BUILDING, VISUALIZING AND EXECUTING DEEP LEARNING MODELS AS DATAFLOW GRAPHS

Gábor KRUPPAI*, Péter LEHOTAY-KÉRY**, Attila KISS***
** ***Department of Information Systems, Faculty of Informatics, ELTE Eötvös Loránd University,
E-mail: *kruppaigabor@gmail.com, **lkp@caesar.elte.hu, ***kiss@inf.elte.hu

ABSTRACT

In recent years many frameworks have appeared, which enable users to easily build, visualize and execute deep learning networks on graphical interfaces. However, they do not always provide enough opporunities to automate this process.

Generally, data processing programs can be organized into dataflow graphs that define the operations to be performed sequentially on the data. The operation of deep learning neural networks can also be interpreted in a similar way, in which the input data to be processed is a specific data set and the operations to be performed on the data are the layers of the net.

Due to architectural reasons, the entire deep learning neural network graph must be built before actual running, thus it is necessary to change topological execution of dataflows to evaluation preceding graph building since knowing the layers separately is not enough to operate the nets. As a solution for displaying editable program graphs, we created a framework in which data processing related to Python packages can be described and the programs built from them can be visualized and executed (mostly) automatically.

Keywords: *artificial neural networks, dataflow, deep learning, graphs, visualization*

1. INTRODUCTION

The main goal of this research was to help create and run models as quickly, transparently and easily as possible when creating and testing models related to data processing. Our approach was to create a framework that can (mostly) automatically generate a graphical interface for arbitrary Python packages which can be used to construct computational models related to the chosen package.

As an example implementation, we choose Keras [1], which contains high-level APIs for deep learning neural networks, where the visual representation of structures could contribute to the development process.

Keras is capable of running on top of TensorFlow, CNTK, or Theano but there are many other frameworks available (to use). First, we are going to present and discuss some of these.

Second, we are going to give some theoretical background for dataflow graphs and neural network structures.

Third, we are going to present our approach in the transformation of callable entities into nodes and then go into our implementation, discussing the main modules.

Fourth, we are going to present our solution in graph editing and program execution, discussing our web interface, the saving and submitting of models and the graph execution.

Last, we are going to discuss our experiments about testing our framework, building deep learning neural network models with it.

2. RELATED WORKS

There are some works discussing various machine learning tools, for example *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow* [2] introduces Scit-Learn, Keras and TensorFlow as possible Python frameworks and *Deep learning with Python: develop deep learning models on Theano and TensorFlow using Keras*, which discusses Keras on Theano and TensorFlow [3].

CNTK [4] (Computational Network Toolkit) is a powerful computation-graph based deep-learning toolkit for training and evaluating deep neural networks. It is used for example to create the Cortana speech models and web ranking.

Theano [5] is a framework in Python for defining, optimizing and evaluating expressions involving high-level operations on tensors. It is a general mathematical tool, but was developed with the goal of facilitating research in deep learning.

Scikit-Learn [6] is a Python module integrating a wide range of machine learning algorithms for medium scale problems. This package focuses on bringing machine learning to non-specialists using a general purpose high-level language.

Visualization tools are already available for similar tasks (such as TensorBoard [7]).

Authors of *Visualizing Dataflow Graphs of Deep Learning Models in TensorFlow* [8] presented a design study of the TensorFlow Graph Visualizer, part of the TensorFlow machine intelligence platform. They built a clustered graph using the hierarchical structure annotated in the source code to provide an overview. They described example usage scenarios and reported user feedback, to demonstrate the utility of the visualizer.

In *Tensorflow: Large-scale machine learning on heterogeneous distributed systems* [9], authors described the TensorFlow interface and an implementation of that interface that they had built at Google.

However, these are mostly processing program codes made for displaying purposes only, so the programs cannot be modified with these tools.

There also exist program-graph editors, for example Node- RED for IOT (i.e. Internet of Things) devices.

Authors of *Toward a Distributed dataflow Platform for the Web of Things (Distributed Node-RED)* [10] explored how to extend existing IoT dataflow platforms to create a system suitable for execution on a range of run time environments, toward supporting distributed IoT programs that can be partitioned among servers, gateways and devices. They aimed to automate the distribution of dataflows using appropriate distribution mechanism, and optimization heuristics based on participating resource capabilities and constraints imposed by the developer.

