AutoPypline

This is the documentation for the AutoPypline library. It explains the functionality of the module, how and where the module can be used. It also contains examples to demonstrate the use of each of the features.

Author	**Sumukha Manjunath**
Maintainer	**Sumukha Manjunath**

Contents

1. Introduction

1.1 What is AutoPypline?

It is a module which can be used for automating python experiment/project workflow. The high level functions of AutoPypline can be divided into three parts:

- Design of an acyclic graph data structure using the experiment/project flow defined by the user
- Generation of parallel flows (if any) in the graph □ Execution of the flows generated

1.2 How does it work?

The modules (functions or classes) of the project/experiment have to be defined in a configuration file (yaml file or others) in a particular format. Each function or class which is part of the project workflow is defined as a node of a graph (Node here refers to a single unit of a graph). Once the graph is constructed, parallel flows (a string of functions/classes part of the graph which can be executed independently from remaining parts of the graph) are identified and executed.

Note: For simplicity and uniformity, yaml documents will be used to design the project workflow throughout the documentation.

1.2.1 Defining a node in the configuration file

Consider a simple function which takes two integer values as input and returns the sum of the two integers. The python code for this function is as follows:

```
def adder(a, b):
    """
    Returns the sum of two integers
    """
    return a + b
```

Figure 1.1

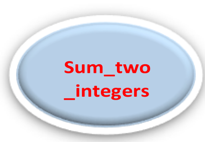
This function should be defined in the configuration file as follows:

```yaml
control_flow:
  sum_two_integers:
    function: example_functions.general.adder
    params:
      a: 10
      b: 4
  outputs:
    result: sum_two_integers
```

Figure 1.2

- The node is defined as a dictionary. An identifier “sum_two_integers” is used as the key for the function. The identifier can be any string. The attributes of the function are defined under this identifier, namely, the relative path to the function and the parameters of the function.
- The attributes are again defined as dictionaries. The relative path is provided as value against the key “function”. The function “adder” is defined in the path “example_functions/general.py” with respect to the project directory. So the relative path is provided as “example_functions.general.adder”.
- If the module is a python class, the key to the relative path should be “factory”. As shown in this example, if it’s a python function, the key “function” should be used.
- The parameters of the module are defined under the key “params”. In the current example, the parameters “a” and “b” are provided as 10 and 4 in the configuration file (Notice that this is defined as a dictionary as well).
- Apart from the nodes defined for the python classes/functions, a node “outputs” should be defined if the output of any node is required. The value under the key “outputs” can be defined as a string, list or dictionary. The format used under “outputs” indicates the format in which the AutoPypline returns the required output. In the above example, the value returned by the AutoPypline will be a dictionary with key “result” and value equal to the output of the node “sum_two_integers” All the modules (even if one) should be defined under the key “control_flow”.

Since it contains only a single module, the graph would be a single node:



Once the configuration file is designed, the AutoPypline class is used as follows:

```python
if __name__ == "__main__":
    from orchestrator_v3 import Orchestrator
    from utils import yaml_reader
    config_path = "Path to the configuration file"
    config = yaml_reader(config_path)
    orchestrator = Orchestrator(config=config.get("control_flow"),
                                sample_generator_def=config.get("sample_generator", None),
                                multiprocessing=config.get("multiprocessing", False))
    outputs = orchestrator()
```

Figure 1.3

This is common for any type of configuration file. The other parameters in the instantiation of AutoPypline will be discussed in the later sections of the documentation.

1.2.2 Defining a simple project workflow in the configuration file

Consider a solution containing two modules, one module to compute the sum of two integers (Figure1.1) and the other to compute the product of two integers (Figure1.4).

```python
def multiplier(a, b):
    """
    Returns the product of two integers
    """
    return a * b
```

Figure 1.4

Let the operation to be performed be defined as:

Result = (a + b) * c

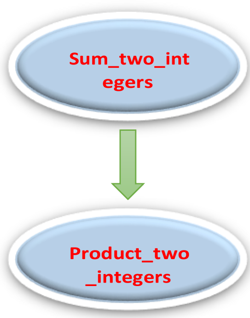
This operation can be performed using the two modules defined above. We first compute the sum of two integers and then multiply the result with another integer. The design of the configuration for this solution would be as follows:

```
control_flow:
  sum_two_integers:
    function: example_functions.general.adder
    params:
      a: 10
      b: 4
  product_two_integers:
    function: example_functions.general.multiplier
    params:
      a: 2
    inputs:
      b: sum_two_integers
```

Figure 1.4

The node “sum_two_integers” is defined the same as in Figure1.1. For the node “product_two_integers”, one of the parameters “a” is defined under params with a value 2. The other parameter is equal to the output of the node “sum_two_integers”. The parameters of a module which are dependent on the output values of other nodes are defined under the key “inputs”. The parameter “b” is defined under “inputs” and is assigned the value “sum_two_integers”.

