Eötvös Loránd University
Faculty of Informatics
Department of Information Systems

Adversarial Framework for Neural Networks

dr. Kiss Attila
Head of department

Kruppai Gábor
Computer Science

Budapest, 2019

Eötvös Loránd Tudományegyetem
Informatikai Kar

SZAKDOLGOZAT-TÉMA BEJELENTŐ

Név:	Kruppai Gábor	Neptun kód:	A180FM
Tagozat:	nappali	Szak:	programtervező informatikus

Témavezető neve: dr. Kiss Attila

Munkahelyének neve és címe: ELTE Informatikai Kar,
Információs rendszerek tanszék
Budapest
1117, Pázmány Péter sétány 1/C

Beosztása és iskolai végzettsége: Tanszékvezető docens, matematikus

A dolgozat címe: Adversarial Framework for Neural Networks

A dolgozat témája:

Adversarial examples are intentionally created inputs to machine learning models that can could unexpected mistakes in the output. These malicious inputs can be dangerous for systems where high reliability are required. In order to test (or deceive) our models against such attacks, we can generate test inputs with various methods.

The purpose of my work is to create a universal framework which can simplify the process of adversarial example generation by hiding the mathematical background of the generation with a simple interface. The program will be capable to load (TensorFlow or Keras based) neural network models and generate adversarial inputs with built in methods.

With the help of the framework the potential weakness of the neural models can be easily discovered and be used for model development.

A témavezetést vállalom:

.....

(témavezető)

Kérem a szakdolgozat témájának jóváhagyását.

Budapest, 2018.11.28.

.....

(hallgató)

A szakdolgozat-témát az Informatikai Kar jóváhagyta.

Budapest, ...

.....

(a témát engedélyező tanszék vezetője)

Table of Content

1 Introduction

1.1 Image Processing with Neural Networks

Image processing and computer vison are probably the most researched area in machine learning. It can be applied in autonomous car driving, medical diagnosis, computer graphic and many more. Most of the methods in the field use the structure called "convolutional neural network" to extract useful information from the given inputs. During the learning phase, the neural model is trained with gradient descent algorithms which are on the one hand the cores behind the whole theory. But on the other hand, they also could be drawbacks when the security comes up.

1.2 Adversarial Example

Adversarial examples are inputs for neural networks that result incorrect outputs. Those inputs – most often images – could be called "visual illusions" to neural networks as they are very similar or almost identical with the original data but cause the model to make wrong decisions.

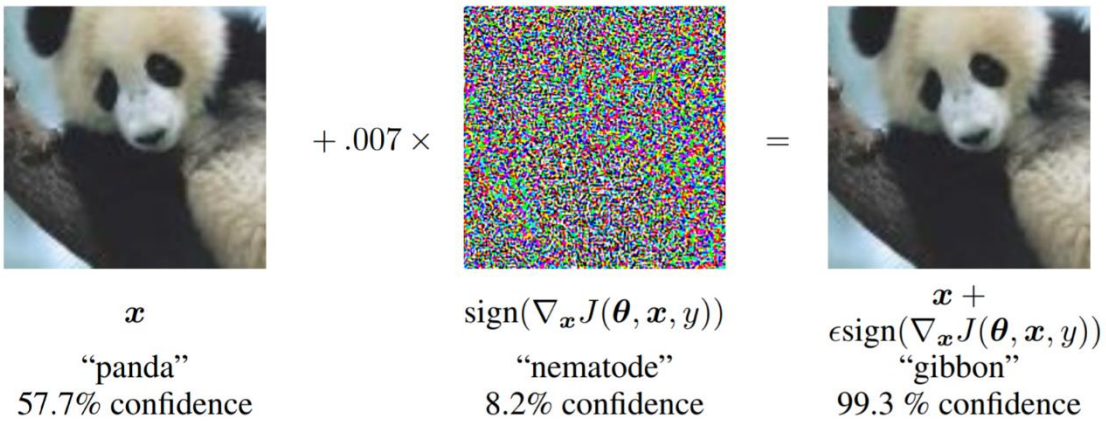

Figure 1: Adversarial image example from [1]. The two inputs on the sides look like the same to human observers but the well-constructed additional noise distracts the neural network based recognition.

For the suitable noise generation there are many and various methods which are mostly based on gradient methods although genetic algorithms are spreading quickly – since they are not required the model to be known. With the help of these algorithms and an appropriate framework, it is relatively easy to construct our own image for fooling.

Recent papers [2], [3] shows that even the most advanced machine learning models are affected by the vulnerability to deliberately perturbed inputs: adversarial examples. As the fields of using these techniques is rapidly broaden and most of the cases, we do not have full understanding about the models we use, security and reliability is getting more important.

1.3 Simplify the Data Generation Process

One approach for improving the "resistance" against malicious inputs is generating such inputs for the network then train the model on them.

Data related frameworks and some programming languages contains functional interface for treating the process as a convey belt by providing transformation functions and collections. For visualization, the functions can be nodes of a graph, the function calls can be the directed edges of the graph so that the actual processing is the evaluation of the graph.

This thesis work is trying to give a possible solution for speeding up the data generation process by implementing a visual interface for users to help building and evaluate transformation programs without coding. The final framework has been become more general and could handle multiple libraries if needed but the implementation and documentation focus on adversarial example generation connected functions.

2 User manual

The program was created to provide simple interface to create dataflow-like data processing programs. The module can be globally installed on both windows and linux systems. The basic module shipped with inbuilt adversarial image generator functions for neural network benchmarking (*which was original the purpose of this thesis work*). The implemented functions can be "dragged and dropped" to the editing area and can applied after each other by drawing edges (*data pipes*) between them.

The base program has functions (*elements/nodes*) to find, load, transform and save images, load and parse *Keras* models and apply adversarial attacks. For the detailed element descriptions see 2.3 below.

During the development, a more general framework (*FlowPy*) was evolved and separated from the original adversarial input generator modules. It means that the sufficient python program execution functionalities were extended with a lot of customizable settings. Therefore, *FlowPy*'s structure could be widely used in other application involving data processing in a conveyor belt manner. (*see the Developer documentation below*)

2.1 Installation and startup

2.1.1 Installation

Windows

FlowPy's backend is written in *python* so first python (*version greater than 3.5*) and *pip* package-manager should be installed on the system whose installer can be downloaded from python.org/downloads/windows which contains *pip* too.

After setting up the python environment without any error, for the automatic installation `setup.bat` should be called with administrator rights. If the script finished successfully, the installation finished.

Linux

The installation on Debian-based linux distributions can be done by giving execution permission to `flowpy/setup.sh` shell script and execute as super-user:

```
sudo chmod +x setup.sh
sudo setup.sh
```

2.1.2 Server startup

The web interface in the browser has a backend python server which could be run either on the same and a remote machine to serve the commands coming from the user interface.

FlowPy installs itself as a global python module. Using the "`-m`" parameter on python interpreter can specify a module to be loaded and executed in the current directory. (*for further argument flags and parameters, see 3.2.1 below*)

Windows

```
python -m flowpy
```

Linux

```
python3 -m flowpy
```

When "`Serving on` ..." message appears in the console/terminal, the web UI become accessible on the serving machine's address on the given port (*default: 80, http*). If the program was started on the same machine furthermore, none of the program flags were specified, the website will be available on http://localhost:80. (*It the server is running on a different host, make sure that the specified port can pass the remote's firewall.*)

On both the systems above, the program can be stopped by the user with the `Ctrl+C` keyboard command in the console/terminal (*exiting may takes time depending on the in-progress background processes*).

2.1.3 Uninstallation

Windows

FlowPy package can be removed by running `remove.bat` file as administrator.

Linux

FlowPy package can be removed from Debian-based linux distributions by giving execution permission to `flwpy/remove.sh` shell script and execute as super-user:

```
sudo chmod +x remove.sh
sudo remove.sh
```

2.2 Web interface

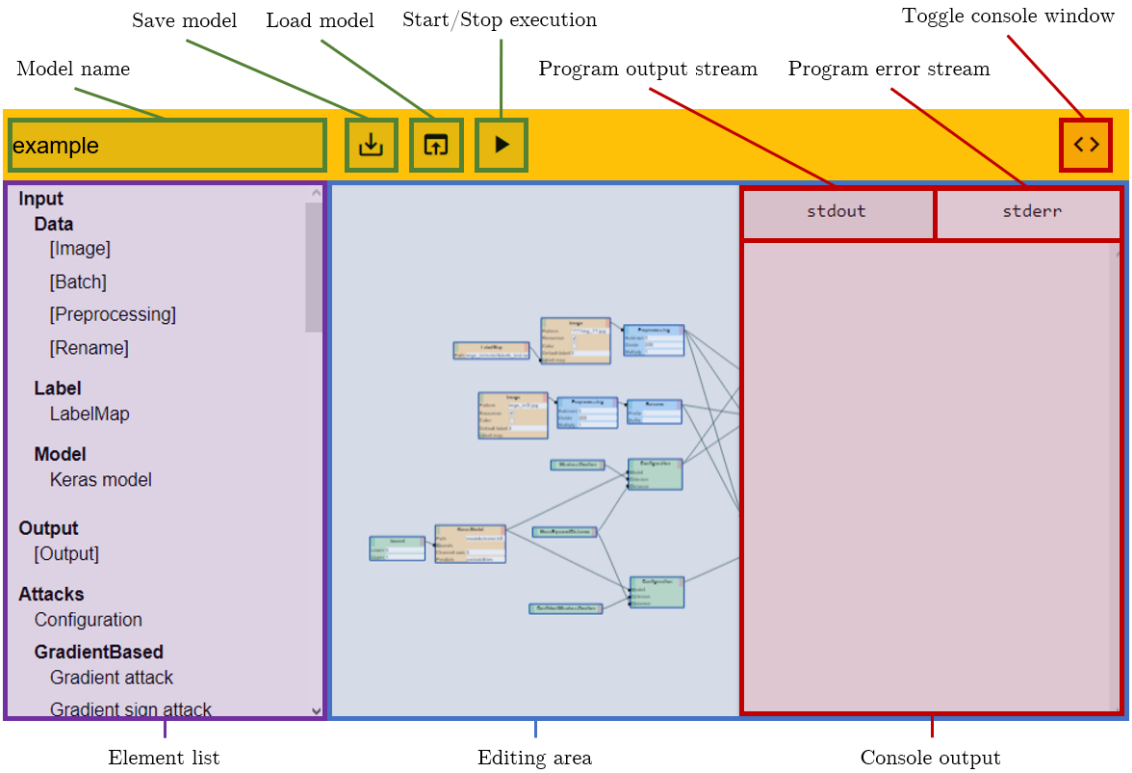

Figure 2: The main components of the web interface.

