Experience in Bridging Keras for Python with Pharo

Alejandro Infante
DCC, Universidad de Chile
Chile
ainfante@dcc.uchile.cl

Alexandre Bergel
DCC, Universidad de Chile
Chile
abergel@dcc.uchile.cl

Abstract

Creating bridges or wrappers for libraries developed in different languages is a challenging task. We successfully created a bridge between Pharo and Python to use the neural network library Keras. The fact that Keras is implemented in Python is completely transparent to a Pharo programmer.

We present and discuss the architectural decisions involved in the development of the bridge. Our decisions include the use of command messages for communication between languages and the use of first-class Pharo objects to generate and operate Python objects while providing live-programming capabilities.

ACM Reference Format:
Alejandro Infante and Alexandre Bergel. 2018. Experience in Bridging Keras for Python with Pharo. In *Proceedings of International Workshop on Smalltalk Technologies (IWST'18).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Nowadays programming languages come with a large amount of frameworks and libraries to solve specific problems in an easy and efficient way. Having access to a vast and powerful set of libraries is fundamental for developing software efficiently in a modern programming language.

Some modern libraries written for C++, Java, Python, and others have undergo an evolution through the years involving thousands of engineering hours. Even though these libraries have a huge impact for solving the problems for which they were designed, it is impractical to port them to many languages because of the cost it would require. This is specially true for new languages or languages with rather small communities or supporting groups.

IWST'18, September 09–14, 2018, Cagliari, Italy
© 2018 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Currently, Python is the most popular language for developing machine learning software. This is explained by the large amount of well written and documented libraries such as Tensorflow [1], Theano [14], Pytorch [11] and others. Trying to port any of these libraries to Smalltalk is expensive and producing bindings for them is a demanding and challenging task.

To address this problem we present our experience for producing a bridge between Pharo and Python to use Keras, a deep learning library for building neural networks [5]. Our bridge allows any Pharo developer to use Keras as if Keras would be written in Pharo.

This paper describes our bridge for Keras and presents our experience with building it. To structure this experience report we took as a running example a neural network built with Keras and we illustrate the necessary steps with their difficulties and design decisions.

To bridge the Keras Python library with Pharo we had to solve the following problems:

- Be able to execute arbitrary pieces of generated Python code within the same Python instance.
- Be able to efficiently communicate with Python without blocking the Pharo image.
- Be able to provide live-programming capabilities to the library from Pharo.
- Be able to send and receive objects between Pharo and Python.
- Be able to wrap library calls in promises that support callbacks.
- Be able to give meaningful feedback to users about errors occurring in Pharo or Python.

Result of our effort is accessible on https://github.com/ObjectProfile/KerasWrapper.

This paper is outlined as follows. Section 2 illustrates the creation of a neural network in Keras using the Python language. Section 3 presents our strategy to make Python objects first-class entities in Pharo. Section 4 describes the communications between Python and Pharo. Section 5 presents our architecture to build promises and callbacks across both languages. Section 6 presents some of the limitations of our approach and outlines our future work. Section 7 concludes the paper.

2 A Simple Keras Neural Network

Keras is a Python neural network API that works on top of Tensorflow [1], Theano [14] or CNTK [13]. The objective of Keras is to deliver an easy way of fast prototyping neural networks that support state of the art architectures that can also run seamlessly on CPU and GPU [5].

We present a very simple neural network in Python to solve the classification problem of the iris dataset using Keras. The iris dataset [7] consists of a list of flower properties and the flower category. To keep the description of our example, we skip the preprocessing of the data, such as performing the one-hot-encoding of the categories and the splitting of the dataset into a training and a testing set.

Firstly we must import the required module and classes:

```
from keras.models import Sequential
from keras.layers import Dense, InputLayer, Activation
```

We then create the sequential model and add the input layer, whose size matches the dimension of our dataset variables:

```
model = Sequential()
model.add(InputLayer(input_shape=(4,)) #Number of flower properties
```

We then add a layer with 16 neurons using a sigmoid activation function:

```
model.add(Dense(16))
model.add(Activation('sigmoid'))
```

After that we add the last layer using a softmax activation function. The number of neurons of this layer must match with the number of categories (*i.e.,* there are 3 kind of iris flowers), therefore we use 3 neurons:

```
model.add(Dense(3))
model.add(Activation('softmax'))
```

The next step is compiling the neural network, where we must specify the loss function to be used, the optimizing technique and the metrics to gather. In our case we use Categorical cross-entropy, the Adam optimizer and the Accuracy metric:

```
model.compile(loss='categorical_crossentropy', optimizer='adam',
    metrics=['accuracy'])
```

Now the model is ready for being trained. To train the model we must provide a training set of properties (`train_X`), the category for each datapoint (`train_Y`) and the number of epochs we want to train the model. The output of the training routine is a dictionary holding the history of the loss value and accuracy at each epoch.

```
training_history = model.fit(train_X, train_Y, verbose=0, batch_size=1,
        epochs=100).history
```

The last step is the evaluation of the model, which is performed by passing a testing dataset with the expected category for each datapoint.