Another example for program-graph editors is RapidMiner [11] for data science projects in Java.

In *Collaborative Analysis of Cancer Patient Data using Rapid Miner* [12], authors performed a collaborative analysis on cancer patient data sets on different attributes of the values. The analysis of data i.e. parameter values had been done through the process of mining by RapidMiner.

Authors of *Data Mining model performance of sales predictive algorithms based on RapidMiner workflows* [13] applied RapidMiner workflows to process a data set containing information about the sales over three years of a large chain of retail stores. They constructed a Deep Learning model performing a predictive algorithm suitable for sales forecasting.

Although solutions have already been made, they do not use Python, but programming languages and their graph nodes have to be prepared one by one without any automatization.

3. DATAFLOW GRAPHS IN PROGRAMMING

Definition 3.1. (see [14]) *A dataflow graph is a bipartite labeled graph where the two types of nodes are called actors and links.*

$$G = (A \cup L, E)$$

where the set of actors, links are

$$A = \{a_1, a_2, .., a_n\}, L = \{l_1, l_2, .., l_m\}$$

and the set of edges is

$$E = (A \times L) \cup (L \times A)$$

Actors represent functions and links are treated as placeholders of data values (tokens) as they flow from actors to actors. Edges are the channels of communication.

Stream-like structures [15] are most commonly used in applications related to data processing. In such programs, there are two main types of components: data transformations and data pipes. These components can be represented in the graphs defined above, where transformation functions can be interpreted as nodes and data pipes as edges.

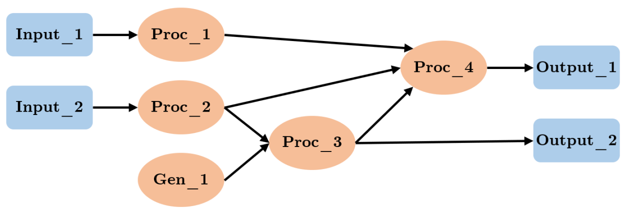


Fig. 1 Schematic diagram of a dataflow graph with multiple inputs and outputs allowing edge splitting and joining.

Directed edges have only one purpose, to transfer the computed/processed data from one node to the proper input of another. The functions that the node represents are only able to process or transform the data if all the required parameters are completely provided so that all the input data ”arrive” from the preceding nodes in the ”pipes”. Finally, the result(s) are forwarded to the output(s) of the node and the execution will continue analogously.

It is indispensable to the program termination that the graph does not contain directed cycles. In this case, there exists a topological order and it can be computed. Evaluating the nodes in that order means that the current node can get all its inputs immediately as all the nodes on which it depends have already been executed.

4. NEURAL NETWORK STRUCTURES

Definition 4.1. (see [16]) *A neural network consists of an input layer of neurons, hidden layers of neurons, and a final layer of output neurons. Each connection is associated with a numeric number called weight.*

$$h_i = o\left(\sum_{j=1}^{N} V_{ij} x_j + T_i^{hid}\right)$$

is the output of neuron i in the hidden layer, where o is called activation function, N the number of input neurons, V_{ij} the weights, x_j inputs to the input neurons, and T_i^{hid} the threshold terms of the hidden neurons. The neurons in the output layer are the last participants in the calculation, they produce the output.

In its internal structure, a neural network can be considered to perform linear algebraic operations. The outer structure of the net can be seen as a multi-parameter function which takes and returns multidimensional arrays with predefined sizes.

The result calculated from the input data is determined by the complex internal structure, which can store up to tens of millions of internal parameters, depending on the application. During the execution, matrices and matrix functions stored in this internal structure are used to compute predefined sequences of operations.

During the training phase, the proper iterative adjustments of the internal variables ensure that the model can get as close as possible to the solution of the given task, which is defined by the training data set providing input-output pairs. To sum up, the algorithm approximates the function represented by the net by using fixed input and output values. During the evaluation phase, the role of the external data and internal parameters are interchanged, so that the internal parameters determined during the training phase are used to calculate the result.

