The ultimate Guide to ARCathon

Lab42*

February 13, 2023

Contents

*https://lab42.global/arcathon/

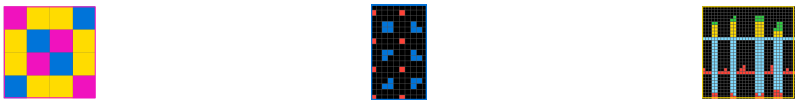
1 Introduction

This is the official manual for the ARCathon competition hosted by Lab42. It contains all relevant information you need to participate in ARCathon. It is not meant to be read from beginning to start but should rather be used as an encyclopedia and things can be looked up when needed. You can find most of the information contained in this manual also on our Website in the active challenges section [1].

2 About ARC

ARC stands for "abstraction and reasoning corpus" and has been introduced by François Chollet in [1]. The dataset has been designed such that only algorithms that show a more "human-like" intelligence can reach high scores. It consists of 800 public tasks for training and evaluation together with a set of 200 private tasks. On 100 of these private tasks we evaluate all solutions that are handed in to ARCathon. You can download the 800 public tasks from our Website [2].

Each task consists of grids with a minimum size of 1x1 and a maximum size of 30x30. The cells of the grid are filled with a number between 0 and 9, represented by ten different colors, cf. Fig. 1.



(a) 4x4 (b) 17x10 (c) 30x30

Figure 1: Examples for ARC Grids.

A task consists of several example input-output pairs of grids, typically three, and test input grids, typically one. From the example pairs a test-taker has to infer a rule that he has to apply to the test input to find the correct test output. A rule can both describe changes of the colors of cells as well as a change in the grid size. An example is given in Fig. 2. A task is only successfully solved if for all of the test inputs an output grid is constructed for which the color of each cell as well as the grid size matches exactly the solution grid. The difficulty of ARC lies in the fact that each of the 1000 tasks is based on a different rule. This makes hard-coding solutions near impossible. We have an ARC playground on our website where you can test your own intelligence on the 800 public tasks and also create your own riddles [3]. Spending some time on the playground is a good starting point for everyone that is not familiar with ARC.

2.1 Data format

As mentioned before you can find the public data set on our Website [4]. In there you will find two folders each containing 400 tasks:

- **Training set**: contains the 400 task files for training. You can use these tasks to prototype your algorithm or train it to learn cognitive priors that are critical to solving ARC. On average the tasks contained in the public training set are considered to be easiest to solve.
- **Evaluation set**: contains the 400 task files for your own evaluation. These tasks are considered to be slightly more difficult than the ones found in the public Training set.

The tasks are stored in **JSON** format. Each JSON file consists of two key-value properties:

- "train": list of example input/output pairs, typically contains three pairs which your algorithm should use to infer a rule.
- "test": list of test input/output pairs, typically contains one pair. Your algorithm should apply the constructed solution from the train pairs to the test input. In case of the public set you have access to the test output as well and can directly check if your algorithm's solution is valid.

[1] https://lab42.global
[2] https://lab42.global/wp-content/uploads/2022/08/ARC-800-tasks.zip
[3] https://arc-editor.lab42.global
[4] https://lab42.global/wp-content/uploads/2022/08/ARC-800-tasks.zip

