PixelPipes Documentation

Release 0.0.8

Luka Cehovin Zajc

Feb 15, 2023

CONTENTS

PixelPipes provides a framework for creating infinite streams of data samples. As it is clear from the its name, PixelPipes is primarily intendend to be used on visual data with the emphasis on deep learning techniques.

Most deep models require a large amound of samples to be processed in a training phase. These samples have to be sampled from a dataset and bundled into batches that can be processed by the model. Besides sampling, another important concept in deep learning for computer vision is data augmentation where each real sample can be expanded in potentially infinite collection of modified samples.

All these steps involve a lot of work that is usually accomplished with many different libraries and is tied to a specific machine learning framework (e.g. PyTorch, TensorFlow). This makes the portability of the generation algorithm is limited. The sequence of training data is usually also not repeatable due to different pseudorandom generators involved, making debugging and experiment reproducably more challenging.

PixelPipes takes a different approach, it combines both sampling and augmentation into a single data processing pipeline of more or less compex operations. This way each sample in a virtually infinite sequence of samples is referenced only by its index (i.e. position) in the stream. The pipeline is constructed from a directed-acyclic graph (DAG) description and is optimized to reduce redundancy. This concept has the following benefits (some of which have not yet been implemented at this state of development, but are possible):

- Because of the way the pipeline is conditioned on a single index, the data stream is easily repeatable, the training or testing procedure reproducably end easy to debug.
- The pipeline is written in C++, making it fast and keeping the memory footprint reasonable. It also exploits the fact that individual samples are generated independently making synchronization easier.
- Each pipeline can be saved and loaded from a file. When loaded, the pipeline can be run from C++ directly, making the stream accessible from other languages besides Python.
- External file dependencies can also be tracked clearly, making data stream easily transferable.
- Despite a C++ core, the framework is extendable with new operations.

The source code for the PixelPipes framework is availabe on GitHub. Contributions to the project are welcome, please read the *development <development.html>* document on how to get started.

CHAPTER

ONE

OVERVIEW

1.1 Installation and quick-start

The source-code for the PixelPipes framwork can be downloaded from Github, but the most convenient way of testing it is by installing a prebuilt version. The package can be installed as a Python wheel package, currently from a testing PyPi compatible repository located here.

```
` > pip install pixelpipes -i https://data.vicos.si/lukacu/pypi/ `
```

1.1.1 Simple example

To demonstrate a very simple example of using a PixelPipes pipeline (without any image operations), lets sample random numbers from a pre-defined list and display them.

Note that this sequence will be the same every time you run the script with the same sample indices. This example is very simple, but shows how a pipleine is built. More complex examples with image operations are presented here.

1.2 Architecture and concepts

This document describes the concepts of the PixelPipes framework in extensive technical details. If you are more interested in practical examples, check out the collection of tutorials.

Similar to some machine learning frameworks, most notably TensorFlow, a PixelPipes stream is formalized as a computational directed acyclic graph (DAG). This graph is constructed in Python. The graph is then transformed into a sequence of operations that are executed in C++. These operations can be very primitive, e.g. summing up two numbers, or they can be very specialized, e.g. a specific image operation. Most users will only work with Python frontend to describe a stream, but it can be beneficial to know what lies beneath.

The framework is therefore divided into two parts with a binding API bridge:

- a C++ core containing all low-level operations together with some binding code,
- a Python frontend that provides a high-level way of describing a computational graph.

TODO: image overview

1.2.1 Python

As already said, the Python part of the framework is just a frontent that makes assembling a pipeline easy. It is not involved in the execution in any way and once a pipeline is assembled, it can be executed from C++ directly (or from any other language). The frontend makes graph description easier by introducing organizational concepts like macros and resources and by allowing the user to leverage abundance of Python tools to import data and transform it into pipeline primitives.

Below is a list of Python-level concepts that are important to know when you start using the framework:

- Node: represents an high-level operaton with zero or more inputs that produces a single output
- Graph: a collection of connected nodes, a graph describes dependencies between operations, each operation, represented as a node in a graph can accept zero or more inputs and produces a single output. There are two types of nodes, operations and macros.
- Operation: special nodes that map directly to individual operations in the final pipeline.
- Macro: A macro is a combination of operations that are frequently used together and are represented as a single node. During compilation macros are expanded to their inner subgraph until only the basic operation nodes remain. Macros are written in Python and combined normal Python language together with DAG generation, they can therefore base generation on input types, use loops and conditional statements.
- Constant: Special type of operation node, used to introduce constant values into the pipeline. Constats can be scalars, but also tensors and lists.
- Compilation: a process of iterativelly reducing macro nodes in a graph to basic operations, removing duplicates where possible and ordering operations according to dependencies.
- Resource: Special type of macros that handle multi-field type abstractions, can be used to make the graph more organized, but get dissolved during compilation.

1.2.2 C++

The native part of the framework is written in C++, this part is less accessible when only using the framework, but it is important for

- Operation: A stateless algorithm that accepts zero or more inputs and produces a single output. Direct mapping of the operation node concept.
- Token: Tokens are data units passed from operation to operation.
- Type: Describes type trait of a token.
- Pipeline: A sequence of operations.
- Module: Operations are organized into modules. Some modules are a part of the core framework and the separation is less noticeable. Other modules can be written as extensions to the framework, providing additional operations.

