DeepArchitect Documentation

Renato Negrinho

Oct 30, 2019

Contents

DeepArchitect is a framework for architecture search in arbitrary domains. DeepArchitect was designed with a focus on **modularity**, **ease of use**, **reusability**, and **extensibility**. DeepArchitect aims to impact the workflows of both researchers and practitioners by reducing the burden resulting from the large number of choices needed to design deep learning models. We recommend the reader to start with the *overview*, tutorials (e.g., *here*) and simple examples (e.g., here) to get a gist of the framework. Currently, we support Tensorflow, Keras, and PyTorch. See here for minimal complete examples for each of these frameworks. It should be straightforward to adapt these examples for your use cases. See *here* to learn how to support new frameworks, which should require minimal adaptation of the existing framework helpers found here. Questions and bug reports should be submitted through Github issues. See *here* for details on how to contribute.

CHAPTER 1

Overview

1.1 Overview

[CODE] [DOCUMENTATION] [PAPER] [BLOG POST]

DeepArchitect: Architecture search so easy you'll think it's magic!

1.1.1 Introduction

DeepArchitect is a framework for automatically searching over computational graphs in arbitrary domains, designed with a focus on **modularity**, **ease of use**, **reusability**, and **extensibility**. DeepArchitect has the following **main components**:

- a language for writing composable and expressive search spaces over computational graphs in arbitrary domains (e.g., Tensorflow, Keras, Pytorch, and even non deep learning frameworks such as scikit-learn and preprocessing pipelines);
- search algorithms that can be used for arbitrary search spaces;
- logging functionality to easily track search results;
- visualization functionality to explore search results.

DeepArchitect aims to impact the workflows of both machine learning researchers and practitioners. For researchers, DeepArchitect aims to make architecture search research more reusable and reproducible by providing them with a modular framework that they can use to implement new search algorithms and new search spaces while reusing code. For practitioners, DeepArchitect aims to augment their workflow by providing them with a tool to easily write search spaces encoding a large number of design choices and use search algorithms to automatically find good architectures.

DeepArchitect has better integration than current hyperparameter optimization tools, e.g., hyperparameters are directly related to computational elements. This saves the expert the effort of writing from scratch an ad-hoc mapping from hyperparameter values to the corresponding computational graph. DeepArchitect is explicitly concerned with extensibility, ease of use, and programmability, e.g., we designed a language to write composable and expressive search spaces. Existing work on architecture search relies on ad-hoc encodings of search spaces, therefore being hard to adapt and reuse for new settings.

1.1.2 Installation

Run the following code snippet for a local installation:

```
git clone git@github.com:negrinho/deep_architect.git deep_architect
cd deep_architect
pip install -e .
```

After installing DeepArchitect, attempt to run one of the examples to check that no dependencies are missing, e.g., `python examples/framework_starters/main_keras.py` or `python examples/mnist_with_logging/main.py --config_filepath examples/mnist_with_logging/configs/debug.json`. We omitted many of the deep learning framework dependencies to avoid installing unnecessary software that may not be used by a particular user.

We have included utils.sh with useful functionality to develop for DeepArchitect, e.g., to build documentation, extract code from documentation files, and build Singularity containers.

1.1.3 A minimal DeepArchitect example with Keras

Consider the following short example that we minimally adapt from this Keras example by defining a search space of models and sampling a random model from it. The original example considers a single fixed three-layer neural network with ReLU activations in the hidden layers and dropout with rate equal to *0.2*. We construct a search space by relaxing the number of layers that the network can have, choosing between sigmoid and ReLU activations, and the number of units that each dense layer can have. Check the following minimal search space:

```
from __future__ import print_function

import keras
from keras.datasets import mnist
from keras.models import Model
from keras.layers import Dense, Dropout, Input
from keras.optimizers import RMSprop

import deep_architect.helpers.keras_support as hke
import deep_architect.modules as mo
import deep_architect.hyperparameters as hp
import deep_architect.core as co
import deep_architect.visualization as vi
from deep_architect.searchers.common import random_specify

batch_size = 128
num_classes = 10
epochs = 20

# the data, split between train and test sets
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
```

(continues on next page)

(continued from previous page)

```
# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

# model = Sequential()
# model.add(Dense(512, activation='relu', input_shape=(784,)))
# model.add(Dropout(0.2))
# model.add(Dense(512, activation='relu'))
# model.add(Dropout(0.2))
# model.add(Dense(num_classes, activation='softmax'))

D = hp.Discrete


def dense(h_units, h_activation):
    return hke.siso_keras_module_from_keras_layer_fn(Dense, {
        'units': h_units,
        'activation': h_activation
    })


def dropout(h_rate):
    return hke.siso_keras_module_from_keras_layer_fn(Dropout, {'rate': h_rate})


def cell(h_units, h_activation, h_rate, h_opt_drop):
    return mo.siso_sequential([
        dense(h_units, h_activation),
        mo.siso_optional(lambda: dropout(h_rate), h_opt_drop)
    ])


def model_search_space():
    h_activation = D(['relu', 'sigmoid'])
    h_rate = D([0.0, 0.25, 0.5])
    h_num_repeats = D([1, 2, 4])
    return mo.siso_sequential([
        mo.siso_repeat(
            lambda: cell(
                D([256, 512, 1024]), h_activation, D([0.2, 0.5, 0.7]), D([0, 1])
            ), h_num_repeats),
        dense(D([num_classes]), D(['softmax']))
    ])


(inputs, outputs) = mo.SearchSpaceFactory(model_search_space).get_search_space()
random_specify(outputs)
inputs_val = Input((784,))
co.forward({inputs["in"]: inputs_val})
outputs_val = outputs["out"].val
vi.draw_graph(outputs, draw_module_hyperparameter_info=False)
model = Model(inputs=inputs_val, outputs=outputs_val)
model.summary()

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

(continues on next page)

(continued from previous page)

```
history = model.fit(
    x_train,
    y_train,
    batch_size=batch_size,
    epochs=epochs,
    verbose=1,
    validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
```

This example is introductory and it is meant to show how to introduce the absolute minimal architecture search capabilities given an existing Keras example. In this case, we compactly express a substantial number of structural transformations of the computational graph. Our search space encodes that our network will be composed of a sequence of *1*, *2*, or *4* cells, followed by a final dense module that outputs probabilities over classes. Each cell is a sub-search space (again, exhibiting the modularity and composability of DeepArchitect). The choice of the type of activation for the dense layer in the cell search space is shared among all cell search spaces used. All other hyperparameter of the cell search space are chosen independently for each occurrence of the cell search space in the sequence.

We left the original single Keras model commented out in the code above for the reader to get a sense of how little code we need to add to support a nontrivial search space. We encourage the reader to think about how to support such a search space using current hyperparameter optimization tools or in an ad-hoc manner. For example, using existing tools, how much code would be required to encode the search space and sample a random architecture from it.

We have not yet discussed other important aspects of DeepArchitect. For example, more complex searchers are able to explore the search space in a more purposeful and sample efficient manner, and the logging functionality is useful to keep a record of the performance of different architectures. These and other aspects are better covered in existing tutorials. We recommend looking at the tour of the repository for deciding what to read next. This slightly more complex example shows the use of the search and logging functionalities. The framework starters are minimal architecture search examples in DeepArchitect across deep learning frameworks. These should be straightforward to adapt to implement your custom examples.

1.1.4 Framework components

In this section, we briefly cover the principles that guided the design of DeepArchitect. Some of the main concepts that we deal with in DeepArchitect are:

- **Search spaces**: Search spaces are constructed by arranging modules (both basic and substitution modules) and hyperparameters (independent and dependent). Modules are composed of inputs, outputs, and hyperparameters. The search spaces are often passed around as a dictionary of inputs and a dictionary of outputs, allowing us to seamlessly deal with search spaces with multiple modules and easily combine them. In designing substitution modules, we make extensive use of ideas of delayed evaluation. Graph transitions resulting from value assignments to independent hyperparameters are important language mechanics. Good references to peruse to get acquainted with these ideas are deep_architect/core.py and deep_architect/modules.py.
- **Searchers**: Searchers interact with search spaces through a simple API. A searcher samples a model from the search space by assigning values to each of the independent hyperparameters, until there are no unassigned independent hyperparameters left. A searcher object is instantiated with a search space. The base API for the searcher has two methods `sample`, which samples an architecture from the search space, and `update`, which takes the results for a sampled architecture and updates the state of the searcher. The reader can look at deep_architect/searchers/common.py, deep_architect/searchers/random.py, and deep_architect/searchers/smbo.py for examples of the common API. It is also worth to look at deep_architect/core.py and for the traversal functionality to iterate over the independent hyperparameters in the search space.

- **Evaluators**: Evaluators take a sampled architecture from the search space and compute a performance metric for that architecture. Evaluators often have a single method named `eval` that takes an architecture definition and returns a dictionary with the evaluation results. In the simplest case, there is a single performance metric of interest. See here for an example implementation of an evaluator.
- **Logging**: When we run an architecture search workload, we evaluate multiple architectures in the search space. To keep track of the generated results, we designed a folder structure that maintains a single folder per evaluation. This structure allows us to keep the information about the configuration evaluated, the results for that configuration, and additional information that the user may wish to maintain for that configuration, e.g., example predictions or the model checkpoints. Most of the logging functionality can be found in deep_architect/search_logging.py. A simple example using logging is found here.
- **Visualization**: The visualization functionality allows us to inspect the structure of a search space and to visualize graph transitions resulting from assigning values to the independent hyperparameters. These visualizations can be useful for debugging, e.g., checking if the search space is encoding the expected design choices. There are also visualizations to calibrate the necessary evaluation effort to recover the correct performance ordering for architectures in the search space, e.g., how many epochs do we need to invest to identify the best architecture or make sure that the best architecture is at least in the top 5. Good references for this functionality can be found in deep_architect/visualization.py.

1.1.5 Main folder structure

The most important source files in the repository live in the deep_architect folder, excluding the contrib folder, which contains auxiliary code to the framework that is potentially useful, but that we do not necessarily want to maintain. We recommend the user to peruse it. We also recommend the user to read the tutorials as they cover much of the information needed to extend the framework. See below for a high-level tour of the repo.

- core.py: Most important classes to define search spaces.
- hyperparameters.py: Basic hyperparameters and auxiliary hyperparameter sharer class.
- modules.py: Definition of substitution modules along with some auxiliary abstract functionality to connect modules or construct larger search spaces from simpler search spaces.
- search_logging.py: Functionality to keep track of the results of the architecture search process, allowing to maintain structured folders for each search experiment.
- utils.py: Utility functions not directly related to architecture search, but useful in many related contexts such as logging and visualization.
- visualization.py: Simple visualizations to inspect search spaces as graphs or sequences of graphs.

There are also a few folders in the deep_architect folder.

- communicators: Simple functionality to communicate between master and worker processes to relay the evaluation of an architecture and retrieve the results once finished.
- contrib: Functionality that it will not necessarily be maintained over time but that users may find useful in their own examples. Contributions by the community will live in this folder. See here for an in-depth explanation for the rationale behind the project organization and the contrib folder.
- helpers: Helpers for the various frameworks that we support. This allows us to take the base functionality defined in core.py and expand it to provide compilation functionality for computational graphs across frameworks. It should be instructive to compare support for different frameworks. One file per framework.
- searchers: Searchers that can be used for search spaces defined in DeepArchitect. One searcher per file.
- surrogates: Surrogate functions over architectures in the search space. searchers based on sequential model based optimization are used frequently in DeepArchitect.

1.1.6 Roadmap for the future

Going forward, the core authors of DeepArchitect expect to continue extending and maintaining the codebase and use it for their own research. The community will have a fundamental role in extending DeepAchitect. For example, authors of existing architecture search algorithms can reimplement them in DeepArchitect, allowing the community to use them widely and compare them on the same footing. This sole fact will allow progress on architecture search to be measured more reliably. New search spaces for new tasks can be implemented and made available, allowing users to use them (either directly or in the construction of new search spaces) in their own experiments. New evaluators can also be implemented. New visualizations can be added, leveraging the fact that architecture search workloads train many models. Ensembling capabilities may be added to DeepArchitect to easily construct ensembles from the many models that were explored as a result of the architecture search workload.

The reusability, composability, and extensibility of DeepArchitect will be fundamental going forward. We ask willing contributors to check the contributing guide. We recommend using GitHub issues to engage with the authors of DeepArchitect and ask clarification and usage questions. Please, check if your question has already been answered before creating a new issue.

1.1.7 Reaching out

You can reach the main researcher behind of DeepArchitect at negrinho@cs.cmu.edu. If you tweet about DeepArchitect, use the tag `#DeepArchitect` and/or mention me (@rmpnegrinho) in the tweet. For bug reports, questions, and suggestions, use Github issues.

1.1.8 License

DeepArchitect is licensed under the MIT license as found here. Contributors agree to license their contributions under the MIT license.

1.1.9 Contributors and acknowledgments

The main researcher behind DeepArchitect is Renato Negrinho. Daniel Ferreira played an important initial role in designing APIs through discussions and contributions. This work benefited immensely from the involvement and contributions of talented CMU undergraduate students (Darshan Patil, Max Le, Kirielle Singajarah, Zejie Ai, Yiming Zhao, Emilio Arroyo-Fang). This work benefited greatly from discussions with faculty (Geoff Gordon, Matt Gormley, Graham Neubig, Carolyn Rose, Ruslan Salakhutdinov, Eric Xing, and Xue Liu), and fellow PhD students (Zhiting Hu, Willie Neiswanger, Christoph Dann, and Matt Barnes). This work was partially done while Renato Negrinho was a research scientist at Petuum. This work was partially supported by NSF grant IIS 1822831. We thank a generous GCP grant for both CPU and TPU compute.

1.1.10 References

If you use this work, please cite:

```
@article{negrinho2017deeparchitect,
  title={Deeparchitect: Automatically designing and training deep architectures},
  author={Negrinho, Renato and Gordon, Geoff},
  journal={arXiv preprint arXiv:1704.08792},
  year={2017}
}

@article{negrinho2019towards,
```

(continues on next page)

(continued from previous page)

```
  title={Towards modular and programmable architecture search},
  author={Negrinho, Renato and Patil, Darshan and Le, Nghia and Ferreira, Daniel and
↪Gormley, Matthew and Gordon, Geoffrey},
  journal={Neural Information Processing Systems},
  year={2019}
}
```

The code for `negrinho2017deeparchitect` can be found here. The ideas and implementation of `negrinho2017deeparchitect` evolved into the work of `negrinho2019towards`, found in this repo. See the paper, documentation, and blog post.

CHAPTER 2

Tutorials

2.1 Tutorials

2.1.1 Search space constructs

This tutorial should give the reader a good understanding of how to write search spaces in DeepArchitect. All the visualizations generated in this tutorial can be found here.

Starting with a fixed Keras model

A simple example in Keras is a good starting point for understanding the search space representation language. DeepArchitect can be used with arbitrary frameworks (deep learning or otherwise) as the core codebase is composed mainly of wrappers. Consider the following example using Keras functional API pulled verbatim from here.

```
from keras.layers import Input, Dense
from keras.models import Model
from keras.utils import plot_model

inputs = Input(shape=(784,))
x = Dense(64, activation='relu')(inputs)
x = Dense(64, activation='relu')(x)
predictions = Dense(10, activation='softmax')(x)
```

Introducing independent hyperparameters

The code above defines a fixed two-layer network in Keras. Unfortunately, this code commits to specific values for the number of units and the activation type. A natural first step is to be less specific about these hyperparameters by searching over the number of layers and activations of each layer. We defined a few simple helper functions that allow us to take a function that returns a Keras layer and wraps it in a DeepArchitect module.

See below for a minimal adaptation of the above example in DeepArchitect.

```
import deep_architect.core as co
import deep_architect.modules as mo
import deep_architect.hyperparameters as hp
import deep_architect.helpers.keras_support as hke
from deep_architect.searchers.common import random_specify, specify

D = hp.Discrete
wrap_search_space_fn = lambda fn: mo.SearchSpaceFactory(fn).get_search_space


def dense(h_units, h_activation):
    return hke.siso_keras_module_from_keras_layer_fn(Dense, {
        "units": h_units,
        "activation": h_activation
    })


def search_space0():
    return mo.siso_sequential([
        dense(D([32, 64, 128, 256]), D(["relu", "sigmoid"])),
        dense(D([32, 64, 128, 256]), D(["relu", "sigmoid"])),
        dense(D([10]), D(["softmax"]))
    ])


(inputs, outputs) = wrap_search_space_fn(search_space0)()
```

This code defines a search space where the nonlinearities and number of units are chosen from a set of values rather than being fixed upfront. In this case, the hyperparameters are independent for each of the modules. The search space captures possible values for these hyperparameters.

In DeepArchitect, we have implemented some auxiliary tools to visualize the search search as a graph.

```
import deep_architect.visualization as vi
vi.draw_graph(
    outputs,
    draw_module_hyperparameter_info=False,
    graph_name='graph0_first')
```

The connections between modules are fixed. In the construction of the search space, all function calls return dictionaries of inputs and outputs. Typically, we use a searcher, but in this example we will use a simple function from the search tools that randomly chooses values for all unsassigned independent hyperparameters in the search space.

In the graph, modules are represented by rectangles and hyperparameters are represented by ovals. An edge between two rectangles represents the output of a module going into an input of the other modules. An edge between a rectangle and a oval represents a dependency of a module on a hyperparameter. Check here for all search space visualizations generated in this tutorial.