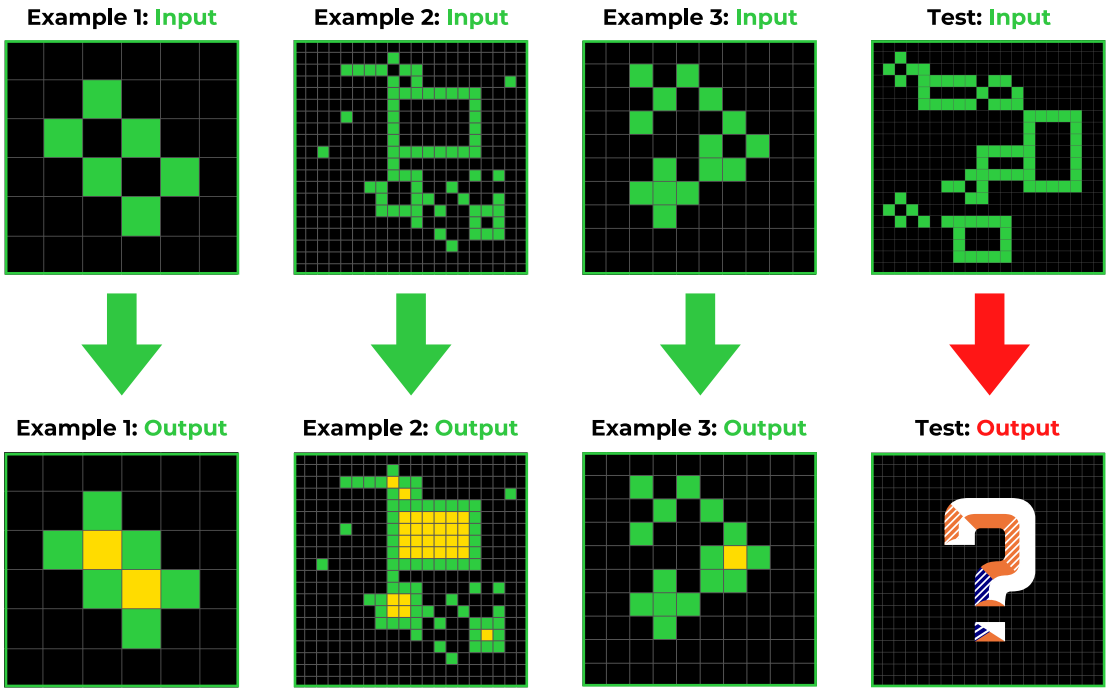


Figure 2: Example of an ARC task. From the 3 example grids a test-taker can infer the rule that all black cells that are surrounded by green cells have to be filled with yellow. This rule has then to be applied to the test grid to construct the correct solution to this task.

Each pair in the lists is itself a key-value property:

- "input": input grid in form of a list for the corresponding pair.
- "output": output grid in form of a list for the corresponding pair.

A very simple of an ARC task JSON file with three training pairs and one test pair looks as follows:

```
{
    "train":
        [
            {"input":[[1,0],[0,0]],"output":[[1,1],[1,1]]},
            {"input":[[0,0],[4,0]],"output":[[4,4],[4,4]]},
            {"input":[[0,0],[6,0]],"output":[[6,6],[6,6]]}
        ],
    "test":
        [
            {"input":[[0,0],[0,8]],"output":[[8,8],[8,8]]}
        ]
}
```

3 About ARCathon

ARCathon was launched by Lab42 in 2022 to build up a new community and renew interest in ARC as an unsolved benchmark on our path towards human level AI. ARC was first hosted as a competition on Kaggle [5] but the best solutions in the competition did not exceed the 20% mark. In agreement with François Chollet we relaunched ARC as a challenge with some changes in the way you hand in your solutions to allow more flexibility in the choice of programming languages and platforms. However, the secret test set on which your algorithms will be evaluated in ARCathon consists of the same tasks that have been used in the original Kaggle competitoin.

[5] https://www.kaggle.com/competitions/abstraction-and-reasoning-challenge

In the first iteration over 100 teams from more than 40 countries tried their best to find new solutions to ARC. While no significant leaps were achieved during ARCathon 2022, the most promising individuals and teams were identified globally, and the challenge has been institutionalized. We congratulate our 1st place finisher Michael Hodel from Zurich, Switzerland, he was also the one who managed to raise the ARC world record above 30% after ARCathon 2022 ended with a solution combining his and several others approaches. 2nd place took Jozef Kopanicak with his team from the Mirus Software AG based in Davos, Switzerland, and in Zilina, Slovakia. On 3rd place finished iOS developer Simon Strandgaard from Copenhagen, Denmark.

For ARCathon 2023 we upped the stakes with a running price money of up to 69'000 CHF. Anyone familiar with the rules of ARCathon 2022 will have no problems to continue to take part in this new iteration.