1.3 Tutorials

Below is a collection of tutorials covering different concepts and using different well-known datasets. The tutorials cover topics linearly, so each of them bases on the terminology used in the previous tutorials.

Note: Note that many examples use OpenCV python package for sample visualization purposes. The package is not a requirement for the PixelPipes package and has to be installed manually.

1.3.1 Building your first graph

A graph can be constructed by leveraging Python concepts like context and operator overloading (it can also be created manually by adding every node to a graph, but this is a much less readable approach). Lets start by extending the initial example of sampling numbers from a list. We can insert a list of lists, sample one of them then sample two elements from it and sum them together.

Note: Be careful when using graph contexts, use only one context in normal situations. The context mechanism supports also hierarchical contexts however, nodes will only be added to the most recent context opened.

Utility decorators wrap even more boilerplate for frequently used cases, the example above can be slightly modified so that entire function is wrapped in a graph context.

To compile the pipeline use

1.3.2 Sampling image patches

In this tutorial we will first sample random images from a directory, then we will cut rectangular patches from them at random positions. As you will see, all this can be achieved using a few operations, since suitable image manipulation macros are already available as standalone nodes.

For this example we will be using example images, located in *examples/images* directory. The entire code for the example is available here, lets look at the pipeline.

The pipeline first generates a list of images from a given directory. Then a random image entry is selected from a list. Images are resources, to actually load an image from a file, you have to access the *image* field. The sampling of a patch can be achieved using a combination of `pixelpipes.image.geometry.RandomPatchView` and `pixelpipes.image.geometry.ViewImage` macros.

1.3.3 Getting started with MNIST

MNIST is something like a hello-world dataset of deep learning community. In this example we will look at how to get real data into the pipeline and how to sample it. The full code for the example is available <mnist.py>, we will just commend on the core parts.

Sampling MNIST data

1.3.4 CIFAR and resource lists

CIFAR is one of the most well known datasets for computer vision, especially for representation learning. This tutorial will show how to generate a stream of random samples from CIFAR-10 collection. In contract to the previous tutorial on MNIST, we will introduce the concept of resources and resource lists as a high-level way of structuring data into datasets. The full code for the example is available here, we will just commend on the core parts.

Resources

First, a few words about resources. Resources are an organizational tool that helps with complex data. They allow us to group several data connections between nodes and group them together into a structure-like connections. Individual fields can then be queried and manipulated.

It is important to know that resources are built upon macros and are therefore not really used in the final pipeline, they are dissolved during compilation and all the their fields that are not required to produce stream output are stripped away.

Define a resource list

Lets define a macro that will produce a dataset resource. This can be done manually by overloading the Macro class, but it is recommended to use a ResourceListSource as a base since it makes this easier.

Note: This example assumes that you have downloaded the Python version of the CIFAR dataset from the dataset website and that you have extracted the files to the example directory.

The ResourceListSource expects subclasses to implement the load method that generates the fields, each field is expected to be a list or a NumPy array, they are also expected to be of equal length (for NumPy arrays this means the number of rows). We can also have virtual fields that generate a snippet once their content is requested, but this is a topic for another tutorial where such fields are needed.

Since we have defined the dataset as a resource list, the final graph is now quite simple:

Notice that special macros are available to process resource lists efficiently. In this case a random resource is sampled from a list and its two fields are returned as output.

1.3.5 Augmenting images

Augmenting images can be viewed as generating multiple different samples from a same image by transforming it with image processing operations that change its pixel-level congent, but not its semantic meaning.

1.3.6 VOT and segmented resource lists

Visual Object Tracking (VOT) datasets are sequence datasets for visual tracking performance evaluation. Usually, these kind of datasets would not be used for training, however, other similar datasets exist and the example can be adapted for them. In this example we will introduce a segmented resource sequence, a data abstraction for storing many sequences of resources into a single list. We will also use the official toolkit of the VOT challenge to retrieve data. Note that the toolkit is not required anymore once a pipeline is constructed (and serialized).

1.3.7 Batching data for training

Simply generating data sequentially is ok for testing pipeline output, but for training deep models using SGD or related optimization methods, we would like to efficiently generate batches of samples utilizing multiple cores. Since the execution of pipeline is done in C++, this is possible to do from Python using a thread pool. But the frameworks also provides helper classes called sinks that do this. Special sinks are provided for the most popular deep learning frameworks that anable easy integration.

One important thing that sinks assume is that all outputs are scalars or have a fixed size for every sample. This allows stacking into tensors that are necessary for efficient deep learning.

For the bervity of examples below we will be using the MNIST pipeline that we have created in the MNIST tutorial.

NumPy sink

The default sink generates a tuple of

PyTorch sink

A simple example on how to download, prepare and convert PyTorch MNIST dataset into acceptable type for injecting it into pixelpipes graph.

TensorFlow sink

A simple example on how to download, prepare and convert TensorFlow MNIST dataset into acceptable type for injecting it into pixelpipes graph.