```
import deep_architect.searchers.common as seco
vs = seco.random_specify(outputs)
x = Input(shape=(784,))
co.forward({inputs["in"]: x})
y = outputs["out"].val
print(vs)
```

`deep_architect.searchers.common.random_specify()` iterates over independent hyperparameters that have not yet been assigned a value and chooses a value uniformly at random from the set of possible values. After all hyperparameters have been assigned values, we have the following search space:

```
vi.draw_graph(
    outputs,
    draw_module_hyperparameter_info=False,
    graph_name='graph0_last')
```

Edges between hyperparameters and modules have been labeled with the values chosen for the hyperparameters. The graph transitions with each value assignment to an independent hyperparameter. We can visualize these graph transitions as a frame sequence:

```
(inputs, outputs) = wrap_search_space_fn(search_space0)()

vi.draw_graph_evolution(
    outputs,
    vs,
    '.',
    draw_module_hyperparameter_info=False,
    graph_name='graph0_evo')
```

We ask the reader to pay attention to how the edges connecting hyperparameters to modules change with each transition. This search space is very simple. This functionality is more insightful for more complex search spaces.

Sharing hyperparameters across modules

In the previous search space, the hyperparameter values were chosen independently for each of the layers. If we wished to tie hyperparameters across different parts of the search space, e.g., use the same nonlinearity for all modules, we would have to instantiate a single hyperparameter and use it in multiple places. Adapting the first search space to reflect this change is straightforward.

```
def search_space1():
    h_activation = D(["relu", "sigmoid"])
    return mo.siso_sequential([
        dense(D([32, 64, 128, 256]), h_activation),
        dense(D([32, 64, 128, 256]), h_activation),
        dense(D([10]), D(["softmax"]))
    ])


(inputs, outputs) = wrap_search_space_fn(search_space1)()
vi.draw_graph(
    outputs,
    draw_module_hyperparameter_info=False,
    graph_name='graph1_first')
```

Redrawing the initial graph for the search space, we see that that now there is a single hyperparameter associated to activations of all dense modules.

Expressing dependencies between hyperparameters

A dependent hyperparameters has its value assigned as a function of the values of other hyperparameters. We will adapt our running example by making the number of hidden units of the second layer of the network twice as large as the number of hidden units of the first layer. This allows us to naturally encode a more restricted search space.

```
def search_space2():
    h_activation = D(["relu", "sigmoid"])
```

(continues on next page)

(continued from previous page)

```
    h_units = D([32, 64, 128, 256])
    h_units_dep = co.DependentHyperparameter(lambda dh: 2 * dh["units"],
                                             {"units": h_units})

    return mo.siso_sequential([
        dense(h_units, h_activation),
        dense(h_units_dep, h_activation),
        dense(D([10]), D(["softmax"]))
    ])


(inputs, outputs) = wrap_search_space_fn(search_space2)()
vi.draw_graph(
    outputs,
    draw_module_hyperparameter_info=False,
    graph_name='graph2_first')
```

As we can see in the graph, there is an edge going from the independent hyperparameter to the hyperparameter that it depends on. Dependent hyperparameters can depend on other dependent hyperparameters, as long as there are no directed cycles.

See below for the graph transition with successive value assignments to hyperparameters.

```
vs = seco.random_specify(outputs)
(inputs, outputs) = wrap_search_space_fn(search_space2)()

vi.draw_graph_evolution(
    outputs,
    vs,
    '.',
    draw_module_hyperparameter_info=False,
    graph_name='graph2_evo')
```

A dependent hyperparameter is assigned a value as soon as the hyperparameters that it depends on have been assigned values.

Delaying sub-search space creation through substitution

We have talked about modules and hyperparameters. For hyperparameters, we distinguish between independent hyperparameters (hyperparameters whose value is set independently of any other hyperparameters), and dependent hyperparameters (hyperparameters whose value is computed as a function of the values of other hyperparameters). For modules, we distinguish between basic modules (modules that stay in place when all hyperparameters that the module depends on have been assigned values), and substitution modules (modules that disappear, giving rise to a new graph fragment in its place with other modules, when all hyperparameters that the module depends on have been assigned values).

So far, we have only concerned ourselves with basic modules (e.g., the dense module in the example search spaces above). Basic modules are used to represent computations, i.e., the module implements some well-defined computation after values for all the hyperparameters of the module and values for the inputs are available. By contrast, substitution modules encode structural transformations (they do not implement any computation) based on the values of their hyperparameters. Substitution modules are inspired by ideas of delayed evaluation from the programming languages literature.

We have implemented many structural transformations as substitution modules in DeepArchitect. Substitution modules are that they are independent of the underlying framework of the basic modules (i.e., they work without any

adaptation for Keras, Tensorflow, Scikit-Learn, or any other framework). Let us consider an example search space using a substitution module that either includes a submodule or not.

```
def search_space3():
    h_activation = D(["relu", "sigmoid"])
    h_units = D([32, 64, 128, 256])
    h_units_dep = co.DependentHyperparameter(lambda dh: 2 * dh["units"],
                                             {"units": h_units})
    h_opt = D([0, 1])

    return mo.siso_sequential([
        dense(h_units, h_activation),
        mo.siso_optional(lambda: dense(h_units_dep, h_activation), h_opt),
        dense(D([10]), D(["softmax"]))
    ])


(inputs, outputs) = wrap_search_space_fn(search_space3)()
```

The optional module takes a thunk (this terminology comes from programming languages) which returns a graph fragment (returned as a dictionary of input names to inputs and a dictionary of output names to outputs) which is called if the hyperparameter that determines if the thunk is to be called or not, takes the value "1" (i.e., the thunk is to be called, and the resulting graph fragment is to be included in the place of the substitution module). The visualization functionality is insightful in this case. Consider the graph evolution for a random sample from this search space.

```
vs = seco.random_specify(outputs)
(inputs, outputs) = wrap_search_space_fn(search_space3)()

vi.draw_graph_evolution(
    outputs,
    vs,
    '.',
    draw_module_hyperparameter_info=False,
    graph_name='graph3_evo')
```

Once the hyperparameter that the optional substitution module depends on is assigned a value, the substitution module disappears and is replaced by a graph fragment that depends on the hyperparameter value, i.e., if we decide to include it, the thunk is called returning a graph fragment; if we decide to not include it, an identity module (that passes the input to the output without changes) is substituted in its place.

Another simple substitution module is one that repeats a graph fragment in a serial connection. In this case, the substitution hyperparameter refers to how many times will the thunk returning a graph fragment will be called; all repetitions are connected in a serial connection.

```
def search_space4():
    h_activation = D(["relu", "sigmoid"])
    h_units = D([32, 64, 128, 256])
    h_units_dep = co.DependentHyperparameter(lambda dh: 2 * dh["units"],
                                             {"units": h_units})
    h_opt = D([0, 1])
    h_num_repeats = D([1, 2, 4])

    return mo.siso_sequential([
        mo.siso_repeat(lambda: dense(h_units, h_activation), h_num_repeats),
        mo.siso_optional(lambda: dense(h_units_dep, h_activation), h_opt),
        dense(D([10]), D(["softmax"]))
    ])
```

(continues on next page)

(continued from previous page)

```
(inputs, outputs) = wrap_search_space_fn(search_space4)()
```

In the search space above, the hyperparameter for the number of units of the dense modules inside the repeat share the same hyperparameter, i.e., all these modules will have the same number of units.

```
vs = seco.random_specify(outputs)
(inputs, outputs) = wrap_search_space_fn(search_space4)()

vi.draw_graph_evolution(
    outputs,
    vs,
    '.',
    draw_module_hyperparameter_info=False,
    graph_name='graph4_evo')
```

In the graph evolution, we see that once we assign a value to the hyperparameter for the number of repetitions of the graph fragment returned by the thunk, a graph fragment with the serial connection of those many repetitions is substituted in its place. These example search spaces, along with their visualizations, should give the reader a sense about what structural decisions are expressible in DeepArchitect.

Substitution modules can be used in any place a module is needed, e.g., they can nested. For example, consider the following example

```
def search_space5():
    h_activation = D(["relu", "sigmoid"])
    h_units = D([32, 64, 128, 256])
    h_units_dep = co.DependentHyperparameter(lambda dh: 2 * dh["units"],
                                             {"units": h_units})
    h_opt = D([0, 1])
    h_num_repeats = D([1, 2, 4])

    return mo.siso_sequential([
        mo.siso_repeat(lambda: dense(h_units, h_activation), h_num_repeats),
        mo.siso_optional(
            lambda: mo.siso_repeat(lambda: dense(h_units_dep, h_activation),
                                   h_num_repeats), h_opt),
        dense(D([10]), D(["softmax"]))
    ])


(inputs, outputs) = wrap_search_space_fn(search_space5)()
```

Take one minute to think about the graph transitions for this search space; then run the code below to generate the actual visualization.

```
vs = seco.random_specify(outputs)
(inputs, outputs) = wrap_search_space_fn(search_space5)()
vi.draw_graph_evolution(
    outputs,
    vs,
    '.',
    draw_module_hyperparameter_info=False,
    graph_name='graph5_evo')
```

By using basic modules, substitution modules, independent hyperparameters, and dependent hyperparameters we are

able to represent a large variety of search spaces in a compact and natural manner. As the reader becomes more comfortable with these concepts, it should become progressively easier to encode search spaces and appreciate the expressivity and reusability of the language.

Minor details

We cover additional details not yet discussed in the tutorial.

Search space wrapper: Throughout the instantiation of the various search spaces, we have seen this call to `wrap_search_space_fn`, which internally uses *deep_architect.modules.SearchSpaceFactory*. *deep_architect.modules.SearchSpaceFactory* manages the global scope and buffers the search space to make sure that there are no substitution modules with unconnected inputs or outputs (i.e., at the border of the search space).

Scope: We use the global the scope to assign unique names to the elements that show up in the search space (currently, modules, hyperparameters, inputs, and outputs). Every time a module, hyperparameter, input, or output is created, we use the scope to assign a unique name to it. Every time that we want to start the search from scratch with a new search space, we should clear the scope to avoid keeping the names and objects from the previous samples around. In most cases, the user does not have to be concerned with the scope as *deep_architect.modules.SearchSpaceFactory* can be used to handle the global scope.

Details about substitution modules: The search space cannot have substitution modules at its border as effectively substitution modules disappear once the substitution is done, and therefore references to the module and its inputs and outputs become invalid. *deep_architect.modules.SearchSpaceFactory* creates and connects extra identity modules, which are basic modules (as opposed to substitution modules), before (in the case of inputs) or after (in the case of outputs) for each input and output belonging to a substitution module at the border of the search space.

Auxiliary functions: Besides basic modules and substitution modules, we also use several auxiliary functions for easily arranging graph fragments in different ways. These auxiliary function often do not create new modules, but use graph fragments or functions that return graph fragments to create a new graph fragment by using the arguments in a certain way. An example of a function of this type is *deep_architect.modules.siso_sequential()*, which just connects the graph fragments (expressed as a dictionary of inputs and a dictionary of outputs), in a serial connection, which just require us to connect inputs and outputs of the fragments passed as arguments to the function. Similarly to substitution modules, these auxiliary functions are framework independent as they only rely on properties of the module API. Using and defining auxiliary functions will help the user have a more effective and pleasant experience with DeepArchitect. Auxiliary functions are very useful to construct larger search spaces made of complex arrangements of smaller search spaces.

Supporting other frameworks: Basic modules are the only concepts that need to be specialized to the new framework. We recommend reading deep_architect/core.py for extensive information about basic DeepArchitect API components. This code is the basis of DeepArchitect and has been extensively commented. Everything in deep_architect/core.py is framework-independent. To better understand substitution modules and how they are implemented, read deep_architect/modules.py . We also point the reader to the tutorial about supporting new frameworks.

Rerouting: While we have not covered rerouting in this tutorial, it is reasonably straightforward to think about how to implement rerouting with, either as a substitution module or a basic module. For example, for a rerouting operation that takes k inputs and k outputs, and does a permutation of the inputs and outputs based on the value of an hyperparameter, if we implement this operation using a basic module, the basic module has to implement the chosen permutation when forward is called. If a substitution module is used instead, the module disappears once the value for the hyperparameter is chosen and the result of rerouting shows up in its place. After the user becomes proficient with the ideas of basic and substitution modules, the user will realize that oftentimes there are multiple ways of expressing the same search space.

Concluding remarks

In this tutorial, we only covered basic functionality to encode search spaces over architectures. For learning more about the framework, please read more tutorials on aspects or use cases which you may find important and/or hard to understand. DeepArchitect is composed of many other components such as search, evaluation, logging, visualization and multiworking, so please read additional tutorials if you wish to become familiar with these other aspects.

2.1.2 Implementing new modules

In this tutorial, we will cover the implementation of new modules in DeepArchitect. We will use Keras to discuss the implementation of new modules. These aspects are similar across frameworks. See the corresponding examples and tutorials for discussions of how to support other frameworks.

Starting with the framework helper module

The starting point for implementing a new module is the the helper for the framework that we are using (in this case, Keras):

```
from __future__ import absolute_import
import deep_architect.core as co


class KerasModule(co.Module):
    """Class for taking Keras code and wrapping it in a DeepArchitect module.

    This class subclasses :class:`deep_architect.core.Module` as therefore inherits␣
↪all
    the functionality associated to it (e.g., keeping track of inputs, outputs,
    and hyperparameters). It also enables to do the compile and forward
    operations for these types of modules once a module is fully specified,
    i.e., once all the hyperparameters have been chosen.

    The compile operation in this case creates all the variables used for the
    fragment of the computational graph associated to this module.
    The forward operation takes the variables that were created in the compile
    operation and constructs the actual computational graph fragment associated
    to this module.

    .. note::
        This module is abstract, meaning that it does not actually implement
        any particular Keras computation. It simply wraps Keras
        functionality in a DeepArchitect module. The instantiation of the Keras
        variables is taken care by the `compile_fn` function that takes a two
        dictionaries, one of inputs and another one of outputs, and
        returns another function that takes a dictionary of inputs and creates
        the computational graph. This functionality makes extensive use of closures.

        The keys of the dictionaries that are passed to the compile
        and forward function match the names of the inputs and hyperparameters
        respectively. The dictionary returned by the forward function has keys
        equal to the names of the outputs.

        This implementation is very similar to the implementation of the Tensorflow
        helper :class:`deep_architect.helpers.tensorflow_support.TensorflowModule`.
```

(continues on next page)

(continued from previous page)

```
    Args:
        name (str): Name of the module
        name_to_hyperp (dict[str,deep_architect.core.Hyperparameter]): Dictionary of
            hyperparameters that the model depends on. The keys are the local
            names of the hyperparameters.
        compile_fn ((dict[str,object], dict[str,object]) -> (dict[str,object] ->␣
↪dict[str,object])):
            The first function takes two dictionaries with
            keys corresponding to `input_names` and `output_names` and returns
            a function that takes a dictionary with keys corresponding to
            `input_names` and returns a dictionary with keys corresponding
            to `output_names`. The first function may also return
            two additional dictionaries mapping Tensorflow placeholders to the
            values that they will take during training and test.
        input_names (list[str]): List of names for the inputs.
        output_names (list[str]): List of names for the outputs.
        scope (deep_architect.core.Scope, optional): Scope where the module will be
            registered.
    """

    def __init__(self,
                 name,
                 name_to_hyperp,
                 compile_fn,
                 input_names,
                 output_names,
                 scope=None):
        co.Module.__init__(self, scope, name)

        self._register(input_names, output_names, name_to_hyperp)
        self._compile_fn = compile_fn

    def _compile(self):
        input_name_to_val = self._get_input_values()
        hyperp_name_to_val = self._get_hyperp_values()

        self._fn = self._compile_fn(input_name_to_val, hyperp_name_to_val)

    def _forward(self):
        input_name_to_val = self._get_input_values()
        output_name_to_val = self._fn(input_name_to_val)
        self._set_output_values(output_name_to_val)

    def _update(self):
        pass
```

With this helper we can instantiate modules by passing values for the name of the module, the names of the inputs and outputs, the hyperparameters, and the compile function. Compile captures most of the functionality for the specific module. Calling the compile function passed as argument returns a function (the forward function). _compile is called only once. It may be informative to revisit the definition of a general module in core.py.

Instances of this class are sufficient for most use cases that we have encountered, but there may exist special cases where inheriting from this class and implementing _compile and _forward directly may be necessary.

Working with inputs and outputs instead of modules

When writing down search spaces, we work mostly with inputs and outputs, so the following auxiliary function is useful, albeit a bit redundant.