4 How to start

For anyone who likes scientific papers a good start is to read Chollet's paper "On the Measure of Intelligence" [1] - especially pp. 46-58 - and read through our ARC Page. There you find the motivation for ARC as well as the "core" knwoledge which is used in the rules to solve ARC tasks. Furthermore, Lab42 has created an ARC interface [6] where you can solve the original ARC tasks and even create new ones: ARC Playground.

From there, you can start brainstorming about your own approach. Some more inputs you can find in Mehran Kazeminia article [2] with some answers from François Chollet.

> *"If you don't know how to get started, I would suggest the following template:"* [7]

We recommend the first two steps of the template to everyone new to ARC:

1. *"Take a bunch of tasks from the training or evaluation set — around 10. For each task, write by hand a simple program that solves it. It doesn't matter what programming language you use — pick what you're comfortable with.*
2. *Think about what these programs have in common."*

From here, deviation from this template could lead to a breakthrough. We only know that Abstract Reasoning and the concept of Core Knowledge – described in Chollet's paper – play an essential role in solving ARC.

3. *"Now, look at your programs, and ponder the following:*
   (a) *Could they be expressed more naturally in a different medium (what we call a DSL, a domain-specific language)?*
   (b) *What would a search process that outputs such programs look like (regardless of conditioning the search on the task data)?*
   (c) How could you simplify this search by conditioning it on the task data?
   (d) *Once you have a set of generated candidates for a solution program, how do you pick the one most likely to generalize?*

You will not find tutorials online on how to do any of this. The best you can do is read past literature on program synthesis, which will help with step 3). But even that may not be that useful :)

This challenge is something new. You are expected to think on your own and come up with novel, creative ideas. It's what's fun about it!"

5 Submission rules

The general format for submissions is Docker. Docker images have the advantage that they are portable but still leave a lot of freedom to the developers. It uses virtualization at the operating system level to deploy software in packages called containers. All code submitted to ARCathon must conform to the format outlined in the Rules below and be submitted as a Docker image!

Starting February 13, you can submit three Docker images per calendar week until December 1, 2023, which will be evaluated by our team within a maximum of 72 hours. Without requesting

[6] https://arc-editor.lab42.global
[7] François Chollet

a GPU your maximal runtime will be 12 hours with GPU the maximal runtime is 5 hours. Before submitting please make sure that your Docker image runs on a Google Cloud virtual machine with Container-Optimized OS (details below), you can then submit your Docker image as a link to public repository such as Docker Hub [8] or Google Container registry [9]. If you are worried about making your Docker publicly available on a registry you can also send us a direct download link through OneDrive, Google Drive, Dropbox, or your preferred platform. Solutions must be submitted via the submission form[10].

5.1 Virtual machine

The solutions will be evaluated on a Google N1-standard-4 series server [11]:

- Container-Optimized OS 101-17162.40.34 LTS [12]
- 4 vCPU's, 15GB RAM, 300 GB storage
- Max. 1 Nvidia T4 GPU; 16 GB RAM [13]
- No internet access

5.2 Data structure

On the virtual machine you will have access to a folder called *secret_data* that itself contains two folders, *evaluation* and *solution*. In the *evaluation* folder you can access the files of the private test set, the *solution* folder is where you should write your solution file (see below for details). A diagram depicting the folder structure is shown in Fig. 3.

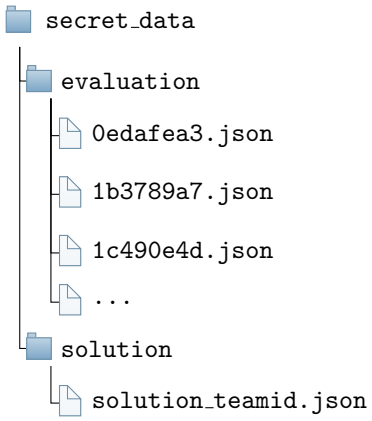


Figure 3: Folder structure on the virtual machine.

5.3 Private test set

We will evaluate your submissions on the private test set consisting of 100 Tasks in JSON form as described in 2.1. The only difference to the public set is that the output grids of the test pairs are replaced with [[0]], as your algorithm should not have access to the solutions it needs to construct.