1.4 List of nodes

Despite being Python classes, nodes follow a consistent logic are documented separately from the rest of the API. Below is a list of nodes included in the PixelPipes core.

1.4.1 Core nodes

operation pixelpipes.graph.**Constant**(*value*)

Generates a constant in the pipeline

value [any]:

operation pixelpipes.graph.**Debug**(*source*, *prefix*)

Debug operation enables low-level terminal output of the content that is provided to it. The token content will usually not be printed entierly, only its shape, the value will only be displayed for simple scalar types as well as strings.

Note that tese nodes will be passed to the pipeline only if the compiler is configered with debug flag, otherwise they will be stripped from the graph.

source [Token [None]]: Result of which node to print

prefix [str =]: String that is prepended to the output

operation pixelpipes.graph.**Output**(*output*, *label*)

Output node that accepts a single input, enables outputting tokens from the final pipeline. Tokens are returned as a tuple, their order is determined by the order of adding output nodes to the graph. Additionally you may also label outputs with non-unique lables that can be used to resolve outputs.

output [Token [None]]: Output token

label [str = default]: Nonunique label of the output

operation pixelpipes.graph.**RandomSeed**

Returns a pseudo-random number, useful for initializing pseudo-random operations. The seed itself is sampled from a pseudo-random generator that produces the same sequence of seeds for a specific position in the data sequence. This is the corner-stone of repeatability of the pipeline.

operation pixelpipes.graph.**ReadFile**(*filename*)

Read file from disk to memory buffer. File is read in binary mode.

filename [Token [char, None]]: Path to the image file

operation pixelpipes.graph.**SampleIndex**

Returns current sample index. This information can be used instead of random seed to initialize random generators where sequential consistentcy is required.

node pixelpipes.compiler.**Variable**(*name*, *default*)

Variable placeholder that can be overriden later

name [str]:

default []:

1.4.2 Scalar operations

operation pixelpipes.numbers.**Add**(*a*, *b*)

a [Token [float]]: First operand

b [Token [float]]: Second operand

operation pixelpipes.numbers.**Ceil**(*source*)

Ceil number and convert to integer.

source [Token [float]]: Number on which ceil operation is performed

operation pixelpipes.numbers.**Divide**(*a*, *b*)

a [Token [float]]: First operand

b [Token [float]]: Second operand

operation pixelpipes.numbers.**Equal**(*a*, *b*)

a [Token [float]]: First operand

b [Token [float]]: Second operand

operation pixelpipes.numbers.**Floor**(*source*)

Floor number and convert to integer.

source [Token [float]]: Number to be rounded

operation pixelpipes.numbers.**Greater**(*a*, *b*)

a [Token [float]]: First operand

b [Token [float]]: Second operand

operation pixelpipes.numbers.**GreaterEqual**(*a*, *b*)

a [Token [float]]: First operand

b [Token [float]]: Second operand

operation pixelpipes.numbers.**Lower**(*a*, *b*)

a [Token [float]]: First operand

b [Token [float]]: Second operand

operation pixelpipes.numbers.**LowerEqual**(*a*, *b*)

a [Token [float]]: First operand

b [Token [float]]: Second operand

operation pixelpipes.numbers.**Maximum**(*a*, *b*)

a [Token [float]]: First operand

b [Token [float]]: Second operand

operation pixelpipes.numbers.**Minimum**(*a*, *b*)

a [Token [float]]: First operand

b [Token [float]]: Second operand

operation pixelpipes.numbers.**Modulo**(*a*, *b*)

a [Token [int]]:

b [Token [int]]:

operation pixelpipes.numbers.**Multiply**(*a*, *b*)

a [Token [float]]: First operand

b [Token [float]]: Second operand

operation pixelpipes.numbers.**NotEqual**(*a*, *b*)

a [Token [float]]: First operand

b [Token [float]]: Second operand

operation pixelpipes.numbers.**Power**(*a*, *b*)

a [Token [float]]: First operand

b [Token [float]]: Second operand

macro pixelpipes.numbers.**RandomBoolean**(*seed*)

Samples a boolean value with equal probability

seed [Token [int] = @[random]]:

operation pixelpipes.numbers.**Round**(*source*)

Round number to closest integer and convert to integer type.

source [Token [float]]: Number to be rounded

operation pixelpipes.numbers.**SampleNormal**(*mean*, *sigma*, *seed*)

Samples values between from normal distribution.

mean [Token [float] = 0]: Mean value of normal distribution

sigma [Token [float] = 1]: Standard deviation

seed [Token [int] = @[random]]:

operation pixelpipes.numbers.**SampleUnform**(*min*, *max*, *seed*)

Samples random value between min and max value.

min [Token [float]]: Minimun value

max [Token [float]]: Maximum value

seed [Token [int] = @[random]]:

operation pixelpipes.numbers.**Subtract**(*a*, *b*)

a [Token [float]]: First operand

b [Token [float]]: Second operand

operation pixelpipes.numbers.**TensorAdd**(*a*, *b*, *saturate*)

a [Token [None]]: First operand

b [Token [None]]: Second operand

saturate [bool = False]: Saturate cast

operation pixelpipes.numbers.**TensorDivide**(*a*, *b*, *saturate*)