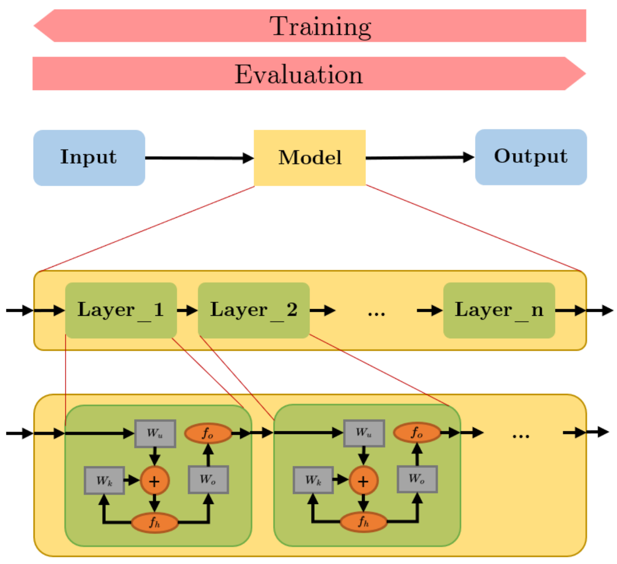


Fig. 2 Detailed example of neural networks inner structure. A model consists of layers, and a layer consists of linear algebraic operations and applied activation functions.

4.1. Neural Network Libraries

Deep learning software can be divided horizontally into low- and high-level libraries. The low-level, hardware related, linear algebraic and analytics libraries are optimized specially for basic operations and functions, whereas the more complex structures are made by combining them.

High-level libraries implement APIs that handle more complex neural layers rather than just variables. These two levels are strongly connected as they are built on each other, since neural layers implemented by higher-level APIs consist of a pre-compiled composition of variables and functions provided by low-level libraries, where the API hides its internal components to form a higher abstraction level.

The most widespread low-level library is TensorFlow, developed by Google [17]. Its high-level counterpart is Keras, which is written in Python, in which predefined layers can be combined as building blocks to create complete, trainable models.

4.2. Parallels with Dataflow Graphs

From both low and high-level approach, it can be seen that the data stream model could easily be applied to them if there were no need to modify the internal variables of the model during the training phase with respect to the training data set. With the exception of the training phase, the representation would be logically correct, even though the low-level implementation works differently.

In addition, full models have to be built first in order to train and run networks, and only then they can be evaluated. Consequently, we are not able to use and see the separated partial results of the neural model without building the whole model. Ignoring the underlying implementation and eliminating the initialization problem caused by the need for building the whole model, we have a logically correct, easy-to-read graph that could be edited easily to test and build new models without coding.

5. TRANSFORMING CALLABLE ENTITIES INTO NODES

Our goal is to reproduce existing libraries so that their elements can be graphically displayed and combined. This would lead to an interface where the functions and the classes which exist in the current library can be used in a visual interface.

In order to make it work with larger packages, it is necessary to support the migration by the automatic transformation between Python modules and our entity description.

5.1. Python ”inspect” Module

The Python ”inspect” module provides a wide variety of functions for retrieving information about modules, classes, methods, functions and other objects at run time. The most important aspect of the conversion mentioned above is to determine the existing elements of the modules and their attributes (e.g. type or signature). This could be achieved by reading the source code of the objects and analyzing it.

As the main aim is to bind a graph node to a callable object, we need to know their signature, which is the name of the object and its parameters, position, type, default value (if exists), and expected values. Using the ”inspect” module, we can retrieve not only these properties but even code comments and documentation if they are provided in the correct format. If we scan the module thematically, we can tell the path and the usage information for each object in the given module.

5.2. Python ”importlib” Module

In Python, it is also possible to dynamically import any module at run time. The easiest way to do this - beyond the standard ”import” keyword - is to use the built-in ”importlib” package and its ”import_module” function.

If the location of a specific object is known in its module (as a package string), it can be dynamically loaded into the memory at run time. It exactly meets our expectations when we would like to load a component only if it was used in the program.

5.3. Information Extraction

The first step of transforming nested package objects into graph nodes is to collect them into a list, with which the above-mentioned ”inspect” module can help. It is necessary to apply some filter criteria during the recursive package traversal in order to skip undesirable objects and solve reference cycles. After the traversal, we will have a specific set of objects and their location in the tree-like modules.

In order to the use the collected callable objects, the signatures and other calling settings must be determined for each object in the selected set. The signatures can be queried by the above mentioned ”inspect.signature” function, which will provide argument information extracted from the code text.