```
def keras_module(name,
                 compile_fn,
                 name_to_hyperp,
                 input_names,
                 output_names,
                 scope=None):
    return KerasModule(name, name_to_hyperp, compile_fn, input_names,
                       output_names, scope).get_io()
```

See below for a typical implementation of a module using these auxiliary functions:

```
from keras.layers import Conv2D, BatchNormalization


def conv_relu_batch_norm(h_filters, h_kernel_size, h_strides):

    def compile_fn(di, dh):
        m_conv = Conv2D(
            dh["filters"], dh["kernel_size"], dh["strides"], padding='same')
        m_bn = BatchNormalization()

        def forward_fn(di):
            return {"out": m_bn(m_conv(di["in"]))}

        return forward_fn

    return keras_module('ConvReLUBatchNorm', compile_fn, {
        "filters": h_filters,
        "kernel_size": h_kernel_size,
        'strides': h_strides
    }, ["in"], ["out"])
```

The forward function is defined via a closure. When compile is called, we have specific values for the module inputs (which in this example, are Keras tensor nodes). We can interact with these objects during compilation (e.g., look up dimensions for the input tensors). The compile function is called with a dictionary of inputs (whose keys are input names and whose values are input values) and a dictionary of outputs (whose keys are hyperparameter names and whose values are hyperparameter values). The forward function is called with a dictionary of input values. Values for the hyperparameters are accessible (due to being in the closure), but they are often not needed in the forward function.

Simplifications for single-input single-output modules

While the above definition is a bit verbose, we expect it to be clear. We introduce a additional functions to make the creation of modules less verbose. For example, we are often dealing with single-input single-output modules, so we define the following function:

```
def siso_keras_module(name, compile_fn, name_to_hyperp, scope=None):
    return KerasModule(name, name_to_hyperp, compile_fn, ['in'], ['out'],
                       scope).get_io()
```

This saves us writing the names of the inputs and outputs for the single-input single-output case. As the reader becomes familiar with DeepArchitect, the reader will notice that we use in/out names for single-input/single-output modules

and in0, in1, .../out0, out1, ... for modules that often have multiple inputs/outputs. These names are arbitrary and can be chosen differently.

Using this function, the above example would be similar except that we would not need to name the input and output explicitly.

```
def conv_relu_batch_norm(h_filters, h_kernel_size, h_strides):

    def compile_fn(di, dh):
        m_conv = Conv2D(
            dh["filters"], dh["kernel_size"], dh["strides"], padding='same')
        m_bn = BatchNormalization()

        def forward_fn(di):
            return {"out": m_bn(m_conv(di["in"]))}

        return forward_fn

    return siso_keras_module('ConvReLUBatchNorm', compile_fn, {
        "filters": h_filters,
        "kernel_size": h_kernel_size,
        'strides': h_strides
    })
```

Easily creating modules from framework functions

Another useful auxiliary function creates a module from a function (e.g., most functions in keras.layers).

```
def siso_keras_module_from_keras_layer_fn(layer_fn,
                                          name_to_hyperp,
                                          scope=None,
                                          name=None):

    def compile_fn(di, dh):
        m = layer_fn(**dh)

        def forward_fn(di):
            return {"out": m(di["in"])}

        return forward_fn

    if name is None:
        name = layer_fn.__name__

    return siso_keras_module(name, compile_fn, name_to_hyperp, scope)
```

This function is convenient for functions that return a single-input single-output Keras module. For example, to get a convolutional module, we do

```
def conv2d(h_filters, h_kernel_size):
    return siso_keras_module_from_keras_layer_fn(Conv2D, {
        "filters": h_filters,
        "kernel_size": h_kernel_size
    })
```

If additionally, we would like to set some attributes to fixed values and have other ones defined through hyperparameters, we can do

```
def conv2d(h_filters, h_kernel_size):
    fn = lambda filters, kernel_size: Conv2D(
        filters, kernel_size, padding='same')
    return siso_keras_module_from_keras_layer_fn(
        fn, {
            "filters": h_filters,
            "kernel_size": h_kernel_size
        }, name="Conv2D")
```

Implementing new substitution modules

So far, we covered how can we easily implement new modules in Keras. These aspects transfer mostly without changes across frameworks. Examples correspond to modules that implement computation. We will now look at modules whose purpose is not to implement computation, but to perform a structural transformation based on the value of its hyperparameters. We call these modules substitution modules.

Substitution modules are framework independent. Only basic modules need to be reimplemented when porting search spaces from one framework to a different one. Substitution modules and auxiliary functions work across frameworks. As a result, a large amount of code is reusable between frameworks. Basic modules are often simple to implement, as most of the complexity of the search space implementation is contained in auxiliary functions and substitution modules.

First, consider the definition of a substitution module.

```
class SubstitutionModule(co.Module):
    """Substitution modules are replaced by other modules when the all the
    hyperparameters that the module depends on are specified.

    Substitution modules implement a form of delayed evaluation.
    The main component of a substitution module is the substitution function.
    When called, this function returns a dictionary of inputs and a dictionary
    of outputs. These outputs and inputs are used in the place the substitution
    module is in. The substitution module effectively disappears from the
    network after the substitution operation is done.
    Substitution modules are used to implement many other modules,
    e.g., :func:`mimo_or`, :func:`siso_optional`, and :func:`siso_repeat`.

    Args:
        name (str): Name used to derive an unique name for the module.
        name_to_hyperp (dict[str, deep_architect.core.Hyperparameter]): Dictionary of
            name to hyperparameters that are needed for the substitution function.
            The names of the hyperparameters should be in correspondence to the
            name of the arguments of the substitution function.
        substitution_fn ((...) -> (dict[str, deep_architect.core.Input], dict[str,␣
↪deep_architect.core.Output]):
            Function that is called with the values of hyperparameters and
            returns the inputs and the outputs of the
            network fragment to put in the place the substitution module
            currently is.
        input_names (list[str]): List of the input names of the substitution module.
        output_name (list[str]): List of the output names of the substitution module.
        scope ((deep_architect.core.Scope, optional)) Scope in which the module will␣
↪be
            registered. If none is given, uses the default scope.
        allow_input_subset (bool): If true, allows the substitution function to
            return a strict subset of the names of the inputs existing before the
```

(continues on next page)

(continued from previous page)

```
            substitution. Otherwise, the dictionary of inputs returned by the
            substitution function must contain exactly the same input names.
        allow_output_subset (bool): If true, allows the substitution function to
            return a strict subset of the names of the outputs existing before the
            substitution. Otherwise, the dictionary of outputs returned by the
            substitution function must contain exactly the same output names.
    """

    def __init__(self,
                 name,
                 name_to_hyperp,
                 substitution_fn,
                 input_names,
                 output_names,
                 scope=None,
                 allow_input_subset=False,
                 allow_output_subset=False):
        co.Module.__init__(self, scope, name)
        self.allow_input_subset = allow_input_subset
        self.allow_output_subset = allow_output_subset

        self._register(input_names, output_names, name_to_hyperp)
        self._substitution_fn = substitution_fn
        self._is_done = False
        self._update()

    def _update(self):
        """Implements the substitution operation.

        When all the hyperparameters that the module depends on are specified,
        the substitution operation is triggered, and the substitution operation
        is done.
        """
        if (not self._is_done) and all(
                h.has_value_assigned() for h in self.hyperps.values()):
            dh = {name: h.get_value() for name, h in self.hyperps.items()}
            new_inputs, new_outputs = self._substitution_fn(dh)

            # test for checking that the inputs and outputs returned by the
            # substitution function are valid.
            if self.allow_input_subset:
                assert len(self.inputs) <= len(new_inputs) and all(
                    name in self.inputs for name in new_inputs)
            else:
                assert len(self.inputs) == len(new_inputs) and all(
                    name in self.inputs for name in new_inputs)

            if self.allow_output_subset:
                assert len(self.outputs) <= len(new_outputs) and all(
                    name in self.outputs for name in new_outputs)
            else:
                assert len(self.outputs) == len(new_outputs) and all(
                    name in self.outputs for name in new_outputs)

            # performing the substitution.
            for name, old_ix in self.inputs.items():
                old_ix = self.inputs[name]
```

(continues on next page)

(continued from previous page)

```
                if name in new_inputs:
                    new_ix = new_inputs[name]
                    if old_ix.is_connected():
                        old_ix.reroute_connected_output(new_ix)
                    self.inputs[name] = new_ix
                else:
                    if old_ix.is_connected():
                        old_ix.disconnect()

            for name, old_ox in self.outputs.items():
                old_ox = self.outputs[name]
                if name in new_outputs:
                    new_ox = new_outputs[name]
                    if old_ox.is_connected():
                        old_ox.reroute_all_connected_inputs(new_ox)
                    self.outputs[name] = new_ox
                else:
                    if old_ox.is_connected():
                        old_ox.disconnect_all()

            self._is_done = True
```

A substitution module has hyperparameters and a substitution function that returns a graph fragment to replace the substitution module. Update is called each time one of the hyperparameters of the substitution module is assigned a value. The substitution is performed when all hyperparameter have been assigned values.

Substitution modules disappear from the graph when the substitution is performed. The substitution function may itself return a graph fragment containing substitution modules. When there are only basic modules left and all the hyperparameters have been assigned values, the search space is fully specified and we can call the compile and forward functions for each of the basic modules in in it.

Substitution modules delay the choice of a structural property of the search space until some hyperparameters are assigned values. These are very helpful to encode complex and expressive search spaces. We have defined a few useful substitution modules in deep_architect/modules.py. Similar to the basic module definition that we looked above, it is more convenient to deal with the dictionaries of inputs and the dictionaries of outputs than with the modules, so we define this function

```
def substitution_module(name,
                        name_to_hyperp,
                        substitution_fn,
                        input_names,
                        output_names,
                        scope,
                        allow_input_subset=False,
                        allow_output_subset=False,
                        unpack_kwargs=True):
    """Same as the substitution module, but directly works with the dictionaries of
    inputs and outputs.

    A dictionary with inputs and a dictionary with outputs is the preferred way
    of dealing with modules when creating search spaces. Using inputs and outputs
    directly instead of modules allows us to return graphs in the
    substitution function. In this case, returning a graph resulting of the
    connection of multiple modules is entirely transparent to the substitution
    function.
```

(continues on next page)

(continued from previous page)

```
    See also: :class:`deep_architect.modules.SubstitutionModule`.

    Args:
        name (str): Name used to derive an unique name for the module.
        name_to_hyperp (dict[str, deep_architect.core.Hyperparameter]): Dictionary of
            name to hyperparameters that are needed for the substitution function.
            The names of the hyperparameters should be in correspondence to the
            name of the arguments of the substitution function.
        substitution_fn ((...) -> (dict[str, deep_architect.core.Input], dict[str,␣
↪deep_architect.core.Output]):
            Function that is called with the values of hyperparameters and
            values of inputs and returns the inputs and the outputs of the
            network fragment to put in the place the substitution module
            currently is.
        input_names (list[str]): List of the input names of the substitution module.
        output_name (list[str]): List of the output names of the substitution module.
        scope (deep_architect.core.Scope): Scope in which the module will be␣
↪registered.

    Returns:
        (dict[str,deep_architect.core.Input], dict[str,deep_architect.core.Output]):
            Tuple with dictionaries with the inputs and outputs of the module.
    """
    return SubstitutionModule(
        name,
        name_to_hyperp,
        substitution_fn,
        input_names,
        output_names,
        scope,
        allow_input_subset=allow_input_subset,
        allow_output_subset=allow_output_subset,
        unpack_kwargs=unpack_kwargs).get_io()
```

We will now look at two specific examples of substitution modules. The or substitution module. is one of the simplest, but also most useful substitution modules:

```
def mimo_or(fn_lst, h_or, input_names, output_names, scope=None, name=None):
    """Implements an or substitution operation.

    The hyperparameter takes values that are valid indices for the list of
    possible substitution functions. The set of keys of the dictionaries of
    inputs and outputs returned by the substitution functions have to be
    the same as the set of input names and output names, respectively. The
    substitution function chosen is used to replace the current substitution
    module, with connections changed appropriately.

    .. note::
        The current implementation also works if ``fn_lst`` is an indexable
        object (e.g., a dictionary), and the ``h_or`` takes values that
        are valid indices for the indexable (e.g., valid keys for the dictionary).

    Args:
        fn_lst (list[() -> (dict[str,deep_architect.core.Input], dict[str,deep_
↪architect.core.Output])]):
            List of possible substitution functions.
```

(continues on next page)

(continued from previous page)

```
        h_or (deep_architect.core.Hyperparameter): Hyperparameter that chooses which
            function in the list is called to do the substitution.
        input_names (list[str]): List of inputs names of the module.
        output_names (list[str]): List of the output names of the module.
        scope (deep_architect.core.Scope, optional): Scope in which the module will be
            registered. If none is given, uses the default scope.
        name (str, optional): Name used to derive an unique name for the
            module. If none is given, uses the class name to derive
            the name.

    Returns:
        (dict[str,deep_architect.core.Input], dict[str,deep_architect.core.Output]):
            Tuple with dictionaries with the inputs and outputs of the
            substitution module.
    """

    def substitution_fn(idx):
        return fn_lst[idx]()

    return substitution_module(
        _get_name(name, "Or"), {'idx': h_or}, substitution_fn, input_names,
        output_names, scope)
```

The implementation is extremely short. This module has a single hyperparameter that chooses which function in the list (or dictionary) to call. Each of the functions in the list returns a dictionary of inputs and a dictionary of outputs when called.

```
def dnn_cell(h_num_hidden, h_nonlin_name, h_swap, h_opt_drop, h_opt_bn,
             h_drop_keep_prob):
    return mo.siso_sequential([
        affine_simplified(h_num_hidden),
        nonlinearity(h_nonlin_name),
        mo.siso_permutation([
            lambda: mo.siso_optional(lambda: dropout(h_drop_keep_prob),
                                     h_opt_drop),
            lambda: mo.siso_optional(batch_normalization, h_opt_bn),
        ], h_swap)
    ])
```

Optional is a special case of a substitution module. If the hyperparameter is such that the function is to be used, then the function (in the example above, a lambda function) is called. Otherwise, an identity module that passes the input unchanged to the output is used.

```
def siso_optional(fn, h_opt, scope=None, name=None):
    """Substitution module that determines to include or not the search
    space returned by `fn`.

    The hyperparameter takes boolean values (or equivalent integer zero and one
    values). If the hyperparameter takes the value ``False``, the input is simply
    put in the output. If the hyperparameter takes the value ``True``, the search
    space is instantiated by calling `fn`, and the substitution module is
    replaced by it.

    Args:
        fn (() -> (dict[str,deep_architect.core.Input], dict[str,deep_architect.core.
↪Output])):
```

(continues on next page)

(continued from previous page)

```
            Function returning a graph fragment corresponding to a sub-search space.
        h_opt (deep_architect.core.Hyperparameter): Hyperparameter for whether to
            include the sub-search space or not.
        scope (deep_architect.core.Scope, optional): Scope in which the module will be
            registered. If none is given, uses the default scope.
        name (str, optional): Name used to derive an unique name for the
            module. If none is given, uses the class name to derive the name.

    Returns:
        (dict[str,deep_architect.core.Input], dict[str,deep_architect.core.Output]):
            Tuple with dictionaries with the inputs and outputs of the
            substitution module.
    """

    def substitution_fn(opt):
        return fn() if opt else identity()

    return substitution_module(
        _get_name(name, "SISOOptional"), {'opt': h_opt}, substitution_fn,
        ['in'], ['out'], scope)
```

An example of a complex substitution module

Let us now look at a more complex use of a custom substitution module.

```
def motif(submotif_fn, num_nodes):
    assert num_nodes >= 1

    def substitution_fn(dh):
        print dh
        node_id_to_node_ids_used = {i: [i - 1] for i in range(1, num_nodes)}
        for name, v in dh.items():
            if v:
                d = ut.json_string_to_json_object(name)
                i = d["node_id"]
                node_ids_used = node_id_to_node_ids_used[i]
                j = d["in_node_id"]
                node_ids_used.append(j)
        for i in range(1, num_nodes):
            node_id_to_node_ids_used[i] = sorted(node_id_to_node_ids_used[i])

        (inputs, outputs) = mo.identity()
        node_id_to_outputs = [outputs]
        in_inputs = inputs
        for i in range(1, num_nodes):
            node_ids_used = node_id_to_node_ids_used[i]
            num_edges = len(node_ids_used)

            outputs_lst = []
            for j in node_ids_used:
                inputs, outputs = submotif_fn()
                j_outputs = node_id_to_outputs[j]
                inputs["in"].connect(j_outputs["out"])
                outputs_lst.append(outputs)
```

(continues on next page)

(continued from previous page)

```
            # if necessary, concatenate the results going into a node
            if num_edges > 1:
                c_inputs, c_outputs = combine_with_concat(num_edges)
                for idx, outputs in enumerate(outputs_lst):
                    c_inputs["in%d" % idx].connect(outputs["out"])
            else:
                c_outputs = outputs_lst[0]
            node_id_to_outputs.append(c_outputs)

        out_outputs = node_id_to_outputs[-1]
        return in_inputs, out_outputs

    name_to_hyperp = {
        ut.json_object_to_json_string({
            "node_id": i,
            "in_node_id": j
        }): D([0, 1]) for i in range(1, num_nodes) for j in range(i - 1)
    }
    return mo.substitution_module(
        "Motif", name_to_hyperp, substitution_fn, ["in"], ["out"], scope=None)
```

This substitution module implements the notion of a motif inspired by this paper. This substitution module delays the creation of the motif structure until hyperparameters determining the connections of the motif are assigned values. The notion of a motif defined in the paper is recursive. The motif function takes a submotif function.

Concluding remarks

This concludes our discussion about how to implement new modules in a specific framework that the reader is working with. We point the reader to the new_frameworks tutorial for learning about how to support a new framework by specializing the module class and to the search space constructs tutorials for a more in-depth coverage of how search spaces can be created by interconnecting modules.

2.1.3 Implementing new searchers

Searchers determine the policy used to search a search space. Search spaces encode sets of architectures to be considered.

Searcher API

We have a very simple API for a searcher, composed of two main methods and two auxiliary methods.