Divides image with another image or scalar (per-element multiplication).

a [Token [None]]: First operand

b [Token [None]]: Second operand

saturate [bool = False]: Saturate cast

operation pixelpipes.numbers.**TensorMultiply**(*a*, *b*, *saturate*)

Multiplies image with another image or scalar (per-element multiplication).

a [Token [None]]: First operand

b [Token [None]]: Second operand

saturate [bool = False]: Saturate cast

operation pixelpipes.numbers.**TensorSubtract**(*a*, *b*, *saturate*)

Subtracts two images with same size and number of channels or an image and a number.

a [Token [None]]: First operand

b [Token [None]]: Second operand

saturate [bool = False]: Saturate cast

macro pixelpipes.expression.**Expression**(*source*, *variables*)

Numeric expression with variables

Macro that expands into an arithmetic expression parsed from an input string.

Inputs:

- source: text representation of arithmetic expression
- variables: a map of inputs that are inserted into the expression

Category: arithmetic, macro

source [str]:

variables []:

1.4.3 Flow control

operation pixelpipes.flow.**Conditional**(*true*, *false*, *condition*)

Node that executes conditional selection, output of branch "true" will be selected if the "condition" is not zero, otherwise output of branch "false" will be selected. Note that the inferred type of these two branches should match as much as possible, otherwise the inferred type of this node will cause problems with dependent nodes.

true [Token [None]]: Use this data if condition is true

false [Token [None]]: Use this data if condition is false

condition [Token [int]]: Condition to test

macro pixelpipes.flow.**Switch**(*inputs*, *weights*, *seed*)

Random switch between multiple branches, a macro that generates a tree of binary choices based on a random variable. The probability of choosing a defined branch

inputs []: Two or more input branches

weights []: Corresponing branch weights

seed [Token [int] = @[random]]:

1.4.4 Lists

operation pixelpipes.list.**CompareEqual**(*a*, *b*)

a [Token [float, None]]:

b [Token [float, None]]:

operation pixelpipes.list.**CompareGreater**(*a*, *b*)

a [Token [float, None]]:

b [Token [float, None]]:

operation pixelpipes.list.**CompareGreaterEqual**(*a*, *b*)

a [Token [float, None]]:

b [Token [float, None]]:

operation pixelpipes.list.**CompareLower**(*a*, *b*)

a [Token [float, None]]:

b [Token [float, None]]:

operation pixelpipes.list.**CompareLowerEqual**(*a*, *b*)

a [Token [float, None]]:

b [Token [float, None]]:

operation pixelpipes.list.**CompareNotEqual**(*a*, *b*)

a [Token [float, None]]:

b [Token [float, None]]:

operation pixelpipes.list.**Concatenate**(*inputs*)

inputs []: Two or more input lists

operation pixelpipes.list.**FileList**(*list*)

String list of file patchs. Use this operation to inject file dependencies into the pipeline.

list []:

operation pixelpipes.list.**FilterSelect**(*parent*, *filter*)

Generate a sublist based on values from a filter list

parent [Token [None]]:

filter [Token [int, None]]:

operation pixelpipes.list.**GetElement**(*parent*, *index*)

Returns an element from a list for a given index

parent [Token [None]]:

index [Token [int]]:

macro pixelpipes.list.**GetRandom**(*source*, *seed*)

source [Token [None]]:

seed [Token [int] = @[random]]:

operation pixelpipes.list.**Length**(*parent*)

Returns a list length

parent [Token [None]]:

operation pixelpipes.list.**ListAsTable**(*parent*, *row*)

Transform list to table

parent [Token [None, None]]: Source list

row [Token [int]]: Row size, total length of list must be its multiple

operation pixelpipes.list.**LogicalAnd**(*a*, *b*)

a [Token [bool, None]]:

b [Token [bool, None]]:

operation pixelpipes.list.**LogicalNot**(*a*)

a [Token [bool, None]]:

operation pixelpipes.list.**LogicalOr**(*a*, *b*)

a [Token [bool, None]]:

b [Token [bool, None]]:

operation pixelpipes.list.**MakeList**(*inputs*)

Builds list from inputs. All inputs should be of the same type as the first input, it determines the type of a list.

inputs []:

operation pixelpipes.list.**Modulo**(*a*, *b*)

a [Token [int, None]]:

b [Token [int, None]]:

operation pixelpipes.list.**Permutation**(*length*, *seed*)

Generates a list of numbers from 0 to length in random order.

length [Token [int]]:

seed [Token [int] = @[random]]:

operation pixelpipes.list.**Permute**(*source*, *seed*)

Randomly permutes an input list

source [Token [None]]: Input list

seed [Token [int] = @[random]]:

operation pixelpipes.list.**Range**(*start*, *end*, *length*, *round*)

Generates a list of numbers from start to end of a given length

start [Token [float]]:

end [Token [float]]:

length [Token [int]]:

round [Token [bool] = False]:

operation pixelpipes.list.**Remap**(*source*, *indices*)