However, the information obtained by the above method is still not sufficient to describe all the object calling styles. In order to achieve proper object calls with ”type-correct” (i.e. classes that the function expects) arguments, additional call options and argument type information have to be gathered.

If the default values are provided in the signature of the function, the type of the argument can be deduced, although, it may vary since Python are dynamically typed and there is a possibility to accept various types/classes (e.g. string and int). Unfortunately, accepted types cannot be predicted from the plain code text without having some kind of structured documentation, thus, it cannot be automated and requires human revision.

Unfortunately, the call options, such as instantiation, application or optional type checks, also require manual completions:

- if the node represents a callable object, we have to provide if the class has to be instantiated or just passed as is to the nodes following;
- if a function call has side effects on an object, we have to consider deep-copy – if possible – the object before the function call to prevent future anomalies;
- if an object is called multiple times in series by different nodes (application), we should indicate that in the node description wherein it is represented;
- if the function has special argument types and we want to ensure the type correctness, we should change the automatic argument determination (as its working for basic types only);

There could be cases where intermediate calls or extra data/code transformations are necessary without the ability of calling an existing function from any package (no such function exists or a special operation is needed). To solve this problem, we created a custom module, called ”fallback module” where all the user-defined functionality extensions can be placed. The advantage of the fallback module is to create and import utilities and wrapper functions in the framework without the need of installation.

After the automatic data extraction and the human revision performed, the following node (Python object) descriptor format will be available for each of the callable objects (we used JSON formatting to store the properties) to configure the graphical editor and inform the execution server about the usage of the objects:

```
{
  // Node name
  "name": "Input",

  // Object's location (fallback)
  "package": "kw.models.Input",

  // Input type(s) of the node
  "itype": "Dataset",

  // Output type of the node
  "otype": "Layer",

  // Instantiate or not
  "instance": true,

  // Call "previous" node
  // with this args.
  "apply": false,

  // Deep-copy on pass
  "copy": false,

  // Maximum number of
  // merged input(s)
  "conns": 1,

  // Argument descriptors
  "args": [{

      // Name of the arg.
      // in the code
      "arg": "shape",

      // Input type
      "itype": "shape",

      // Required argument
      "required": true

      // "otype", "default",
      // "conns", etc.

    }, {
      "arg": "name",
      "itype": "string",
      "default": null
    }, {
      "arg": "dtype",
      "itype": "nosupport",
      "default": null
    }, {
      "arg": "sparse",
      "itype": "bool",
      "default": false
    },

    // further arguments
    // ...

  ]
}
```

6. GRAPH EDITING AND PROGRAM EXECUTION

6.1. Web Interface

To properly visualize the components of the processed package objects as nodes, the previously extracted node descriptions are used. In our solution, we decided to implement a browser-based program-graph editor with the help of an existing diagram editor library called ”Go.js” [18], which was used with academic license in this work. The library’s main profile is to provide templatization and highly customizable diagram rendering.

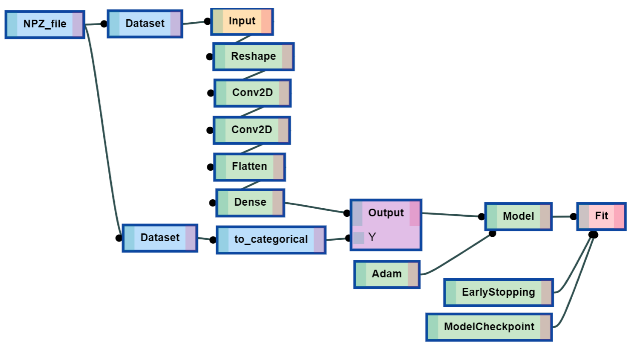
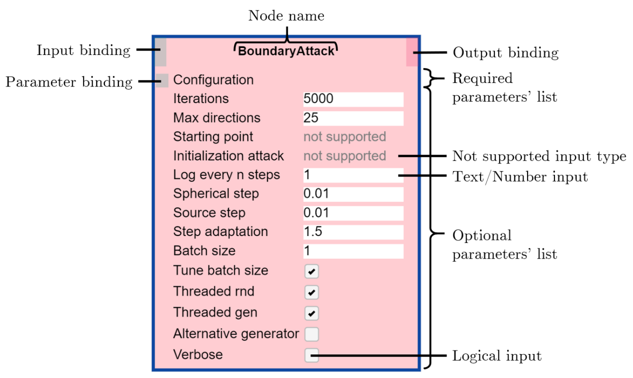


Fig. 3 The prepared functions can be dragged and dropped to the editing area and can be applied after each other by drawing edges (data pipes) between them. Every node can be displayed in ”overview” or ”detailed” mode (see Figure 4.)