Web interface is accessible after the server successfully started (*see 2.1.2 above*). If the server was started on the same machine as the client wants to connect from and port number (*default: 80 – http*) was not specified through command line arguments, the interface can be reached by any browser on `http://localhost web address`. If the server is running on a different machine and/or port number was changed by "`-p`" option on start, the address of the web interface will change to `http://<host>:<port>` respectively to the given network environment and port settings. If the page is hosted on remote machine, make sure if the port is not blocked by the remote system's firewall (*usually, incoming network traffic is blocked on all ports, see your firewall's documentation to open ports*).

Page components

Header: contains the model name and action buttons

Element list: list of elements from which the processing model can be built up.

Editing area: visual diagram editing area where the elements can be arranged, connected and configured.

Console: real-time standard output/error stream from the model.

2.2.1 Nodes

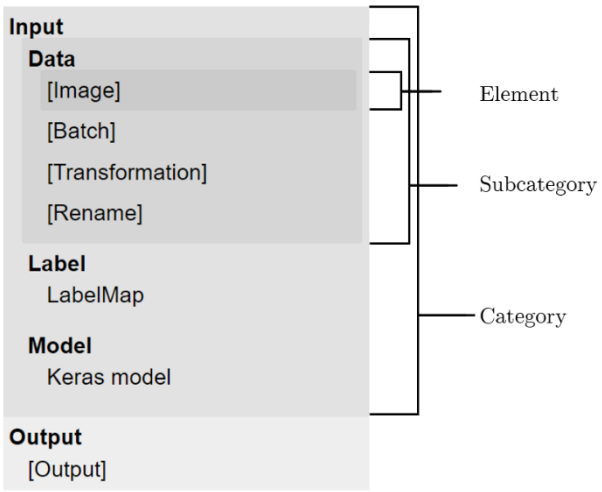

Figure 3: Node elements' list.

Usable model elements are listed on the left side of the page (*for element definition, see 3.4.1 in the developer documentation*). These elements can be added to the model by dragging the chosen element holding the left mouse button. The bold names indicate logical packages (*sub-packages are allowed*) which help to the user to find elements/functions. Brackets have been also placed around data transformer element's name (*like* "`[Batch]`") as an indication of plain input transformation. (*Note, that the element names in the list and on the editing area may differ due to restriction in the implementation. By convention, on editing area spaces and parentheses are removed from the name and multiple words are resolved with UpperCamelCase naming style.*)

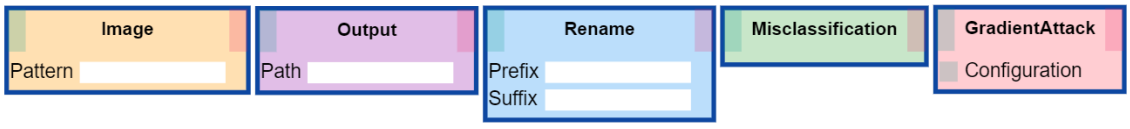

Figure 4: Elements has five main categories which are marked with different node colors for easier readability: input (ocher), output (purple), transformational (blue), configurational (green) and attack (red).

Different element categories can have different colors which is not correlated with the element packages in the element list, which mentioned above. (*Element style categories can be added, changed and removed in the configuration file, see 3.4.1 in the developer documentation.*)

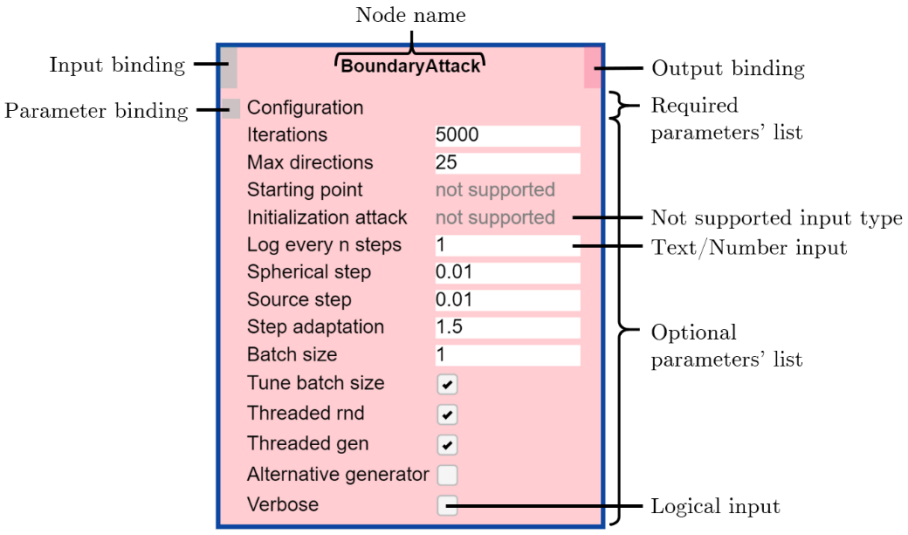

Figure 5: Node structure's parts and fields.

Node input and output

Each node has input and output data binding port (*top-left and top-right corners*) which connect the node to "data streams". The input and output ports are marked with greenish and reddish rectangles which other nodes' output or input can connects to.

Arguments

Arguments are separated into two parts: required and optional arguments. Required arguments must be specified before model execution otherwise program error will be thrown (*to standard error stream*) and the execution will stop. On the other hand, optional parameters can be left blank however, default values are usually set by the framework.

Arguments can have several input types. The input field type is chosen by the framework according to the accepted argument types.

If the argument type is `string`, an editable text box will be shown.

If the input type is `number` (*int, float, positive integer, float between -1 and 1, etc.*), the node will show a string input field, but the entered value will be checked against the format of the required numerical value.

If the input type is a `logical` value (*true or false*) a checkbox will be shown.

If the input type is `not defined`, the argument cannot be specified through the interface and "`not supported`" message will be shown instead.

If a greenish rectangle is shown before the argument name, the argument can be specified by using data `binding`(s) and can be connected to other nodes' output.

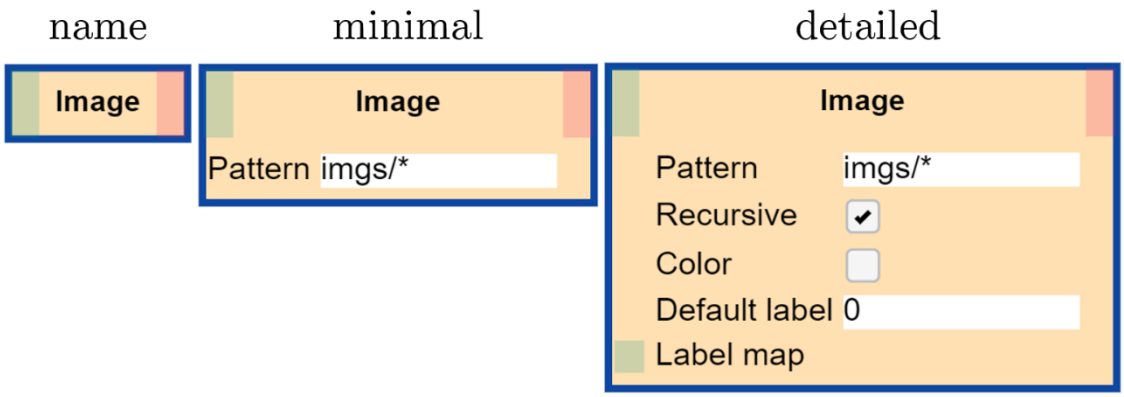

Figure 6: Node views: name, minimal and detailed.

Visual node elements have three view options which can be cyclically changed by double click the node's name.

Name: only node's name will be visible

Minimal: only node's name and the required arguments' list will be visible

Detailed: all option will be shown, including node's name and all the arguments

2.2.2 Data flow

Node inputs, node outputs and argument inputs can be connected with each other via edges. Port's output types are predefined in the framework configuration as well as input binding ports types or list of types which will be accepted from output ports (*i.e. node A's output type must be accepted as an input type at node B's input port*). On binding port connection attempts, the input and output ports' type compatibility is checked and the two ports can be connected only if they are type compatible. This mechanism ensures that during the execution, no error will be thrown because of type errors.

Both input and output ports can have multiple connections. In that case, output values will be copied and passed to input ports separately and multiple input values will be joined and passed as input altogether.

As the splitted outputs are completely independents and do not have any side effects on each other, multiple data processing algorithm can be applied to the same dataset at the same time (*in the same model execution*). This behavior gives the opportunity to easily generate different benchmark datasets or apply different transformation functions to the same data as a conversion.

Input files will be loaded (*from the disk*) only if they are passed to at least one attack. (*If the file is not used, it won't be loaded to save memory.*)

2.2.3 Model operations

For the mentioned interface elements, see Figure 2 on page 11.

Model loading and saving

The web application gives the possibility to save and load models to file. With the "Model name" input field, a name can be assigned to the model (*will be stored in filename*) and the model can be exported (*downloaded*) to the client machine by pressing the "Save model" button.

Exported modes can be loaded (*uploaded*) by pressing the "Load model" button on the header bar. The model's name will be automatically set (*from the filename*).

Model execution

Model execution can be started and stopped with the "Start/Stop execution" button on the header bar. The button icon is always referring to the action will be made in case of the user clicks. Diagram editing is disabled during execution.

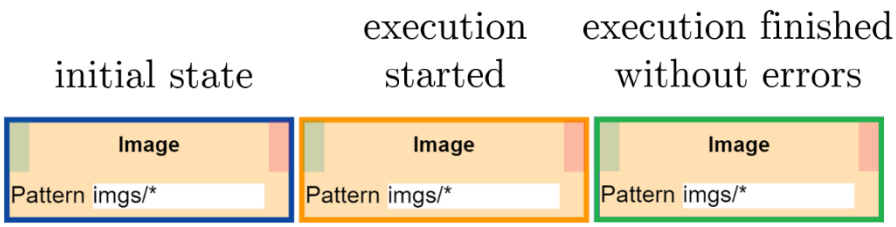

Figure 7: Node elements execution states.

The execution progress is tracked by the *FlowPy* framework and every progress change can be followed on the diagram. In the initial state, every node has dark blue border. If the server starts to evaluate a specific node, its border color changes to orange. If the evaluation finished without error, the node's border changes to green. If error has been thrown, the execution stops, and node which has the error will be still indicated with orange border.

Console

Standard output and error window can be opened and closed with the "Toggle console window" button on the header bar. The standard output and error streams are captured and displayed on their own tabs separately. Console tabs can be changed by clicking on the stream's name. If the user starts a new model execution, all the previous console data will be deleted.

2.3 Node descriptions

The listed elements in the web application can be described with their arguments and input/output types. Some of the methods use third-party functions from *Foolbox* package [4], see their documentation on *Foolbox*'s documentation [4] page.

Inputs

`Image`
`path`: *string* (regular expression path, relative to the working directory)
`recursive`: *bool* (recursive file search)
`color`: *bool* (if `true`, the image will be loaded with three channel – `RGB`)
`default_label`: *bool* (label to be assigned to all loaded images if *Labelmap* is not defined or the label map does not contain the given label)
`label_map`: *Labelmap* (label map binding)
Input: –
Outputs: *ImageBatch*
Loads images which paths are matching with the path argument's regular expression. For the attack functions, the original image labels must be known which can be defined with the `default_label` or the `label_map` arguments.