```
class Searcher:
    """Abstract base class from which new searchers should inherit from.

    A search takes a function that when called returns a new search space, i.e.,
    a search space where all the hyperparameters have not being specified.
    Searchers essentially sample a sequence of models in the search space by
    specifying the hyperparameters sequentially. After the sampled architecture
    has been evaluated somehow, the state of the searcher can be updated with
    the performance information, guaranteeing that future architectures
    are sampled from the search space in a more informed manner.
```

(continues on next page)

(continued from previous page)

```
    Args:
        search_space_fn (() -> (dict[str,deep_architect.core.Input], dict[str,deep_
↪architect.core.Output], dict[str,deep_architect.core.Hyperparameter])):
            Search space function that when called returns a dictionary of
            inputs, dictionary of outputs, and dictionary of hyperparameters
            encoding the search space from which models can be sampled by
            specifying all hyperparameters (i.e., both those arising in the
            graph part and those in the dictionary of hyperparameters).
    """

    def __init__(self, search_space_fn):
        self.search_space_fn = search_space_fn

    def sample(self):
        """Returns a model from the search space.

        Models are encoded via a dictionary of inputs, a dictionary of outputs,
        and a dictionary of hyperparameters. The forward computation for the
        model can then be done as all values for the hyperparameters have been
        chosen.

        Returns:
            (dict[str, deep_architect.core.Input], dict[str, deep_architect.core.
↪Output], dict[str, deep_architect.core.Hyperparameter], list[object], dict[str,␣
↪object]):
                Tuple encoding the model sampled from the search space.
                The positional arguments have the following semantics:
                1: Dictionary of names to inputs of the model.
                2: Dictionary of names to outputs of the model.
                3: List with list of values that can be to replay the sequence
                of values assigned to the hyperparameters, and therefore,
                reproduce, given the search space, the model sampled.
                4: Searcher evaluation token that is sufficient for the searcher
                to update its state when combined with the results of the
                evaluation.
        """
        raise NotImplementedError

    def update(self, val, searcher_eval_token):
        """Updates the state of the searcher based on the searcher token
        for a particular evaluation and the results of the evaluation.

        Args:
            val (object): Result of the evaluation to use to update the state of the␣
↪searcher.
            searcher_eval_token (dict[str, object]): Searcher evaluation token
                that is sufficient for the searcher to update its state when
                combined with the results of the evaluation.
        """
        raise NotImplementedError
```

Implementing a new searcher is done by inheriting from this class. A searcher is initialized with a function that returns a search space, which is simply a dictionary of inputs and dictionary of outputs. The sample method is used to pick an architecture from this search space. The architecture returned is generated by assigning values to all the independent hyperparameters in the search space, until no unassigned independent hyperparameters are left.

Sample returns the inputs and outputs, which encode the architecture. The other two returned elements deserve a bit

more explanation. The third element returned by sample is the list of hyperparameter values that led to the returned architecture. If we call the search space function, iterate over the unassigned independent hyperparameters, and assign the values from this list in order, we will obtain back the same architecture. The combination of the search space function and this list of hyperparameters values encodes the architecture sampled.

The fourth element returned is what we call a searcher evaluation token. This token is used to keep any information necessary for the searcher to update its state once the result of evaluating the sample architecture is passed back to the searcher in the call to update. This token is especially useful when we have multiple workers being serviced by a searcher. In this case, the results of architecture evaluations may arrive in a different order than they were relayed to the workers. The searcher evaluation token allows the searcher to identify the architecture for which the results correspond to.

Finally, we look at the update function of the searcher. The update function takes the results of the evaluation and the searcher evaluation token (which allows the searcher to identify which architecture the results refer to) and updates the state of the searcher with this new information. Updates to the searcher change the state of the searcher and therefore, the behavior of the searcher.

The other two searcher auxiliary functions (that we have ommitted) are `save_state` and `load_state`, which allows us to save and load the state of the searcher to disk. This is especially useful for long running searches that require multiple times due to job length limits or hardware issues.

We will now go over a two different searchers to help the reader ground the discussion.

Random searcher

The simplest possible searcher is a random searcher, which assigns a random value to each of the unassigned hyperparameters.

```
from deep_architect.searchers.common import random_specify, Searcher


class RandomSearcher(Searcher):

    def __init__(self, search_space_fn):
        Searcher.__init__(self, search_space_fn)

    def sample(self):
        inputs, outputs = self.search_space_fn()
        vs = random_specify(outputs)
        return inputs, outputs, vs, {}

    def update(self, val, searcher_eval_token):
        pass
```

The implementation of this searcher is very short. It uses the implementation of random_specify, which is also very compact. We copy it here for reference.

```
def random_specify_hyperparameter(hyperp):
    """Choose a random value for an unspecified hyperparameter.

    The hyperparameter becomes specified after the call.

    hyperp (deep_architect.core.Hyperparameter): Hyperparameter to specify.
    """
    assert not hyperp.has_value_assigned()

    if isinstance(hyperp, hp.Discrete):
```

(continues on next page)

(continued from previous page)

```
        v = hyperp.vs[np.random.randint(len(hyperp.vs))]
        hyperp.assign_value(v)
    else:
        raise ValueError
    return v


def random_specify(output_lst):
    """Chooses random values to all the unspecified hyperparameters.

    The hyperparameters will be specified after this call, meaning that the
    compile and forward functionalities will be available for being called.

    Args:
        output_lst (list[deep_architect.core.Output]): List of output which by being
            traversed back will reach all the modules in the search space, and
            correspondingly all the current unspecified hyperparameters of the
            search space.
    """
    hyperp_value_lst = []
    for h in co.unassigned_independent_hyperparameter_iterator(output_lst):
        v = random_specify_hyperparameter(h)
        hyperp_value_lst.append(v)
    return hyperp_value_lst
```

These are the two main auxiliary functions to randomly specify hyperparameters and to pick a random architecture from the search space by picking values for all the hyperparameters independently at random. These are concise and self-explanatory.

SMBO searcher

Let us now see a SMBO searcher, which is more complex than the random searcher. We copy the implementation here for ease of reference.

```
from deep_architect.searchers.common import random_specify, specify, Searcher
from deep_architect.surrogates.common import extract_features
import numpy as np


class SMBOSearcher(Searcher):

    def __init__(self, search_space_fn, surrogate_model, num_samples, eps_prob):
        Searcher.__init__(self, search_space_fn)
        self.surr_model = surrogate_model
        self.num_samples = num_samples
        self.eps_prob = eps_prob

    def sample(self):
        if np.random.rand() < self.eps_prob:
            inputs, outputs = self.search_space_fn()
            best_vs = random_specify(outputs)
        else:
            best_model = None
            best_vs = None
            best_score = -np.inf
```

(continues on next page)

(continued from previous page)

```
            for _ in range(self.num_samples):
                inputs, outputs = self.search_space_fn()
                vs = random_specify(outputs)

                feats = extract_features(inputs, outputs)
                score = self.surr_model.eval(feats)
                if score > best_score:
                    best_model = (inputs, outputs)
                    best_vs = vs
                    best_score = score

            inputs, outputs = best_model

        searcher_eval_token = {'vs': best_vs}
        return inputs, outputs, best_vs, searcher_eval_token

    def update(self, val, searcher_eval_token):
        (inputs, outputs) = self.search_space_fn()
        specify(outputs, searcher_eval_token['vs'])
        feats = extract_features(inputs, outputs)
        self.surr_model.update(val, feats)
```

This searcher can be found in searchers/smbo_random.py. A SMBO (surrogate model based optimization) searcher relies on a surrogate function on the space of architectures that gives us a performance estimate or a score.

An architecture from the search space is sampled by optimizing the surrogate function. In the implementation above, the optimization of the surrogate function is done by sampling a number of random architectures from the search space, evaluating the surrogate function for each of them, and picking the best one. Additionally, we pick an architecture at random with fixed probability.

In this case, updates to the searcher correspond to updates to the surrogate function with observed results. The searcher policy hopefully improves as the surrogate function becomes more accurate as we get more data for the search space. The API definition for a surrogate function can be found in surrogates/common.py.

Concluding remarks

Implementing a new searcher amounts to implementing the sample and update methods for it. We point the reader to the searchers folder for more examples. There is a single searcher per file. We very much welcome searcher contributions. If you would like to contribute with a search algorithm that you developed, please write a issue to discuss the implementation.

2.1.4 Supporting new frameworks

Supporting a framework in DeepArchitect requires the specialization of the module definition for that particular framework. For the frameworks that we currently support, the necessary changes across frameworks are minimal resulting mostly from framework idiosyncrasies, e.g., what information is needed to create a computational graph.

DeepArchitect is not limited to the frameworks that we currently support. Aside from module specialization, most of the other code in DeepArchitect is general and can be reused without changes across frameworks, e.g., searchers, logging, and visualization. We will walk the reader over the implementation of some of the helpers for the frameworks that are currently supported.

Specializing the module class

The main module functions to look at are:

```
# def _compile(self):
#     """Compile operation for the module.

#     Called once when all the hyperparameters that the module depends on,
#     and the other hyperparameters of the search space are specified.
#     See also: :meth:`_forward`.
#     """
#     raise NotImplementedError

# def _forward(self):
#     """Forward operation for the module.

#     Called once the compile operation has been called. See also: :meth:`_compile`.
#     """
#     raise NotImplementedError

# def forward(self):
#     """The forward computation done by the module is decomposed into
#     :meth:`_compile` and :meth:`_forward`.

#     Compile can be thought as creating the parameters of the module (done
#     once). Forward can be thought as using the parameters of the module to
#     do the specific computation implemented by the module on some specific
#     data (done multiple times).

#     This function can only called after the module and the other modules in
#     the search space are fully specified. See also: :func:`forward`.
#     """
#     if not self._is_compiled:
#         self._compile()
#         self._is_compiled = True
#     self._forward()
```

After all hyperparameters for a search space are specified, we can compile its modules. After all the independent hyperparameters have been assigned values, all substitution modules will have been disappeared and only basic modules will be in place. We can then execute the network. This can be seen in *deep_architect.core.Module.forward()*. When `forward` is called, `_compile` is called once and then `forward` is called right after. Compilation can be used to instantiate any state that is used by the module for the forward computation, e.g., creation of parameters. After compilation, `forward` can be called multiple times to perform the computation repeatedly.

Let us look in detail at concrete examples for frameworks that are currently supported in DeepArchitect.

Keras helpers

Let us look at the Keras helper defined in deep_architect/helpers/keras.py.

```
import deep_architect.core as co
import tensorflow as tf
import numpy as np


class KerasModule(co.Module):
    """Class for taking Keras code and wrapping it in a DeepArchitect module.
```

(continues on next page)

(continued from previous page)

```
    This class subclasses :class:`deep_architect.core.Module` as therefore inherits␣
↪all
    the functionality associated to it (e.g., keeping track of inputs, outputs,
    and hyperparameters). It also enables to do the compile and forward
    operations for these types of modules once a module is fully specified,
    i.e., once all the hyperparameters have been chosen.

    The compile operation in this case creates all the variables used for the
    fragment of the computational graph associated to this module.
    The forward operation takes the variables that were created in the compile
    operation and constructs the actual computational graph fragment associated
    to this module.

    .. note::
        This module is abstract, meaning that it does not actually implement
        any particular Keras computation. It simply wraps Keras
        functionality in a DeepArchitect module. The instantiation of the Keras
        variables is taken care by the `compile_fn` function that takes a two
        dictionaries, one of inputs and another one of outputs, and
        returns another function that takes a dictionary of inputs and creates
        the computational graph. This functionality makes extensive use of closures.

        The keys of the dictionaries that are passed to the compile
        and forward function match the names of the inputs and hyperparameters
        respectively. The dictionary returned by the forward function has keys
        equal to the names of the outputs.

        This implementation is very similar to the implementation of the Tensorflow
        helper :class:`deep_architect.helpers.tensorflow_support.TensorflowModule`.

    Args:
        name (str): Name of the module
        name_to_hyperp (dict[str,deep_architect.core.Hyperparameter]): Dictionary of
            hyperparameters that the model depends on. The keys are the local
            names of the hyperparameters.
        compile_fn ((dict[str,object], dict[str,object]) -> (dict[str,object] ->␣
↪dict[str,object])):
            The first function takes two dictionaries with
            keys corresponding to `input_names` and `output_names` and returns
            a function that takes a dictionary with keys corresponding to
            `input_names` and returns a dictionary with keys corresponding
            to `output_names`. The first function may also return
            two additional dictionaries mapping Tensorflow placeholders to the
            values that they will take during training and test.
        input_names (list[str]): List of names for the inputs.
        output_names (list[str]): List of names for the outputs.
        scope (deep_architect.core.Scope, optional): Scope where the module will be
            registered.
    """

    def __init__(self,
                 name,
                 name_to_hyperp,
                 compile_fn,
                 input_names,
                 output_names,
```

(continues on next page)

(continued from previous page)

```
                 scope=None):
        co.Module.__init__(self, scope, name)

        self._register(input_names, output_names, name_to_hyperp)
        self._compile_fn = compile_fn

    def _compile(self):
        input_name_to_val = self._get_input_values()
        hyperp_name_to_val = self._get_hyperp_values()
        self._fn = self._compile_fn(input_name_to_val, hyperp_name_to_val)

    def _forward(self):
        input_name_to_val = self._get_input_values()
        output_name_to_val = self._fn(input_name_to_val)
        self._set_output_values(output_name_to_val)

    def _update(self):
        pass
```

The code is compact and self-explanatory. We pass a function called `compile_fn` that returns a function called `forward_fn` function upon compilation. To instantiate a module, we simply have to provide a compile function that returns a forward function when called. For example, for implementing a convolutional module from scratch relying on this module (check the Keras docstring for `Conv2D`), we would do:

```
from keras.layers import Conv2D


def conv2d(h_filters, h_kernel_size, h_strides, h_activation, h_use_bias):

    def compile_fn(di, dh):
        m = Conv2D(**dh)

        def forward_fn(di):
            return {"out": m(di["in"])}

        return forward_fn

    return KerasModule(
        "Conv2D", {
            "filters": h_filters,
            "kernel_size": h_kernel_size,
            "strides": h_strides,
            "activation": h_activation,
            "use_bias": h_use_bias
        }, compile_fn, ["in"], ["out"]).get_io()
```

A few points to pay attention to:

- Input, output and hyperparameter names are specified when creating an instance of `KerasModule`.
- `di` and `dh` are dictionaries with inputs names mapping to input values and hyperparameter names mapping to hyperparameter values, respectively.
- `Conv2D(**dh)` uses dictionary unpacking to call the Keras function that instantiates a Keras layer (as in the Keras API). We could also have done the unpacking manually.
- Upon the instantiation of the Keras module, we call `get_io` to get a pair `(inputs, outputs)`, where both `inputs` and `outputs` are dictionaries, where `inputs` maps input names to input objects (i.e., an object

from the class *deep_architect.core.Input*), and `outputs` maps output names to output objects (i.e., an object from the class *deep_architect.core.Output*). Working directly with dictionaries of inputs and outputs is more convenient than working with modules, because we can transparently work with subgraph structures without concerning ourselves about whether they are composed of multiple modules or not.

A minimal example to go from this wrapper code to an instantiated Keras model is:

```
from keras.layers import Input
import deep_architect.hyperparameters as hp
import deep_architect.core as co
from deep_architect.searchers.common import random_specify
from keras.models import Model

D = hp.Discrete
# specifying all the hyperparameters.
x = Input((32, 32, 3), dtype='float32')
h_filters = D([32, 64])
h_kernel_size = D([1, 3, 5])
h_strides = D([1])
h_activation = D(['relu', 'sigmoid'])
h_use_bias = D([0, 1])
(inputs, outputs) = conv2d(h_filters, h_kernel_size, h_strides, h_activation,
                           h_use_bias)
random_specify(outputs)
co.forward({inputs["in"]: x})
out = outputs["out"].val
model = Model(inputs=x, outputs=out)
model.summary()

import deep_architect.visualization as vi
vi.draw_graph(outputs, draw_module_hyperparameter_info=False)
```

As modules with a single input and a single output are common, we defined a few simplified functions that directly work with the Keras definition. The goal of these functions is to reduce boilerplate and provide a more concise workflow. For example, the above function could be expressed in the same way as:

```
import deep_architect.helpers.keras_support as hke


def conv2d(h_filters, h_kernel_size, h_strides, h_activation, h_use_bias):
    return hke.siso_keras_module_from_keras_layer_fn(
        Conv2D, {
            "filters": h_filters,
            "kernel_size": h_kernel_size,
            "strides": h_strides,
            "activation": h_activation,
            "use_bias": h_use_bias
        })


(inputs, outputs) = conv2d(h_filters, h_kernel_size, h_strides, h_activation,
                           h_use_bias)
co.forward({inputs["in"]: x})
out = outputs["out"].val
model = Model(inputs=x, outputs=out)
model.summary()
vi.draw_graph(outputs, draw_module_hyperparameter_info=False)
```

We refer the reader to deep_architect.helpers.keras_support if the reader wishes to inspect the implementation of this

function and how does it fit with the previous definition for a Keras module. These functions require minimal additional code. These auxiliary functions are convenient to reduce boilerplate for some of the most common use cases. As we have seen, it is possible to express everything that we need using `KerasModule`, with the other functions used for convenience for common specific cases.

Calls to *deep_architect.core.forward()* call the individual module forward and compile functions as defined in `KerasModule` and passed as argument during the instantiation. We invite the reader to inspect *deep_architect.core.forward()* (found here) to understand how it is implemented using graph traversal. This is sufficient to specialize the general module code in *deep_architect.core* to support basic modules from Keras.

Pytorch helpers

The reader may think that Pytorch does not fit well in our framework due to being a dynamic framework where the graph used for back propagation is defined for each instance, i.e., defined by run, rather than static (as it is the case of Keras) where the graph is defined upfront and used across instances. Static versus dynamic is not an important distinction for architecture search in DeepArchitect. For example, we can search over computational elements that are used dynamically by the model, e.g., a recurrent cell.

Let us quickly walk through the DeepArchitect module specialization for PyTorch. We omit the docstring due to the similarity with the one for `KerasModule.forward`.