The output of the evaluation routine is a list of metric values:

```
loss, accuracy = model.evaluate(test_X, test_Y)
```

With these few lines of code in Python we can create a simple, but functional, neural network. In the next sections we will show how each part of this network is built with Pharo and how it is then run in Python.

3 Python Entities as First-Class Objects in Pharo

One of the key aspects of Pharo is the live-programming environment and the inspecting capabilities it gives to developers. We represent each Python entity as a particular Pharo class.

These classes have three main objectives. The first objective is defining the Pharo API for using the Python library. Secondly, being able to keep track of the entities living in Python and interact with them. The third objective is being able to provide live-programming features to the entities running in Python.

In Keras, the very first action to create a neural network is creating the Sequential model which we define as an object that holds a list of layers and activation functions. Then we implemented the `add:` method to the sequential model for adding new layers.

```
"Pharo code"
model := SequentialModel new.
model add: (InputLayer inputDim: 4). "Add input layer"
model add: (DenseLayer neurons: 16). "Add dense layer"
model add: (SigmoidActivation new). "Add sigmoid activation function"
```

The code above generates the following Python code:

```
"Python generated code"
knRandomVar = Sequential()
knRandomVar.add(InputLayer(input_shape=(4,))
knRandomVar.add(Dense(16))
knRandomVar.add(Activation('sigmoid'))
```

Each class is responsible for holding the Python variable name associated with the represented entity. In this example, the `SequentialModel` instance stores the variable name `knRandomVar` which is used in Python to hold the Sequential object. For simplifying the code snippet we are using `knRandomVar` as an example, the variable name in working examples is randomly generated and consists on a 25 random character string. This behavior is repeated on all variables in Python generated code to avoid name conflicts.

Another important responsibility of the class model is producing the necessary Python code for using the library and deciding when to create the objects in Python. Each object knows how to produce the corresponding Python code to use the library. In our bridge we opted to use Python3Generator to generate syntactically correct Python code.

The following method is responsible of generating the code of a new Dense layer:

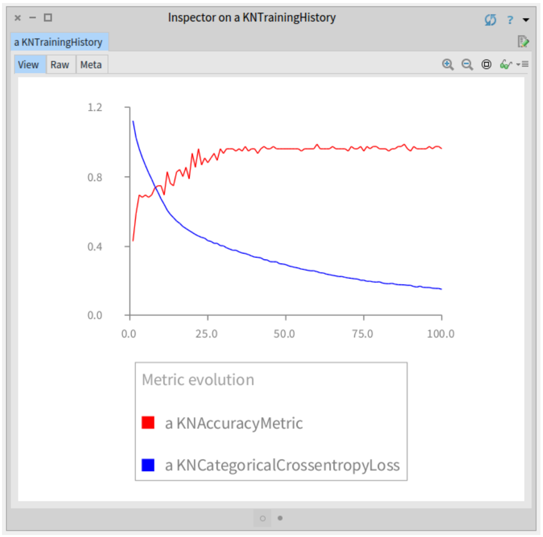

Figure 1. Visualization of training history during 100-epochs evolution.

```
DenseLayer>>pyCreate
  ^ 'keras' asP3GIdentifier => #layers => #Dense callWith: { self neurons }
```

In contrast to `SequentialModel`, the `DenseLayer` Python code is not executed when the Pharo object is created. Instead, the message `SequentialModel>>add:` is the responsible of creating the layer in the Python side by sending the result of `DenseLayer>>pyCreate` and appending the layer to the `SequentialModel` Python object.

The decision of when to dispatch the execution to the Python environment is entirely delegated to the developer because we want the bridge to be as flexible as possible to the requirements of the developers. In the context of the `SequentialModel`, we decided that a `DenseLayer` does not make sense outside of the model itself, and therefore, the creation of the layer is delayed until appended to a model.