`Batch`
Input: *ImageBatch* (multiple)
Outputs: *ImageBatch*
Identity element for organizing data flow.

`Transformation`
`subtract`: *float* (number to be subtracted, first operation)
`divide`: *float* (number to be divided with, second operation)
`multiple`: *float* (number to be multiplied with, third operation)
Input: *ImageBatch* (multiple)
Outputs: *ImageBatch*
Subtract the given value then divide and multiply the input with the given numbers.

`Rename`
`prefix`: *string* (*string* to be prepended)
`suffix`: *string* (*string* to be appended)
Input: *ImageBatch* (multiple)
Outputs: *ImageBatch*
Prepend and append the given strings to filename.

`LabelMap`
`path`: *string* (path to the label map file)
Input: –
Outputs: *ImageBatch*
Loads and parse the specified label map file. The file structure must be ‘‘`<filename> <label>`’’ in each row.

`KerasModel`
`path`: *string* (path to *Keras* model file)
`bounds`: *bound* (minimum and maximum values which the model accepts)
`channel_axis`: *int* (color channel axis)
`predicts`: *string* (can be “`logits`” and “`probabilities`”, see *Foolbox* docs [4])
Input: –
Outputs: *FoolboxModel*

Outputs

`Output`
`path`: *string* (path to the output directory)
Input: *ImageBatch* (multiple)
Outputs: *ImageBatch*
Saves the input images (from input port) to the specified folder.

Attacks

`Configuration`
`model`: *FoolboxModel*
`criterion`: *Criteria*
`distance`: *Distance*
Input: –
Outputs: *Configuration*

Each attack has the following common structure:

`<Attacks>`
`configuration`: *Configuration*
Input: *Configuration*
Outputs: *ImageBatch*
The other arguments are different in each attack methods. See *Foolbox* documentation [4] for the individual arguments.

Criterion

Each criterion has the following common structure:

`<Criterion>`
Input: –
Outputs: *Criteria*
The arguments are different in each criterion. See *Foolbox* documentation [4] for the individual arguments.

Distances

Each distance has the following common structure:

`<Distances>`
Input: –
Outputs: *Distance*
See *Foolbox* documentation [4] for more information.

Utils

`bound`
`lower`: *float*
`upper`: *float*
Input: –
Outputs: bound (int, int tuple)
Constructs a bound object which can be passed to Keras models.

`epsilon`
`epsilon`: *float*
Input: –
Outputs: *epsilon* (*float*)
Constructs an epsilon value.

3 Developer documentation

Installation

FlowPy was packed to behave as a simple global python module. It has its own setup script (`setup.py`) which downloads and installs all the required third-party libraries; moreover, it also sets *FlowPy* as a global module. Installation can be easily done with the help of *pip3* python package-manager.

Installation commands (*as administrator*):

Windows:

```
pip install setuptools requests
pip install path/to/the/directory/flowpy
```

Linux:

```
sudo pip3 install setuptools requests
sudo pip3 install path/to/the/directory/flowpy
```

Usage

The global module can be run anywhere on the system with the python interpreter (*indicated with the `"-m"` module flag*). By default, the execution location will be the working directory (*if "-w" is not specified*).

Running from console/terminal:

```
python -m flowpy
or
python3 -m flowpy
```

Uninstall

The installed module can also be removed with *pip* package-manager:

```
pip uninstall flowpy
or
pip3 uninstall flowpy
```

3.1 Overview

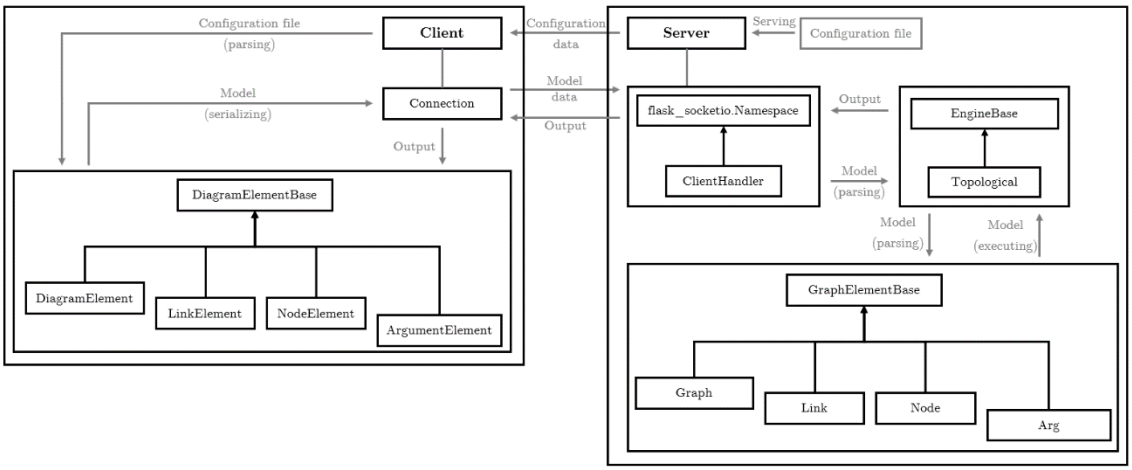

Figure 8: The structure of the whole application ("flowpy"). For more detailed structure description, see 3.2 (Server) and 3.3 (Client) sections.

The whole application has only one fix-structured element: the *Configuration file*, which describes how the supported (*i.e. described in the framework*) python classes and functions can be instantiated or called automatically. One of the most important principle during the development was to create a fairly modular structure, keeping the possibility for future extensibility. Program components (*diagram and graph elements, execution strategies, etc.*) can be easily modified and extended with extra functionalities without breaking the original application structure (*see Figure 8*).

3.2 Server ⇄ Backend

The server-side code was written in python programming language (*version 3.6*). The final application can be separated into two main parts: framework and diagram components where the *FlowPy* framework provides interface to the adversarial input generator functions (*see 3.2.5 below*).

In the final server-side code, the following third-party resources were used:

- Flask[1] (*BSD license*)
- Flask SocketIO[2] (*MIT license*)
- Waitress[3] (*ZPL license*)
- Foolbox[4] (*MIT license*)

3.2.1 Server

The server can be started (configured) with the following parameters and flags:

Command line arguments
flowpy.py [-h] [-b host] [-p port] [-w workdir] [-c config] [-m module] [-d]

`-h, --help`	Shows the help message
`-b host, --bind host`	Bind the server to a specific interface (*default*: `"0.0.0.0"`)
`-p port, --port port`	Server port to listen on (*default*: `"80"`)
`-w workdir, --workdir workdir`	Working directory. (*default*: `inherited from terminal`)
`-c config, --config config`	Configuration file path (*default*: `"lib/data/data.json"`)
`-m module, --module module`	Fallback module directory (*default*: `"lib/data"`)
`-d, --debug`	Start in debug mode (*default*: `false`)

[1] http://flask.pocoo.org
[2] https://flask-socketio.readthedocs.io
[3] https://docs.pylonsproject.org/projects/waitress
[4] https://foolbox.readthedocs.io

Request handling functions were written in *Web Server Gateway Interface* (*WSGI*) format. *WSGI* format is used by multiple server-side python frameworks. Depending on the usage intention, different *WSGI* backend can be used for dealing with the actual requests. In development mode, *Flask* package [5] is used as the implementation of the *WSGI* environment due to its extensive debugging possibilities however, for final usage *Waitress* server [6] is preferred as it was written for production providing more stable environment. In both cases, the *WebSocket* connection is supported by long-polling technique within *WSGI*.

3.2.2 Model execution

When a new client connects to the server via the *WebSocket* (*WS*) API, the new session will be marked and stored with the *WS* session id.

The actual server-side model parsing, building and execution is confined to the `"lib/backend/execute.py"` file when run as a separated process with the specified arguments (*see below*).

execute.py
lib/backend/execute.py

Arguments

`-e escape, --escape escape`	Metadata escape sequence. (*default*: `"::::::::"`)
`-w workdir, --workdir workdir`	Working directory. (*default*: `inherited from terminal`)
`-m module, --module module`	Fallback module directory. (*default*: `"lib/data"`)

Fallback module is used when the given (*node*) package cannot be imported as an installed python package. If the requested class or function cannot be found neither in the fallback module directory, error will be thrown.

With its event handler functions (*callback, progress*), it also mixes metadata into the standard error stream starting with the specified escape sequence (*see below*).

Mixed in metadata

`"progress <nodeid> <progress> <time>"`
where
`nodeid`: the id of a node (if `None`, the progress belongs to the whole model)
`progress`: type of the progress (can be `"progress"` or `"finished"`)
`time`: elapsed time during the progress (in milliseconds)

`"callback <progress> <time>"`
where
`progress`: type of the progress (can be `"progress"` or `"finished"`)
`time`: total elapsed time (in milliseconds)

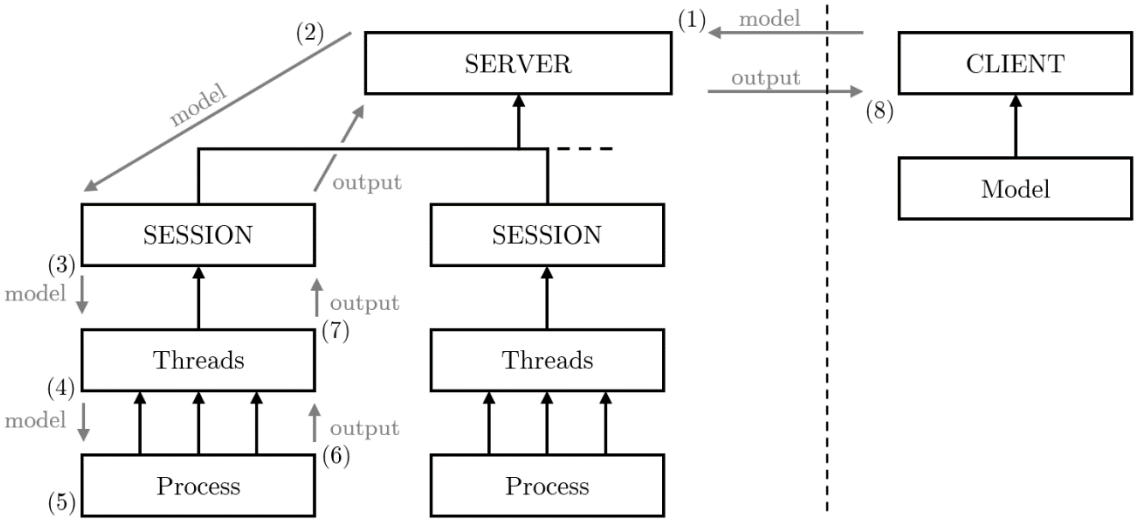

Figure 9: The server can handle multiple separated client connections and model executions at the same time managed by WebSocket's session ids.