The graph constructed for this configuration file is as follows:



There are no parallel flows in the constructed graph.

1.2.3 A slightly complex project workflow in the configuration file

Consider a solution containing three modules, one module to compute the sum, difference and product of two integers (Figure1.7), the next to compute the square of a given integer (Figure1.6) and the last module to compute the sum of three integers (Figure1.5).

```python
def adder_3(a, b, c):
    """
    Returns the sum of three integers
    """
    return a + b + c
```

Figure 1.5

```python
def square(a):
    """returns the square of a given integer"""
    return a * a
```

Figure 1.6

```python
def ams(a, b):
    """
    returns the sum, difference and product of two integers
    """
    return {"sum": a + b, "diff": a - b, "prod": a * b}
```

Figure 1.7

Let the operation to be performed be defined as:

Result = (a + b) **2 + (a * b) + (a – b)

This operation can be performed using the three modules defined above. We first compute the sum, difference and product of two integers using the function “ams”. The sum of the two integers which is part of the output returned by “ams” is passed as input to the function “square”. The output from the function “square” is passed as input to the function “adder_3” along with the product and difference values from “ams”. The design of the configuration for this solution would be as follows:

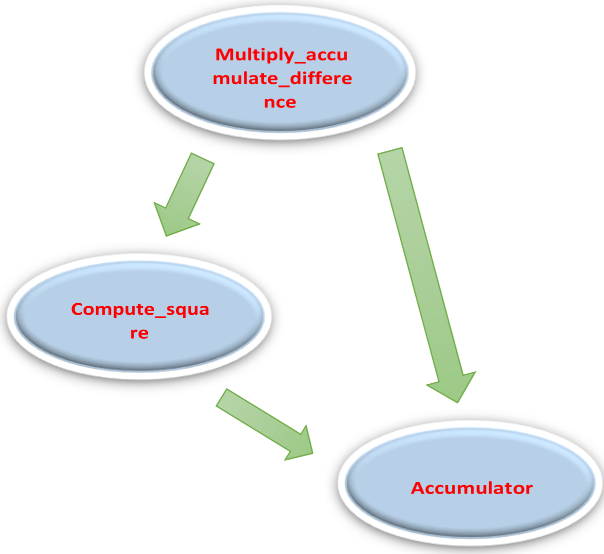
```yaml
control_flow:
  multiply_accumulate_difference:
    function: example_functions.general.ams
    params:
      a: 2
      b: 4
  compute_square:
    function: example_functions.general.square
    inputs:
      multiply_accumulate_difference.outputs.sum
  accumulator:
    function: example_functions.general.adder_3
    inputs:
      - compute_square
      - multiply_accumulate_difference.outputs.prod
      - multiply_accumulate_difference.outputs.diff
  outputs:
    accumulator
```

Figure 1.8

- As discussed in previous example, we define all the nodes under the key "control_flow".
- Each node has an identifier and the path and params of the nodes are defined as explained in the previous examples.
- One constraint in the design of code is that if the python class/function returns more than one value, the output should be returned in the form of a dictionary. This is demonstrated with the function "ams" in the above example. This constraint ensures that a part of the output can be used by the other nodes.
- As explained above, "compute_square" node computes the square of a given integer. The input integer is equal to the sum of two integers computed by "multiply_accumulate_difference" node. The format to access only part of a node's output is as follows:
    - ❖ node_name.outputs.required_output_key
- The sum of the two integers from "multiply_accumulate_difference" node is accessed as "multiply_accumulate_difference.outputs.sum". Observe that the "required_output_key" should be equal to one of the keys in the output dictionary of the node.
- Similarly, the difference and product of the two integers are accessed as "multiply_accumulate_difference.outputs.diff" and "multiply_accumulate_difference.outputs.prod". These two values are provided as input to the node "accumulator" along with the output from "compute_square".

- The output of “accumulator” node is the required solution.

The graph constructed for this configuration file is as follows:



There are no parallel flows in the constructed graph. The nodes are executed one after another in the correct order.

Note:

- If the function only contains a single parameter, the value under “inputs” can be a string apart from a dictionary. For example, in Figure1.8, inputs to “compute_square” is provided as a string.
- If all the parameters are dependent on a single node (other node in the configuration), then the values under “inputs” can be defined as a string aside from being defined as a dictionary.
- If all the parameters are dependent on the other nodes, then the values under “inputs” can be defined as a list aside from being defined as a dictionary. In such case, care should be taken by the user to ensure that the values are defined in the same order as the parameters in the code (If the order matters). For example, in Figure1.8, inputs to “accumulator” is provided as a list.