Beside the calling instructions of the represented objects, the other important thing is to list the available argument fields. The arguments with basic typed default values are recognized during the automatic information extraction (5.3) and associated with the arguments, however, it is still impossible to give a general solution which always determines all possible types. Even the predicted basic types can differ as Python is dynamically typed so they may accept other types as well as the predicted ones, so human revision is important cannot be omitted.



BoundaryAttack

Configuration	
Iterations	5000
Max directions	25
Starting point	not supported
Initialization attack	not supported
Log every n steps	1
Spherical step	0.01
Source step	0.01
Step adaptation	1.5
Batch size	1
Tune batch size	✓
Threaded rnd	✓
Threaded gen	✓
Alternative generator	☐
Verbose	☐

Fig. 4 Nodes can be generated from the configuration objects prepared based on 5.3. As some of the input types cannot be determined automatically, ”not supported”* will be shown as they are not basic types but objects.
*Unfortunately, it can only be eliminated or resolved by modifying the configurations by hand.

6.2. Saving and Submitting Models

In order to run a Python program from the graphical model in the browser, it has to be serialized and sent to a Python interpreter (on the server) where it can be parsed and executed based on the extracted serialized model. The visual models can also be exported into JSON format which is identical to the serialized model used by the model submission to the execution server.

This JSON object contains all the necessary properties for both the execution server and the ”Go.js” visualization, like node definitions and links as well as the nodes’ actual arguments and execution state. Since this is the complete inner state of the client-side visualization, not only can it be exported, but easily imported as well.

6.3. Graph Execution

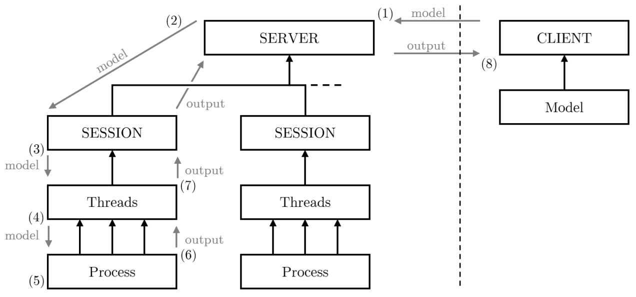


Fig. 5 Structure of the execution server. (1) Model serialization, sending to the server; (2) Execution, session creation; (3) Spawn new threads for model execution; (4) Parse the received program and create topological evaluation order; (5) Evaluate the graph nodes in a dedicated process; (6) Capture standard outputs; (7) Notify the session about the progress; (8) Send back progress and output data.

To execute the client-edited program, we also implemented an execution server (in Python) where the serialized program model (i.e. graph) can be sent to server, parsed and executed. As the server receives the executable model from the client, it parses the serialized data and rebuilds the same graph as it was created at client-side. It also checks if the received graph is cycle-free and then sorts the nodes into topological order to prepare them for sequential processing.

Checking if the received graph is cycle-free:

```
def getOrder(G):
    order = []
    for key, node in G.nodes.items():
        if node.getAttr('visited')
                is None:
            NNGraph.DFSorder(
                G, node, order)
    order.reverse()
    return order
```

Sorting the nodes into topological order:

```
def DFSorder(G, root, order = []):
    root.setAttr('visited', False)
    for link in root.links_to:
        node = G.nodes[link.to_node]
        state =
            node.getAttr('visited')
        if state == False:
            raise Error('Graph
                is not a DAG!')
        elif state is None:
                NNGraph.DFSorder(
                    G, node, order)
        order.append(root.id)
        root.setAttr('visited', True)
```

The evaluation starts from the first node of the topological order and moves on one by one, taking the previously computed values from preceding nodes – if they are necessary for the current node. During the evaluation of the current node, the server-side framework reads the represented object package path from the model graph and tries to import it from the installed packages first.

If the requested module or function does not exist globally, a user defined fallback module will be tried, which can be useful not only for error handling but for package extension. If the fallback module and its function is also missing, an error will be thrown and the whole execution will stop.