(1) When a user presses the run button in the browser, the client-side model object will be serialized and sent to the server.

(2) The WS server assigns the session id to the call and looks up the stored process (*if exists*).

(3) The server starts new threads in order to asynchronously spawn a new execution process and watch its output streams. One thread for managing the process and additional two to capture the standard output and error stream.

(4) The "spawner" thread spawns a new process and passes the serialized model to it so it will run in a different environment.

(5) The execution process parses the serialized model got on the standard input and build a graph (*similar than it was on the client side*), compute the nodes' execution order with the given engine (*see Topological in 3.2.3 below*) and starts to evaluate the nodes (*see execute function in 3.2.3 below*).

(6) The management threads watch the execution process and capture its outputs and events (*mixed-in metadata is sorted out from the stream by with the help of an escape sequence, see ClientHandler in 3.2.3 below*)

(7) Stream listeners send the captured output to the specified clients. (*see 3.2.4 below*)

(8) The client receives and process the output/meta data and update its state.

3.2.3 Classes

Server-side classes can be separated for graph- and execution-connected classes based on their functionality. Graph elements are representing a model structure which come from clients (see 3.4.2). Execution (engine) classes use the parsed graph structure, which was built from the client-side model object, to evaluate the user-built program.

Most of the graph element classes are logically equal to the client-side diagram element classes. For further information about the stored model properties in graphs see the model object structure in 3.3.3.

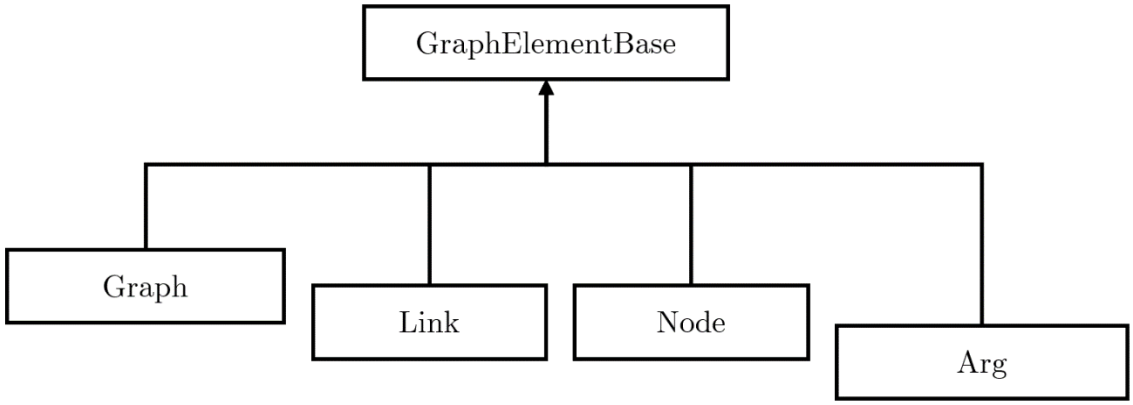

Figure 10: Graph elements implement a graph structure on the server side in order to build and execute the model by evaluating the nodes in the proper order.

GraphElementBase

GraphElementBase is a superclass for all the graph elements on the server side. It contains functions for setting and getting class attributes, which used in the child classes. As it is a base class, should not be instantiated.

GraphElementBase

lib/backend/GraphElement.py

`__init__()`
Base class, should be not instantiated.

Properties

–

Methods

`setAttr(key, val)`
`key`: *string* (attribute key)
`val`: *any* (attribute value)
Returns: *void*
Sets the given value to the given attribute (name).

`getAttr(key, fb)`
`key`: *string*
`fb`: *any* (fallback value, *default*: `None`)
Returns: *any* (the value of the attribute)
Returns with the requested attribute or with the fallback value if the attribute does not exist.

`delAttr(key)`
`key`: *string*
Returns: *void*
Delete the given attribute.

Arg

Arg(*GraphElementBase*) class represents an argument of a function.

Arg
lib/backend/GraphElemet.py

`__init__(arg_data)`
`arg_data`: *argDescriptor* (*see Argument section in 3.4.1*)
Parse the given `arg_data` and sets its object attributes.

Properties

The properties (`arg`, `name`, `itype`, `value`, `conns`, `required`) has same meaning, type and function as defined in the configuration file (*see Argument in 3.4.1*).

Methods

`typarr(types)`
`types`: *string* or *list of strings* (argument type names)
Returns: *list of strings*
Ensure that the return value is a list of strings (if the input is not a list, converts it).

`identity(el)`
`el`: *any*
Returns: *any*
Identity function (for consistent data parsing).

`arglist(arglist)`
`arglist`: *list of argDescriptors* (*see Argument section in 3.4.1*)
Returns: *list of Arg objects* (constructed from `arglist`)
Iterate through and parse `arglist` by calling the *Arg* constructor for all items.

`intinf(val)`
`val`: *int* or *float* or *string* or *math.inf*
Returns: *int* or *math.inf*
Checks if the parameter is infinity. If not, converts to integer.

`conversion(val, typarr)`
`val`: any
`typarr`: *list of strings* (type names)
Returns: *one of typarr typed value*
Tries to convert `val` to the given types defined in `typarr`. If cannot, returns with the original value.

Node

Node(*GraphElementBase*) represent a python function/class/static variable.

Node

lib/backend/GraphElement.py

`__init__(node_data)`
`node_data`: *nodeDescriptor* (*see Node section in 3.4.1 and 3.4.2*)
Parse the given `node_data` and sets its object attributes.

Properties

The properties (`id`, `$$`, `name`, `package`, `args`, `itype`, `otype`, `instance`, apply, `copy`, `conns`) has same meaning, type and function as defined in the configuration file (*see Node in 3.4.1 and 3.4.2*).

Methods

`package(pckg)`
`pckg`: *string* (fully qualified name)
Returns: *tuple of strings* (package name, callable name)

Link

Link(*GraphElementBase*) represent an edge in the graph between two nodes.

Link
lib/backend/GraphElement.py

`__init__(node_data)`
node_data: nodeDescriptor (see Node section in 3.4.1 and 3.4.2)
Parse the given node_data and sets its object attributes.

Properties

`_id_auto_increase`
Type: *int*
Auto incremented id for the next link constructed (in order to unique link identification)

The properties (`from_node`, `from_port`, `to_node`, `to_port`) has same meaning, type and function as defined in the configuration file (*see Link in 3.4.2*).

Methods

–

Graph

Graph(*GraphElementBase*) represents the entire user-created model. It consists of the parsed nodes and links. In the execution phase, the engine classes will use this structure to compute the evaluation order, store the return values and track the progress.

Graph

lib/backend/GraphElement.py

`__init__(graph)`

`graph`: model object (*see 3.4.2*)

Parse the given model object, constructs and connects the nodes and links.

Properties

`nodes`

Type: *dict of Nodes*

Stores all the nodes of the model with the key of the node id.

`links`

Type: *dict of Links*

Stores all the connection between nodes with the key of the link id.

Methods

–

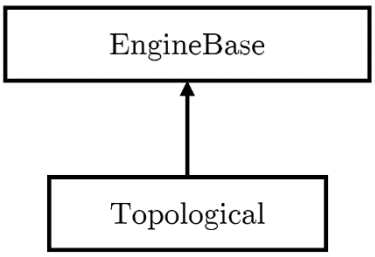

Figure 11: Different evaluation techniques can be implemented by creating a custom engine (class) which inherits from EngineBase. Currently, only Topological evaluation was defined as it is the most suitable for most of the programs.

EngineBase

EngineBase is a superclass for execution engines. It contains static helper functions for importing packages, flushing standard output, cloning objects, preparing function/class arguments and evaluating graph nodes. As it is a base class, should not be instantiated.

EngineBase

lib/backend/Engines.py

`__init__(G)`

`G`: *Graph*

Sets the `G` property and initialize empty the order.

Properties

`G`

Type: *Graph*

Inner *Graph* store.

`order`

Type: *list*

Stores the nodes execution order (computed by the engine classes).

Methods

`EngineBase.importClass(pckg_name, cls_name, clsFallback)`

`pckg_name`: *string* (name of the package)

`cls_name`: *string* (name of the required element in the package)

`clsFallback`: *string* (fallback module in case of the requested does not exists)

Returns: (*imported*) function/class/variable

Imports the requested python function/class/variable from the given package. If the element or the package does not exist, the fallback module

will be used instead (*if the member cannot be found in the fallback module, error is thrown, and the execution stops*).

`EngineBase.flush()`
Returns: *void*
Flushes the standard output and standard error.

`EngineBase.copy(obj)`
`obj`: *object*
Returns: *object* (copy of the object)
Tries to "deep copy" the object. If deep copy is not possible returns with a reference copy.

`prepareArguments(node)`
`node`: Node
Returns: any tuple (with three items: `input`, `required`, `kwargs`)
Separates node arguments to three groups:
`input`: the main input of the node (*see input port on Figure 16*)
`required`: required arguments (with required "flag")
`kwargs`: any other arguments
If an argument has multiple binding (*connected edges*), their values will be passed as a list.

`evaluate(node, clsFallback)`
`node`: *Node*
`clsFallback`: *string* (fallback module in case of the requested does not exists)
Returns: *void*
Import the required python element, prepare the node's arguments, capture the return value and assign is to the node. If any error happens during execution, error will be thrown, and the process will stop.

Topological

Topological(*EngineBase*) is a specialized execution engine. The evaluation order will be one of the graph's topological order. This engine is only applicable for directed acyclic graphs (DAGs), otherwise error will be thrown.

Topological

lib/backend/Engins.py

`__init__(G)`
`G`: *Graph*
Sets the `G` property and the evaluation order.

Properties

–

Methods

`Topological.getOrder(G)`
G: *Graph*
Returns: *list of strings* (*Node* ids)
Returns the topological order of the graph as the list of node ids in the proper order (*if it exists*).

`Topological.DFSorder(G, root, order)`
`G`: *Graph*
`root`: *Node*
`order`: *list*
Returns: *void*
Recursive depth-first search algorithm to get the topological order.

`run(clsFallback, progress)`
`clsFallback`: *string* (fallback module in case of the requested does not exists)
`progress`: *function* or *None*
Returns: *void*
Execute the whole model following the preset evaluation order. Progress function will be called right before and after node evaluation with the node in progress, progress type (can be `"progress"` or `"finished"`) and the elapsed time in milliseconds.

`execute`

execute is a function for executing models wrapped into an engine.

`execute(engine, clsFallback, progress, callback)`
`engine`: subclass of EngineBase
`clsFallback`: *string* (fallback module in case of the requested does not exists)
`progress`: *function* or *None* (*see Topological section above*)
`callback`: *function* or *None*
Execute the model with the given engine. Callback function will be called at the beginning and at the end of the model execution with the engine, progress type (can be `"progress"` or `"finished"`) and the total elapsed time in milliseconds.