```
class PyTorchModule(co.Module):

    def __init__(self,
                 name,
                 name_to_hyperp,
                 compile_fn,
                 input_names,
                 output_names,
                 scope=None):
        co.Module.__init__(self, scope, name)
        self._register(input_names, output_names, name_to_hyperp)
        self._compile_fn = compile_fn

    def _compile(self):
        input_name_to_val = self._get_input_values()
        hyperp_name_to_val = self._get_hyperp_values()
        self._fn, self.pyth_modules = self._compile_fn(input_name_to_val,
                                                       hyperp_name_to_val)
        for pyth_m in self.pyth_modules:
            assert isinstance(pyth_m, nn.Module)

    def _forward(self):
        input_name_to_val = self._get_input_values()
        output_name_to_val = self._fn(input_name_to_val)
        self._set_output_values(output_name_to_val)

    def _update(self):
        pass
```

We can see that the implementation for PyTorch is essentially the same as the one for Keras. The main difference is that the compile_fn function that returns both `forward_fn` and the list of Pytorch modules (as in `nn.Module`) that have been used in the computation. This list is used to keep track of which Pytorch modules are used by the DeepArchitect module. For example, this is necessary to move them to the GPU or CPU, or get their parameters. Changes from Tensorflow to Pytorch are mainly a result of the differences in how these two frameworks declare computational graphs.

```
import deep_architect.helpers.pytorch_support as hpy


def conv2d(h_filters, h_kernel_size, h_strides, h_activation, h_use_bias):

    def compile_fn(di, dh):
        m = Conv2D(**dh)

        def forward_fn(di):
            return {"out": m(di["in"])}

        return forward_fn

    return PyTorchModule(
        "Conv2D", {
            "filters": h_filters,
            "kernel_size": h_kernel_size,
            "strides": h_strides,
            "activation": h_activation,
            "use_bias": h_use_bias
        }, compile_fn, ["in"], ["out"]).get_io()


def conv2d_pytorch(h_filters, h_kernel_size, h_strides, h_activation,
                   h_use_bias):
    return hpy.siso_pytorch_module_from_pytorch_layer_fn(
        Conv2D, {
            "filters": h_filters,
            "kernel_size": h_kernel_size,
            "strides": h_strides,
            "activation": h_activation,
            "use_bias": h_use_bias
        })
```

Concluding remarks

DeepArchitect is not limited to deep learning frameworks—any domain that for which we can define notions of compile and forward (or potentially, other operations) as they were discussed above can be supported. Another aspect to keep in mind is that there is not a need for all the modules of the computational graph to be in the same domain (e.g., a preprocessing component followed by the actual graph propagation).

We showcased support for both static (Keras) and dynamic deep learning (PyTorch) frameworks. The notions of basic modules, substitution modules, independent hyperparameters, and dependent hyperparameters are general and can be used across a large range of settings (e.g., scikit-learn or data augmentation pipelines). We leave the consideration of other frameworks (deep learning or otherwise) to the reader.

2.1.5 Logging and visualization

Architecture search provides tremendous opportunity to create insightful visualizations based on architecture search results.

The logging functionality in DeepArchitect allows us to create a folder for a search experiment. This search log folder contains a folder for each evaluation done during search. Each evaluation folder contains a fixed component and a component that is specified by the user. The fixed component contains a JSON file with the hyperparameters values that define the architecture and a JSON file with the results obtained for that architecture. The user component

allows the user to store additional information for each architecture, e.g., model parameters or example predictions. The logging functionality makes it convenient to manage this folder structure. The log folder for the whole search experiment keeps user information at the search level, e.g., searcher checkpoints.

We point the reader to examples/mnist_with_logging for an example use of logging, and to deep_architect/search_logging.py for the API definitions.

Starting Keras example to adapt for logging

Let us start with a Keras MNIST example (copied from the Keras website), and adapt to get a DeepArchitect example using the logging functionality.

```
'''Trains a simple deep NN on the MNIST dataset.

Gets to 98.40% test accuracy after 20 epochs
(there is *a lot* of margin for parameter tuning).
2 seconds per epoch on a K520 GPU.
'''

from __future__ import print_function

import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import RMSprop

batch_size = 128
num_classes = 10
epochs = 1

# the data, split between train and test sets
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(784,)))
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(num_classes, activation='softmax'))
print(y_train.shape, y_test.shape)
model.summary()

model.compile(
```

(continues on next page)

(continued from previous page)

```
    loss='categorical_crossentropy', optimizer=RMSprop(), metrics=['accuracy'])

history = model.fit(
    x_train,
    y_train,
    batch_size=batch_size,
    epochs=epochs,
    verbose=1,
    validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

This is the simplest MNIST example that we can get from the Keras examples website.

Evaluator and search space definitions

We will adapt this example to create a DeepArchitect example showcasing some of the logging and visualization functionalities.

We first create a small validation set out of training set:

```
import deep_architect.core as co
from keras.layers import Input
from keras.models import Model


class Evaluator:

    def __init__(self, batch_size, epochs):
        self.batch_size = 128
        self.num_classes = 10
        self.epochs = 1

        # the data, split between train and test sets
        (x_train, y_train), (x_test, y_test) = mnist.load_data()

        x_train = x_train.reshape(60000, 784)
        x_test = x_test.reshape(10000, 784)
        x_train = x_train.astype('float32')
        x_test = x_test.astype('float32')
        x_train /= 255
        x_test /= 255
        y_train = keras.utils.to_categorical(y_train, num_classes)
        y_test = keras.utils.to_categorical(y_test, num_classes)

        num_val = 10000
        x_train, x_val = (x_train[:num_val], x_train[num_val:])
        y_train, y_val = (y_train[:num_val], y_train[num_val:])
        self.x_train = x_train
        self.y_train = y_train
        self.x_val = x_val
        self.y_val = y_val
        self.x_test = x_test
        self.y_test = y_test
        self.last_model = None
```

(continues on next page)

(continued from previous page)

```
    def eval(self, inputs, outputs):
        x = Input((784,), dtype='float32')
        co.forward({inputs["in"]: x})
        y = outputs["out"].val
        model = Model(inputs=x, outputs=y)

        model.summary()

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

        history = model.fit(
            self.x_train,
            self.y_train,
            batch_size=self.batch_size,
            epochs=self.epochs,
            verbose=1)
        self.last_model = model
        train_metrics = model.evaluate(self.x_train, self.y_train, verbose=0)
        val_metrics = model.evaluate(self.x_val, self.y_val, verbose=0)
        test_metrics = model.evaluate(self.x_test, self.y_test, verbose=0)
        return {
            "train_loss": train_metrics[0],
            "validation_loss": val_metrics[0],
            "test_loss": test_metrics[0],
            "train_accuracy": train_metrics[1],
            "validation_accuracy": val_metrics[1],
            "test_accuracy": test_metrics[1],
            "num_parameters": model.count_params(),
        }


import deep_architect.helpers.keras_support as hke
import deep_architect.hyperparameters as hp
import deep_architect.searchers.common as sco
import deep_architect.modules as mo
from keras.layers import Dense, Dropout, BatchNormalization

D = hp.Discrete

km = hke.siso_keras_module_from_keras_layer_fn


def cell(h_opt_drop, h_opt_batchnorm, h_drop_rate, h_activation, h_permutation):
    h_units = D([128, 256, 512])
    return mo.siso_sequential([
        mo.siso_permutation(
            [
                lambda: km(Dense, {
                    "units": h_units,
                    "activation": h_activation
                }),  #
                lambda: mo.siso_optional(
                    lambda: km(Dropout, {"rate": h_drop_rate}), h_opt_drop),
```

(continues on next page)

(continued from previous page)

```
                lambda: mo.siso_optional(  #
                    lambda: km(BatchNormalization, {}), h_opt_batchnorm)
            ],
            h_permutation)
    ])


def model_search_space():
    h_opt_drop = D([0, 1])
    h_opt_batchnorm = D([0, 1])
    h_permutation = hp.OneOfKFactorial(3)
    h_activation = D(["relu", "tanh", "elu"])
    fn = lambda: cell(h_opt_drop, h_opt_batchnorm, D([0.0, 0.2, 0.5, 0.8]),
                      h_activation, h_permutation)
    return mo.siso_sequential([
        mo.siso_repeat(fn, D([1, 2, 4])),
        km(Dense, {
            "units": D([num_classes]),
            "activation": D(["softmax"])
        })
    ])


search_space_fn = mo.SearchSpaceFactory(model_search_space).get_search_space
```

Main search loop with logging

This creates an initial folder structure that is progressively filled with each of the evaluations. The basic architecture search loop with a single process is as follows:

```
from deep_architect.searchers.mcts import MCTSSearcher
import deep_architect.search_logging as sl
import deep_architect.visualization as vi
import deep_architect.utils as ut

searcher = MCTSSearcher(search_space_fn)
evaluator = Evaluator(batch_size, epochs)
num_samples = 3

search_logger = sl.SearchLogger(
    'logs', 'logging_tutorial', delete_if_exists=True, abort_if_exists=False)

for evaluation_id in range(num_samples):
    (inputs, outputs, hyperp_value_lst, searcher_eval_token) = searcher.sample()
    results = evaluator.eval(inputs, outputs)
    eval_logger = search_logger.get_evaluation_logger(evaluation_id)
    eval_logger.log_config(hyperp_value_lst, searcher_eval_token)
    eval_logger.log_results(results)
    user_folderpath = eval_logger.get_evaluation_data_folderpath()
    vi.draw_graph(
        outputs,
        draw_module_hyperparameter_info=False,
        out_folderpath=user_folderpath)
    model_filepath = ut.join_paths([user_folderpath, 'model.h5'])
    evaluator.last_model.save(model_filepath)
```

(continues on next page)

(continued from previous page)

```
    searcher.update(results["validation_accuracy"], searcher_eval_token)
```

The above code samples and evaluates three architectures from the search space. The results, the corresponding graph, and the saved models are logged to each of the evaluation folders. Typically, we may not want to store weights for all the architectures evaluated as it will lead to a large storage being consumed. In case only the weights for few architectures are to be kept around, then the user can employ different logic to guarantee that the number of stored models remains small during search (e.g., keeping only the best ones).

After running this code, we ask the reader to explore the resulting log folder to get a sense for the information stored. We made the resulting folder available here in case the reader does not wish to run the code locally, but still wishes to inspect the resulting search log folder.

Concluding remarks

Log folders are useful for visualization and exploration. Architecture search allows us to try many architectures and explore different characteristics of each of them. We may set the search space to explore what characteristics lead to better performance. Architecture search, and more specifically, DeepArchitect and the workflow that we suggest allows us to formulate many of these questions easily and explore the results for insights. We encourage users of DeepArchitect to think about interesting visualizations that can be constructed using architecture search workflows.

CHAPTER 3

API documentation

3.1 API Documentation

3.1.1 deep_architect.core

class deep_architect.core.**Addressable**(*scope*, *name*)

Bases: object

Base class for classes whose objects have to be registered in a scope.

Provides functionality to register objects in a scope.

Parameters

- **scope** (deep_architect.core.Scope) – Scope object where the addressable object will be registered.
- **name** (*str*) – Unique name used to register the addressable object.

_get_base_name()

Get the class name.

Useful to create unique names for an addressable object.

Returns Class name.

Return type str

get_name()

Get the name with which the object was registered in the scope.

Returns Unique name used to register the object.

Return type str

class deep_architect.core.**DependentHyperparameter**(*fn*, *hyperps*, *scope=None*, *name=None*)

Bases: *deep_architect.core.Hyperparameter*

Hyperparameter that depends on other hyperparameters.

The value of a dependent hyperparameter is set by a calling a function using the values of the dependent hyperparameters as arguments. This hyperparameter is convenient when we want to express search spaces where the values of some hyperparameters are computed as a function of the values of some other hyperparameters, rather than set independently.

Parameters

- **fn** (*dict[str, object] -> object*) – Function used to compute the value of the hyperparameter based on the values of the dependent hyperparameters.
- **hyperps** (*dict[str, deep_architect.core.Hyperparameter]*) – Dictionary mapping names to hyperparameters. The names used in the dictionary should correspond to the names of the arguments of `fn`.
- **scope** (*deep_architect.core.Scope, optional*) – The scope in which to register the hyperparameter in.
- **name** (*str, optional*) – Name from which the name of the hyperparameter in the scope is derived.

_update()
Checks if the hyperparameter is ready to be set, and sets it if that is the case.

class deep_architect.core.**Hyperparameter**(*scope=None*, *name=None*)
Bases: *deep_architect.core.Addressable*

Base hyperparameter class.

Specific hyperparameter types are created by inheriting from this class. Hyperparameters keep references to the modules that are dependent on them.

Note: Hyperparameters with easily serializable values are preferred due to the interaction with the search logging and multi-GPU functionalities. Typical valid serializable types are integers, floats, strings. Lists and dictionaries of serializable types are also valid.

Parameters

- **scope** (*deep_architect.core.Scope, optional*) – Scope in which the hyperparameter will be registered. If none is given, uses the default scope.
- **name** (*str, optional*) – Name used to derive an unique name for the hyperparameter. If none is given, uses the class name to derive the name.

_check_value(*val*)
Checks if the value is valid for the hyperparameter.

When `set_val` is called, this function is called to verify the validity of `val`. This function is useful for error checking.

_register_dependent_hyperparameter(*hyperp*)
Registers an hyperparameter as being dependent on this hyperparameter.

Parameters **module** (*deep_architect.core.Hyperparameter*) – Hyperparameter dependent of this hyperparameter.

_register_module(*module*)
Registers a module as being dependent of this hyperparameter.

Parameters **module** (deep_architect.core.Module) – Module dependent of this hyperparameter.

assign_value(*val*)
Assigns a value to the hyperparameter.

The hyperparameter value must be valid for the hyperparameter in question. The hyperparameter becomes set if the call is successful.

Parameters **val** (*object*) – Value to assign to the hyperparameter.

get_value()
Get the value assigned to the hyperparameter.

The hyperparameter must have already been assigned a value, otherwise asserts `False`.

Returns Value assigned to the hyperparameter.

Return type object

has_value_assigned()
Checks if the hyperparameter has been assigned a value.

Returns `True` if the hyperparameter has been assigned a value.

Return type bool

class deep_architect.core.**Input**(*module*, *scope*, *name*)
Bases: *deep_architect.core.Addressable*

Manages input connections.

Inputs may be connected to a single output. Inputs and outputs are associated to a single module.

See also: *deep_architect.core.Output* and *deep_architect.core.Module*.

Parameters

- **module** (deep_architect.core.Module) – Module with which the input object is associated to.
- **scope** (deep_architect.core.Scope) – Scope object where the input is going to be registered in.
- **name** (*str*) – Unique name with which to register the input object.

connect(*from_output*)
Connect an output to this input.

Changes the state of both the input and the output. Asserts `False` if the input is already connected.

Parameters **from_output** (deep_architect.core.Output) – Output to connect to this input.

disconnect()
Disconnects the input from the output it is connected to.

Changes the state of both the input and the output. Asserts `False` if the input is not connected.

get_connected_output()
Get the output to which the input is connected to.

Returns Output to which the input is connected to.

Return type *deep_architect.core.Output*

get_module()
Get the module with which the input is associated with.

Returns Module with which the input is associated with.

Return type *deep_architect.core.Module*

is_connected()
Checks if the input is connected.

Returns `True` if the input is connected.

Return type bool

reroute_connected_output(*to_input*)
Disconnects the input from the output it is connected to and connects the output to a new input, leaving this input in a disconnected state.

Changes the state of both this input, the other input, and the output to which this input is connected to.

Note: Rerouting operations are widely used in *deep_architect.modules.SubstitutionModule*. See also: *deep_architect.core.Output.reroute_all_connected_inputs()*.

Parameters **to_input** (deep_architect.core.Input) – Input to which the output is going to be connected to.

class deep_architect.core.**Module**(*scope=None*, *name=None*)
Bases: *deep_architect.core.Addressable*

Modules inputs and outputs, and depend on hyperparameters.