Maps elements from source list to a result list using indices from indices list.

source [Token [None]]:

indices [Token [int, None]]:

operation pixelpipes.list.**Repeat**(*source*, *length*)

Repeat list element a number of times

source [Token [None]]: Element to repeat

length [Token [int]]: Number of repetitions

operation pixelpipes.list.**SublistSelect**(*parent*, *begin*, *end*)

Selects a range from the source list as a new list.

parent [Token [None]]: Source list

begin [Token [int]]: Start index

end [Token [int]]: End index

macro pixelpipes.list.**Table**(*source*)

Constant Table

Inputs:

- source: Table type

Category: list

source []:

1.4.5 Geometery

1.4.6 Images

1.4.7 Resources

macro pixelpipes.resource.**AppendField**(*source*, *name*, *value*)

Produce a resource from an input resource and another field. Essentially just node renaming.

source [Resource ()]: Original resource

name [str]: Name of new field

value [Token [None]]: Value for new field

macro pixelpipes.resource.**ConditionalResource**(*true*, *false*, *condition*)

Node that executes conditional selection, output of branch "true" will be selected if the "condition" is not zero, otherwise output of branch "false" will be selected.

true [Resource ()]: Use this data if condition is true

false [Resource ()]: Use this data if condition is false

condition [Token [int]]: Condition to test

macro pixelpipes.resource.**GetField**(*source*, *element*)

This macro exposes only selected field of an input structure as an output, enabling processing of that data.

source [Resource ()]: Input resource

element [str]: Name of the structure field

macro pixelpipes.resource.**MakeResource**(*inputs*)

Macro that generates a resource from given inputs

inputs [= {}]: A map of inputs that are inserted into the expression

macro pixelpipes.resource.list.**GetLastResource**(*resources*)

resources [Resource (__list_length)]:

macro pixelpipes.resource.list.**GetResource**(*resources*, *index*)

resources [Resource (__list_length)]:

index [Token [int]]:

macro pixelpipes.resource.list.**GetResourceListLength**(*resources*)

resources [Resource (__list_length)]:

macro pixelpipes.resource.list.**ListInterval**(*resources*, *begin*, *end*)

resources [Resource (__list_length)]:

begin [Token [int]]:

end [Token [int]]:

macro pixelpipes.resource.list.**PermuteResourceSegments**(*resources*, *seed*)

resources [Resource (__list_length, __list_seg_begin, __list_seg_end)]:

seed [Token [int] = @[random]]:

macro pixelpipes.resource.list.**PermuteResources**(*resources*)

Randomly permutes the resource list

resources [Resource (__list_length)]:

macro pixelpipes.resource.list.**RandomResource**(*resources*, *seed*)

Select a random resource from an input list of resources

resources [Resource (__list_length)]:

seed [Token [int] = @[random]]:

macro pixelpipes.resource.list.**RandomResourceSegment**(*resources*, *seed*)

resources [Resource (__list_length, __list_seg_begin, __list_seg_end)]:

seed [Token [int] = @[random]]:

macro pixelpipes.resource.list.**RepeatResource**(*resource*, *length*)

Returns a list of resources where an input resource is repeated a number of times

resource [Resource ()]: Resource to repeat

length [Token [int]]: Number of repetitions

macro pixelpipes.resource.list.**ResourceSegment**(*resources*, *index*)

resources [Resource (__list_length, __list_seg_begin, __list_seg_end)]:

index [Token [int]]:

macro pixelpipes.resource.list.**SegmentCount**(*resources*)

resources [Resource (__list_length, __list_seg_begin, __list_seg_end)]:

1.5 API documentation

The reference API documentation for the Python wrapper as well as the C++ exported symbols. Note that the wrapper documentation does not contain node classes that are instead documented separately as a list of available nodes..

1.5.1 Python API

Pipeline

class pixelpipes.**LazyLoadEnum**(*name*)

Bases: Mapping

Special enum class used to load mappings from the core library when they are needed for the first time.

class pixelpipes.**Pipeline**(*data: Iterable[*PipelineOperation*]*, *optimize=True*)

Bases: object

Wrapper for the C++ pipeline object, includes additional metadata. This wrapper should be used instead of interacting with the C++ object directly.

property metadata: mappingproxy

Accesses the pipeline metadata storage.

Returns:

MappingProxyType: A string to string key-value storage.

property outputs: List[str]

Returns labels for individual elements of the output tuple.

run(*index: int*) → Tuple[np.ndarray]

Executes the pipeline for a given index and resturns result

Args:
index (int): Index of sample to generate. Starts with 1.

Returns:
Tuple[np.ndarray]: Generated sample, a sequence of NumPy objects.

class pixelpipes.**PipelineOperation**(*id*, *name*, *arguments*, *inputs*)

Bases: tuple

property arguments

Alias for field number 2

property id

Alias for field number 0

property inputs

Alias for field number 3

property name

Alias for field number 1

pixelpipes.**include_dirs**() → List[str]

Returns a list of directories with C++ header files for pixelpipes core library. Useful when building pixelpipes modules.