We can then continue adding elements to the network and compile the model when it is ready.

```
"Pharo code"
model add: (DenseLayer neurons: 3). "Add dense layer"
model add: (SoftmaxActivation new). "Add softmax activation function"
model
  compileLoss: KNCategoricalCrossentropyLoss new
  optimizer: KNAdamOptimizer new.
```

The code above generates the following Python code:

```
"Python generated code"
knRandomVar.add(Dense(3))
knRandomVar.add(Activation('softmax'))
knRandomVar.compile( optimizer=Adam(),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

Inspecting library entities. Because we have first class objects we can add views to the inspector to improve the live-programming experience when using external libraries. For gathering the data we can communicate with the Python entity, if it defines appropriate getters, or hold a partial state of the object in Pharo. We prefer the second option because it simplifies the debugging when developing the library.

For Keras we implemented a new view in the GTInspector [3] for the `TraniningHistory`, which holds the evolution of the training metrics during the training of the model. Figure 1 presents the new view that contains a line graph of the metrics using Roassal [2].

Modeling the domain. We have not developed a translating mechanism that generates Pharo bindings from Python code. Instead, we manually designed how Keras should be exposed in the Pharo environment. Up to now, the design is driven by user stories, as the presented in Section 2 and is highly influenced by Keras Python object model. An example of this is that all first-class objects presented in this section do also have a first-class object representation in the original Keras library written in Python.

For this reason, our Keras implementation for Pharo is not complete, but it features a working subset of Keras that allow us to create functional neural networks.

4 Communication with the Python Library

Until now we have stated that our wrapper objects can communicate with the Python entities, but we have not explained how they do it. The main architectural decision we have made is that the basic communication element from Pharo to Python is a command. Each of them includes a piece of Python code and a set of bindings to build the Python environment where the code will run.

When a command is sent, it is serialized as a message and asynchronously sent to Python, which appends it to a list of commands to be executed. The Python main thread is continuously consuming and executing the commands to finally return the resulting expression of each of them back to Pharo.

4.1 Python code generation

To generate Python code we use Python3Generator [6], a library that implements a domain-specific language for generating syntactically correct Python code. One of the greatest strengths of this library is that you can easily compose elements to build complex code snippets. An example of this is the implementation of `compileLoss:optimizer:metrics:` method in `SequentialModel`:

```
compileLoss: loss optimizer: optimizer metrics: metricsArray
  ...
  Keras send: (
    pyVar => #compile                    "Send message compile to pyVar"
          callWith:#()                   "Sequential arguments"
          with: {                        "Named arguments"
            #optimizer –> optimizer pyCreate.
            #loss –> loss pyCreate.
            #metrics –> (metricsArray collect: #pyCreate)
        } asDictionary)
```

The last instruction executed by this method is sending the message `Keras send:`, which receives as argument a P3GGenerable and sends a command to Python with the code generated with it.

A P3GGenerable is a representation of a Python expression built using Python3Generator. In this case, the expression generated is sending the message `compile` to the object stored in the variable `pyVar` with no sequential arguments and three named arguments: the optimizer, the loss function and the metrics.

Each of the named arguments is represented as an association which has the argument name as key and the argument expression as value. Since optimizer, loss and the metrics are modeled as first class objects, we send the message `pyCreate` to generate the P3GGenerable for each of these objects.

The following code snippet shows the implementation of `AdamOptimizer>>pyCreate` method:

```
AdamOptimizer>>pyCreate
  ^ #Adam asP3GI callWith: #()
```

The method above returns the Python expression `Adam()` which is used to build the model `compile` expression.

Finally, the resulting expression from `SequentialModel>>compileLoss:optimizer:metrics:` method of the example is:

```
knRandomVar.compile(
  optimizer=Adam(),
  loss=categorical_crossentropy,
  metrics= [ 'acc' ] )
```

Even though the idea to treat optimizers, loss functions and metrics as first class objects may be seen as over-engineering, the models required to represent these components are not trivial and accept many parameters that enable their customization. An example of this is that SGD optimizer have 4 different parameters being `SGD(lr=0.01, momentum=0.0, decay=0.0, nesterov=False)` a reasonable construction.

4.2 Command pattern for the communication

The command is the most important unit of communication between Pharo and Python. It enables execution of arbitrary code in Python, defining the environment variables for the call and communicating the result of the last Python expression in the command back to Pharo. The Command has three essential parts: A command id, the Python code and the bindings.

Command Id. Each command has a unique id used to identify a command at any moment in the execution. This id is used to verify if the command has been already executed in Python or to know in which position is it in the command list queue. This id is also used to feedback Pharo about errors occurred when executing a command or to deliver back a value after the execution of one.

Python code. Each command holds a Python script that will be executed in Python. The code snippet is stored as a P3GGenerable and it is only converted to source code as part of the serialization of the command.

Python bindings. Each command has a list of associations where the key corresponds to a Python variable name and the value is an object that is serialized and then bounded to the declared variable name.

Knowing how bindings work we can describe how to train the Keras model of our example. The following message is sent to train the model:

```
SequentialModel>>fit: trainX labels: trainY epochs: numberOfEpochs
  ^ Keras
      send:
        (pyVar => #fit
          callWith: { #trainXVar asP3GI. #trainYVar asP3GI }
          with: {
            #verbose –> 0.
            #batch_size –> 1.
            #epochs –> numberOfEpochs.
            } asDictionary)
      bindings: {
        #trainXVar –> trainX.
        #trainYVar –> trainY }.
```

The arguments *trainX* and *trainY* are bounded to the Python variables `trainXVar` and `trainYVar`, which are then used by the generated code as arguments of the `fit` function. The code generated by this method is the following:

```
knRandomVar.fit(trainXVar, trainYVar,
      verbose=0, batch_size=1, epochs=100)
```

Notice that we can also replace `trainXVar` and `trainYVar` variable names with randomly generated variable names to reduce conflicts with other variables.

4.3 Serialization of objects

An important step for delivering the commands between Pharo and Python is the serialization of the bindings. The serialization component is designed for receiving any kind of Pharo object. Before the command is sent to Python a serialization visitor is responsible of building a representation for each object that will be deserialized in Python for its use.

Currently we only have used a simple JSON serialization for the most common objects, such as Dictionaries, Arrays, Strings, Numbers, Booleans and Nil. At this stage we use JSON because most languages already have out of the box JSON serialization tools, but we are aware that we could greatly improve memory consumption by using a compact binary serialization. Despite that, we will soon extend our serialization component to support sparse matrices to cope with one-hot-encoded matrices, which are very large tables where most values are 0.

4.4 Python command queue and execution

The length of a command execution can vary a lot depending on the task asked to Python. For this reason we opted for using asynchronous messaging for commands. To support this, Python has a queue of commands waiting to be executed. Each new command sent by Pharo is queued in and an answer is immediately sent back to Pharo, preventing blocking Pharo for a long time.

The first step to execute a command is to include the bindings of the command into a Python environment scope. This scope is initially empty, but is shared between all the commands executed so the command Python code can find and use objects created by other commands.

Python then executes the generated code by using the `exec` function [8]. This allow us to manipulate at will the environment scope and to generate and execute arbitrary code from Pharo and executing it in Python without the need of restarting it.

4.5 Message delivery between Pharo and Python

Because we are sending asynchronous messages between the two runtimes, both of them need to be capable of sending and receiving messages. Pharo needs to be able to send commands to Python, and Python needs to be able to send notifications to Pharo once a command execution has finished. Architecturally speaking, both will behave as servers and clients at the same time.

For simplicity we are using HTTP for delivering messages. We are aware that we can improve the performance replacing the HTTP protocol with a dedicated messaging protocol over a socket. We decided to postpone that change in order to push and validate other design decisions presented in this paper.

Messages sent by Pharo. The messages sent by Pharo are the commands to be executed by Python we have detailed before.

Messages sent by Python. Python sends two kind of messages. The first message is *notify*, which is used to let Pharo know that a command execution has finished and deliver the return value of the command to Pharo. The second message is *notifyError*, which is used to let Pharo know that an error has occurred in the execution of a specific command. The error message contains the command id and an error message, which allow us to improve the debugging experience.