1.2.4 A complex project workflow in the configuration file

In this section, we provide an example for the image processing domain. The objectives of this example are as follows:

- Read an image and its corresponding label from the given location (Both in rgb color space).
- Extract a random crop from both image and label.
- Convert the cropped label from rgb space to gray
- Save the cropped rgb image and cropped gray scale label into the given location.

```python
def img_reader(img_name, img_folder):
    """Reads an image given the image folder and name"""
    return cv2.imread(os.path.join(img_folder, img_name))
```

Figure 1.9

```python
def img2gray(img):
    """
    Converts rgb image to gray scale
    """
    return cv2.cvtColor(np.array(img, dtype=np.uint8), cv2.COLOR_BGR2GRAY)
```

Figure 1.10

```python
def img_writer(img, save_folder, save_name):
    """
    Writes the given img to the given save_folder with save_name.
    """
    cv2.imwrite(os.path.join(save_folder, save_name), img)
```

Figure 1.11

```python
class RandomCrop:
    """
    Returns a randomly cropped patch of img and lbl of size "crop_size"
    """
    def __init__(self, crop_size):
        self.crop_size = crop_size

    def __call__(self, img, lbl):
        initial_width = random.randint(0, img.shape[0] - self.crop_size[0])
        initial_height = random.randint(0, img.shape[1] - self.crop_size[1])
        img = img[initial_width: initial_width + self.crop_size[0],
                  initial_height: initial_height + self.crop_size[1], :]
        lbl = lbl[initial_width: initial_width + self.crop_size[0],
                  initial_height: initial_height + self.crop_size[1], :]
        return {'img': img, 'lbl': lbl}
```

Figure 1.12

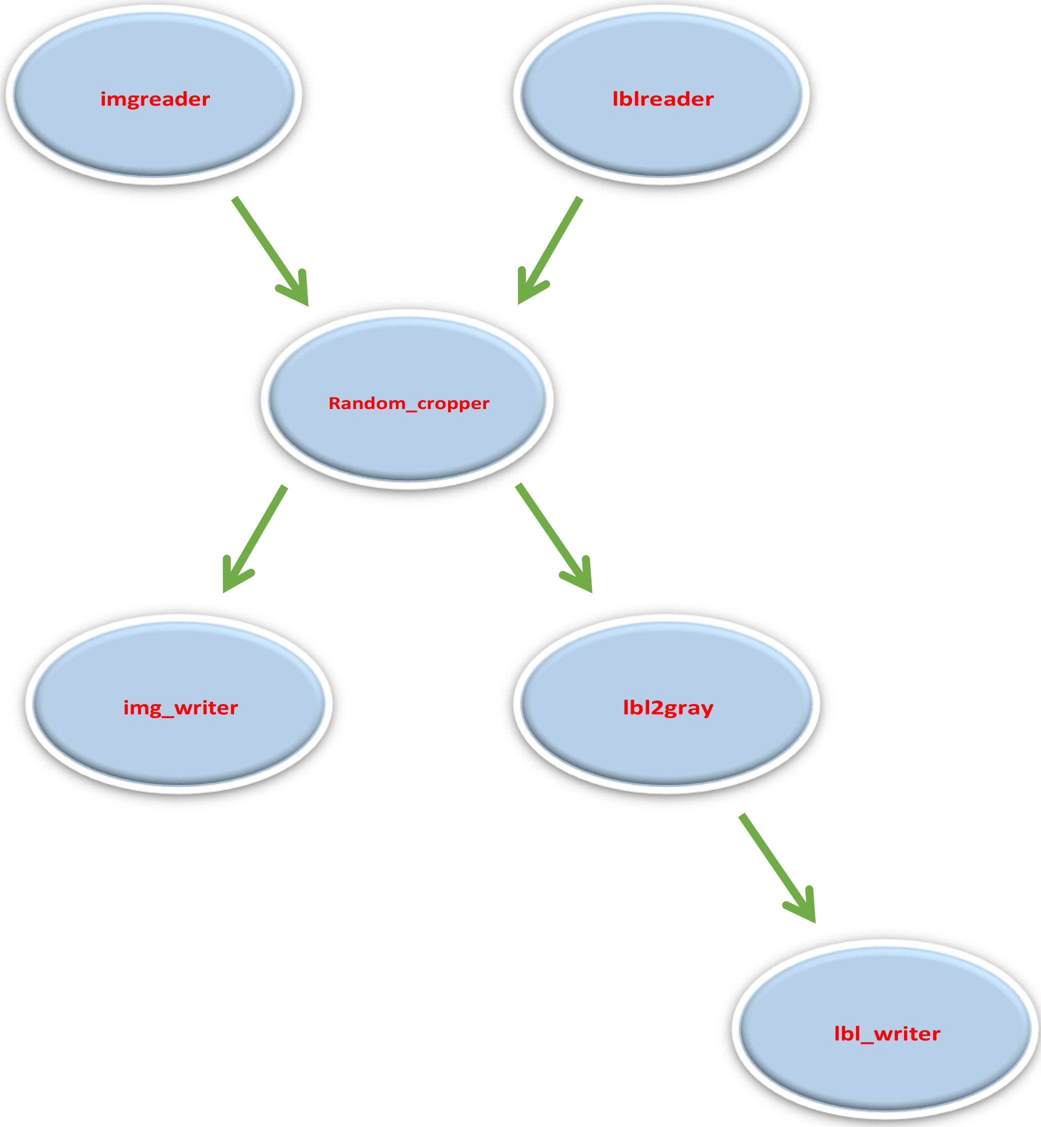
The objectives listed above can be achieved using the four modules defined above. The functionality of each of these modules is specified along with the module definition.