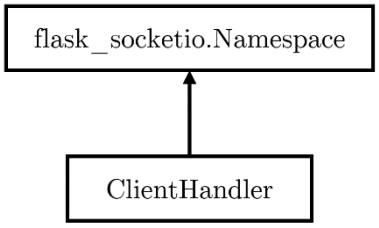

Figure 12: ClientHandler class extends a SocketIO namespace (in the actual implementation: root namespace) in order to keep the business logic encapsulated.

ClientHandler

ClientHandler(*flask_socketio.Namespace*) is a *WebSocket* (*WS*) handler class for Flask SocketIO package [7]. Its methods are named after *WS* event names (*see 3.2.4 below*).

ClientHandler

lib/backend/Socket.py

`__init__(namespace, workdir, module)`
`namespace`: *string* or *None* (*WS* server namespace)
`workdir`: *string* (working directory)
`module`: *string* (fallback module directory, *see EngineBase.importClass above*)
Initialize its properties.

Properties

`WORKDIR`
Type: *string*
Working directory for the execution process (inherited from terminal by default)

`MODULE`
Type: *string*
Fallback module directory for user-defined classes and functions.

`STREAM_ESCAPE_SEQUENCE`
Type: *string*
Binary sequence which will be used as an escape sequence when processing captured subprocess' outputs. If an output line starts with the escape sequence, it means that line contains metadata for progress notification.

`PROCS`
Type: *dict of ints*
Registry for session processes where the key is the WS session id and the value is (operating system's) process id.

Methods

`on_connect()`
Returns: *void*
Event. Called when new client connects to the server and registers its session id.

`on_disconnect()`
Returns: *void*
Event. Called when a client disconnects and delete its process and session.

`on_cmd_run(data)`
`data`: *object*
Returns: *void*
Event. Called when a client starts the model execution (*see 3.2.4 below*). Parse the received model, spawns execution process and listener threads.

`on_cmd_stop(data)`
`data`: *object*
Returns: *void*
Event. Called when a client stops the model execution (*see 3.2.4 below*). Kill the client's ongoing execution process.

`_notify(SID, event, status, message, state, data)`
`SID`: *string* (session id)
`event`: *string* (event name)
`status`: *int* (status code)
`message`: *string* (response description)
`state`: *string* (event state)
`data`: *dict* (data to send)
Returns: *void*
Send the specified data (`data`) to the specified client (`SID`) in a structured format (*see 3.2.4 below*).

`_on_spawn(SID)`
`SID`: *string* (session id)
Returns: *void*
Inner event for process spawn acknowledgement.

`_on_exit(SID)`
`SID`: *string* (session id)
Returns: *void*
Inner event for process termination acknowledgement.

`_process_wrapper(SID, stdin_data, on_spawn, on_exit, stdout_handler, stderr_handler, *popen_args, **popen_kwargs)`

`SID`: *string* (session id)
`stdin_data`: *string* (standard input data)
`on_spawn`: *function* (callback function after process spawn)
`on_exit`: *function* (callback function after process termination)
`stdout_handler`: *function*
`stderr_handler`: *function*
`*popen_args`: *list of any* (`subprocess.Popen` options)
`**popen_kwargs`: *dict* (`subprocess.Popen` options)
Returns: *void*
Create an asynchronous execution process in a separated thread.

`_stream_handler(SID, stream, label)`

`SID`: *string* (session id)
`stream`: *stream* object (e.g.: `stdout`)
`label`: *string* (stream label, e.g.: `"stdout"`)
Returns: *void*
Capture the stream output coming from the given stream and send it to the specified client (`SID`).

`_kill(SID)`

`SID`: *string* (session id)
Returns: *void*
Kills the specified client's process (*if exists*).

3.2.4 WebSocket API

WebSocket is a communication protocol, providing persistent bidirectional connection between server and client. The technology is widely supported among browsers and can provide real time data connection. (*If the protocol is not supported, "long-polling" technique will be used instead.*)

The *WebSocket* connection is session based, so in case of connection lost during execution, that client's execution progress cannot be restored.

Events

Accepted events

The required properties for the calls are listed after the event name.

`cmd_run(graph)`
: `graph`: *Model* object
Starts the execution of the given model. Send `"state_change"` event after the operation.

`cmd_stop()`
: Stops the execution of the running model. Send `"state_change"` event after the operation.

Sent events

All the server-sent events have a unified structure and contain the following properties:
`status`: *int* (`200`: success, `500`: server error)
`state`: *string* (state of the notification type, varies among different events, see below)
`message`: *string* (event description)
`data`: *any* (additional information to pass)
The sent properties are listed after the event name.

`state_change(state)`
: `state`: *string* (`"running"`: the spawned process is running without any problem `"stopped"`: the spawned process is terminated)
Informs about the state of the spawned execution process.

`progress_change(state, { node, state, time })`
: `state`: *string* (`"progress"`: model evaluation in progress, `"finished"`: model evaluation finished successfully)
`data.node`: *int* or *None* (node id if related to a specific node or *None* if the information is about the whole model)

`data.state`: *string* (`"progress"`: evaluation started, `"finished"`: evaluation finished successfully)
`data.time`: *float* (evaluation time in milliseconds)

`console_stream(state, { label, line })`
`state`: *string* (`"connected"`)
`data.label`: *string* (name of the standard output)
`data.line`: *string* (printed line by the execution process)

3.2.5 Adversarial wrappers

Adversarial input generator functions are part of the base *FlowPy* module which are defined with the help of the configuration file and custom modules (in `data/adv` folder).

"Adversarial" functions and wrappers were placed to "lib/data/adv" folder. The functions were separated into three files: "`fileop.py`", "`attacks.py`" and "`utils.py`".

fileop.py
This module contains functions and classes related to file operations: loading, transforming and saving data files.
For further information, see the comments in code.

Main classes

`BatchItem`
Wraps and manage an input data entity.
Can load, save and transform the data.

`Batch`
Data array wrapper class.
Can load and join data batches consist of *BatchItems*.

`Image(BatchItem)`
Specialized *BatchItem* class which can load, store and save image data.

attacks.py

This module contains adapter functions for Foolbox's attack function set [4]*. For further information, see the comments in code.*

Main functions

`Configuration(model, criterion, distance)`
`model`: *FoolboxModel* (*Foolbox*-wrapped neural network model [4])
`criterion`: *Criteria* (success criteria for the attack)
`distance`: *Distance*
Constructs a three-element tuple with the given parameters. This tuple will be passed to attacks as the "Configuration" of the attack.

`_Attack(*batch, attack, **kwargs)`
`batch`: list of *Batches* and the configuration tuple (as the last element)
`attack`: `foolbox.attacks.*` (*Foolbox* attack function [4])
`kwargs`: *dict* (parameters to be set for the attack)
Apply the specified attack with the given parameters to the given *Batches* in batches.

utils.py

This module contains utility functions and custom data types constructors which are neither file operations nor attack related functions. For further information, see the comments in code.

3.3 Client ⇆ Frontend

The client side of the application was written for web browsers as it is easily customizable, and it is easier to quickly implement a better-looking user interface on the web than in a natively supported programming language.

In the final client-side code, the following third-party resources were used:

- Go.js[5] (*Academic license*)
- Socket.io[6] (*MIT license*)
- Google Material Design Icon font[7] (*Apache license 2.0*)

3.3.1 Structure

File tree

All the frontend-connected files are in `"lib/static/"` directory which is served as a static directory on the root location. The user interface was created with *HTML* (*Hypertext Markup Language*) + *CSS* (*Cascading Style Sheets*) + *JavaScript*.

The source of the main page was placed at `"lib/static/index.html"` which served on the root location by default. The other resources are separated into distinct directories: `"css"`, `"js"`. (The `"js"` directory has a `"class"` subdirectory which contains all the *JavaScript* classes created in the context of the thesis work.)

[5] https://gojs.net
[6] https://socket.io
[7] https://material.io/tools/icons

Page

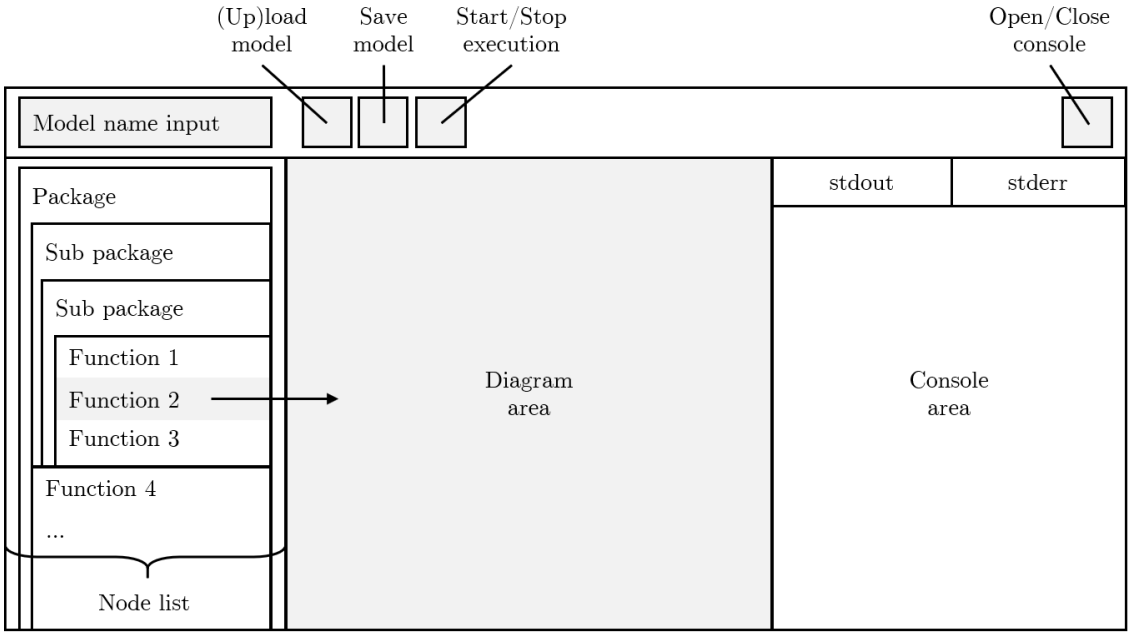

Figure 13: The frame of the main page. It can be easily modified or replaced by changing the HTML parameters in the CONFIG object.

The structure was created with the help of a small self-made *CSS* "flexbox" library (`lib/static/css/layout.css`) which greatly simplify markup code.

3.3.2 Initialization

The "main" code is placed in `"lib/static/js/main.js"` file and most of the page initialization happens there.