Another crucial point is the side effect and "noncopyable" object handling. If a node's output value is bounded to multiple subsequent nodes but calling a function on it causes modification in its value, only the first node will get the original value and thee others may fail because of the "unintended" value change through references.

To solve this problem, it should be ensured that the node's original return value does not change by deep-copying – if possible – it on demand. To avoid most of the unexpected behaviors, the framework tries to make a deep-copy from all objects if possible; otherwise it leaves them as they are.

Each evaluation step (evaluation of a single node) starts and ends with meta signals which are captured and forwarded to the client to inform the user about the progress and the time consumed. Beside meta signals, standard contents of outputs are also captured and sent back to the client.

Evaluation:

```
def run(clsFallback='module',
        progress=None):
    for key in order:
        if progress is not None:
            progress(G.nodes[key],
                'progress', 0)
        time_start = time.clock()
        evaluate(G.nodes[key],
            clsFallback)
        time_end = time.clock()
        if progress is not None:
            progress(G.nodes[key],
                'finished',
                time_end-time_start)

def evaluate(node,
        clsFallback='module'):
    cls = EngineBase.importClass(
        *node.package, clsFallback)
    ret = None
    (inputargs, required, kwargs)=
        prepareArguments(node)
    if node.instance and node.apply:
        ret = cls(*required,
            **kwargs)(*inputargs)
    elif node.instance and
            not node.apply:
        ret = cls(*inputargs,
            *required, **kwargs)
    elif not node.instance and
            node.apply:
        ret = cls(*inputargs)
    elif not node.instance and
            not node.apply:
        ret = cls
    node.setAttr('return', ret)
    EngineBase.flush()
```

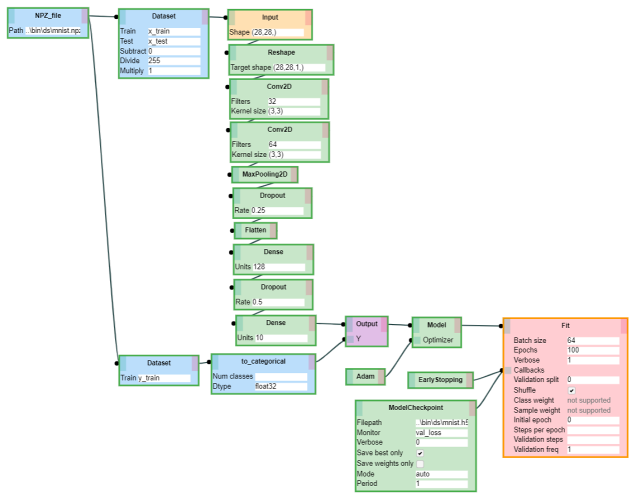
7. EXPERIMENTS

To properly test the framework, we built several neural network models with it. In order to manage the required data for the nets, we used exported "numpy" objects (".npz" files) to store and load data sets for both training and validation, so additional "npz" data loader nodes were implemented to work with them. To show the working mechanism and capabilities, a simple sequential (Figure 6) and a functional (Figure 7, with layer splitting and merging) model were created.

The nodes with different logical functionality are marked with different colors. In the screenshots below, blue indicates data managing (e.g. loaders, transformations), yellow and purple means model input and output, green nodes belong to the neural model (e.g. layers, optimizers, functions, configurations) and red indicates the compute-intensive calls (such as training and evaluation).

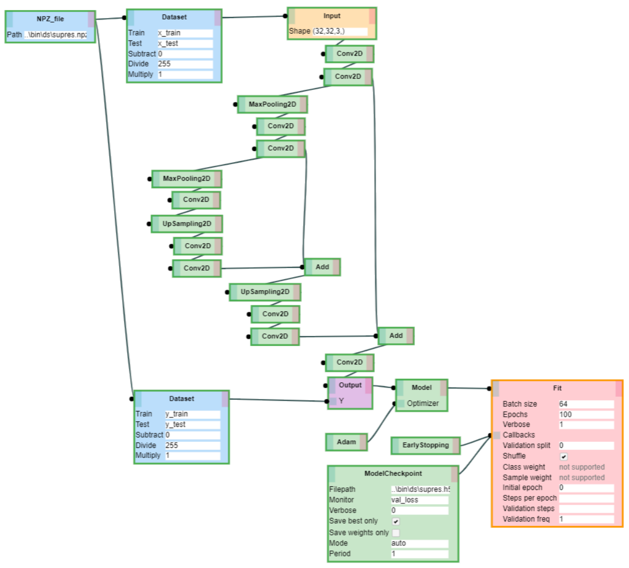
In total, we made 91 nodes available from Keras package in our framework, 76 of which were auto-generated, 10 required a wrapper class and 5 needed to redefined including the 3 data nodes. In addition, another library – Foolbox [19] was also transferred and tested with our program in order to generate adversarial attack inputs with a wide variety of methods.