Modules are some of the main components used to define search spaces. The inputs, outputs, and hyperparameters have names local to the module. These names are different than the ones used in the scope in which these objects are registered in.

Search spaces based on modules are very general. They can be used across deep learning frameworks, and even for purposes that do not involve deep learning, e.g., searching over scikit-learn pipelines. The main operations to understand are compile and forward.

Parameters

- **scope** (deep_architect.core.Scope, *optional*) – Scope object where the module is going to be registered in.
- **name** (*str*, *optional*) – Unique name with which to register the module.

_compile()
Compile operation for the module.

Called once when all the hyperparameters that the module depends on, and the other hyperparameters of the search space are specified. See also: *_forward()*.

_forward()
Forward operation for the module.

Called once the compile operation has been called. See also: *_compile()*.

_get_hyperp_values()
Get the values of the hyperparameters.

Returns Dictionary of local hyperparameter names to their corresponding values.

Return type dict[str, object]

_get_input_values()
Get the values associated to the inputs of the module.

This function is used to implement forward. See also: *_set_output_values()* and *forward()*.

Returns Dictionary of local input names to their corresponding values.

Return type dict[str, object]

_register(*input_names*, *output_names*, *name_to_hyperp*)
Registers inputs, outputs, and hyperparameters locally for the module.

This function is convenient to avoid code repetition when registering multiple inputs, outputs, and hyperparameters.

Parameters

- **input_names** (*list[str]*) – List of inputs names of the module.
- **output_names** (*list[str]*) – List of the output names of the module.
- **name_to_hyperp** (*dict[str, deep_architect.core.Hyperparameter]*) – Dictionary of names of hyperparameters to hyperparameters.

_register_hyperparameter(*name*, *h*)
Registers an hyperparameter that the module depends on.

Parameters

- **name** (*str*) – Local name to give to the hyperparameter.
- **h** (deep_architect.core.Hyperparameter) – Hyperparameter that the module depends on.

_register_input(*name*)
Creates a new input with the chosen local name.

Parameters **name** (*str*) – Local name given to the input.

_register_output(*name*)
Creates a new output with the chosen local name.

Parameters **name** (*str*) – Local name given to the output.

_set_output_values(*output_name_to_val*)
Set the values of the outputs of the module.

This function is used to implement forward. See also: *_get_input_values()* and *forward()*.

Parameters **output_name_to_val** (*dict[str, object]*) – Dictionary of local output names to the corresponding values to assign to those outputs.

_update()
Called when an hyperparameter that the module depends on is set.

forward()
The forward computation done by the module is decomposed into *_compile()* and *_forward()*.

Compile can be thought as creating the parameters of the module (done once). Forward can be thought as using the parameters of the module to do the specific computation implemented by the module on some specific data (done multiple times).

This function can only called after the module and the other modules in the search space are fully specified. See also: *forward()*.

get_hyperps()

Returns Dictionary of local hyperparameter names to the corresponding hyperparameter objects.

Return type dict[str, *deep_architect.core.Hyperparameter*]

get_io()

Returns Pair with dictionaries mapping the local input and output names to their corresponding input and output objects.

Return type (dict[str,*deep_architect.core.Input*], dict[str,*deep_architect.core.Output*])

class deep_architect.core.**OrderedSet**

Bases: object

add(*x*)

update(*xs*)

class deep_architect.core.**Output**(*module*, *scope*, *name*)

Bases: *deep_architect.core.Addressable*

Manages output connections.

Outputs may be connected to multiple inputs. Inputs and outputs are associated to a single module.

See also: *deep_architect.core.Input* and *deep_architect.core.Module*.

Parameters

- **module** (deep_architect.core.Module) – Module with which the output object is associated to.
- **scope** (deep_architect.core.Scope) – Scope object where the output is going to be registered in.
- **name** (*str*) – Unique name with which to register the output object.

connect(*to_input*)

Connect an additional input to this output.

Changes the state of both the input and the output.

Parameters **to_input** (deep_architect.core.Input) – Input to connect to this output.

disconnect_all()

Disconnects all the inputs connected to this output.

Changes the state of the output and all the inputs connected to it.

get_connected_inputs()

Get the list of inputs to which is the output is connected to.

Returns

List of the inputs to which the output is connect to.

Return type list[*deep_architect.core.Input*]

get_module()

Get the module object with which the output is associated with.

Returns

Module object with which the output is associated with.

Return type *deep_architect.core.Module*

is_connected()
Checks if the output is connected.

Returns `True` if the output is connected.

Return type bool

reroute_all_connected_inputs(*from_output*)
Reroutes all the inputs to which the output is connected to a different output.

Note: Rerouting operations are widely used in *deep_architect.modules.SubstitutionModule*. See also: *deep_architect.core.Input.reroute_connected_output()*.

Parameters **from_output** (deep_architect.core.Output) – Output to which the connected inputs are going to be rerouted to.

class deep_architect.core.**Scope**
Bases: object

A scope is used to help assign unique readable names to addressable objects.

A scope keeps references to modules, hyperparameters, inputs, and outputs.

default_scope = <deep_architect.core.Scope object>

get_elem(*name*)
Get the object that is registered in the scope with the desired name.

The name must exist in the scope.

Parameters **name** (*str*) – Name of the addressable object registered in the scope.

Returns Addressable object with the corresponding name.

Return type str

get_name(*elem*)
Get the name of the addressable object registered in the scope.

The object must exist in the scope.

Parameters **elem** (deep_architect.core.Addressable) – Addressable object registered in the scope.

Returns Name with which the object was registered in the scope.

Return type str

get_unused_name(*prefix*)
Creates a unique name by adding a numbered suffix to the prefix.

Parameters **prefix** (*str*) – Prefix of the desired name.

Returns Unique name in the current scope.

Return type str

register(*name*, *elem*)
Registers an addressable object with the desired name.

The name cannot exist in the scope, otherwise asserts `False`.

Parameters

- **name** (*str*) – Unique name.
- **elem** (deep_architect.core.Addressable) – Addressable object to register.

static reset_default_scope()
Replaces the current default scope with a new empty scope.

deep_architect.core.**determine_input_output_cleanup_seq**(*inputs*)
Determines the order in which the outputs can be cleaned.

This sequence is aligned with the module evaluation sequence. Positionally, after each module evaluation, the values stored in val for both inputs and outputs can be deleted. This is useful to remove intermediate results to save memory.

Note: This function should be used only for fully-specified search spaces.

Parameters **inputs** (*dict[str, deep_architect.core.Input]*) – Dictionary of named inputs which by being traversed forward will reach all the modules in the search space.

Returns List of lists with the inputs and outputs in the order they should be cleaned up after they are no longer needed.

Return type (list[list[*deep_architect.core.Input*]], list[list[*deep_architect.core.Output*]])

deep_architect.core.**determine_module_eval_seq**(*inputs*)
Computes the module forward evaluation sequence necessary to evaluate the computational graph starting from the provided inputs.

The computational graph is a directed acyclic graph. This function sorts the modules topologically based on their dependencies. It is assumed that the inputs in the dictionary provided are sufficient to compute forward for all modules in the graph. See also: *forward()*.

Parameters **inputs** (*dict[str, deep_architect.core.Input]*) – dictionary of inputs sufficient to compute the forward computation of the whole graph through propagation.

Returns List of modules ordered in a way that allows to call forward on the modules in that order.

Return type list[*deep_architect.core.Module*]

deep_architect.core.**extract_unique_modules**(*input_or_output_lst*)
Get the modules associated to the inputs and outputs in the list.

Each module appears appear only once in the resulting list of modules.

Parameters **input_or_output_lst** (*list[deep_architect.core.Input or deep_architect.core.Output]*) – List of inputs or outputs from which to extract the associated modules.

Returns Unique modules to which the inputs and outputs in the list belong to.

Return type list[*deep_architect.core.Module*]

deep_architect.core.**forward**(*input_to_val*, *_module_seq=None*)
Forward pass through the graph starting with the provided inputs.

The starting inputs are given the values in the dictionary. The values for the other inputs are obtained through propagation, i.e., through successive calls to *deep_architect.core.Module.forward()* of the appropriate modules.

Note: For efficiency, in dynamic frameworks, the module evaluation sequence is best computed once and reused in each forward call. The module evaluation sequence is computed with *determine_module_eval_seq()*.

Parameters

- **input_to_val** (*dict[deep_architect.core.Input, object]*) – Dictionary of initial inputs to their corresponding values.
- **_module_seq** (*list[deep_architect.core.Module], optional*) – List of modules ordered in a way that calling *deep_architect.core.Module.forward()* on them starting from the values given for the inputs is valid. If it is not provided, the module sequence is computed.

deep_architect.core.**get_all_hyperparameters**(*outputs*)

Going backward from the outputs provided, gets all hyperparameters.

Hyperparameters that can be reached by traversing dependency links between hyperparameters are also included. Setting an hyperparameter may lead to the creation of additional hyperparameters, which will be most likely not set. Such behavior happens when dealing with, for example, hyperparameters associated with substitutition modules such as *deep_architect.modules.siso_optional()*, *deep_architect.modules.siso_or()*, and *deep_architect.modules.siso_repeat()*.

Parameters **outputs** (*dict[str, deep_architect.core.Output]*) – Dictionary of named outputs to start the traversal at.

Returns Ordered set of hyperparameters that are currently present in the graph.

Return type *OrderedSet*[*deep_architect.core.Hyperparameter*]

deep_architect.core.**get_modules_with_cond**(*outputs*, *cond_fn*)

deep_architect.core.**get_unassigned_independent_hyperparameters**(*outputs*)

Going backward from the outputs provided, gets all the independent hyperparameters that are not set yet.

Setting an hyperparameter may lead to the creation of additional hyperparameters, which will be most likely not set. Such behavior happens when dealing with, for example, hyperparameters associated with substitutition modules such as *deep_architect.modules.siso_optional()*, *deep_architect.modules.siso_or()*, and *deep_architect.modules.siso_repeat()*.

Parameters **outputs** (*dict[str, deep_architect.core.Output]*) – Dictionary of named outputs to start the traversal at.

Returns Ordered set of hyperparameters that are currently present in the graph and not have been assigned a value yet.

Return type *OrderedSet*[*deep_architect.core.Hyperparameter*]

deep_architect.core.**get_unconnected_inputs**(*outputs*)

Get the inputs that are reachable going backward from the provided outputs, but are not connected to any outputs.

Often, these inputs have to be provided with a value when calling *forward()*.

Parameters **outputs** (*list[deep_architect.core.Output]*) – Dictionary of named outputs to start the backward traversal at.

Returns Unconnected inputs reachable by traversing the graph backward starting from the provided outputs.

Return type list[*deep_architect.core.Input*]

deep_architect.core.**get_unconnected_outputs**(*inputs*)
Get the outputs that are reachable going forward from the provided inputs, but are not connected to outputs.

Often, the final result of a forward pass through the network will be at these outputs.

Parameters **inputs** (*dict[str, deep_architect.core.Input]*) – Dictionary of named inputs to start the forward traversal at.

Returns Unconnected outputs reachable by traversing the graph forward starting from the provided inputs.

Return type list[*deep_architect.core.Output*]

deep_architect.core.**is_specified**(*outputs*)
Checks if all the hyperparameters reachable by traversing backward from the outputs have been set.

Parameters **outputs** (*dict[str, deep_architect.core.Output]*) – Dictionary of named outputs to start the traversal at.

Returns `True` if all the hyperparameters have been set. `False` otherwise.

Return type bool

deep_architect.core.**jsonify**(*inputs*, *outputs*)
Returns a JSON representation of the fully-specified search space.

This function is useful to create a representation of model that does not rely on the graph representation involving *deep_architect.core.Module*, *deep_architect.core.Input*, and *deep_architect.core.Output*.

Parameters

- **inputs** (*dict[str, deep_architect.core.Input]*) – Dictionary of named inputs which by being traversed forward will reach all the modules in the search space.
- **outputs** (*dict[str, deep_architect.core.Output]*) – Dictionary of named outputs which by being traversed back will reach all the modules in the search space.

Returns JSON representation of the fully specified model.

Return type (dict)

deep_architect.core.**sorted_values_by_key**(*d*)

deep_architect.core.**traverse_backward**(*outputs*, *fn*)
Backward traversal function through the graph.

Traverses the graph going from outputs to inputs. The provided function is applied once to each module reached this way. This function is used to implement other functionality that requires traversing the graph. `fn` typically has side effects, e.g., see *is_specified()* and `get_unassigned_hyperparameters()`. See also: *traverse_forward()*.

Parameters

- **outputs** (*dict[str, deep_architect.core.Output]*) – Dictionary of named of outputs to start the traversal at.
- **fn** (*(deep_architect.core.Module) -> (bool)*) – Function to apply to each module. Returns `True` if the traversal is to be stopped.

deep_architect.core.**traverse_forward**(*inputs*, *fn*)
Forward traversal function through the graph.

Traverses the graph going from inputs to outputs. The provided function is applied once to each module reached this way. This function is used to implement other functionality that requires traversing the graph. `fn` typically has side effects, e.g., see *get_unconnected_outputs()*. See also: *traverse_backward()*.

Parameters

- **inputs** (*dict[str, deep_architect.core.Input]*) – Dictionary of named inputs to start the traversal at.
- **fn** (*(deep_architect.core.Module) -> (bool)*) – Function to apply to each module. Returns `True` if the traversal is to be stopped.

deep_architect.core.**unassigned_independent_hyperparameter_iterator**(*outputs*)

Returns an iterator over the hyperparameters that are not specified in the current search space.

This iterator is used by the searchers to go over the unspecified hyperparameters.

Note: It is assumed that all the hyperparameters that are touched by the iterator will be specified (most likely, right away). Otherwise, the iterator will never terminate.

Parameters **outputs** (*dict[str, deep_architect.core.Output]*) – Dictionary of named outputs which by being traversed back will reach all the modules in the search space, and correspondingly all the current unspecified hyperparameters of the search space.

Yields *(deep_architect.core.Hyperparameter)* – Next unspecified hyperparameter of the search space.

3.1.2 deep_architect.hyperparameters

3.1.3 deep_architect.modules

class deep_architect.modules.**HyperparameterAggregator**(*name_to_hyperp*, *scope=None*, *name=None*)

Bases: *deep_architect.core.Module*

class deep_architect.modules.**Identity**(*scope=None*, *name=None*)

Bases: *deep_architect.core.Module*

Module passes the input to the output without changes.

Parameters

- **scope** (*deep_architect.core.Scope, optional*) – Scope in which the module will be registered. If none is given, uses the default scope.
- **name** (*str, optional*) – Name used to derive an unique name for the module. If none is given, uses the class name to derive the name.

class deep_architect.modules.**SearchSpaceFactory**(*search_space_fn*, *reset_default_scope_upon_get=True*)

Bases: `object`

Helper used to provide a nicer interface to create search spaces.

The user should inherit from this class and implement `_get_search_space()`. The function get_search_space should be given to the searcher upon creation of the searcher.

Parameters

- **search_space_fn** (*() -> (dict[str,deep_architect.core.Input], dict[str,deep_architect.core.Output])*) – Returns the inputs and outputs of the search space, ready to be specified.
- **reset_default_scope_upon_get** (*bool*) – Whether to clean the scope upon getting a new search space. Should be `True` in most cases.

get_search_space()
Returns the buffered search space.

class deep_architect.modules.**SubstitutionModule**(*name*, *substitution_fn*, *name_to_hyperp*, *input_names*, *output_names*, *scope=None*, *allow_input_subset=False*, *allow_output_subset=False*)

Bases: *deep_architect.core.Module*

Substitution modules are replaced by other modules when the all the hyperparameters that the module depends on are specified.

Substitution modules implement a form of delayed evaluation. The main component of a substitution module is the substitution function. When called, this function returns a dictionary of inputs and a dictionary of outputs. These outputs and inputs are used in the place the substitution module is in. The substitution module effectively disappears from the network after the substitution operation is done. Substitution modules are used to implement many other modules, e.g., *mimo_or()*, *siso_optional()*, and *siso_repeat()*.

Parameters

- **name** (*str*) – Name used to derive an unique name for the module.
- **substitution_fn** (*dict[str, object] -> (dict[str, deep_architect.core.Input], dict[str, deep_architect.core.Output]*) – Function that is called with the values of hyperparameters and returns the inputs and the outputs of the network fragment to put in the place the substitution module currently is.
- **name_to_hyperp** (*dict[str, deep_architect.core.Hyperparameter]*) – Dictionary of name to hyperparameters that are needed for the substitution function. The names of the hyperparameters should be in correspondence to the name of the arguments of the substitution function.
- **input_names** (*list[str]*) – List of the input names of the substitution module.
- **output_name** (*list[str]*) – List of the output names of the substitution module.
- **scope** (*(deep_architect.core.Scope, optional)*) – registered. If none is given, uses the default scope.
- **allow_input_subset** (*bool*) – If true, allows the substitution function to return a strict subset of the names of the inputs existing before the substitution. Otherwise, the dictionary of inputs returned by the substitution function must contain exactly the same input names.
- **allow_output_subset** (*bool*) – If true, allows the substitution function to return a strict subset of the names of the outputs existing before the substitution. Otherwise, the dictionary of outputs returned by the substitution function must contain exactly the same output names.

_update()
Implements the substitution operation.