A simple example JSON that could be in the private test set looks as follows:

```
{
    "train":
        [
            {"input":[[1,0],[0,0]],"output":[[1,1],[1,1]]},
            {"input":[[0,0],[4,0]],"output":[[4,4],[4,4]]},
```

[8] https://hub.docker.com
[9] https://cloud.google.com/container-registry
[10] https://lab42.global/arcathon/submission/
[11] https://cloud.google.com/compute/docs/general-purpose-machines#n1-standard
[12] https://cloud.google.com/container-optimized-os/docs
[13] https://cloud.google.com/compute/docs/gpus

```
            {"input":[[0,0],[6,0]],"output":[[6,6],[6,6]]}
        ],
    "test":
        [
            {"input":[[0,0],[0,8]],"output":[[0]]}
        ]
}
```

5.4 Solution file

Your submission should output a solution file in form of a JSON file named *solution_teamid.json* and write it to the *secret_data/solution* folder. The file's name consists of "solution" followed by an underscore and your team ID (same as your team name) provided during registration.

Structure of the solution file The structure of the solution file should be as follows:

- It is an array of solutions for each task that are given as objects:

  `[{object_task_1}, ..., {object_task_n}]`

- Each of the solution task objects in the list above have two key-value pairs:
  - "task_name": "filename"
    (The value is the filename of the task without ".json")
  - "test": [object_solution_1, object_solution_2]
    (The value is a list of the solutions to the tests of the task – some tasks have two inputs and two output solutions that have to be predicted)
  - The objects in the list with they key "test" have the following key-value pairs:
    - "output_id": 0 *or* 1
      (Depending on input number (first "test" task is 0, the second one is 1)
    - "number_of_predictions": 0, 1, 2 *or* 3
      (Max. 3 predictions per output are allowed)
    - "predictions": [object_prediction_1, object_prediction_2, object_prediction_3]
      (List of objects containing predictions — max. 3)
  - The objects in the list with key "predictions" have the following key-value pairs:
    - "prediction_id": 1, 2 *or* 3
      (Prediction number, the order does not matter)
    - "output": [[...], ...] *Your solution to the test input as list.*

Example solution file A solution file that contains the solutions for two different tasks could look as follows:

```
[
    {
        "task_name": "0edafea3",
        "test":
            [
                {
                    "output_id": 0,
                    "number_of_predictions": 2,
                    "predictions":
                        [
                            {
                                "prediction_id": 0,
                                "output": [[1,1],[1,1]]
                            },
                            {
                                "prediction_id": 1,
                                "output": [[2,2],[2,2]]
                            }
```

```
                        ]
                    }
                ]
        },
        {
            "task_name": "1c490e4d",
            "test":
                [
                    {
                        "output_id": 0,
                        "number_of_predictions": 1,
                        "predictions":
                            [
                                {
                                    "prediction_id": 0,
                                    "output": [[1,2],[2,1]]
                                }
                            ]
                    },
                    {
                        "output_id": 1,
                        "number_of_predictions": 2,
                        "predictions":
                            [
                                {
                                    "prediction_id": 0,
                                    "output": [[1,1,2],[1,1,9]]
                                },
                                {
                                    "prediction_id": 1,
                                    "output": [[1,1],[1,1],[2,2]]
                                }
                            ]
                    }
                ]
        }
    ]
```


You can also download an example file from our Website [14]

5.5 Docker image

Any docker image that runs on the above virtual machine is allowed. It can contain all kinds of data – publicly available or not – including pre-trained models. The image has to be supplied together with a run command such that when running the container with the provided command it takes the private test set files located in *secret_data/evaluation* and outputs the solution file to *secret_data/solution*. The docker image should also indicate whether it is still running by, for example, outputting how many tasks have already been solved, and by indicating when the program has finished (e.g., by displaying the message "Done!"

Accessing the private test set Your docker container can access the private test set by mounting the corresponding *secret_data* folder present on the virtual machine. An example for the mount part of the Docker run command is the following:

```
mount type=bind,source="$(pwd)"/secret_data, target=/data
```

In this example the *secret_data* directory is mounted in the Docker container and made accessible as *data*.