Foolbox required a little more (10) functions to be wrapped in totally self-defined objects, and 32 functions needed almost identical one-lined wrapper functions, which could be eliminated in the future by recognizing even more calling information. These manual extensions were enough to cover more than 45% of Keras and around 95% of Foolbox functions and classes.



Path ..\bin\ds\mnist.npz
Train x_train
Test x_test
Subtract 0
Divide 255
Multiply 1
Shape (28,28,)
Target shape (28,28,1,)
Filters 32
Kernel size (3,3)
Filters 64
Kernel size (3,3)
Rate 0.25
Units 128
Rate 0.5
Units 10
Y
Optimizer
Train y_train
Num classes
Dtype float32
Filepath ..\bin\ds\mnist.h5
Monitor val_loss
Verbose 0
Save best only ✓
Save weights only
Mode auto
Period 1
Batch size 64
Epochs 100
Verbose 1
Callbacks
Validation split 0
Shuffle ✓
Class weight not supported
Sample weight not supported
Initial epoch 0
Steps per epoch
Validation steps
Validation freq 1

Fig. 6 Simple MNIST convolutional neural network sequential model from https://keras.io/examples/mnist_cnn



NPZ_file: Path ..\bin\ds\supres.npz

Dataset: Train x_train; Test x_test; Subtract 0; Divide 255; Multiply 1

Input: Shape (32,32,3,)

Dataset: Train y_train; Test y_test; Subtract 0; Divide 255; Multiply 1

Output: Y

Model: Optimizer

ModelCheckpoint: Filepath ..\bin\ds\supres.h5; Monitor val_loss; Verbose 0; Save best only ✓; Save weights only ☐; Mode auto; Period 1

Fit: Batch size 64; Epochs 100; Verbose 1; Callbacks; Validation split 0; Shuffle ✓; Class weight not supported; Sample weight not supported; Initial epoch 0; Steps per epoch; Validation steps; Validation freq 1

Fig. 7 Deep Denoising Super Resolution (DDSRCNN) from the collection at https://github.com/titu1994/Image-Super-Resolution, referring to [20]

7.1. Extending functionalities

Adding new packages and functions are also possible by generating (see 5.3) additional node (object) descriptors and appending them to the configuration file which contains these data. In our examples, we only ported specific parts from popular neural network related libraries. What is more, other fields can also benefit from the base framework.

Our original goal was to build programs by visual tools with as little preparation overhead as possible. This technique could also be useful in data processing and transformation if the model is extended with the required functions.

8. CONCLUSION

The dataflow structure has shortcomings over general programming, namely with circular object dependencies and side effect handling. In statically typed languages, the function-node transformation would be feasible in contrast to dynamically typed languages, where some of the nodes can be generated automatically but there is no fully general solution to handle every function signature.

However, creating editable graph view for Python packages with a relatively small amount of work is quite useful. In addition, this tool can be helpful for professionals in data processing and testing tasks like multimedia (image, audio, video) manipulation pipelines (e.g. with ”ffmpeg”), data mining applications and cycle-free network modelling. Finally, it can be used in education too, for example to teach basic programming skills for children as no language and coding knowledge is required.

9. FUTURE WORK

Possible improvements could be made if the execution system was changed from single node evaluations to actual program generation and running. That would bring a completely different backend structure and possibly increase the graph complexity on editing, which is against the endeavor of simplification but worth testing in the future works.

ACKNOWLEDGEMENT

The project has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

REFERENCES

[1] F. CHOLLET et al. Keras documentation. *keras. io*, 2015.

[2] A. GÉRON. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O’Reilly Media, 2019.

[3] J. BROWNLEE. *Deep learning with Python: develop deep learning models on Theano and TensorFlow using Keras*. Machine Learning Mastery, 2016.