Returns:
List[str]: List of directories

pixelpipes.**link_dirs**() → List[str]

Returns a list of directories where a pixelpipe library can be found. Useful when building pixelpipes modules.

Returns:
List[str]: List of directories

pixelpipes.**load_module**(*name*) → bool

pixelpipes.**read_pipeline**(*filename: str*)

pixelpipes.**visualize_pipeline**(*pipeline:* Pipeline)

pixelpipes.**write_pipeline**(*filename: str*, *pipeline:* Pipeline, *compress: Optional[bool] = True*) → None

Serializes pipeline to a file with optional compression

Args:
filename (str): Filename to use. pipeline (Pipeline): Pipeline to serialize. compress (Optional[bool], optional): Use GZIP compression or not. Defaults to True.

Graph

operation pixelpipes.graph.**Constant**(*value*)

Generates a constant in the pipeline

value [any]:

node pixelpipes.graph.**Copy**(*source*)

source [Token [None]]:

operation pixelpipes.graph.**Debug**(*source*, *prefix*)

Debug operation enables low-level terminal output of the content that is provided to it. The token content will usually not be printed entierly, only its shape, the value will only be displayed for simple scalar types as well as strings.

Note that tese nodes will be passed to the pipeline only if the compiler is configered with debug flag, otherwise they will be stripped from the graph.

source [Token [None]]: Result of which node to print

prefix [str =]: String that is prepended to the output

class pixelpipes.graph.**EnumerationInput**(*options*, *default=None*, *description=''*)

Bases: *Input*

coerce(*value*, _)

dump(*value*)

class pixelpipes.graph.**Graph**(*prefix: Optional[Union[str,* Reference*]] = ''*)

Bases: object

add(*node: Node*, *name: Optional[Union[str,* Reference*]] = None*)

static add_default(*node: Node*, *name: str*) → bool

commit()

copy()

static default() → *Graph*

static has_default()

nodes()

pipeline(*fixedout=False*, *variables=None*, *output=None*)

reference(*node: Node*)

remove(*node: Union[Node,* Reference*]*)

replace(*oldnode: Union[Node,* Reference*]*, *newnode: Node*)

subgraph(*prefix: Optional[Union[str,* Reference*]] = ''*) → *Graph*

class pixelpipes.graph.**InferredReference**(*ref: str*, *typ:* Data)

Bases: *Reference*, *OperationProxy*

A node reference with type already inferred. Using during compilation in macro expansion.

property type

class pixelpipes.graph.**Input**(*reftype:* Data, *default: Optional[Union[str, float, int]] = None*, *description: Optional[str] = ''*)

Bases: Attribute

coerce(*value*, _)

dump(*value*)

reftype()

macro pixelpipes.graph.**Macro**

node pixelpipes.graph.**Node**

Base class for all nodes in a computation graph.

exception pixelpipes.graph.**NodeException**(**args*, *node: Optional[Node] = None*)

Bases: Exception

property node

nodestack()

print_nodestack()

class pixelpipes.graph.**NodeOperation**(*value*)

Bases: Enum

An enumeration.

ADD = 4

DIVIDE = 7

EQUAL = 10

GREATER = 14

GREATER_EQUAL = 15

INDEX = 2

LENGTH = 3

LOWER = 12

LOWER_EQUAL = 13

MODULO = 9

MULIPLY = 6

NEGATE = 1

NOT_EQUAL = 11

POWER = 8

SUBTRACT = 5

operation pixelpipes.graph.**Operation**

Base class of all atomic nodes that generate pipeline operations

class pixelpipes.graph.**OperationProxy**

Bases: object

static query_operation(*operation:* NodeOperation, **qargs:* Data)

static register_operation(*operation:* NodeOperation, *generator: Callable*, **args:* Data)

operation pixelpipes.graph.**Output**(*output*, *label*)

Output node that accepts a single input, enables outputting tokens from the final pipeline. Tokens are returned as a tuple, their order is determined by the order of adding output nodes to the graph. Additionally you may also label outputs with non-unique lables that can be used to resolve outputs.

output [Token [None]]: Output token

label [str = default]: Nonunique label of the output

operation pixelpipes.graph.**RandomSeed**

Returns a pseudo-random number, useful for initializing pseudo-random operations. The seed itself is sampled from a pseudo-random generator that produces the same sequence of seeds for a specific position in the data sequence. This is the corner-stone of repeatability of the pipeline.

operation pixelpipes.graph.**ReadFile**(*filename*)

Read file from disk to memory buffer. File is read in binary mode.

filename [Token [char, None]]: Path to the image file

class pixelpipes.graph.**Reference**(*ref: Union[str,* Reference*]*)

Bases: object

property name

static parse(*value*)

operation pixelpipes.graph.**SampleIndex**

Returns current sample index. This information can be used instead of random seed to initialize random generators where sequential consistentcy is required.

class pixelpipes.graph.**SeedInput**(*description=''*)

Bases: *Input*

exception pixelpipes.graph.**ValidationException**(**args*, *node: Optional[Node] = None*)

Bases: *NodeException*

pixelpipes.graph.**hidden**(*node_class*)

pixelpipes.graph.**outputs**(**inputs*, *label='default'*)

pixelpipes.graph.**wrap_pybind_enum**(*bindenum*)

Compiler

A compiler converts a graph to a sequence of operations.

class pixelpipes.compiler.**Compiler**(*fixedout=False*, *debug=False*)

Bases: object

Compiler object contains utilities to validate a graph and compiles it to a pipeline (a sequence of operations, written in native code) that can be executed to obtain output variables.

build(*graph:* Graph, *variables: Optional[Mapping[str, Number]] = None*, *output: Optional[Union[Container, Callable]] = None*, *optimize=True*) → *Pipeline*

Compiles the graph and builds a pipeline from it in one function.