Communication schema. Figure 2 presents a schema detailing how different components in Pharo and Python interact when `SequentialModel>>fit:labels:epochs:` is called.

The main thread in Pharo creates a new command, a promise for the return value and sends the command to Python for its execution. In this example we have introduced a waiting promise, which halts the execution until the command has been executed in Python. Promises are discussed in more detail in Section 5.

When receiving the command, the Python HTTP server immediately pushes the new command to the Command Queue. A second Python process is executing the commands in the queue, and eventually, will execute the command described in this example. When the command execution is finished, a `notify` HTTP message is sent back to Pharo.

A background Pharo process running an HTTP server is responsible of processing the `notify` message. The first step is retrieving the promise associated to the command using the command id. The return value included in the `notify` is then deserialized and introduced into the promise. Lastly, the promised is signaled allowing the main thread to continue its execution.

4.6 Error inspection and debugging

When executing arbitrary code in Python there is always the possibility of an error raising on the execution. We use a try-catch handler responsible of delivering a *notifyError* message back to Pharo when an error occur in Python. This routine halts the execution until a decision about how to deal with the error is sent back from Pharo.

When Pharo receives *notifyError* it crafts a Pharo exception called `P3PythonError`. This object holds the Python command that signaled the error and the message generated by Python about the error. After that, the `P3PythonError` instance is signaled and the developer can decide how to recover from the error.

We currently support 3 different actions for recovering control of the execution:

- *Ignore command*: Discard the command that raised the error and continue the execution with the next command in the command queue.
- *Replace command*: Discard the command that raised the error and execute instead another command to recover the state of the execution. Then continue executing the next command in the command queue.
- *Drop queue*: Discard all the following commands in the command queue. Then Python waits until new commands are pushed from Pharo and execute them normally.

5 Promises and Callbacks

When a command is sent, Pharo does not wait to receive the result of the execution from Python because it could take a long time. Instead, it creates a promise, a value holder object that is aware of the Python command execution and is responsible of update the held value when the execution successfully finish.

The API also allows developers to define callbacks for promises. Callbacks fulfill two particular objectives: letting the user define a block to use the value when received and being able to transform the received object in a business-domain object defined in Pharo.

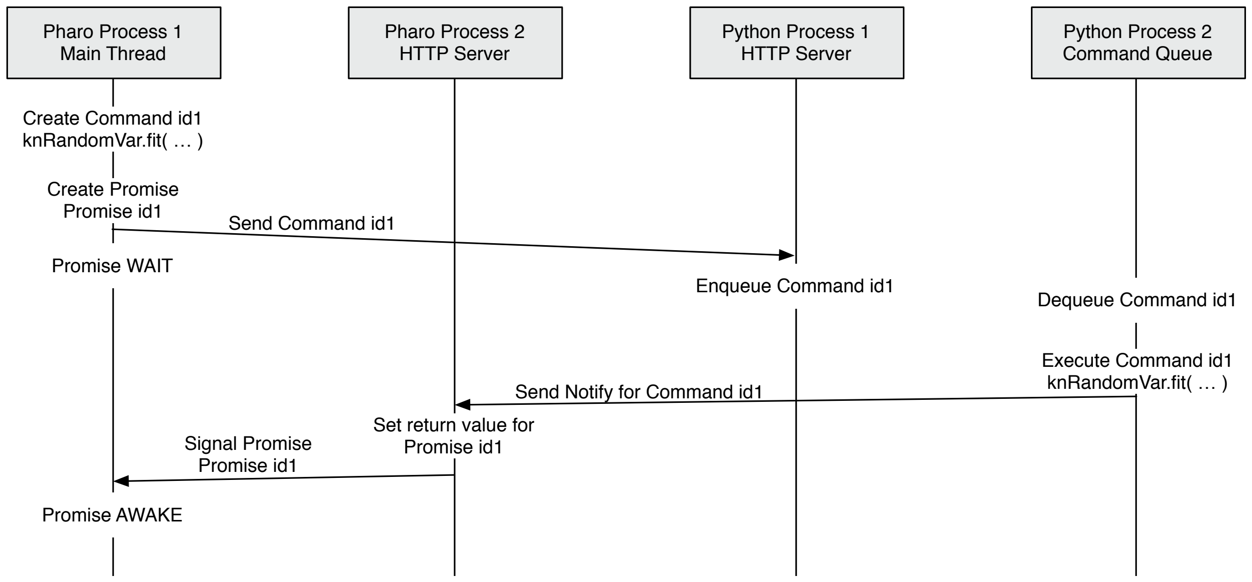

Figure 2. Communication schema for SequentialModel»fit:labels:epochs:

We use this capability to receive a TrainingHistory object as result of the `SequentialModel>>fit:` method, instead of a simple dictionary:

```
SequentialModel>>fit: trainX labels: trainY epochs: numberOfEpochs
  ^ Keras
      send: ...
      bindings: ...
      transformBlock: [ :dictionary |
          TrainingHistory model: self history: dictionary ]
```

The *TrainingHistory* object allow us to define new methods to query for the history of specific metrics, generate visualizations easier and generate new inspector views for the data.

Sometimes the business logic can not continue without receiving the result from Python. The API offers the capability of waiting for the value using simply the `waitForValue` message. We use this message in our example to store the training history and the result of evaluating the network in a variable:

```
trainingHistory := (model fit: trainX labels: trainY epochs: 100)
                        waitForValue.
trainingHistory metrics. "–> #('loss' 'accuracy')"
metricValues := (model evaluate: testX labels: testY) waitForValue.
metricValues. "–> #(0.124 0.987) –– loss and accuracy"
...
```

6 Limitations and Future Work

We have identified three important improvements to be implemented:

Messaging protocol instead of HTTP. HTTP is one of the most used application level protocol, but it was not designed to excel at message delivering. To simulate a messaging protocol we defined URL end-points. There are specialized protocols for message delivering such as ZeroMQ or gRPC that would lower the latency of the communication.

Compact binary serialization instead of JSON. JSON is an open-standard format that transmit data objects as human-readable text. It is well known that JSON format file size is not optimal and a significant amount of memory can be saved by using binary encoding [10].

Memory management tools. Currently we do not offer any memory management support to the developers. If they create variables or bindings, they are responsible for detaching those references to prevent memory leaks. To solve this problem we will define tools for declaring and enforcing the life-expectancy of each reference using different policies:

- *Eternal*: The binding will not be removed.
- *Command*: The binding will be removed after the command execution is finished.
- *Pharo garbage collection*: The binding is associated to a Pharo object and it will be removed once the Pharo object is garbage collected.

Comparison with other bridges. As future work we expect to exhaustively compare our bridging architecture with other solutions. Two bridges that are relevant to compare with are:

- *Atlas*: As stated in their webpage, it consists on a Python bridge that uses inter process communication stream sockets to allow communication between Pharo and Python. [4]
- *IPython*: It consists on a library to provide a Python kernel for interactive computing. [12] The most notable usage of IPython is the Jupyter Notebook, a web

application to create and share documents with live code. [9]

7 Conclusion

We presented our experience in designing a bridge between a Python library and Pharo and the implementation of it for bridging Keras. Our bridge defines first-class objects in Pharo to wrap Python entities and generate Python code. We also describe how we implement remote executed code in a Python environment from Pharo while preserving a live-programming experience.

Acknowledgement

We gratefully thank LAM Research for its financial support.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Alexandre Bergel. 2016. *Agile Visualization.* LULU Press. http://AgileVisualization.com

[3] Andrei Chiş, Oscar Nierstrasz, Aliaksei Syrel, and Tudor Gîrba. 2015. The Moldable Inspector. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!) (Onward! 2015).* ACM, New York, NY, USA, 44–60. DOI: https://doi.org/10.1145/2814228.2814234

[4] Dimitris Chloupis. 2016. Atlas. https://github.com/kilon/Atlas. (2016). Accessed on 2018-08-15.

[5] François Chollet and others. 2015. Keras. https://keras.io. (2015).

[6] Julien Delplanque. 2017. Python3Generator. https://github.com/juliendelplanque/Python3Generator. (2017). Accessed on 2018-08-15.

[7] Dua Dheeru and Efi Karra Taniskidou. 2017. UCI Machine Learning Repository. (2017). http://archive.ics.uci.edu/ml

[8] Python Software Foundation. 2018. Python Standard Library, Built-in Functions. https://docs.python.org/3/library/functions.html#exec. (2018). Accessed on 2018-08-15.

[9] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. 2016. Jupyter Notebooks – a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt (Eds.). IOS Press, 87 – 90.

[10] K. Maeda. 2012. Performance evaluation of object serialization libraries in XML, JSON and binary formats. In *Digital Information and Communication Technology and it's Applications (DICTAP), 2012 Second International Conference on.* 177–182. DOI: https://doi.org/10.1109/DICTAP.2012.6215346

[11] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[12] Fernando Pérez and Brian E. Granger. 2007. IPython: a System for Interactive Scientific Computing. *Computing in Science and Engineering* 9, 3 (May 2007), 21–29. DOI: https://doi.org/10.1109/MCSE.2007.53

[13] Frank Seide and Amit Agarwal. 2016. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16).* ACM, New York, NY, USA, 2135–2135. DOI: https://doi.org/10.1145/2939672.2945397

[14] Theano Development Team. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints* abs/1605.02688 (May 2016). http://arxiv.org/abs/1605.02688