When all the hyperparameters that the module depends on are specified, the substitution operation is triggered, and the substitution operation is done.

deep_architect.modules.**buffer_io**(*inputs*, *outputs*)

deep_architect.modules.**dense_block**(*h_num_applies*, *h_end_in_combine*, *apply_fn*, *combine_fn*, *scope=None*, *name=None*)

deep_architect.modules.**get_hyperparameter_aggregators**(*outputs*)

deep_architect.modules.**hyperparameter_aggregator**(*name_to_hyperp*, *scope=None*, *name=None*)

deep_architect.modules.**identity**(*scope=None*, *name=None*)

Same as the Identity module, but directly works with dictionaries of inputs and outputs of the module.

See *Identity*.

Returns Tuple with dictionaries with the inputs and outputs of the module.

Return type (dict[str,*deep_architect.core.Input*], dict[str,*deep_architect.core.Output*])

deep_architect.modules.**mimo_nested_repeat**(*fn_first*, *fn_iter*, *h_num_repeats*, *input_names*, *output_names*, *scope=None*, *name=None*)

Nested repetition substitution module.

The first function function returns a dictionary of inputs and a dictionary of outputs, and it is always called once. The second function takes the previous dictionaries of inputs and outputs and returns new dictionaries of inputs and outputs. The names of the inputs and outputs returned by the functions have to match the names of the inputs and outputs of the substitution module. The resulting network fragment is used in the place of the substitution module.

Parameters

- **fn_first** (*() -> (dict[str, deep_architect.core.Input], dict[str, deep_architect.core.Output])*) – Function that returns the first network fragment, represented as dictionary of inputs and outputs.
- **fn_iter** (*(dict[str, deep_architect.core.Input], dict[str, deep_architect.core.Output]) -> (dict[str, deep_architect.core.Input], dict[str, deep_architect.core.Output])*) – Function that takes the previous dictionaries of inputs and outputs and it is applied to generate the new dictionaries of inputs and outputs. This function is applied one time less that the value of the hyperparameter for the number of repeats.
- **h_num_repeats** (deep_architect.core.Hyperparameter) – Hyperparameter for how many times should the iterative construct be repeated.
- **input_names** (*list[str]*) – List of the input names of the substitution module.
- **output_name** (*list[str]*) – List of the output names of the substitution module.
- **scope** (deep_architect.core.Scope, *optional*) – Scope in which the module will be registered. If none is give'n, uses the default scope.
- **name** (*str*, *optional*) – Name used to derive an unique name for the module. If none is given, uses the class name to derive the name.

Returns Tuple with dictionaries with the inputs and outputs of the substitution module.

Return type (dict[str,*deep_architect.core.Input*], dict[str,*deep_architect.core.Output*])

deep_architect.modules.**mimo_or**(*fn_lst*, *h_or*, *input_names*, *output_names*, *scope=None*, *name=None*)

Implements an or substitution operation.

The hyperparameter takes values that are valid indices for the list of possible substitution functions. The set of keys of the dictionaries of inputs and outputs returned by the substitution functions have to be the same as the set of input names and output names, respectively. The substitution function chosen is used to replace the current substitution module, with connections changed appropriately.

Note: The current implementation also works if `fn_lst` is an indexable object (e.g., a dictionary), and the `h_or` takes values that are valid indices for the indexable (e.g., valid keys for the dictionary).

Parameters

- **fn_lst** (*list[() -> (dict[str,deep_architect.core.Input], dict[str,deep_architect.core.Output])]*) – List of possible substitution functions.
- **h_or** (deep_architect.core.Hyperparameter) – Hyperparameter that chooses which function in the list is called to do the substitution.
- **input_names** (*list[str]*) – List of inputs names of the module.
- **output_names** (*list[str]*) – List of the output names of the module.
- **scope** (deep_architect.core.Scope, *optional*) – Scope in which the module will be registered. If none is given, uses the default scope.
- **name** (*str, optional*) – Name used to derive an unique name for the module. If none is given, uses the class name to derive the name.

Returns Tuple with dictionaries with the inputs and outputs of the substitution module.

Return type (dict[str,*deep_architect.core.Input*], dict[str,*deep_architect.core.Output*])

deep_architect.modules.**preproc_apply_postproc**(*preproc_fn*, *apply_fn*, *postproc_fn*)

deep_architect.modules.**siso_nested_repeat**(*fn_first*, *fn_iter*, *h_num_repeats*, *scope=None*, *name=None*)

Nested repetition substitution module.

Similar to *mimo_nested_repeat()*, the only difference being that in this case the function returns an or substitution module that has a single input and a single output.

The first function function returns a dictionary of inputs and a dictionary of outputs, and it is always called. The second function takes the previous dictionaries of inputs and outputs and returns new dictionaries of inputs and outputs. The resulting network fragment is used in the place of the current substitution module.

Parameters

- **fn_first** (*() -> (dict[str, deep_architect.core.Input], dict[str, deep_architect.core.Output])*) – Function that returns the first network fragment, represented as dictionary of inputs and outputs.
- **fn_iter** (*(dict[str, deep_architect.core.Input], dict[str, deep_architect.core.Output]) -> (dict[str, deep_architect.core.Input], dict[str, deep_architect.core.Output])*) – Function that takes the previous dictionaries of inputs and outputs and it is applied to generate the new dictionaries of inputs and outputs. This function is applied one time less that the value of the number of repeats hyperparameter.
- **h_num_repeats** (deep_architect.core.Hyperparameter) – Hyperparameter for how many times to repeat the iterative construct.

- **scope** (deep_architect.core.Scope, *optional*) – Scope in which the module will be registered. If none is given, uses the default scope.
- **name** (*str*, *optional*) – Name used to derive an unique name for the module. If none is given, uses the class name to derive the name.

Returns Tuple with dictionaries with the inputs and outputs of the substitution module.

Return type (dict[str,*deep_architect.core.Input*], dict[str,*deep_architect.core.Output*])

deep_architect.modules.**siso_optional**(*fn*, *h_opt*, *scope=None*, *name=None*)

Substitution module that determines to include or not the search space returned by *fn*.

The hyperparameter takes boolean values (or equivalent integer zero and one values). If the hyperparameter takes the value `False`, the input is simply put in the output. If the hyperparameter takes the value `True`, the search space is instantiated by calling *fn*, and the substitution module is replaced by it.

Parameters

- **fn** (*() -> (dict[str,*deep_architect.core.Input*], dict[str,*deep_architect.core.Output*])*) – Function returning a graph fragment corresponding to a sub-search space.
- **h_opt** (deep_architect.core.Hyperparameter) – Hyperparameter for whether to include the sub-search space or not.
- **scope** (deep_architect.core.Scope, *optional*) – Scope in which the module will be registered. If none is given, uses the default scope.
- **name** (*str*, *optional*) – Name used to derive an unique name for the module. If none is given, uses the class name to derive the name.

Returns Tuple with dictionaries with the inputs and outputs of the substitution module.

Return type (dict[str,*deep_architect.core.Input*], dict[str,*deep_architect.core.Output*])

deep_architect.modules.**siso_or**(*fn_lst*, *h_or*, *scope=None*, *name=None*)

Implements an or substitution operation.

The hyperparameter takes values that are valid indices for the list of possible substitution functions. The substitution function chosen is used to replace the current substitution module, with connections changed appropriately.

See also *mimo_or()*.

Note: The current implementation also works if `fn_lst` is an indexable object (e.g., a dictionary), and the `h_or` takes values that are valid indices for the dictionary.

Parameters

- **fn_lst** (*list[() -> (dict[str,*deep_architect.core.Input*], dict[str,*deep_architect.core.Output*])]*) – List of possible substitution functions.
- **h_or** (deep_architect.core.Hyperparameter) – Hyperparameter that chooses which function in the list is called to do the substitution.
- **input_names** (*list[str]*) – List of inputs names of the module.
- **output_names** (*list[str]*) – List of the output names of the module.
- **scope** (deep_architect.core.Scope, *optional*) – Scope in which the module will be registered. If none is given, uses the default scope.

- **name** (*str, optional*) – Name used to derive an unique name for the module. If none is given, uses the class name to derive the name.

Returns Tuple with dictionaries with the inputs and outputs of the substitution module.

Return type (dict[str,*deep_architect.core.Input*], dict[str,*deep_architect.core.Output*])

deep_architect.modules.**siso_permutation**(*fn_lst*, *h_perm*, *scope=None*, *name=None*)

Substitution module that permutes the sub-search spaces returned by the functions in the list and connects them sequentially.

The hyperparameter takes positive integer values that index the possible permutations of the number of elements of the list provided, i.e., factorial in the length of the list possible values (zero indexed). The list is permuted according to the permutation chosen. The search spaces resulting from calling the functions in the list are connected sequentially.

Parameters

- **fn_lst** (*list[() -> (dict[str,deep_architect.core.Input], dict[str,deep_architect.core.Output])]*) – List of substitution functions.
- **h_perm** (deep_architect.core.Hyperparameter) – Hyperparameter that chooses the permutation of the list to consider.
- **scope** (deep_architect.core.Scope, *optional*) – Scope in which the module will be registered. If none is given, uses the default scope.
- **name** (*str, optional*) – Name used to derive an unique name for the module. If none is given, uses the class name to derive the name.

Returns Tuple with dictionaries with the inputs and outputs of the substitution module.

Return type (dict[str,*deep_architect.core.Input*], dict[str,*deep_architect.core.Output*])

deep_architect.modules.**siso_repeat**(*fn*, *h_num_repeats*, *scope=None*, *name=None*)

Calls the function multiple times and connects the resulting graph fragments sequentially.

Parameters

- **fn** (*() -> (dict[str,deep_architect.core.Input], dict[str,deep_architect.core.Output])*) – Function returning a graph fragment corresponding to a sub-search space.
- **h_num_repeats** (deep_architect.core.Hyperparameter) – Hyperparameter for the number of times to repeat the search space returned by the function.
- **scope** (deep_architect.core.Scope, *optional*) – Scope in which the module will be registered. If none is given, uses the default scope.
- **name** (*str, optional*) – Name used to derive an unique name for the module. If none is given, uses the class name to derive the name.

Returns Tuple with dictionaries with the inputs and outputs of the substitution module.

Return type (dict[str,*deep_architect.core.Input*], dict[str,*deep_architect.core.Output*])

deep_architect.modules.**siso_residual**(*main_fn*, *residual_fn*, *combine_fn*)

Residual connection of two functions returning search spaces encoded as pairs of dictionaries of inputs and outputs.

The third function returns a search space to continue the main and residual path search spaces. See also: *siso_split_combine()*. The main and residual functions return search space graphs with a single input and a single output . The combine function returns a search space with two inputs and a single output.

Note: Specific names are assumed for the inputs and outputs of the different search spaces.

Parameters

- **main_fn** (*() -> (dict[str,deep_architect.core.Input], dict[str,deep_architect.core.Output])*) – Function returning the dictionaries of the inputs and outputs for the search space of the main path of the configuration.
- **residual_fn** (*() -> (dict[str,deep_architect.core.Input], dict[str,deep_architect.core.Output])*) – Function returning the dictionaries of the inputs and outputs for the search space of the residual path of the configuration.
- **combine_fn** (*() -> (dict[str,deep_architect.core.Input], dict[str,deep_architect.core.Output])*) – Function returning the dictionaries of the inputs and outputs for the search space for combining the outputs of the main and residual search spaces.

Returns Tuple with dictionaries with the inputs and outputs of the resulting search space graph.

Return type (dict[str,*deep_architect.core.Input*], dict[str,*deep_architect.core.Output*])

deep_architect.modules.**siso_sequential**(*io_lst*)

Connects in a serial connection a list of dictionaries of the inputs and outputs encoding single-input single-output search spaces.

Parameters **io_lst** (*list[(dict[str,deep_architect.core.Input], dict[str,deep_architect.core.Output])]*) – List of single-input single-output dictionaries encoding

Returns Tuple with dictionaries with the inputs and outputs of the resulting graph resulting from the sequential connection.

Return type (dict[str,*deep_architect.core.Input*], dict[str,*deep_architect.core.Output*])

deep_architect.modules.**siso_split_combine**(*fn*, *combine_fn*, *h_num_splits*, *scope=None*, *name=None*)

Substitution module that create a number of parallel single-input single-output search spaces by calling the first function, and then combines all outputs with a multiple-input single-output search space returned by the second function.

The second function returns a search space to combine the outputs of the branch search spaces. The hyperparameter captures how many branches to create. The resulting search space has a single input and a single output.

Note: It is assumed that the inputs and outputs of both the branch search spaces and the reduce search spaces are named in a specific way.

Parameters

- **fn** (*() -> (dict[str,deep_architect.core.Input], dict[str,deep_architect.core.Output])*) – Substitution function that return a single input and single output search space encoded by dictionaries of inputs and outputs.
- **combine_fn** (*(int) -> (dict[str,deep_architect.core.Input], dict[str,deep_architect.core.Output])*) – Returns a search space with a

number of inputs equal to the number of of branches and combines them into a single output.

- **h_num_splits** (deep_architect.core.Hyperparameter) – Hyperparameter for the number of parallel search spaces generated with the first function.
- **scope** (deep_architect.core.Scope, *optional*) – Scope in which the module will be registered. If none is given, uses the default scope.
- **name** (*str*, *optional*) – Name used to derive an unique name for the module. If none is given, uses the class name to derive the name.

Returns Tuple with dictionaries with the inputs and outputs of the resulting search space graph.

Return type (dict[str,*deep_architect.core.Input*], dict[str,*deep_architect.core.Output*])

deep_architect.modules.**substitution_module**(*name*, *substitution_fn*, *name_to_hyperp*, *input_names*, *output_names*, *scope=None*, *allow_input_subset=False*, *allow_output_subset=False*)

Same as the substitution module, but directly works with the dictionaries of inputs and outputs.

A dictionary with inputs and a dictionary with outputs is the preferred way of dealing with modules when creating search spaces. Using inputs and outputs directly instead of modules allows us to return graphs in the substitution function. In this case, returning a graph resulting of the connection of multiple modules is entirely transparent to the substitution function.

See also: *deep_architect.modules.SubstitutionModule*.

Parameters

- **name** (*str*) – Name used to derive an unique name for the module.
- **substitution_fn** (*dict[str, object] -> (dict[str,* deep_architect.core.Input*], dict[str,* deep_architect.core.Output*]*) – Function that is called with the values of hyperparameters and values of inputs and returns the inputs and the outputs of the network fragment to put in the place the substitution module currently is.
- **name_to_hyperp** (*dict[str,* deep_architect.core.Hyperparameter*]*) – Dictionary of name to hyperparameters that are needed for the substitution function. The names of the hyperparameters should be in correspondence to the name of the arguments of the substitution function.
- **input_names** (*list[str]*) – List of the input names of the substitution module.
- **output_name** (*list[str]*) – List of the output names of the substitution module.
- **scope** (deep_architect.core.Scope) – Scope in which the module will be registered.
- **allow_input_subset** (*bool*) – If true, allows the substitution function to return a strict subset of the names of the inputs existing before the substitution. Otherwise, the dictionary of inputs returned by the substitution function must contain exactly the same input names.
- **allow_output_subset** (*bool*) – If true, allows the substitution function to return a strict subset of the names of the outputs existing before the substitution. Otherwise, the dictionary of outputs returned by the substitution function must contain exactly the same output names.

Returns Tuple with dictionaries with the inputs and outputs of the module.

Return type (dict[str,*deep_architect.core.Input*], dict[str,*deep_architect.core.Output*])

3.1.4 deep_architect.search_logging

class deep_architect.search_logging.**EvaluationLogger**(*folderpath*, *search_name*, *evaluation_id*, *abort_if_exists=False*, *abort_if_notexists=False*)

Bases: object

Evaluation logger for a simple evaluation.

The logging is divided into config and results. Both are represented as JSON files in disk, i.e., dictionaries. The config JSON encodes the architecture to be evaluated. This encoding is tied to the search space the evaluation was drawn from, and it can be used to reproduce the architecture to be evaluated given the search space.

The results JSON contains the results of the evaluating the particular architecture. In the case of deep learning, this often involves training the architecture for a given task on a training set and evaluating it on a validation set.

Parameters

- **all_evaluations_folderpath** (*str*) – Path to the folder where all the evaluation log folders lie. This folder is managed by the search logger.
- **evaluation_id** (*int*) – Number of the evaluation with which the logger is associated with. The numbering starts at zero.

config_exists()

get_evaluation_data_folderpath()

Path to the user data folder where non-standard logging data can be stored.

This is useful to store additional information about the evaluated model, e.g., example predictions of the model, model weights, or model predictions on the validation set.

See *deep_architect.search_logging.EvaluationLogger.get_evaluation_folderpath()* for the path for the standard JSON logs for an evaluation.

Returns Path to the folder where the evaluations logs are written to.

Return type str

get_evaluation_folderpath()

Path to the evaluation folder where all the standard evaluation logs (e.g., `config.json` and `results.json`) are written to.

Only standard logging information about the evaluation should be written here. See *deep_architect.search_logging.EvaluationLogger.get_evaluation_data_folderpath()* for a path to a folder that can be used to store non-standard user logging information.

Returns Path to the folder where the standard logs about the evaluation are written to.