Args:
graph (Graph): _description_ variables (typing.Optional[typing.Mapping[str, numbers.Number]], optional): _description_. Defaults to None. output (typing.Optional[typing.Union[Container, typing.Callable]], optional): _description_. Defaults to None. optimize (bool, optional): Optimize conditional operations by inserting jumps into the pipeline.

Returns:
Pipeline: Pipeline object

static build_graph(*graph: Union[*Graph*, Mapping[str, Node]]*, *variables: Optional[Mapping[str, Number]] = None*, *output: Optional[str] = None*, *fixedout: bool = False*) → *Pipeline*

compile(*graph:* Graph, *variables: Optional[Mapping[str, Number]] = None*, *output: Optional[Union[Container, Callable]] = None*) → Iterable[*PipelineOperation*]

Compile a graph into a pipeline of native operations.

Args:
graph (Graph): Graph representation

Raises:
CompilerException: raised if graph is not valid

Returns:
engine.Pipeline: resulting pipeline

validate(*graph: Union[*Graph*, Mapping[str, Node]]*)

Validates graph by interring input and output types for all nodes. An exception will be thrown if dependencies cannot be resolved or if output of a node is not compatible with an input specification of a dependant node.

Args:
graph (typing.Mapping or Graph): Graph representation

Raises:
ValidationException: Different validation errors share this exception type

Returns:
dict: resolved types of all nodes

exception pixelpipes.compiler.**CompilerException**

Bases: Exception

node pixelpipes.compiler.**Variable**(*name*, *default*)

Variable placeholder that can be overriden later

name [str]:

default []:

pixelpipes.compiler.**infer_type**(*node: Union[Reference, str]*, *graph: Optional[Graph] = None*, *type_cache: Optional[Mapping[str, Data]] = None*) → *Data*

Computes output type for a given node by recursively computing types of its dependencies and calling validate method of a node with the information about their computed output types.

Args:
node (typing.Union[Reference, typing.Type[Node]]): Reference of the node or raw value graph (Graph): Mapping of all nodes in the graph type_cache (typing.Mapping[str, types.Type], optional): Optional cache for already computed types. Makes repetititve calls much faster. Defaults to None.

Raises:
ValidationException: Contains information about the error during node validation process.

Returns:
types.Type: Computed type for the given node.

pixelpipes.compiler.**toposort**(*data*)

Dependencies are expressed as a dictionary whose keys are items and whose values are a set of dependent items. Output is a list of sets in topological order. The first set consists of items with no dependences, each subsequent set consists of items that depend upon items in the preceeding sets.

Sinks

Sink is a utility class that execute a pipeline in multiple threads and stack sample outputs to batches.

class pixelpipes.sink.**AbstractDataLoader**(*batch: int*, *workers: Optional[Union[int, WorkerPool]] = None*, *offset: int = 0*)

Bases: object

class **_BatchIterator**(*commit*, *size: int*, *offset: int = 0*)

Bases: *BatchIterator*

benchmark(*n=100*)

class pixelpipes.sink.**BatchIterator**(*commit*, *size: int*, *offset: int = 0*)

Bases: object

Abstract batch iterator base with most functionality for consumer agnostic multithreaded batching of samples.

class pixelpipes.sink.**PipelineDataLoader**(*pipeline: Pipeline*, *batch: int*, *workers: Optional[Union[int, WorkerPool]] = None*, *offset: Optional[int] = 0*)

Bases: *AbstractDataLoader*

class **_BatchIterator**(*commit*, *size: int*, *offset: int = 0*)

Bases: *BatchIterator*

property pipeline

class pixelpipes.sink.**WorkerPool**(*max_workers: int = 1*)

Bases: ThreadPoolExecutor

Utilities

Utilites for more efficient common usecases.

class pixelpipes.utilities.**Counter**

Bases: object

Object based counter, each time it is called it returns a value greater by 1

class pixelpipes.utilities.**PersistentDict**(*root: str*)

Bases: object

A dictionary interface to a folder, with memory caching.

pixelpipes.utilities.**collage**(*pipeline:* Pipeline, *index: int*, *rows: int*, *columns: int*, *offset: Optional[int] = 0*) → ndarray

pixelpipes.utilities.**find_nodes**(*module=None*)

pixelpipes.utilities.**graph**(*constructor*)

pixelpipes.utilities.**pipeline**(*variables=None*, *fixedout=False*, *debug=False*)

Types

Token type representation wrapper.

class pixelpipes.types.**Anything**

Bases: *Data*

Denotes type that accepts all inputs.

