if $(f(\text{state}))\{\ \text{action}_1\}$ else $\{\ \text{action}_2\}$. For some actions, such branching can also be implemented in classical deep learning. For example: if $(c(\text{state}) > 0)\{v = a\}$ else $\{v = b\}$ can be transformed into $w = \sigma(c(\text{state})), v = wa + (1 - w)b$, where $c : \mathbb{R}^n \to \mathbb{R}$.

3.3. Object as a Product and a Set of Features. Object can be represented as product of features. For example: Iris = SepalLength $\times$ SepalWidth $\times$ PetalLength $\times$ PetalWidth. The resulting architecture will be equivalent to a matrix multiplication to V if sum approach for product type is used and the auxiliary functions g_i are the product of the input scalar x_i and the trainable vector v_i, where V is a matrix stacked from v_i.

Object type can be represented as a set of feature types. For example: IrisFeature = SepalLength + SepalWidth + PetalLength + PetalWidth and Iris = Set(IrisFeature). The resulting architecture will be equivalent to a matrix multiplication again, if the same auxiliary functions from product type were selected for sum type and summation was used to aggregate the set type. Also this proves that the architecture with a Linear transformation is invariant to permutations of features in the dataset.

3.4. Missing Values Encoding. The previous model can handle missing values, because if a value is missing, the corresponding feature will not be added to the set. The resulting architecture is equivalent to the standard zero-filling approach.

However, using ADT, it is possible to directly encode missing values. The object type again will be specified as a feature product type. Each feature can be specified as a sum type of a number and an empty type. Now this type of object will take into account possible missing values that may occur in it. Therefore, the DL architecture built for this type of data will be able to work with missing values without pre-imputing it. Note that although there is an autoencoder that claims to impute missing values, it

actually restores the object after imputing missing values with another method.

If the previously proposed approaches are used to implementing functions for the corresponding ADT constructors, the following function will be obtained. For each feature of the object a vector is obtained. If the feature value was missed, then the trainable constant vector is taken. Otherwise, the feature value is multiplied by another trainable vector. After this, the vectors are added or concatenated depending on the chosen approach to working with a type product. In this work the sum of vectors will be used, since the resulting architecture will be more similar to the standard one that simply processes vectors. The resulting architecture will be equivalent to zero-filling approach, but with the addition of a new missing indicator feature. An example at Figure 4.

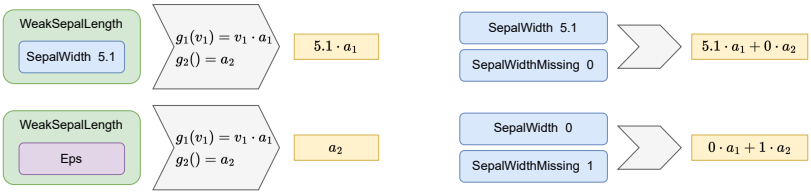

Figure 4. An example of obtained architecture for working with missing values (left) and standard imputing with an indicator addition (right). The upper case is when the value of the feature is present, the lower case is when it is missing.

3.5. Category Encoding. As mentioned in the Introduction, a category can be represented as a sum of the category's value types. And type of each value can be represented as an empty type. It turns out that the auxiliary functions for the sum type will be trainable constant vectors. So each category is associated with a trainable vector, as in equivalent trainable embeddings approach.

This approach can be extended to recommender systems with the following encodings: User $= \sum_i \text{User}_i$, Item $= \sum_i \text{Item}_i$, Object $=$ User $\times$ Item. If concatenation will be used for type product and decision function will be dot product, the resulting architecture will be equivalent to matrix decomposition.

§4. Experiments with Handling Data with Missing Values

For practical testing of the proposed method, the task of classifying data with missing values was chosen. Because, to build the corresponding architecture, it is necessary to implement the construction of the architecture for all basic ADT constructors: Eps, Vec, product and sum.

4.1. Datasets. The experiment was conducted on 459 medium-sized datasets from the OpenML database. The distributions of the basic characteristics of the datasets are presented in Figure 5.

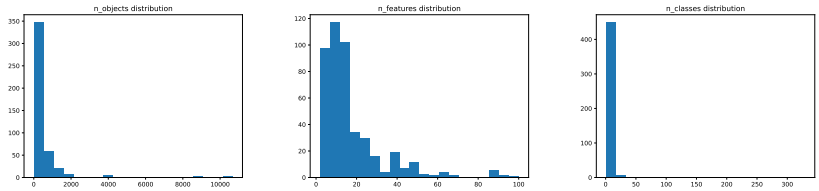

Figure 5. The distributions of number of objects, features and classes in the considered datasets.

All categories except the target one were transformed using one-hot encoding and PCA. From each dataset, 25% and 50% and 75% of values were removed in two ways: a random subset of values and values not exceeding the corresponding quantile value were removed. The missing values were then imputed using three standard imputers from the scikit-learn [13] library: SimpleImputer, KNNImputer, IterativeImputer. Each imputer was run twice: with standard parameters and with the addition of a missing indicator feature in dataset. Such imputers will be named with the suffix WI.