[14] https://lab42.global/wp-content/uploads/2022/08/solution_teamid.json

Writing the solution file If you have mounted the *secret_data* folder correctly and can access the private test set you can just write your solution file to the *solution* folder and it will be accessible by us once your code has finished.

5.6 Submission form

You have to submit your solution via the submission form available on our website [15]. Among details about your team you have to provide the following things in the form:

- Download link to your Docker image.
- If the download link is not for a repository also provide a command to load the docker image. For example "docker load ¡ file.tar"
- Docker run command, e.g.,

```
docker run --mount type=bind,source="$(pwd)"/secret_data,target=/data IMAGE
```

5.7 Summary

To make an ARCathon submission do the following steps:

1. Create a Docker image containing your algorithm that runs on the virtual machines specified in 5.1 taking the private tasks as an input 5.3 and outputs a solution in the correct format 5.4.
2. Upload the Docker image to a Docker repository and make it accessible. In case you prefer to send us a direct download link save the docker image as a tar file ("docker save IMAGE > /path/to/file.tar").
3. Fill out the submission form. We send you a confirmation once we receive your submission.
4. You will receive a score or feedback from us within 72 hours.

5.8 Frequent issues

In this section we list a few issuces that frequently occur with the submissions to help avoid them in the future.

Building images on ARM machines (like the newer apple devices) If you build your image on a machine with an architecture that differs from the virtual machine one used for evaluation make sure that you specify the architecture of the virtual machine when building your image. It will still run on your local machine albeit not as efficient. The easiest way to do this is to specify the platform in the Dockerfile when you pull your base image. For example when you use an existing python base image:

```
# Use the Python base image for the correct architecture:
FROM --platform=linux/amd64 python:3.9
# Install dependencies
:
```

Using a GPU When using a GPU make sure that it works on our virtual machine with Container Optimized OS. The - *-gpus all* flag does not work and instead on the virtual machine you need a run command like the following [16]:

```
  docker run \
--volume /var/lib/nvidia/lib64:/usr/local/nvidia/lib64 \
--volume /var/lib/nvidia/bin:/usr/local/nvidia/bin \
--device /dev/nvidia0:/dev/nvidia0 \
--device /dev/nvidia-uvm:/dev/nvidia-uvm \
--device /dev/nvidiactl:/dev/nvidiactl \
IMAGE
```

[15] https://lab42.global/arcathon/submission/

[16] https://cloud.google.com/container-optimized-os/docs/how-to/run-gpus#configuring_docker_containers_to_consume_gpus

Furthermore, CUDA applications running in Docker containers that are consuming NVIDIA GPUs need to dynamically discover CUDA libraries. This requires including */usr/local/nvidia/lib64* in the *LD_LIBRARY_PATH* environment variable.

We recommend using Ubuntu-based CUDA Docker base images for CUDA applications on Container-Optimized OS, where *LD_LIBRARY_PATH* is already set appropriately [17]. As an example the beginning of a Dockerfile that is working properly:

```
# Use a Cuda Docker base image with LD_LIBRARY_PATH set appropriately
FROM --platform=linux/amd64 nvidia/cuda:11.8.0-runtime-ubuntu22.04
# Install dependencies
:
```

6 Docker basics

Docker is a platform to develop, deploy, and run applications in containers. A container is a lightweight, standalone, and executable package of software that includes everything needed to run a piece of software, including the code, runtime, system tools, libraries, and settings.

6.1 Basic steps

When using Docker for the first time, here are some of the basic steps explaining the workflow:

1. Install Docker: Before using Docker, you need to install it on your computer. You can find installation instructions for various operating systems on the Docker website [18].
2. Pull a Docker Image: Docker images are the building blocks of containers. You can pull an existing image from a registry like Docker Hub or create your own. To pull an image, use the docker pull command followed by the image name.
3. Run a Container: Once you have an image, you can use it to run a container. To run a container, use the docker run command followed by the image name. You can also specify options like the inclusion of external data, port mapping and environment variables when running a container.
4. Explore Container: To check the details of a running container, use the docker ps command to list all running containers and the docker inspect command to inspect a specific container.
5. Stop and Remove Containers: When you're done with a container, you can stop it using the docker stop command followed by the container name or ID. You can also remove it using the docker rm command.