The design of the configuration for this solution would be as follows:

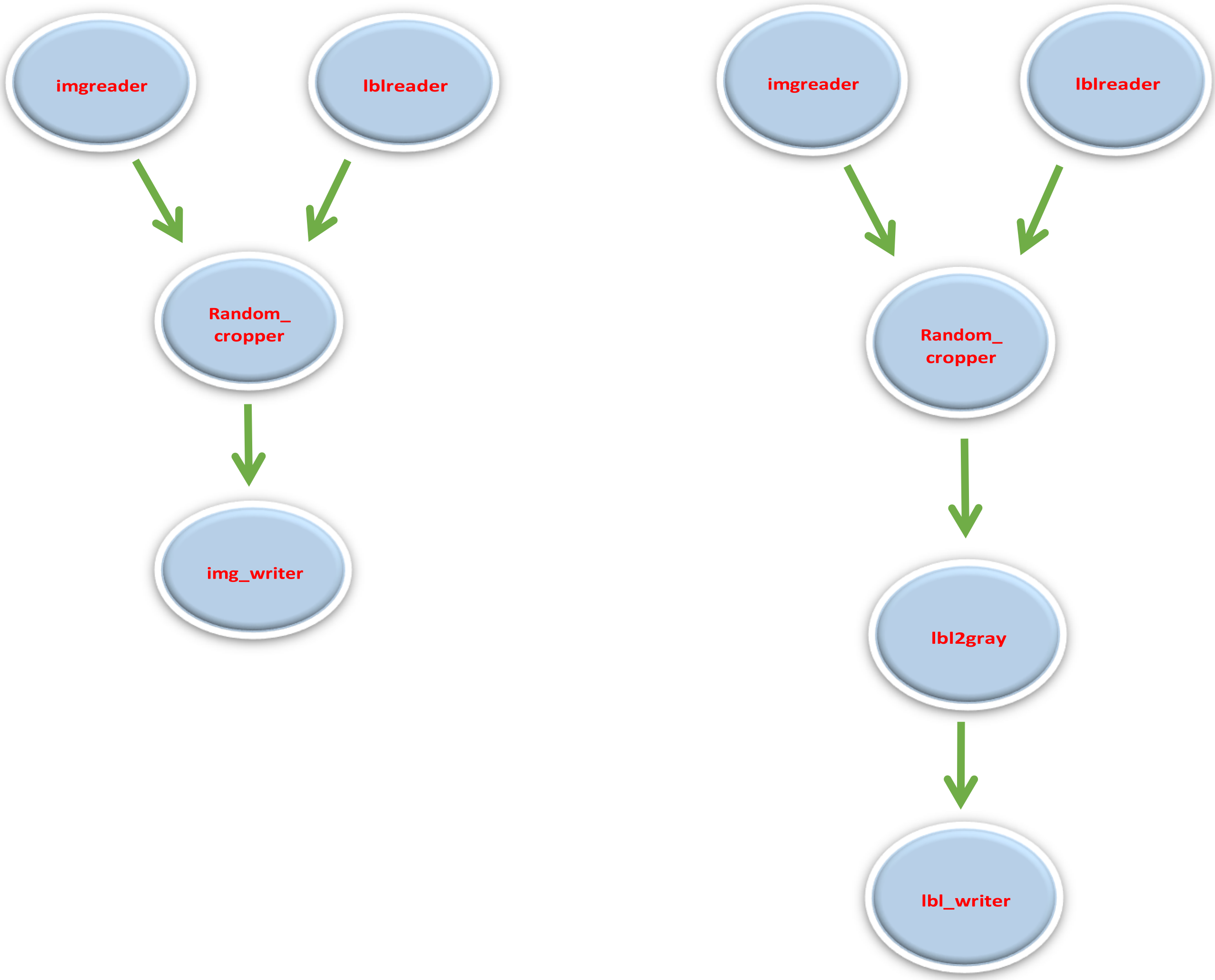
```yaml
control_flow:
  imgreader:
    function: utils.img_reader
    params:
      img_name: "test_img.png"
      img_folder: sample_data/set1/images

  lblreader:
    function: utils.img_reader
    params:
      img_name: "test_lbl.png"
      img_folder: sample_data/set1/labels

  random_cropper:
    factory: utils.RandomCrop
    params:
      crop_size: [150, 150]
    inputs:
      img: imgreader
      lbl: lblreader

  img_writer:
    function: utils.img_writer
    params:
      save_folder: Project_Orchestrator/outputs_folder/images
      save_name: "test_img.png"
    inputs:
      img: random_cropper.outputs.img

  lbl2gray:
    function: utils.img2gray
    inputs: random_cropper.outputs.lbl

  lbl_writer:
    function: utils.img_writer
    params:
      save_folder: Project_Orchestrator/outputs_folder/labels/
      save_name: "test_label.png"
    inputs:
      lbl: lbl2gray
```