The `CONFIG` object wraps all the parameters required for setting up the user interface. If the *HTML* structure changes, this object's properties should be changed respectively. (*further information can be found as comments in the code*)

The initialization has four main steps:

1) set global constants (constant variables with capitalized names)
2) connect to the server by the *Connection* class (*see Connection section in 3.3.3*).
3) download, parse and share all necessary data (*the configuration file located at* `CONFIG.DS_PATH`) among components, create the *DiagramElement* class and insert further *HTML* elements (node list, diagram) into the interface (`init` and `render_node_tree` functions)

4) setting up all the user interaction handler functions (load, save, run, consoles)

Further information can be found as comments in the code and the used global functions are described below.

Global functions

`_(selector, all, root)`
`selector`: *string* (*HTML* selector)
`all`: *bool* (select all match)
`root`: *string* or *HTMLElement* (root node for search)
Returns: *HTMLElement* or array of *HTMLElements* or *null*
Wrapper and shortcut function for `HTMLElement.querySelector[All]`.

`init(CONFIG)`
`CONFIG`: *object*
Returns: *void*
Downloads and parses the configuration file, initialize shared data (*DiagramElementBase*) and sets event handlers for dragging node elements from side list to the diagram area.

`render_node_tree_element($$, data)`
`$$`: *string* (selector)
`data`: *object* (node descriptor)
Returns: generated *HTMLElement* node, used the given node descriptor data
If the data object is not a node descriptor the function will return with null.

`render_node_tree(node, name)`
`node`: *object* (configuration object)
`name`: *string* (node name)
Returns: generated *HTMLElement* list, used the given configuration object
Recursive function in configuration file's logical packaging. Calls `render_node_tree_element` if has found a node descriptor in the configuration object (*tree*).

`console_clear(name)`
`name`: *string* (console name)
Returns: *void*
Clears the specified console content. This is not connected to the console of the browser but the captured outputs of the server-side python programs.
If the name parameter is not provided all the consoles will be cleared which are listed in `CONFIG` object.

3.3.3 Classes

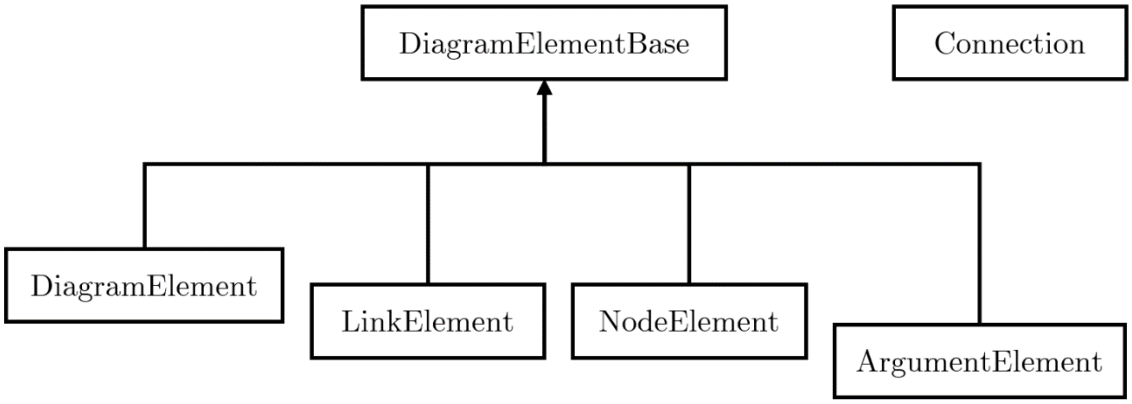

*Figure 14: The *Element classes implement an editable graph by sharing global configuration data in their parent's static properties while the Connection establish real time communication between the diagram and the server.*

DiagramElementBase

DiagramElementBase is a superclass for all the graph elements on client side. It contains user interface settings, configuration data, template builder functions and helper functions for set static class variable in *JavaScript*. All its static variables and methods will inherit to the child classes by reference so they can use the same pre-collected data and function set without further initialization. As the class just for share data and functionality, it should not be instantiated.

DiagramElementBase
lib/js/class/DiagramElementBase.js

`constructor()`
Base class, should not be instantiated.

Properties

`DiagramElementBase.$`
Type: *function*
Go.js template builder function: go.GraphObject.make. (*see: Go.js docs* [8])

`DiagramElementBase._data`
Type: *object*
The whole parsed configuration file. (*see: 3.4.1 below*)

`DiagramElementBase.config`
Type: *object*
Contains parameters about the user interface. (e.g.: *HTML* element ids, dataset locations, *Go.js* [8] settings)

`DiagramElementBase._deferredStaticAssigns`
Type: *array*
Stores deferred static assign requests.
`DiagramElementBase.flushDeferredStaticAssigns()` function call will execute all the stored request and clear the array.

Methods

`DiagramElementBase.init(data, make, config)`
`data`: *object* (configuration object)
`make`: *function* (template builder)
`config`: *object* (configuration object)
Returns: *void*
Initialize the shared static variables: parsed configuration object (`data`), template builder function (`make`) and user interface configuration (`config`).

`DiagramElementBase.assignStatic(context, generator)`
`context`: *object* or *callable*
`generator`: *function*
Returns: *void*
Assigns static variables, defined in the generator function, to the given context. (All the variable set with `"this.*"` in the generator will be assigned.)

`DiagramElementBase.assignStaticDeferred(context, generator)`
`context`: *object* or *callable*
`generator`: *function*
Returns: *void*
Stores the (`context`, `generator`) pairs in `DiagramElementBase._deferredStaticAssigns` for deferred static assigns. Useful is static methods and variables are built on each other.

`DiagramElementBase.flushDeferredStaticAssigns()`
`context`: *object* or *callable*
`generator`: *function*
Returns: *void*
It process all the deferred static assign stored in `DiagramElementBase._deferredStaticAssigns` by calling `DiagramElementBase.assignStatic` for each.

ArgumentElement

ArgumentElement(*DiagramElementBase*) is a helper class for node argument processing and styling. Its purpose is to separate the type system/validation and

argument field styling from node-level logic and make the code more readable. It contains only static function and properties, should not be instantiated.

ArgumentElement

lib/js/class/ArgumentElement.js

`constructor()`
Static share, should not be instantiated.

Properties

`ArgumentElement.style`
Type: *object*
Contains styling information for arguments.
First level keys (namespaces): `type`

`ArgumentElement.templates`
Type: *object*
Contains *Go.js* [8] template objects for argument fields.
First level keys (namespaces): `fields`

Methods

`ArgumentElement.typeValidator(type)`
`type`: *string* (type name)
Returns: a function which requires one `"value"` parameter and will validate it according to the predefined `"type"`. (will call `ArgumentElement.validate`)

`ArgumentElement.validate(type, value)`
`type`: *string* (type name)
`value`: *any*
Returns: true if the value is valid by the given type or the type is unknown, false otherwise (will call the proper `ArgumentElement.validate<Type>(value)` function)

`ArgumentElement.validate<Type>(value)`
`type`: *any*
Returns: true if the value is valid by the given type, false otherwise
Types are:
`bool`, `string`, `int`, `int0p`, `intp`, `int0n`, `intn`, `float`, `float0p`, `floatp`, `float0n`, `floatn`, `float01`, `float11`, `shape`
The ending `"p"`/`"n"` means positive/negative value. `"0"` before `"p"`/`"n"` means `0` is included. Ending `"01"` means between `0` and `1`, `"11"` means between `-1` and `1`.

NodeElement

NodeElement(*DiagramElementBase*) is the most important class from all the diagram elements. It collects the required information from the parsed configuration file in order to create a representation of the instantiated node logically as well as visually.

NodeElement

lib/js/class/NodeElement.js

`constructor($$)`

`$$`: *string* (selector)

Creates a node object by getting all the node, argument, category and type information from the configuration object. `$$` is a selector string which tells the node location in the configuration object. (e.g.: `nodes.utils.bound` or `nodes.Input.Data.Image`)

Properties

`$$`

Type: *string*

Configuration selector (in configuration object).

Same as the constructor gets.

`name`

Type: *string*

Name of the node.

Same as the node name in 3.4.1 below. (*last part of $$, separated by dots*)

`nodeDescriptor`

Type: *object*

Reference to the selected node object in the configuration object.

`categoryDescriptor`

Type: *object*

Reference to the descriptor on the selected node's category.

See *Category* section in 3.4.1 below.

`argDescriptor`

Type: *array of objects*

Array of processed argument descriptors from the node's descriptor.

See *Argument* section in 3.4.1 below.

`argDescriptorRequired`
Type: *array of objects*
Filtered array of references from *argDescriptor*, which only contains required arguments.

`package`
Type: *string*
See *Node* section in 3.4.1 below.

`itype`
Type: *string* or *array of strings*
Input filed type will be chosen based on this property (`unsupported`, `string`, `bool`).
See *Node* section in 3.4.1 below.

`otype`
Type: *string*
See *Node* section in 3.4.1 below.

`instance`
Type: *bool*
See *Node* section in 3.4.1 below.

`conns`
Type: *int*
Input type will be chosen based on this property (`value`, `bind`).
See *Node* section in 3.4.1 below.

`position`
Type: *object*
See *Model* section in 3.4.2 below.

`state`
Type: *string*
Execution state of the node. Possible values are:
`idle`: the node hasn't been executed
`progress`: the node execution started but hasn't been finished yet
`finished`: the node execution finished successfully

`view`
Type: *string*
See *Model* section in 3.4.2 below and Figure 15 at the end of *NodeElement* section.

`NodeElement.style`
Type: *object*
Contains styling information for nodes.
First level keys (namespaces): `category`, `text`, `highlight`, `port`, `descriptor`

`NodeElement.maps`
Type: *object*
Contains *Go.js* [8] template mappings for node visualization. Different model views have different templates, template mapping makes possible to switch automatically based on node properties (`view`, `conns`, `itype`).
First level keys (namespaces): `argTypeMap` (maps for argument field), `argRecordMap` (maps for argument record), `nodeTemplateMap` (maps for node view)
See Figure 16 at the end of *NodeElement* section.

`NodeElement.templates`
Type: *object*
Contains *Go.js* [8] template objects for nodes.
First level keys (namespaces): `parts`, `views`

Methods

`NodeElement.getNodeDescriptor($$)`
`$$`: *string* (selector)
Returns: reference to node descriptor (reference to the selected node in the configuration object) of the selected node, selected by `$$` node selector string
See *Node* section in 3.4.1 below.

`NodeElement.getCategoryDescriptor($$)`
`$$`: *string* (selector)
Returns: *object reference* to category descriptor (reference to the selected node's category descriptor in the configuration object) of the selected node's category, selected by `$$` node selector string
See *Category* section in 3.4.1 below.

`NodeElement.getArgDescriptor($$)`
`$$`: *string* (selector)
Returns: *object reference* to argument descriptor (reference to the selected node's argument descriptor in the configuration object) of the selected node, selected by `$$` node selector string
See *Argument* section in 3.4.1 below.