castable(*typ:* Data)

Can object of given input type description be casted to this type.

pixelpipes.types.**Boolean**()

pixelpipes.types.**BooleanList**(*length=None*)

pixelpipes.types.**Buffer**(*length=None*)

pixelpipes.types.**Char**()

class pixelpipes.types.**Data**

Bases: object

Abstract type base, represents description of token types accepted or returned by nodes.

castable(*typ:* Data) → bool

Can object of given input type description be casted to this type.

common(*typ:* Data) → *Data*

Merge two types by finding their common type. By default this just looks if one type is castable into the other.

pixelpipes.types.**Float**()

pixelpipes.types.**FloatList**(*length=None*)

pixelpipes.types.**Image**(*width: Optional[int] = None*, *height: Optional[int] = None*, *channels: Optional[int] = None*, *depth: Optional[str] = None*)

Represents an image type. This type can be specialized with image width, height, number of channels as well as bit-depth.

pixelpipes.types.**Integer**()

pixelpipes.types.**IntegerList**(*length=None*)

pixelpipes.types.**List**(*element=None*, *length=None*)

Type that represents a list of elements.

pixelpipes.types.**Point**()

pixelpipes.types.**Points**(*length=None*)

pixelpipes.types.**Rectangle**()

pixelpipes.types.**Short**()

pixelpipes.types.**String**(*length=None*)

class pixelpipes.types.**Token**(*element=None*, **shape*)

Bases: *Data*

castable(*typ:* Data) → bool

Can object of given input type description be casted to this type.

common(*typ:* Data) → *Data*

Merge two types by finding their common type. By default this just looks if one type is castable into the other.

property element

pop()

push(*length=None*)

property rank

squeeze()

exception pixelpipes.types.**TypeException**

Bases: Exception

class pixelpipes.types.**Union**(**args:* Data)

Bases: *Data*

Denotes type that accepts any of the given inputs. Do not nest unions.

castable(*typ:* Data) → bool

Can object of given input type description be casted to this type.

common(*typ:* Data) → *Data*

Merge two types by finding their common type. By default this just looks if one type is castable into the other.

pixelpipes.types.**UnsignedChar**()

pixelpipes.types.**UnsignedShort**()

pixelpipes.types.**View**()

class pixelpipes.types.**Wildcard**(*element=None*, *mindim=None*, *maxdim=None*)

Bases: *Token*

castable(*typ:* Data) → bool

Can object of given input type description be casted to this type.

common(*typ:* Data) → *Data*

Merge two types by finding their common type. By default this just looks if one type is castable into the other.

pixelpipes.types.**cast_element**(*source: str*, *destination: str*)

pixelpipes.types.**convert_element**(*element*)

1.5.2 C++ API

1.6 Extending

PixelPipes contains a lot of operations used in data loading and augmentation in computer vision. Still, sometimes additional functionality is needed. Simple cases can be easily implemented by Writing new macros, more complex cases require writing new C++ operations wrapped in a new custom module.

1.6.1 Writing macros

A macro is a combination of operations that are frequently used together. It is written in Python and combined normal Python language together with DAG generation. Macros can change generated subgraph based on input type inference. Macros can also use other macros within them. During compilation all macros are reduced down to primitive operations. For this example lets write a macro that

1.6.2 Creating custom operations

Frequently used or complex operations can be included into the pipeline by crating and building a PixelPipes module. A module is a dynamic library written in C++ that contains operations. Operations are functions that are exposed in a special manner and can be integrated in operation pipeline.

1.7 Compiling and development

PixelPipes is a hybrid source-code project, it contains C++ and Python code. Its main build framework is CMake which is wrapped in distutils.

The C++ library does not require any external dependencies during runtime, internally dependencies (like OpenCV) are pinned to a fixed version, compiled as static libraries and linked into the binary library. The C++ code requires a fairly recent compiler, supporting C++17. Compilation processed was tested on GCC 10, Clang ?? and MSVC ??.

The Python C++ wrapper requires Pybind11 and Numpy. It also uses some other Python packages that are installed via Pip. A PyBind11 header library is used to generate Python bindings for the C++ core, it is installed as a Pip dependency.

For development and testing purposes, the libraries can be compiled inplace using the following commands:

```
pip install cmake pybind11
pip install -r requirements.txt
python setup.py build_lib --inplace
python setup.py build_ext --inplace
```

1.7.1 Submitting issues and patches

Note: At the moment there are no specific rules on submitting issues and patches, just use Github issue tracker.

1.8 Credits

PixelPipes is an open-source project, but the

1.8.1 Funding

The development of this package was supported by Sloveninan research agency (ARRS) projects Z2-1866, J2-316 and J7-2596.

PYTHON MODULE INDEX

p

NODE INDEX

p

pixelpipes.resource.list.RepeatResource (*node*), **??**
pixelpipes.resource.list.ResourceSegment (*node*), **??**
pixelpipes.resource.list.SegmentCount (*node*), **??**
pixelpipes.resource.MakeResource (*node*), **??**

INDEX

L

M

N

O

P

Q

R

S

T

U

V

W