Figure 1.13

The graph constructed for this configuration file is as follows:



From the constructed graph, it can be seen that there are two parallel and independent paths after the random_cropper node. The parallel flows generated from this graph are as follows:



Execution of the flows generated:

Two flows were generated as shown above. The nodes which are common along the multiple flows are first executed. So here the nodes “imgreader”, “lblreader” and “random_cropper” are executed first so as to ensure that they are not repeated. The remainder of the two flows are executed in parallel as they are independent of one another.

There’s no key “outputs” defined here since output from the nodes are not required as the modified image and label are saved directly.

2. Features of AutoPypline

The AutoPypline supports looping, multiprocessing and multi-level graph configurations. In this section, examples from image processing domain will be used to elucidate each of the features, but keep in mind that the features can be applied for any domain.

2.1 Looping

In the examples of section1, all the operations were only performed on a single set of inputs. It is common for the same set of operations to be performed on multiple sets of inputs.

For example, in section 1.2.4, a single image and its corresponding label were passed through a string of functions/classes. When preparing datasets, the same string of operations are usually performed on a set of images and labels. The AutoPypline module can be used for looping over batches of inputs over the same set of operations.

Facilitation of looping with the AutoPypline is demonstrated with an example below.

2.1.1 Example

The same modules defined in section 1.2.4 will be used in this example. The design of the configuration file is as shown in Figure2.1. In comparison to the configuration file in Figure1.13, it can be seen that the "sample_generator" is an addition.

The sample_generator is a python class, which generates a batch of samples in each step. Some design requirements of the sample_generator are fixed. The sample_generator is defined as a node with the identifier "sample_generator" outside the control_flow. It is also defined with "factory" and "params".

Design of sample_generator class:

- In the sample_generator definition, the parameter "batch_size" refers to the number of samples to consider in each step of processing.
- The parameter "sample_name" is the identifier for each sample. The output of the control_flow for a batch sample is stored against its corresponding identifier (It stores a value None if no output is required). If "sample_name" is not provided, the results are accumulated in the form of a list.
- The call method should return the batch samples in the form of a list of dictionaries (each batch sample is a dictionary). This is the only fixed design requirement.
- If the parameter "sample_name" is used, it needs to be one of the keys in the dictionary the sample_generator returns in the call method. The parameter name "sample_name" is fixed.

```yaml
control_flow:
  imgreader:
    function: utils.img_reader
    params:
      img_name: external.imgname
      img_folder: sample_data/set1/images

  lblreader:
    function: utils.img_reader
    params:
      img_name: external.lblname
      img_folder: sample_data/set1/labels

  random_cropper:
    factory: utils.RandomCrop
    params:
      crop_size: [150, 150]
    inputs:
      img: imgreader
      lbl: lblreader

  img_writer:
    function: utils.img_writer
    params:
      save_folder: Project_Orchestrator/outputs_folder/images
      save_name: external.imgname
    inputs:
      img: random_cropper.outputs.img

  lbl2gray:
    function: utils.img2gray
    inputs: random_cropper.outputs.lbl

  lbl_writer:
    function: utils.img_writer
    params:
      save_folder: Project_Orchestrator/outputs_folder/labels/
      save_name: external.lblname
    inputs: lbl2gray

sample_generator:
  factory: utils.SampleGeneratorListFromTxtFile
  params:
    img_list_path: "sample_data/set1/set_1_image_list.txt"
    lbl_list_path: "sample_data/set1/set_1_lbl_list.txt"
    batch_size: 1
```