[4] F. SEIDE and A. AGARWAL. Cntk: Microsoft’s open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2135–2135, 2016.

[5] J. BERGSTRA, F. BASTIEN, O. BREULEUX, P. LAMBLIN, R. PASCANU, O. DELALLEAU, G. DESJARDINS, D. WARDE-FARLEY, I. GOODFELLOW, A. BERGERON, et al. Theano: Deep learning on gpus with python. In *NIPS 2011, BigLearning Workshop, Granada, Spain*, volume 3, pages 1–48. Citeseer, 2011.

[6] F. PEDREGOSA, G. VAROQUAUX, A. GRAMFORT, V. MICHEL, B. THIRION, O. GRISEL, M. BLONDEL, R. PRETTENHOFER, P.and WEISS, V. DUBOURG, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[7] TensorFlow. (2019). TensorBoard, tensorflow.org/tensorboard.

[8] K. WONGSUPHASAWAT, D. SMILKOV, J. WEXLER, J. WILSON, D. MANE, D. FRITZ, D. KRISHNAN, F. B. VIÉGAS, and M. WATTENBERG. Visualizing dataflow graphs of deep learning models in tensorflow. *IEEE transactions on visualization and computer graphics*, 24(1):1–12, 2017.

[9] M. ABADI, A. AGARWAL, P. BARHAM, E. BREVDO, Z. CHEN, C. CITRO, G. S. CORRADO, A. DAVIS, J. DEAN, M. DEVIN, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[10] M. BLACKSTOCK and R. LEA. Toward a distributed data flow platform for the web of things (distributed node-red). In *Proceedings of the 5th International Workshop on Web of Things*, pages 34–39, 2014.

[11] RapidMiner. (2019). Lightning Fast Data Science Platform for Teams, rapidminer.com.

[12] P. JAIN and S. Kr. VISHWAKARMA. Collaborative analysis of cancer patient data using rapid miner. *International Journal of Computer Applications*, 145(2), 2016.

[13] A. MASSARO, V. MARITATI, and A. GALIANO. Data mining model performance of sales predictive algorithms based on rapidminer workflows. *International Journal of Computer Science & Information Technology (IJCSIT)*, 10(3):39–56, 2018.

[14] K. M. KAVI, B. P. BUCKLES, and U. N. BHAT. A formal definition of data flow graph models. *IEEE Transactions on computers*, (11):940–948, 1986.

[15] J. KODOSKY, J. MAC CRISKEN, and G. RYMAR. Visual programming using structured data flow. In *Proceedings 1991 IEEE Workshop on Visual Languages*, pages 34–39. IEEE, 1991.

[16] S.-CH. WANG. Artificial neural network. In *Interdisciplinary computing in java programming*, pages 81–100. Springer, 2003.

[17] TensorFlow. (2019). TensorFlow, tensorflow.org.

[18] F. SHAHZAD, T. R. SHELTAMI, E. M. SHAKSHUKI, and O. SHAIKH. A review of latest web tools and libraries for state-of-the-art visualization. *Procedia Computer Science*, 98:100–106, 2016.

[19] J. RAUBER, W. BRENDEL, and M. BETHGE. Foolbox: A python toolbox to benchmark the robustness of machine learning models. *arXiv preprint arXiv:1707.04131*, 2017.

[20] X.-J. MAO, CH. SHEN, and Y.-B. YANG. Image restoration using convolutional auto-encoders with symmetric skip connections. *arXiv preprint arXiv:1606.08921*, 2016.

Received March 2, 2020, accepted June 1, 2020

BIOGRAPHIES

Gábor Kruppai received his BSc degree in computer science at the Etvs Lornd University Faculty of Informatics in Hungary in 2019 and currently doing his masters degree with the specialization of mathematics. During his studies he attended several projects related to machine learning and taught various programming languages to university students.

Péter Lehotay-Kéry received his MSc degree in computer science at the Etvs Lornd University Faculty of Informatics in Budapest, 2018 and currently doing his PhD studies with specialization in information systems. His scientific research is focusing on databases, security, big data, data mining and bioinformatics.

Attila Kiss defended his PhD in the field of database theory in 1991. His research is focusing on database theory, data mining, artificial intelligence. Since 2010 he has been the head of Department of Information Systems at Etvs Lornd University.