These are some basic steps to start using Docker. For the purpose of ARCathon you will want to create your own image. A short tutorial on that is given in the next section.

6.2 Basic terms

Dockerfile A Dockerfile is a text file that contains instructions for building a Docker image. It specifies the base image to use, the files to copy into the image, the commands to run inside the image, and the environment variables to set.

A Dockerfile is a step-by-step script that automates the process of creating a Docker image. You can think of a Dockerfile as a blueprint for a Docker image. By following the instructions in the Dockerfile, Docker can build an image that is consistent and repeatable, and that can be used to run containers anywhere.

To build an image using a Dockerfile, you can use the *docker build* command. Docker will read the instructions in the Dockerfile, download the base image if necessary, run the specified commands, and produce a new image that you can use to run containers.

Here is an example of a Dockerfile that could be used when working with Python:

```
# Use an existing Python base image
FROM python:3.9
```

[17] https://hub.docker.com/r/nvidia/cuda/tags/
[18] https://www.docker.com

```
# Set the working directory
WORKDIR /app

# Copy the contents of the current directory into the image
COPY . .

# Install required packages
RUN pip install --no-cache-dir -r requirements.txt

# Run the script
CMD ["python", "main.py"]
```

This Dockerfile starts with a base image of Python 3.9 and sets the working directory to */app*. It then copies the contents of the current directory into the image, which includes a *requirements.txt* file with the dependencies required.

Next, it uses the *RUN* directive to run the *pip* command and install the required packages from the *requirements.txt* file. This ensures that all the necessary packages are installed in the image and available to use when running the script.

Finally, the *CMD* directive specifies the command to run when the container starts, which is *python main.py* in this case. This will start running the main.py script.

You can build the image using this Dockerfile with the *docker build* command, and then run a container from the image to perform the data analysis.

Registry A registry is a place where Docker images are stored and distributed to others. A registry can be public or private, and it allows users to upload, store, and manage Docker images. When you create a Dockerfile you typically start with an image that you pull from a registry.

The most well-known public registry is Docker Hub [19]. It is a centralized repository for Docker images, where anyone can upload, download, and share images with others. The free tier on Dockder Hub will be enough for the purpose of ARCathon.

Docker image vs container A Docker image is a blueprint or a snapshot of a Docker application that includes all of its dependencies, configuration files, and other necessary components. Essentially, it's a package that contains everything needed to run an application inside a Docker container. Docker images are immutable, which means that once you create an image, you cannot change it. However, you can create new images based on an existing image by making modifications to it. A docker image is what we would like you to submit.

On the other hand, a Docker container is a running instance of a Docker image. It is a lightweight, standalone, and executable package of software that includes everything needed to run the application. Containers are created from Docker images, and they allow you to run an application in an isolated environment. Each container runs in its own isolated environment, with its own file system, environment variables, and network settings.

In summary, Docker images are the building blocks for containers, and containers are the instances of running images. You can have multiple containers running from the same image, but each container is a separate instance of the application with its own isolated environment. We would like you to submit a Docker image such that we minimze potential failure points when we evaluate your solution.

6.3 Basic commands

Here are some of the basic Docker commands that you will use frequently:

- *docker run*: This command is used to start a new container from an image. The basic syntax is *docker run [OPTIONS] IMAGE [COMMAND] [ARG...]*. For example, to start a new container from the *arc_solution* image, you would run *docker run arc_solution*. Of importance for an ARCathon submission is the - *-mount* option which will allow your docker container to access the private test set. The basic syntax is *docker run –mount type=bind,source=local_folder,target=container_folder [OPTIONS] IMAGE [COMMAND] [ARG...]*. For example if you have a local folder *secret_data* on your host that you want to mount in a Docker container at */app/data* you would run the following command:

[19] https://hub.docker.com

```
docker run --mount type=bind,source=/secret_data,target=/app/data
    [OPTIONS] IMAGE [COMMAND] [ARG...]
```