4.2. Experiment Setup. In the experiment, two similar architectures were trained. The first one processed vector data with missing values imputed by different imputers. Let $n = \#\text{objects}$, $f = \#\text{features}$, $c = \#\text{classes}$, $h =$ hidden size. If new features were used as a missing indicators, $f = \#\text{features} \,/\, 2$. The model consisted of the following PyTorch [1] modules:

$$\text{Sequential}(\text{Linear}(f, h), \text{ReLU}, \text{Linear}(h, h), \text{ReLU}, \text{Linear}(h, c))$$

The value of h was chosen as the solution to the equation: $(fh+h)+(h^2+h)+(hc+c)=p$, where $p=\frac{n}{10}$ is the expected number of parameters. So $h=\max\left(3,\left[\frac{\sqrt{s^2-4(c-p)}-s}{2}\right]\right)$, where $s=f+2+c$.

The second architecture was almost the same, but the first transformation $\mathrm{Linear}(f,h)$ was replaced by a model built from the corresponding ADT description $\prod_i(F_i+\varepsilon)\to\mathbb{R}^h$. Since this model can directly handle missing values, it was fed data without imputation as input.

Each dataset was split into training and testing parts in a ratio of 3:1. Both architectures were trained for 50 epochs using stochastic gradient descent with Adam optimizer [10]. Cross entropy [2] with SoftArgMax was used as the loss function. Weighted F_1-score was used as testing quality function.

4.3. Results. The results of the experiment are presented in Figure 6.

As can be seen from the graphs, the imputors performed approximately equally. When missing values were removed randomly, imputors without adding missing values indicators performed better than imputors that did. The ADT-based model performed slightly worse than the others.

When the positions of missing values were not random and themselves contained dependency from quantiles, imputors that added missing values indicators performed better than imputors that did not. The ADT-based model performed better than the others.

Despite the fact that theoretically the ADT-based model and simple model with missing indicator should be equivalent, in practice it turned out that they differ. Most likely, this is due to the explicit separation of the learning parameters for cases when feature values are present and when they are missed.

§5. Conclusion

In this paper, a method for constructing deep learning models from algebraically typed data was proposed. For this purpose, methods for connecting existing architectures for the main ADT constructors were developed. Theoretical experiments showed that the proposed approach can formalize and derive existing methods of working with data. A practical experiment showed that the proposed approach allows construct deep learning models that are superior in some scenarios to existing approaches for filling missing values. A disadvantage of the proposed approach is the additional

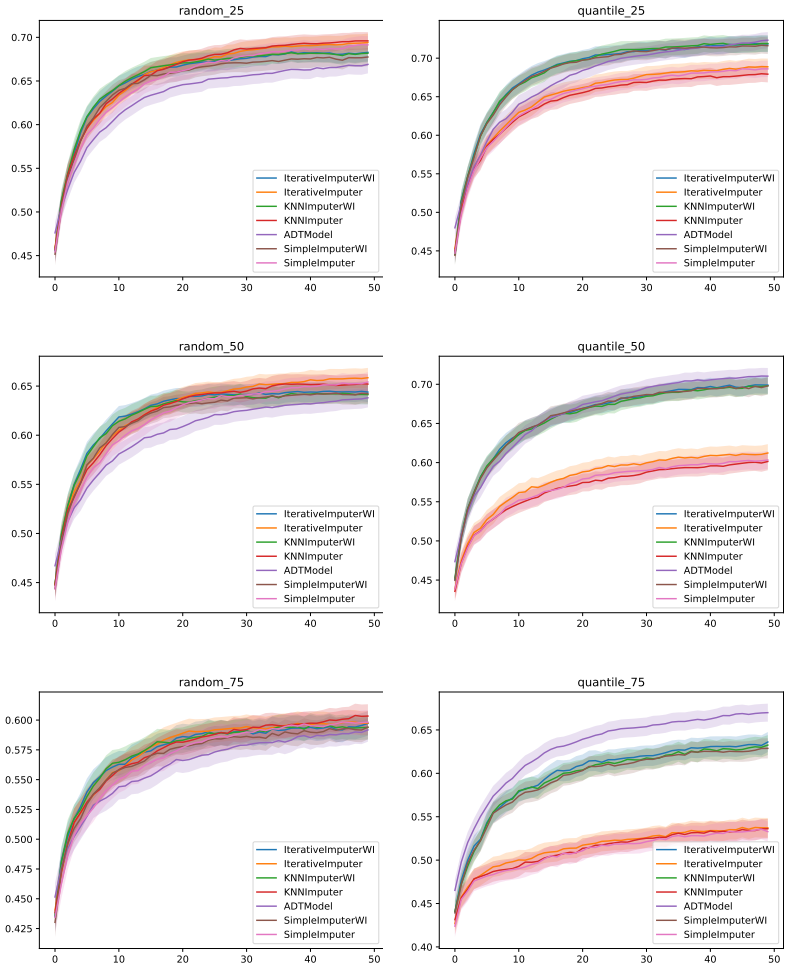

Figure 6. Dependence of the weighted F-score averaged over all datasets on the epoch.

branching of the execution flow, which can slow down the operation of the constructed model due to the use of SIMD parallelization.