Figure 2.1

The sample_generator used in this example is defined as shown in Figure2.2 and this can be used as the base class for all sample generators.

```python
class SampleGeneratorListFromTxtFile:
    def __init__(self, img_list_path, lbl_list_path, batch_size=1, sample_name="imgname"):
        """
        :param img_list_path: Path to a text file containing list of image paths
        :param lbl_list_path: Path to a text file containing list of label paths
        :param batch_size: Number of image and label samples to generate for each batch
        :param sample_name: The key (with respect to the returned outputs) to be used identify the sample
        """
        self.img_list = file_reader(img_list_path)
        self.lbl_list = file_reader(lbl_list_path)
        self.batch_size = batch_size
        self.sample_name = sample_name
        if len(self.img_list) % self.batch_size == 0:
            self.input_length = int(len(self.img_list) / batch_size)
            self.remainder = 0
        else:
            self.input_length = int(len(self.img_list) / batch_size) + 1
            self.remainder = len(self.img_list) % self.batch_size

    def __call__(self, batch_id):
        """
        Returns a batch of samples
        :param batch_id: The id of the batch to be returned
        :return:
        """
        batch_values = []
        batch_items = self.get_batch_items(batch_id)
        for batch_img, batch_lbl in zip(batch_items[0], batch_items[1]):
            batch_values.append({'imgname': batch_img, 'lblname': batch_lbl})
        return batch_values

    def get_batch_items(self, id_val):
        if id_val == self.input_length and self.remainder > 0:
            return [self.img_list[id_val * self.batch_size:],
                    self.lbl_list[id_val * self.batch_size:]]
        else:
            return [self.img_list[id_val * self.batch_size: (id_val + 1) * self.batch_size],
                    self.lbl_list[id_val * self.batch_size: (id_val + 1) * self.batch_size]]
```

Figure 2.2

The sample_generator in Figure2.2 takes in a path to a file containing a list of image names "img_list_path", path to a file containing a list of label names "lbl_list_path" as inputs along with batch_size and sample_name. Here the sample_name is provided as "imgname", so the name of the image is used as the identifier while storing the result of the control_flow for the corresponding batch sample.

The list of image and label paths are obtained by reading the files from their paths. The number of iterations of batch samples "self.input_length" is computed with respect to the batch size. If the length of the list of inputs is not divisible by batch size, then the last batch of samples will only contain number of samples equal to remainder.

The __call__ method generates the batch sample given the batch_id (step number) and returns a list of dictionaries with each dictionary containing two keys “imgname” and “lblname” and their corresponding values. The number of elements in the list is equal to the batch size.

Since the process involves looping over batches of input, the parameters of some of the nodes change dynamically. For example, the “img_name” parameter of the nodes “imgreader” and “lblreader” change dynamically for each step.

The dynamically changing values should correspond to the keys of the values returned by the sample_generator.

To access the values generated by sample_generator, a particular format is used: “external.key_name”. For example, the img_name parameter of imgreader and lblreader are provided as “external.imgname” and “external.lblname” respectively.

2.2 Multi-processing

In configurations where looping is involved, the processing of a single batch sample at a time might not be utilizing the maximum capacity of the available cpu cores. So data parallelization feature has been implemented.

To use multi-processing, a flag “multiprocessing” has to be set to True in the configuration file. It is provided as a key in the same level as the control_flow and sample_generator. If multiprocessing is used, the batch_size parameter in the sample_generator can be increased beyond 1.

2.2.1 Example

```
control_flow:
  imgreader:
    function: utils.img_reader
    params:
      img_name: external.imgname
      img_folder: sample_data/set1/images

  lblreader:
    function: utils.img_reader
    params:
      img_name: external.lblname
      img_folder: sample_data/set1/labels

  random_cropper:
    factory: utils.RandomCrop
    params:
      crop_size: [150, 150]
    inputs:
      img: imgreader
      lbl: lblreader

  img_writer:
    function: utils.img_writer
    params:
      save_folder: Project_Orchestrator/outputs_folder/images
      save_name: external.imgname
    inputs:
      img: random_cropper.outputs.img

  lbl2gray:
    function: utils.img2gray
    inputs: random_cropper.outputs.lbl

  lbl_writer:
    function: utils.img_writer
    params:
      save_folder: Project_Orchestrator/outputs_folder/labels/
      save_name: external.lblname
    inputs: lbl2gray

sample_generator:
  factory: utils.SampleGeneratorListFromTxtFile
  params:
    img_list_path: "sample_data/set1/set_1_image_list.txt"
    lbl_list_path: "sample_data/set1/set_1_lbl_list.txt"
    batch_size: 8

multiprocessing: True
```

Figure 2.3

In Figure2.3, “multiprocessing” flag is set to True and batch_size is set to 8. So 8 pairs of image and label are processed in parallel in this example.