- *docker build*: This command is used to build an image from a Dockerfile. The basic syntax is *docker build [OPTIONS] PATH — URL — -*. For example, to build an image from a Dockerfile in the current directory, you would run

  ```
  docker build .
  ```

- *docker ps*: This command lists all the containers that are currently running on the host. The basic syntax is *docker ps [OPTIONS]*. For example, to list all the containers, you would run

  ```
  docker ps
  ```

- *docker stop*: This command is used to stop a running container. The basic syntax is *docker stop CONTAINER_ID*. For example, to stop a container with the ID *d9b100f2f636*, you would run

  ```
  docker stop d9b100f2f636
  ```

  You can obtain the ID using the *docker ps* command mentioned above.

- *docker images*: This command lists all the images that are currently stored on the host. The basic syntax is *docker images [OPTIONS] [REPOSITORY[:TAG]]*. For example, to list all the images, you would run

  ```
  docker images
  ```

- *docker rmi*: This command is used to delete an image from the host. The basic syntax is *docker rmi IMAGE_ID*. For example, to delete an image with the ID *sha256:cbb...*, you would run

  ```
  docker rmi sha256:cbb...
  ```

  The image ID can be obtained with the *docker images* command mentioned before.

- *docker logs*: This command is used to view the logs of a running container. The basic syntax is *docker logs CONTAINER*. For example, to view the logs of a running container with the ID *arcathon*, you would run the following command:

  ```
  docker logs arcathon
  ```

  This will display the logs generated by the application running inside the container. By using the *–follow* or *-f* option, you can follow the logs in real-time, similar to using the tail *-f* command. The *–since* option allows you to only display logs generated since a specified time or date.

These are just a fw of the basic Docker commands existing. However, they should be enough to get you started and make your first submission.

7 Docker python tutorial

We have created a short tutorial that explains how your code can be integrated in Docker if you use Python as your programming language. We will describe how you create a simple Dockerfile to build your first image containing a python code that reads in the private test set and generates a random solution file. You can find the tutorial with all Code on our Website [20].

The Tutorial contains the following:

1. Simple python file creating a valid solution file.
2. Creation of the Dockerfile
3. Building and running the image

But please note that the very reason for using Docker is that you can use any programming language you want.

[20] https://lab42.global/wp-content/uploads/2022/10/ARCathon-Docker-Python-Tutorial.zip

7.1 Python script

To explain the submission rules, we will "dockerize" the Python script which you can find in the zip folder mentioned before (named: tutorial_script.py). The script reads the evaluation data set in the data folder. It makes a meaningless prediction about the output by taking random grids and filling them with the color of the test input with the highest numeric value. Its goal is to explain how a submission should run in the sense that:

- It takes the evaluation data from outside the container as input.
- It outputs text to indicate progression.
- It outputs the solutions to a JSON file with a structure described in the submission rules.

In the end, it outputs text which indicates it has finished.

The python script consists of a function to load the JSON files of the tasks which splits them into training and test tasks separated into a list of dictionaries:

```
import json
import os
import numpy as np


# Define function to read tasks
def load_tasks(path):
    """
    Function to load .json files of tasks
    :param path: Path to folder where tasks are stored
    :return: - training and test tasks separated into a list of dictionaries
                    where each entry is of the type {'input': [.task.], 'output': [.task.]}
             - list of file names
    """
    # Load Tasks
    # Path to tasks
    tasks_path = path
    # Initialize list to s
    # tore file names of tasks
    tasks_file_names = list(np.zeros(len(os.listdir(tasks_path))))
    # Initialize lists of lists of dictionaries to store training and test tasks
    # Format of items will be [{'input': array,'output': array},...,
    # {'input': array,'output': array}]
    tasks_count = len(os.listdir(tasks_path))
    train_tasks = list(np.zeros(tasks_count))
    test_tasks = list(np.zeros(tasks_count))

    # Read in tasks and store them in lists initialized above
    for i, file in enumerate(os.listdir(tasks_path)):
        with open(tasks_path + file, 'r') as f:
            task = json.load(f)
            tasks_file_names[i] = file
            train_tasks[i] = []
            test_tasks[i] = []

            for t in task['train']:
                train_tasks[i].append(t)
            for t in task['test']:
                test_tasks[i].append(t)

    return train_tasks, test_tasks, tasks_file_names
```