`NodeElement.getTypeDescriptor($$)`
`$$`: *string* (selector)
Returns: *object reference* to type descriptor (reference to the selected type descriptor in the configuration object) of the selected type, selected by `$$` type selector string
See *Type* section in 3.4.1 below.

`NodeElement.getNameFromPackage($$)`
`$$`: *string* (selector)
Returns: *string* (the last part of the `$$` selector)
If the selector contains packages, if will return the name after the last separator dot. If the selector does not contain any package (there is no dot in the selector string), it will return with `$$`.

`NodeElement.getNameFromArg(arg)`
`arg`: *string*
Returns: *string* (formatted argument name of the given name)
It will only change the `"snake_case"` convention by capitalize the first letter and replace the `"_"` characters with spaces.

`NodeElement.getNodeAttributePipe(attr)`
`attr`: *string* (attribute name)
Returns: a *function* which takes an arbitrary descriptor object and returns the given attribute if defined, otherwise returns undefined
(used in *Go.js* [8] template styling)

`NodeElement.changeNodeView(event, node)`
`event`: *go.js event object*
`node`: *NodeElement*
Returns: *void*
Event handler which change the node view type which called the event.
See *NodeElement.view* above.

`NodeElement.positionToPoint(position)`
`position`: *object* (has `x` and `y` properties)
Returns: *go.Point* object with the given `x` and `y` coordinates in the parameter
Converts object with `x`, `y` properties to *go.Point* object.

`NodeElement.pointToPosition(point)`
`point`: *go.Point*
Returns: object with `x` and `y` properties set from the *go.Point* object given as parameter
Converts *go.Point* object to plain *JavaScript* object with `x`, `y` properties.

`NodeElement.getStrokeColorFromState(state)`
`state`: *string* (node state)
Returns: the styling properties for the defined highlight stroke color
See *NodeElement.state* property above.

`NodeElement.getStrokeWidthFromState(state)`
`state`: *string* (node state)

Returns: the styling properties for the defined highlight stroke width
See *NodeElement.state* property above.

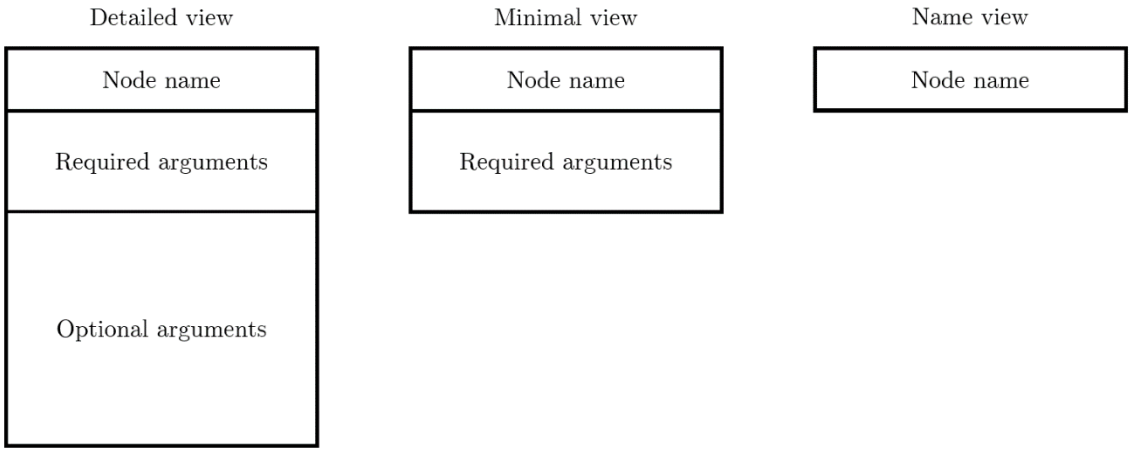

Figure 15: The rendered node element views. The template definitions are placed at "NodeElement.templates.maps.nodeTemplateMap" for views.

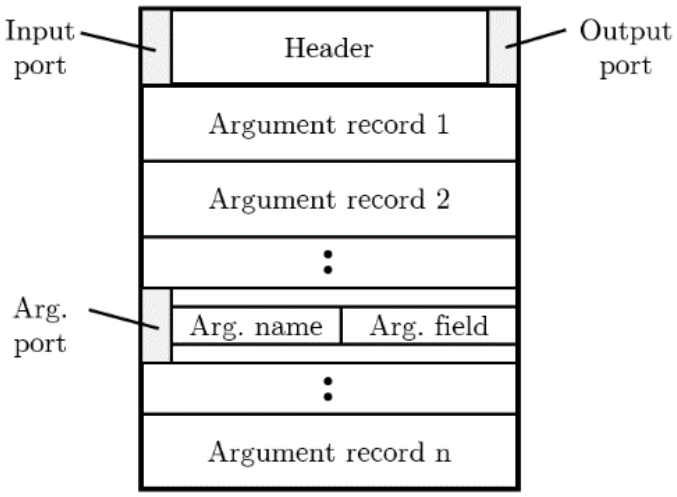

Figure 16: The rendered NodeElement inner structure. The template definitions are placed at "NodeElement.templates.maps.argRecordMap" for "Argument record" and "NodeElement.templates.maps.argTypeMap" for "Argument field".

LinkElement

LinkElement(*DiagramElementBase*) stores data for link validation and styling. It contains only static function and properties, should not be instantiated.

LinkElement

lib/js/class/LinkElement.js

`constructor()`

Static share, should not be instantiated.

Properties

`LinkElement.style`

Type: *object*

Contains styling information for links between nodes.

First level keys (namespaces): `link`

`LinkElement.templates`

Type: *object*

Contains *Go.js* [8] template objects for link between nodes.

First level keys (namespaces): `links`

Methods

`LinkElement.validate(fromNode, fromPort, toNode, toPort)`

`fromNode`: *string* (node key)

`fromPort`: *string* (node port name)

`toNode`: *string* (node key)

`toPort`: *string* (node port name)

Returns: *bool*

Returns `true` if the link connection allowed i.e. the output type of the start node (`fromNode`) is allowed at the input of the destination port (`toPort`).

DiagramElement

DiagramElement(*DiagramElementBase*) is an extended (*configured*) class for the *Go.js* [8] diagram object. Its constructor contains predefined settings for initializing *Go.js* diagrams and its instance will be a configured *Go.js* diagram object. The class also sets event listeners for accepting dragged and dropped node elements (node selector strings) making possible to creating nodes visually.

DiagramElement

lib/js/class/DiagramElement.js

`constructor(elementId, NodeElement, LinkElement)`
`elementId`: *string* (*HTML* id)
`NodeElement`: *class* (*NodeElement*)
`LinkElement`: *class* (*LinkElement*)
Requires the *HTML* element id, which the diagram will be rendered within, and the classes for node and link elements. Creates *go.diagram* object with the given settings in the constructor.

Properties

`DiagramElement.config`
Type: *object*
Contains settings for gs.js diagram initialization. (*see: Go.js docs* [8])

Methods

`loadModel(data)`
`data`: *object* (model)
Returns: *void*
Parse, construct and load the given model to *Go.js* [8] diagram. See 3.4.2 for the data object structure.

`highlightNode(key, state)`
`key`: *number* (node key)
`state`: *string* (node state)
Returns: *void*
Sets the state of the node with the given key. If the key is `null`, all the nodes' `state` will be set. For key value see: *Go.js* [8] docs, for possible state values see *NodeElement.state* section above.

Connection

Connection acts like a wrapper for server API and provide an event-subscription-based interface. It stores the server status, handles communication errors and calls the subscripted event handlers.

Connection

lib/static/js/class/Connection.js

`constructor(domain, port)`
`domain`: *string* (*default*: `document.domain`)
`port`: *number* or *string* (*default*: `location.port`)
Connects to the given domain on the given port.

Properties

`_eventElement`
Type: *HTMLElement*
Dummy *HTML* element which makes possible to efficiently register and dispatch events managed by *HTML DOM*.

`_backend`
Type: *object*
Stores information about server state.

`_backend.connected`
Type: *bool*
Indicates if the *WebSocket* connection is alive

`_backend.running`
Type: *bool*
Indicates if the execution is running

`socket`
Type: *socket object*
Socket.io socket object. (*see socket.io docs* [4])

Methods

`on(event, callback)`
`event`: *string* (event name)
`callback`: *function*
Returns: *void*
Register a new non-wrapped handler for the given event. The callback function first parameter will be the received data. (*see socket.io docs* [9])

`emit(event, data)`
`event`: *string* (event name)
`data`: *object* (data to send)
Returns: *void*
Emits an event to the wrapped socket identified by the event name. (*see socket.io docs* [9])

`addEventListener(event, handler, options)`
`event`: *string* (event name)
`handler`: *function*
`options`: *object*
Returns: *void*
Register a new handler for the given event.

`removeEventListener(event, handler, options)`
`event`: *string* (event name)
`handler`: *function*
`options`: *object*
Returns: *void*
Remove the given handler from the given event.

`_dispatchEvent(event, detail)`
`event`: *string* (event name)
`detail`: *any* (data to send)
Returns: *bool*
Dispatch an event with the given name and data.

`isConnected()`
Returns: *bool*
Returns with `_backend.connected` property.

`isRunning()`
Returns: *bool*
Returns with `_backed.running` property.

`start(model)`
`model`: *object* (model)
Returns: *bool*
Sends the given model to server and starts the execution if other execution is not running. Return `true` if the call was successful, `false` otherwise.

`stop()`
Returns: *bool*
Stops the execution is running. Return `true` if the call was successful, `false` otherwise.

3.3.4 Server connection

The connection with the server is established through *WebSocket*[8] connection with the help of an external library called *socket.io* [9]. It enables to synchronize the standard outputs, execution state and progress between the server and the client in real time.

On page load, the client connects to the hosted socket.io server [7] and after page initialization, all communication is done through that connection, including execution state change, progress change, console data stream and model submission. In case of network error (*connection lost*), socket.io [9] tries to reconnect automatically if possible.

For the communication API and detailed class description, see: 3.2.4 and 3.3.3 above.

[8] https://en.wikipedia.org/wiki/WebSocket

3.4 Data files

The framework based on two types of data/configuration files. The configuration file (*see 3.4.1*) for storing all the information needed during model editing and executing, and the model file (*see 3.4.2*) in order to save the user-built model.

3.4.1 Configuration file

The configuration file of the visualization framework must be placed in `"lib/data"` directory as a valid `"data.json"` *JSON* file (*default*: `"lib/data/data.json"`). The file structure has two main sections: type descriptor (`"types"` attribute) and node descriptor (`"nodes"` attribute). The `"$"` sign means the root object of the file.

Type Descriptor

This section contains configurations for the node categories (`"node"` attribute) and the predefined data types (`"arg"` attribute). For node categorization, categories can be defined by adding new attributes to the `"arg"` object with style modifiers. The same method should be used for type customization: the attribute name refers to the predefined type and the content of the attribute will change the type defaults. Currently, the modifications, that can be applied with type descriptors, are quite limited but – in case of future development – the structure can be easily extended with user defined types and custom properties.