2.3 Multi-level graphs

In some cases, a node/nodes in the control flow might contain a string of operations needed to be performed on a list of inputs. So we need to define a control flow under the node identifier. This is termed as an "internal_graph".

2.3.1 Simple Example

Suppose we want to compute the sum of the squares of a list of integers. We can reuse some of the already defined modules to achieve the same. A list of integers are generated using the module "random_number_generator". The square of each element in the list can be computed using the module "square" defined in Figure1.6. The sum of the square of all elements is computed using the module "list_adder".

```
def random_number_generator(start_value, end_value, number_of_elements):
    """
    Generates a list of random elements in the range [start_value, end_value]
    of length equal to number_of_elements.
    """
    return [random.randint(start_value, end_value) for i in range(number_of_elements)]
```

Figure 2.4

```
def list_adder(input_list):
    """Computes sum of all values in a list"""
    return sum(input_list)
```

Figure 2.5

The design of the configuration file is as shown in Figure2.6.

- At the first level there are three nodes: random_values_generator, compute_square_loop and adder.
- The list of integers is generated by random_values_generator, next the square of each element needs to be computed. This involves looping, so an "internal_graph" is defined under the identifier "compute_square_loop".
- The "internal_graph" consists of a "control_flow" and a "sample_generator" and flag indicating the use of multiprocessing. Notice that these are defined under the key "internal_graph" under the node identifier. Either a factory/function and params should be defined under a node identifier or an internal_graph should be defined. The inputs from other nodes are provided as usual.

- In this example, the control_flow only includes a single node “compute_square” which computes the square of a given integer.
- As mentioned previously, the result for each sample is accumulated. The sample_generator used does not have a “sample_key” parameter, so the results are accumulated in the form of a list.
- This list of squares of all elements are provided as input to “adder” which returns the sum of all the elements provided as input.

```yaml
control_flow:
  random_values_generator:
    function: example_functions.general.random_number_generator
    params:
      start_value: 1
      end_value: 1000
      number_of_elements: 20
  compute_square_loop:
    internal_graph:
      control_flow:
        compute_square:
          function: example_functions.general.square
          params:
            a: external.sample_value
        outputs: compute_square
      sample_generator:
        factory: sample_generators.SampleGeneratorFromList
        params:
          input_list: sample_generator_inputs.input_values
          batch_size: 12
      multiprocessing: True
    inputs:
      input_values: random_values_generator
  adder:
    function: example_functions.general.list_adder
    inputs:
      input_list: compute_square_loop
  outputs:
    adder
```

Figure 2.6

The sample_generator under the node “compute_square_loop” gets the input list as input from the first level. This input is defined under the key “inputs” of “compute_square_loop”.

It can be accessed by the sample_generator as “sample_generator_inputs.input_key_name”. The value “input_key_name” should be one of the keys provided under “inputs” of the node. Here the list of elements from “random_values_generator” are accessed as “sample_generator_inputs.input_values”.
The sample_generator is defined as follows:

```python
class SampleGeneratorFromList:
    """
    Generates samples from a list of values.
    """
    def __init__(self, input_list, batch_size=1, sample_name=None):
        self.input_list = input_list
        self.batch_size = batch_size
        self.sample_name = sample_name
        if len(self.input_list) % self.batch_size == 0:
            self.input_length = int(len(self.input_list) / batch_size)
            self.remainder = 0
        else:
            self.input_length = int(len(self.input_list) / batch_size) + 1
            self.remainder = len(self.input_list) % self.batch_size

    def __call__(self, batch_id):
        batch_values = []
        batch_items = self.get_batch_items(batch_id)
        for sample_id, sample_value in enumerate(batch_items):
            batch_values.append({'sample_value': sample_value})
        return batch_values

    def get_batch_items(self, id_val):
        if id_val == self.input_length and self.remainder > 0:
            return self.input_list[self.batch_size * id_val:]
        else:
            return self.input_list[self.batch_size * id_val: self.batch_size * (id_val + 1)]
```

Figure 2.7

2.3.2 Complex example

Consider an image processing pipeline which taken in high resolution image as input, generates lower resolution patches from the input image. It passes each patch through a neural network designed for semantic segmentation after pre-processing the patch. Once this is repeated for all the patches, the inference results are reconstructed back to the input image resolution.