§6. Future Work

The described approach requires using many basic functions for simple types to build a function for a complex type. This process can theoretically be automated using NAS [4]. It is also necessary to create a full-fledged framework with the ability to check types and integrate it into the NAS system. This is difficult now because datasets typically do not contain such complex type information.

This approach can also be modified to construct operators for evolutionary computations or even hash functions. For example, such approach is used to solve a tree isomorphism problem [14], when Tree = Eps + Set(Tree) and hash of set is a sum of subtree hashes with some non additive function application.

Also it can also be applied to typify hyperparameters. For example, hyperparameters of the nearest neighbor classifier can be specified by the type Dist × Window × Kernel, where Dist = Minkowski($\mathbb{R}^+$) + Cosine, Window = Variable($\mathbb{R}^+$) + Fixed($\mathbb{N}$), Kernel = Gauss + Binominal($\mathbb{R}^+ \times \mathbb{R}^+$).

Types can be extended, which can be useful for automatic transfer learning. For example, an image from a MNIST dataset can be typed as $\mathrm{Img}_{28\times28}(\mathbb{R} \in [0; 255])$ and input of pre-trained VGG19 as $\mathrm{Img}_{224\times224}(R \times G \times B)$, where $R = \mathrm{Norm}_{\mu=0.485,\sigma=0.229}(\mathbb{R} \in [0; 1])$ etc. The task is also much easier to define using ADT. For example, segmentation task target: Img(Class), object detection task target: Set(BoundingBox × Class), Word2Vec task with bag of words: Set(Word) → Word.

The theoretical research can be continued with other ADT constructors. For example, a function of type $X \to Y$ can also be written in ADT as Y^X. Also, ADT can be symbolically differentiated, which makes sense in analytical combinatorics and functional programming, but has not been researched for deep learning.

References

1. J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, *et al.*, *PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation*, in: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, vol. 2, 2024, pp. 929–947.
2. P. Blanchard, D. J. Higham, and N. J. Higham, *Accurately Computing the Log-Sum-Exp and Softmax Functions.* — IMA Journal of Numerical Analysis **41**(4) (2021), 2311–2330.

3. M. M. Bronstein, J. Bruna, T. Cohen, and P. Veličković, *Geometric Deep Learning: Grids, Groups, Graphs, Geodesics and Gauges*, ArXiv preprint arXiv:2104.13478 (2021).
4. T. Elsken, J. H. Metzen, and F. Hutter, *Neural Architecture Search: A Survey.* — Journal of Machine Learning Research **20**(55) (2019), 1–21.
5. P. Flajolet and R. Sedgewick, *Analytic Combinatorics*, Cambridge University Press, 2009.
6. R. Garipov, A. Zabashta, and A. Filchenkov, *Deep Learning Architecture for Processing Tabular Data to Extract Meta-Features*, Записки Научных Семинаров ПОМИ, 2025.
7. B. Gavranović, *Fundamental Components of Deep Learning: A Category-Theoretic Approach*, ArXiv preprint arXiv:2403.13001 (2024).
8. G. Giorgidze, T. Grust, A. Ulrich, and J. Weijers, *Algebraic Data Types for Language-Integrated Queries*, in: Proceedings of the 2013 Workshop on Data Driven Functional Programming, 2013, pp. 5–10.
9. H. S. Jomaa, L. Schmidt-Thieme, and J. Grabocka, *Dataset2Vec: Learning Dataset Meta-Features.* — Data Mining and Knowledge Discovery **35**(3) (2021), 964–985.
10. D. P. Kingma and J. Ba, *Adam: A Method for Stochastic Optimization*, ArXiv preprint arXiv:1412.6980 (2014).
11. A. Mahajan and D. Teneketzis, *Multi-Armed Bandit Problems*, in: Foundations and Applications of Sensor Management, Springer, 2008, pp. 121–151.
12. L. R. Medsker, L. Jain, *et al.*, *Recurrent Neural Networks*, Design and Applications, vol. 5, 2001, pp. 64–67.
13. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, *Scikit-Learn: Machine Learning in Python.* — Journal of Machine Learning Research **12** (2011), 2825–2830.
14. G. Valiente, *Tree Isomorphism*, in: Algorithms on Trees and Graphs, Springer, 2021, pp. 113–180.
15. A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, *Attention Is All You Need*, in: Advances in Neural Information Processing Systems **30**, 2017.
16. C. J. C. H. Watkins and P. Dayan, *Q-Learning.* — Machine Learning **8** (1992), 279–292.
17. M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Poczos, R. R. Salakhutdinov, and A. J. Smola, *Deep Sets*, in: Advances in Neural Information Processing Systems **30**, 2017.
18. L. Zhou, *A Survey on Contextual Multi-Armed Bandits*, ArXiv preprint arXiv:1508.03326 (2015).

ITMO University,
Kronverksky Prospekt 49, bldg. A,
St. Petersburg 197101, Russia
E-mail: azabashta@itmo.ru
E-mail: aaafil@mail.ru

Поступило 28 февраля 2025 г.