Category

`$.types.node.<category>`
where
`$` is the root object
`<category>` is the name of the described category

`fill`
Type: *string* (hex color)
Default: `NodeElement.style.category._base`
If set, the nodes included in the category will have that background color. This option is useful for visually separating the nodes with different functions.

Type

`$.types.arg.<type>`
where
$ is the root object
`<type>` is the name of the described type

`default`
Type: *any*
Default: *null*
This property can set the "global" default value for the given type.

Node descriptor

This section contains the description of the defined nodes. Inside `"$.nodes"` each property means a logical package (independent from the python package system) or a node definition which name will be the name of the property. If the property object contains the `"package"` and `"args"` properties, it will be considered as a node definition, otherwise it will be listed as a logical package.

Node

`$.nodes[.<pckg1>[.<pckg2>] … ].<node>`
where
$ is the root object
`<pckgN>` is the logical package for the user (can differ from python package)
`<node>` is the name of the described node

`package`
Type: *string*
Default: *must be specified*
Fully qualified python name for the callable function or class. Must be specified for node listing. The import path will be relative to the `"lib/data"` directory. Any python file within `"lib/data"` can be imported while running. (If no package name provided `"module"` will be used.) There is a predefined symbol (`"@"`) for easily calling the node input (*if callable*) with the given arguments.

`category`
Type: *string*
Default: –
If specified, the category's descriptor will be applied rendering (if the category is defined in the configuration file). If not, the default style will be used.

`args`
Type: *array of Arguments* (*see Argument section below*)
Default: *must be specified*
Argument list for the callable node. Must be specified for node listing.

`itype`
Type: *null* or *string* or *array of strings*
Default: *null*
Specified input type(s) for the node. If not specified, any types will be accepted as input.

`otype`
Type: *any*
Default: *null*
If specified, the name of the output type of the node will be considered as this value.

`conns`
Type: *int*
Default: *1*
Number of inputs to be accepted. If -1, any number of argument binding will be accepted. If 0, the argument value must be specified by the input filed, binding is not accepted. If its value is greater than 0, the argument will accept this much bindings.

`instance`
Type: *bool*
Default: *true*
If false, the represented function will not be called, or the represented class will not be instantiated.

`apply`
Type: *bool*
Default: *false*
If true, the represented function or class will be called or instantiated with the given parameters and the instance will be called with the input nodes.

`copy`

Type: *bool*

Default: *true*

If false, the represented function or class will not be copied on return value passing.

Argument

$.nodes[.<pckg1>[.<pckg2>] ...].<node>.<nth>
where
$ is the root object
<pckgN> is the logical package for the user (*can differ from python package*)
<node> is the node name
<nth> is the number of the described argument

`arg`
Type: *string*
Default: *must be specified*
Name of the argument to be associated with.

`name`
Type: *string*
Default: `.arg`*'s value*
Displayed name for the argument. If not given, `.arg` will be used (`"snake_case"` will be capitalized and formatted with spaces).

`itype`
Type: *null* or *string* or *array of strings*
Default: *null*
Specified input type(s) for the argument. If null, any types will be accepted as input. If conns is set to `0`, the following type string will be accepted:
`nosupport`: input not supported
`bool`: bool (checkbox)
`string`: strings
`int`: integers
`int0+`: positive integers including zero
`int+`: positive integers
int-: negative integers
`int0-`: negative integers including zero
`float`: floats
`float0+`: positive floats including zero
`float+`: positive floats
`float-`: negative floats
`float0-`: negative flats including zero
`float01`: floats between [0, 1]
`float11`: floats between [-1, 1]

`default`
: Type: *any*
Default: *null*
Default value for the argument.

`conns`
: Type: *int*
Default: *0*
Number of inputs to be accepted. If `-1`, any number of argument binding will be accepted. If `0`, the argument value must be specified by the input filed, binding is not accepted. If its value is greater than `0`, the argument will accept this much bindings.

`required`
: Type: *bool*
Default: *false*
Indicate if the value is required for the node. If true, the argument will be stable sorted before the other arguments on call.

3.4.2 Model file

The visual models can be saved in *Go.js JSON* model format. The file is generated and parsed by the visual framework so the user should not modify that. The file contains necessary properties for *Go.js* visualization (*see: Go.js* [8] *the API docs*[9]), node definitions with actual arguments and link descriptions. The additional properties are below: (*see: 3.4.1 above for all*)

Model

`$.*`
where
$ is the root object

`$.class`
Type: *string*
Defines the model type to be parsed. (*see: Go.js docs* [8])

`$. copiesKey`
Type: *bool*
If true, the node key will be copied on node copying. (*see: Go.js docs* [8])

`$.nodeCategoryProperty`
Type: *string*
Property name for render template selection. (*see: Go.js docs* [8])

`$.linkFromPortIdProperty`
Type: *string*
Node directed port binding reference property (from node). (*see: Go.js docs* [8])

`$.linkToPortIdProperty`
Type: *string*
Node directed port binding reference property (to node). (*see: Go.js docs* [8])

`$.nodeDataArray`
Type: *array of objects* (serialized Node objects)
Contains node definitions for *Go.js*. (*see: Go.js docs* [8])

`$.nodeDataArray[].$$`
Type: *string*
Node location in configuration file/object from the root of $.nodes. (*see: 3.4.1 above*)

[9] https://gojs.net/latest/api/

`$.nodeDataArray[].name`
Type: *string*
Name of the node (come from configuration file object path, *see 3.4.1 above*)

`$.nodeDataArray[].argDescriptor`
Type: *array of objects* (serialized *Arguments* objects)
Same as `args` in *Node* object. (*see: 3.4.1 above*)

`$.nodeDataArray[].position`
Type: *object*
Position object; has `x` and `y` properties. Refers to the position in graph editing area.

`$.nodeDataArray[].view`
Type: *string*
Indicates, how much details will be displayed when editing. The following options are accepted:
`name`: just the name and main node input and output
`minimal`: name extended with required attributes
`detailed`: minimal extended with all types of attributes

`$.nodeDataArray[].key` (*see: Go.js docs* [8])
Type: *number*
Unique identifier for nodes.

`$.linkDataArray` (*see: Go.js docs* [8])
Type: *array of objects* (serialized link objects)
Contains link definitions for *Go.js* [8]. Properties are:
`from`: the key of the node the is coming from
`to`: the key of the node the link is going to
`fromPort`: port name in the specified start node (`"::input"` and `"::output"` is the main input and output of the start node, others are argument names)
`toPort`: port name in the specified destination node (`"::input"` and `"::output"` is the main input and output of the destination, others are argument names)

4 Testing

The framework and the adapter functions for adversarial input generator methods was tested with exported models (`test/*.json` files). For testing purposes, the server can be started in debug mode with the attached "`test/server.bat`" (*windows*) or "`test/server.sh`" (*linux*) files.

Loading and executing the test files on the server is enough for client-side testing too. When the user loads the test file the client-side script also parses the whole model, add the nodes and edges to the diagram model and renders it in the editing area. Actual execution can test the *WebSocket* API and server event handler functions as the model changing its state. Multi-client mode can also be tested with the given test files by opening multiple browser tabs and starting and stopping executions at the same time.

Test files
*test/**

attacks.json	Tests all the implemented attack method adapter functions. *Reads a color image and apply all attack function separately than rename the output files and writes them to "outputs/attacks/" folder.*
criterion.json	Tests all the attack criteria objects. *Constructs all the criteria objects.*
distances.json	Tests all the attack distance metric functions. *Constructs all the attack distance objects.*
general.json	Real world example. *Loads multiple image sets, transforms and renames them, then runs multiple attacks defined with "Configuration" nodes and saves them to "outputs/general/" folder.*
images.json	Tests the image loader functionalities. *Scan the given directory (recursively too) and search for file names matching the given path's regular expression. Loads files in color and grayscale mode.*
labels.json	Tests label file loading. *Loads different label files and assign the labels to the matching images.*

models.json	Test different Keras model loading. *Loads different Keras models and one input image for each model. Label maps are also loaded and used for specifying the original labels. Transforms the data to the model-required format and apply a simple gradient attack then saves the new files to "outputs/models/" folder.*
transformation.json	Test the data transformation functions. *Load greyscale and color images, renames them and inverts their colors and save the new images to "outputs/inverse/" folder.*
utils.json	Tests the functions in the "utils" virtual package. *Constructs "bound" and "epsilon" data types.*

As the program has quite a lot components and classes built on each other, during the development smaller tests were made "by hand" to check if the core components are stable and well-functioning.

In case of hidden bugs, debug mode server, the standard error console of the model and the browser console can provide useful information as the errors which caught during running are printed to these streams.

5 Summary

5.1 Adversarial Image Generation

The base program can provide a relatively simple visual interface to generate perturbed images in order to fool neural networks. It can generate adversarial input images with several methods and save them separately for benchmarking purposes.

The program is also having the potential to be extended with actual automated benchmark functions or event with neural network training functionalities.

5.2 Dataflow visualization framework

FlowPy framework have become an independent python module with its modular and expandable structure. Currently, *FlowPy's* configuration file has to be written "by hand" and in some cases "adapter" functions have to be created to adjust *FlowPy's* abilities to other packages. However, there is "reflection" in *python* which makes it able to list packages' functions and classes, examine object's type and get functions' argument list with default values in runtime. This feature could change the way how the configuration file is edited now.

One possible extension is to create a package adaptation system to generate the configuration file automatically which will make the framework widely used. On the other hand, there are package structures – even in this program – which could be transformed into *FlowPy's* configuration format by using reflection, but the result will be a huge and complex type system with plenty of functions which is the opposite that this project aimed. [4]

6 References

[1] Ian J. Goodfellow, Jonathon Shlens and Christian S., "Explaining and Harnessing Adversarial Examples," 2014.

[2] Alexey Kurakin, Ian J. Goodfellow and Samy Bengio, "Adversarial Examples in the Physical World," 2017.

[3] Xiaoyong Yuan, Pan He, Qile Zhu and Xiaolin Li., "Adversarial Examples: Attacks and Defenses for Deep Learning," 2017.

[4] J. Rauber and B. Wieland, "Foolbox documentation," 2017. [Online]. Available: https://foolbox.readthedocs.io/. [Accessed May 2019].

[5] The Pallets Projects, "Flask python package," [Online]. Available: http://flask.pocoo.org. [Accessed May 2019].

[6] Pylons Project, "Waitress python package," [Online]. Available: https://docs.pylonsproject.org/projects/waitress. [Accessed May 2019].

[7] "Flask SocketIO package," [Online]. Available: https://flask-socketio.readthedocs.io. [Accessed May 2019].

[8] Northwoods Software, "Go.js," Northwoods Software, [Online]. Available: https://gojs.net. [Accessed May 2019].

[9] "Socket.io," [Online]. Available: https://socket.io. [Accessed May 2019].

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