The design of the configuration file is as follows:

```yaml
control_flow:
  model_generator:
    function: segmentation_models.unet
    params:
      input_shape: [320, 320]
      no_classes: 8

  patch_generator:
    function: patch_generator.patch_generator2
    params:
      img_path: Project_Orchestrator/Test_images/01_10_2018_Row_1_Top.png
      patch_size: [320, 320]

  patch_processor:
    internal_graph:
      control_flow:
        patch_rgb_to_gray:
          function: utils.img2gray
          params:
            img: external.img
        semseg_infer:
          function: utils.semseg_infer
          inputs:
            img: patch_rgb_to_gray
            model: external.model
        outputs:
           semseg_infer
      sample_generator:
        factory: utils.SampleGeneratorFromDictModel
        params:
          batch_size: 8
          img_dict: sample_generator_inputs.img_dict
          model: sample_generator_inputs.inference_model
      multiprocessing: True

    inputs:
      img_dict: patch_generator.outputs.patches_dict
      inference_model: model_generator

  patch_stitcher_and_saver:
    function: patch_generator.patch_stitcher_and_writer
    params:
      save_path: Project_Orchestrator/outputs_folder/raw_image/01_10_2018_Row_1_Top.png
    inputs:
      patches_dict: patch_processor
      target_size: patch_generator.outputs.padded_size
```

Figure 2.8

- The node “model_generator” returns a semantic segmentation model. The model is designed to classify each pixel to one of eight possible classes in this example. The input size for the model is fixed at 320x320.
- The “patch_generator” node takes the path of a high resolution image as input and returns a dictionary where the each key and value correspond to the coordinates of the patch with respect to the high resolution image and the patch respectively. Along with patches dictionary, it returns the size of the high resolution image.
- The patch_processor takes the model from model_generator and patches dictionary from patch_generator as input. The internal graph processes each patch individually. Each batch is generated by the sample_generator and each batch is passed through the modules defined under control_flow.

- Finally, “patch_stitcher_and_saver” node reconstructs the inference results from patch_processor back to high resolution.

The sample_generator used is defined as follows:

```python
class SampleGeneratorFromDictModel:
    def __init__(self, img_dict, model, batch_size=1, sample_name="img_name"):
        self.img_dict = img_dict
        self.batch_size = batch_size
        self.model = model
        self.sample_name = sample_name
        if len(self.img_dict) % self.batch_size == 0:
            self.input_length = int(len(self.img_dict) / batch_size)
            self.remainder = 0
        else:
            self.input_length = int(len(self.img_dict) / batch_size) + 1
            self.remainder = len(self.img_dict) % self.batch_size

    def __call__(self, batch_id):
        batch_values = []
        batch_items = self.get_batch_items(batch_id)
        for batch_img_name, batch_img in zip(batch_items[0], batch_items[1]):
            batch_values.append({'img_name': batch_img_name, 'img': batch_img, "model": self.model})
        return batch_values

    def get_batch_items(self, id_val):
        if id_val == self.input_length and self.remainder > 0:
            img_names = [list(self.img_dict.keys())[id_val * self.batch_size:]]
            img_values = [list(self.img_dict.values())[id_val * self.batch_size:]]
            return [*img_names, *img_values]
        else:
            img_names = [list(self.img_dict.keys())[id_val * self.batch_size: (id_val + 1) * self.batch_size]]
            img_values = [list(self.img_dict.values())[id_val * self.batch_size: (id_val + 1) * self.batch_size]]
            return [*img_names, *img_values]
```

Figure 2.9

It takes a dictionary as input (key: image name and value: img) along with a model object. Each batch sample is a dictionary with the values of image name, image and model.

Currently all the values from higher level to lower level should be passed through the sample_generator defined in the lower level. This will be modified in the next version of AutoPypline. So the model object need not be returned from the sample_generator, instead it can be directly accessed from the node “model_generator”.

3. Applications

The AutoPypline module can be used for any python project. This section gives a few examples as to where the AutoPypline class can be used.

- In research and development projects which involves a lot of trial and error. Using the AutoPypline will allow easy configuration of new experiments and also helps to keep track of all the previous experiments through the configuration file. For example, if we want to train an object detection model such as Mask R-CNN on a new dataset, a lot of parameters with respect to the model and data have to be tried out to train the model in the right direction.
- In projects which involve fine-tuning or slightly modifying an existing solution. In such projects, the modifications have to be made only to the configuration file. For example, if you want to replace few of the modules with different modules.
- In solutions which involve design of a pipeline using existing python functions or classes. The communication between the individual modules are handled automatically.
  For example: Automation of Data Analysis by designing a pipeline using a combination of multiple python functions or classes with each module responsible for a particular data analysis objective.

For all the applications, there is no need to change the existing code and there is no fixed structure for the functions or classes. Design and modification needs to be carried out only in the configuration file.

4. Future Work

- Easy Communication between different levels of graphs
- Automation of sample_generator code based on user inputs
- Logging of starting and ending time of execution for each node in the graph. Visualization of the constructed graphs