Return type str

log_config(*hyperp_value_lst*, *searcher_evaluation_token*)

Logs a config JSON that describing the evaluation to be done.

The config JSON has the list with the ordered sequence of hyperparameter values that allow to replicate the same evaluation given the same search space; the searcher evaluation token, that can be given back to the same searcher allowing it to update back its state. The searcher evaluation token is returned by the searcher

when a new architecture to evaluate is sampled. See, for example, `deep_architect.searchers.MCTSSearcher.sample()`. The format of the searcher evaluation token is searcher dependent, but it should be JSON serializable in all cases.

Creates `config.json` in the evaluation log folder.

Parameters

- **hyperp_value_lst** (*list[object]*) – List with the sequence of JSON serializable hyperparameter values that define the architecture to evaluate.
- **searcher_evaluation_token** (*dict[str, object]*) – Dictionary that is JSON serializable and it is enough, when given back to the searcher along with the results, for the searcher to update its state appropriately.

log_results(*results*)

Logs the results of evaluating an architecture.

The dictionary contains many metrics about the architecture.. In machine learning, this often involves training the model on a training set and evaluating the model on a validation set. In domains different than machine learning, other forms of evaluation may make sense.

Creates `results.json` in the evaluation log folder.

Parameters **dict[object]** – Dictionary of JSON serializable metrics and information about the evaluated architecture.

read_config()

read_results()

results_exist()

class deep_architect.search_logging.**SearchLogger**(*folderpath*, *search_name*, *abort_if_exists=True*, *delete_if_exists=False*)

Bases: object

get_evaluation_logger(*evaluation_id*, *abort_if_exists=False*, *abort_if_notexists=False*)

deep_architect.search_logging.**create_search_folderpath**(*folderpath*, *search_name*, *abort_if_exists=False*, *delete_if_exists=False*, *create_parent_folders=False*)

deep_architect.search_logging.**get_all_evaluations_folderpath**(*folderpath*, *search_name*)

deep_architect.search_logging.**get_evaluation_data_folderpath**(*folderpath*, *search_name*, *evaluation_id*)

deep_architect.search_logging.**get_evaluation_folderpath**(*folderpath*, *search_name*, *evaluation_id*)

deep_architect.search_logging.**get_search_data_folderpath**(*folderpath*, *search_name*)

deep_architect.search_logging.**get_search_folderpath**(*folderpath*, *search_name*)

deep_architect.search_logging.**is_search_log_folder**(*folderpath*)

deep_architect.search_logging.**read_evaluation_folder**(*evaluation_folderpath*)

Reads all the standard JSON log files associated to a single evaluation.

See also *deep_architect.search_logging.read_search_folder()* for the function that reads all the evaluations in a search folder.

Parameters **evaluation_folderpath** (*str*) – Path to the folder containing the standard JSON logs.

Returns Nested dictionaries with the logged information. The first dictionary has keys corresponding to the names of the standard log files.

Return type dict[str,dict[str,object]]

deep_architect.search_logging.**read_search_folder**(*search_folderpath*)
Reads all the standard JSON log files associated to a search experiment.

See also *deep_architect.search_logging.read_evaluation_folder()* for the function that reads a single evaluation folder. The list of dictionaries is ordered in increasing order of evaluation id.

Parameters **search_folderpath** (*str*) – Path to the search folder used for logging.

Returns List of nested dictionaries with the logged information. Each dictionary in the list corresponds to an evaluation.

Return type list[dict[str,dict[str,object]]]

deep_architect.search_logging.**recursive_list_log_folders**(*folderpath*)

deep_architect.search_logging.**recursive_read_search_folders**(*folderpath*)

3.1.5 deep_architect.searchers

3.1.6 deep_architect.helpers

3.1.7 deep_architect.surrogates

class deep_architect.surrogates.common.**SurrogateModel**
Bases: object

Abstract class for a surrogate model.

eval(*feats*)
Returns a prediction of performance (or other relevant metrics), given a feature representation of the architecture.

update(*val*, *feats*)
Updates the state of the surrogate function given the feature representation for the architecture and the corresponding ground truth performance metric.

Note: The data for the surrogate model may be kept internally. The update of the state of the surrogate function can be done periodically, rather than with each call to update. This can be done based on values for the configuration of the surrogate model instance.

---

deep_architect.surrogates.common.**extract_features**(*inputs*, *outputs*)
Extract a feature representation of a model represented through inputs and outputs.

This function has been mostly used for performance prediction on fully specified models, i.e., after all the hyperparameters in the search space have specified. After this, there is a single model for which we can compute an appropriate feature representation containing information about the connections that the model makes and the values of the hyperparameters.

Parameters

- **inputs** (*dict[str, deep_architect.core.Input]*) – Dictionary mapping names to inputs of the architecture.
- **outputs** (*dict[str, deep_architect.core.Output]*) – Dictionary mapping names to outputs of the architecture.

Returns Representation of the architecture as a dictionary where each key is associated to a list with different types of features.

Return type dict[str, list[str]]

3.1.8 deep_architect.visualization

CHAPTER 4

Contributing

4.1 Contributing

4.1.1 Introduction

We encourage contributions to DeepArchitect. If DeepArchitect has been useful for your work, please cite it and/or contribute to the codebase. We encourage everyone doing research in architecture search to implement their algorithms in DeepArchitect to make them widely available to other researchers and the machine learning community at large. This will significantly improve reproducibility and reusability of architecture search research.

Contributions can be a result of your own research or from implementations of existing algorithms. If you have developed a searcher, search space, evaluator, or any other component or functionality that would be useful to include in DeepArchitect, please make a pull request that follows the guidelines described in this document.

After reading this document, you will understand:

- what are the different types of contributions that we identify;
- what is the folder structure for contributions;
- what is required in terms of tests and documentation for different types of contributions;
- what are the different levels of conformity that we require for different types of contributions.

If you have a feature that you would like to add to DeepArchitect but you are unsure about its suitability, open a GitHub issue for discussion. This guarantees that your efforts are well-aligned with the project direction and needs. Consider including a code snippet or pseudo-code illustrating a useful use case for the feature.

4.1.2 Types of contributions

Most contributions will live in the contrib folder. The contrib folder is used for functionality that is likely useful, but that won't necessarily be maintained over time. While code lies in the contrib folder, the code owners are responsible for its maintenance. If code in the contrib folder breaks and the code owner does not fix it in a timely manner, we reserve the right to move the code to the dev folder. The dev folder contains code sketching interesting functionality

that may not be fully functional or it has not been refactored well enough to be integrated in contrib. Unmaintained code will be moved to dev upon breakage. Code in dev should not be used directly, but it can inspire further development.

Code that is part of the contrib folder may eventually be refactored into code that is part of the deep_architect folder. Similarly, code in the dev folder may be refactored in code that goes in the contrib folder. If code becomes part of the deep_architect folder, it becomes the responsibility of the developers of DeepArchitect to maintain it. To create a new contrib folder, it is best to first discuss its scope. We do not impose these restrictions for the dev folder. The dev folder should be used lightly though. We will only accept contributions to the dev folder if it they showcase important functionality and there is sufficient reason to justify their incompleteness. If the functionality is complete, we advise the contributor to refactor it into the contrib folder. Including the contribution in the contrib folder can be done either by adding it to an existing contrib subfolder, or by creating a new well-scoped contrib subfolder.

Cross-pollination between contrib and dev folders is expected and encouraged. For example, a few subcontrib folders already contain useful functionality, but a contributor may want to extend it and encapsulate it in a more coherent contrib subfolder. This scheme allows DeepArchitect to evolve without committing to major refactoring decisions upfront. If the foreseen contribution is better seen as an extension or a fix to an existing contrib folder, please open an issue or a pull request to discuss with the most active contributors on how to best incorporate the contribution in the existing files and folders. We may ask for refactoring changes or additional tests.

4.1.3 Required documentation and tests

Your new library in contrib should be placed in `deep_architect/contrib/$YOUR_LIBRARY_NAME`. New folders in contrib should include a `README.md` file providing information about the functionality that the library seeks to implement, the features that are implemented in the folder contributed, and an explanation about how the implementation is split between the different files and folders. Also include an explanation about when would it be natural to use the code in this library. This guarantees that a new user will quickly get a reasonable grasp of how to use the library and what files to look at for specific desired functionality. Comments for each major class and function are also recommended but not mandatory. Check the comments in `deep_architect/core.py` to get a sense of the style and format used for comments. It is also convenient to include in `README.md`, a roadmap for missing functionality that would be nice to include in the future. This informs future contributors about where the contributed project is going and compels them to help, e.g., if they believe that the feature is important.

The following is a typical structure for `README.md`: explanation of the problem that the contributed code tries to solve, some example code, a brief description of the high-level organization of the contributed library, and a roadmap for future work items and nice-to-haves and how other people can contribute to it, additional comments, GitHub handles of the code owners. If another contributor would like to extend an existing contributed library, it is best to reach out to the appropriate owner by writing an issue and mentioning the appropriate owner. The addition of significant new functionality requires adding more tests to exercise the newly developed code.

In addition to `README.md`, it is convenient to add tests and examples. The contributor should place tests in `tests/contrib/$YOUR_LIBRARY_NAME` and examples in `examples/contrib/$YOUR_LIBRARY_NAME`. Both `tests/contrib` and `examples/contrib` are meant to mostly reproduce the folder structure in `deep_architect/contrib`. This guarantees that removing a contributed library can be done easily by removing the corresponding folders in `deep_architect/contrib`, `tests/contrib`, and `examples/contrib`. While an example is not required, we do require a few tests to exercise the contributed code and have some guarantee that specific features remain correct as the contributed code and the development environment change.

4.1.4 Folder structure for contributions

For minimizing coupling between contributions of different people, we adopt a design similar to the one used in Tensorflow. Namely, we have a contrib folder where each new sufficiently different well-scoped contribution gets assigned a folder in `deep_architect/contrib`. The name of the folder should be chosen to reflect the functionality that lies within. All the library code contributed by the developer will be placed in this folder. Main files that are meant to be run should be placed in `examples/contrib` rather than in `deep_architect/contrib`. The same name

should be used for both the folder in `deep_architect/contrib` and in `examples/contrib`. The subfolder in `examples/contrib` is meant for runnable code related to or making extensive use of the library code in the `deep_architect/contrib` subfolder. We recommend checking existing examples in the repo for determining how to structure and document a new example appropriately.

Each configuration to run the example should be placed in a JSON configuration file `$CONFIG_NAME.json` in a folder named `configs` living in the same folder of the main file of the example. JSON configuration files guarantee that the options that determine the behavior of running the code can be kept separated from the code itself. This is more manageable, programmable, and configurable than having a command line interface. This guarantees that it is easy to maintain and store many different configurations, e.g., one configuration where the code is exercised with few resources and another configuration where the code is exercised in a longer run, e.g., see here. Each JSON file corresponds to a different configuration. We suggest including a `debug.json` to run a quick experiment to validate the functionality of both the code under `contrib/examples` and `deep_architect/contrib`. We recommend the use of configuration files for all but the most trivial examples. We often use the signature `python path/to/example/main.py -- config_filepath /path/to/config.json` for running examples, where we put all the configuration information in the JSON file.

Whether contributing examples or libraries, we recommend identifying the search spaces, searchers, evaluators, and datasets and splitting them into different files, e.g., see. Having these components into multiple files makes the dependencies more explicit and improves the reusability of the components. The framework is developed around these modular components. We recommend creating the following files when appropriate: `evaluators.py`, `search_spaces.py`, `searchers.py`, `main.py`, and `config.json`.

4.1.5 Development environment

The recommended code editor is Visual Studio Code with recommended plugins `ms-python.python`, `donjayamanne.githistory`, `eamodio.gitlens`, `donjayamanne.jupyter`, `yzhang.markdown-all-in-one`, `ban.spellright`. These can be installed through the extension tab or in the command line (after Visual Studio Code has been installed) with `code --install-extension $EXTENSION_NAME` where `EXTENSION_NAME` should be replaced by the name of each of the extensions.

We include VS Code settings with the repo which makes uses of yapf to automatically format the code on save. This will allow the contributor to effortlessly maintain formatting consistency with the rest of DeepArchitect.

We provide Singularity and Docker containers recipes for the development environment. These can found in `containers` along with additional information on how to build them.

We have attempted to maintain compatibility with both Python 2 and Python 3. There might exist places in the code base where this is not verified. If you find a place in the codebase that is not simultaneously Python 2 and Python 3 compatible, please issue a pull request fixing the problem.

4.1.6 Code style

All contributions should follow the code style used in most of the code base. When in doubt, mimic the style of `deep_architect`. Code in `deep_architect` is the most carefully designed. Getting the general gist of the design decisions that went in writing this code will help you write code that fits well with the existing code. We provide an autoformatter configuration for VS Code.

Readable variable names are preferred for function names, function arguments, class names, object attributes, object attributes, and dictionary keys. Names for iterator variables or local variables with a short lifespan can be shorter and slightly less readable. `deep_architect/core.py` (and the code in `deep_architect` in general) is a good place to get the gist of how these decisions influenced the naming conventions of the code. Function signatures should be readable without much documentation. Use four spaces for indentation. Upon submission of a pull request, some of these aspects will be reviewed to make sure that the level of conformity is appropriate for the type of contribution.

4.1.7 Examples of contributions

In this section, we identify the most natural contributions for DeepArchitect. These were identified to guarantee that DeepArchitect covers existing architecture search algorithms well. Other contributions are also very much encouraged.

Contributing a searcher

Searchers interact with the search space through a very simple interface: the searcher can ask if all the hyperparameters are specified (and therefore, if the specified search space can be compiled to a single model that can be evaluated); if the search space is not specified, the searcher can ask for a single unspecified hyperparameter and assign a value to it. When a value is assigned to an unspecified hyperparameter, the search space transitions, which sometimes gives rise to additional unspecified hyperparameters, e.g., after choosing the number of repetitions for a repetition substitution module.

The most general searchers rely solely on this simple interface. Good examples of general searchers implemented can be found here. In more specific cases, namely in reimplementations of searchers proposed in specific architecture search papers, there is some coupling between the search space and the searcher. In this case, the developed searcher expects the search space to have certain structure or properties. We recommend these types of searchers and search spaces to be kept in a contrib folder dedicated to the specific pair.

Searchers that work with arbitrary search space are preferred. Searchers that require specific properties from the search space are also often easily implemented in the framework. If the searcher requires specific search space properties, document this, e.g., by including example of search spaces that the searcher operates on, by discussing how do these differences compare with the most general case, and by discussing how are these differences supported by the DeepArchitect framework. All searchers should be accompanied by documentation, at the very least a docstring, and ideally both a docstring and an example exercising the searcher.

Contributing a search space

A search space encodes the set of architecture that will be under consideration by the searcher. Due to the compositionality properties of search spaces, e.g., through the use of substitution modules, or simply via the use of functions that allow us to create a larger search space from a number of smaller search spaces, new search spaces can be reused for defining yet other search spaces. A search space is as an encoding for the set of architectures that the expert finds reasonable, i.e., encodes the expert's inductive bias about the problem under consideration.

For certain search spaces, it may make sense to develop them in a framework independent way. For example, all substitution modules are framework independent. Certain search space functionality that takes other smaller search spaces and put them together into a larger search space are also often framework independent.

Due to the flexibility of the domain specific language in DeepArchitect, it is possible to have search spaces over structures different than deep architectures, e.g., it is possible to have search spaces over scikit-learn pipelines or arithmetic circuits. Due to the generic interface that the searchers use to interface with the search space, any existing general searchers can be directly applied to the problem at hand.

The goal of introducing new search spaces may be to explore new interesting structures and to make them available to other people that want to use them.

Contributing an evaluator

Evaluators determine the function that we are optimizing over the search space. If the evaluator does a poor job identifying the models that we in fact care about, i.e., the models that achieve high performance when trained to convergence, then the validity of running architecture search is undermined.

It is worth to consider the introduction of new evaluators for specific tasks. For example, if people in the field have found that specific evaluators (i.e., specific ways of training the models) are necessary to achieve high-performance, then it is useful replicate them.

Contributing a surrogate model

Sequential model-based optimization (SMBO) searchers use surrogate models extensively. Given a surrogate function predicting a quantity related to performance, a SMBO searcher optimizes this function (often approximately) to pick the architecture to evaluate next. The quantity predicted by the surrogate model does not need to be the performance metric of interest, it can simply be a score that preserves the ordering of the models in the space according to the performance metric of interest. Surrogate models also extend naturally to multi-objective optimization.

The quality of a surrogate function can be evaluated both by the quality of the search it induces, and by how effective it is in determining the relative ordering of models in the search space. A good surrogate functions should be able to embed the architecture to be evaluated and generate accurate predictions. It is unclear which surrogate models predict performance well. We ask contributors to explore different structures and validate their performance. Existing implementations of surrogate functions can be found here.

4.1.8 Conclusion

This document outlines guidelines for contributing to DeepArchitect. Having these guidelines in place guarantees that the focus is placed on the functionality developed, rather than on the specific arbitrary decisions taken to implement it. Please make sure that you understand the main points of this document, e.g., in terms of folder organization, documentation, code style, test requirements, and different types of contributions.

CHAPTER 5

Indices and Tables

- genindex
- modindex
- search

Python Module Index

d

Index

Symbols

A

B

C

D

E

R

S

T

U