We can then use this function to read in all evaluation tasks, make the random predictions and create the solution file:

```
# Read in evaluation tasks
training_tasks, testing_tasks, file_names = load_tasks('data/evaluation/')
# Get number of test tasks for outputting progress later and define counter.
num_test_tasks = len(testing_tasks)
counter = 0
# Do some stuff to generate solution
# Allocate space for overall solution
solution = []
# Iterate over all tasks to generate solution
for test_task, task_filename in zip(testing_tasks, file_names):
    # Allocate space for solutions of task examples
    test = []
    # Store filename
    task_name = task_filename.strip('.json')
    # Iterate over test examples (1 or 2)
    for id_example, example in enumerate(test_task):
        # Get input of example
        example_input = example['input']
        # Do some stuff to generate output
        # Get maximal value of input
        input_max = np.amax(example_input)
        # Do some random stuff to generate outputs
        # Random grid sizes
        random_grid_sizes = np.random.randint(1, 5, size=(3, 2))
        # Allocate space for predicticting objects
        predictions = []
        # Make predictions taking random grid sizes and filling the resulting arrays with
        # color found above
        for prediction_id, grid_size in enumerate(random_grid_sizes):
            # Generate output prediction and change it to a list to create json file later
            output = np.full(grid_size, input_max, dtype=np.uint8).tolist()
            object_prediction = {'prediction_id': prediction_id, 'output': output}
            predictions.append(object_prediction)
        # Generate object solution containing all predictions
        object_solution = {'output_id': id_example, 'number_of_predictions': len(predictions),
                           'predictions': predictions}
        # Add solution of example to list of solutions
        test.append(object_solution)
    # Add solution of examples to overall solution
    object_task = {'task_name': task_name, 'test': test}
    solution.append(object_task)
    # Output progress
    counter += 1
    if counter % 50 == 0:
        print('Generated solution for {} of {} test examples'.format(counter, num_test_tasks))
```

Finally, we store the created solution file as a JSON into the correct folder:

```
# Store solution to json file named solution_teamid where our teamid is lab42
# Store it in solution folder which is mounted
solution_json = json.dumps(solution)
with open('../data/solution/solution_lab42.json', 'w') as outfile:
    outfile.write(solution_json)
# Print that program has finished
print("Program has finished!")
```

7.2 Dockerfile

Before creating a Dockerfile it is best to start with an empty directory (preferably not your root directory) which you can name *docker_tutorial*. In the end, this directory should contain everything needed to build the Docker image. In the case at hand, put the script *tutorial_script.py* in the folder *tutorial_code/* and the folder containing the publicly available tasks in *secret_data*. You

could also choose different names for these folders. However, when you submit your solution, we will put our private task set in the folder *secret_data/evaluation/*, so you can test how this works. We will explain to you how you include this data in your image and make it available to your code below when we discuss how to execute an image.

Now open any text editor at hand and create within the directory a new file called *Dockerfile* without any extension (like .txt). In the end, this file contains all specifications to build the Docker image later. As a launching point, use the official Python 3 image provided by the Docker community [21] and choose the basic one with Python version 3.10.6 tagged 3-slim (we choose the slim one as it is sufficient for our purpose). For this, add the following line to your Dockerfile:

```
FROM python:3-slim
```

The goal is to run the python script called *tutorial_script.py*. For this, the script needs to be added to the Dockerfile. Above, you put the script in the folder *tutorial_code/*, and you can include it by adding the following line to the Dockerfile:

```
ADD tutorial_code/tutorial_script.py /
```

Suppose that the script needs some libraries. In our example, we need the library requires *NumPy*. All dependencies need to be installed before running the image. To install *NumPy*, add the following line to your Dockerfile:

```
RUN pip install numpy
```

Now you are ready to include the command to run the script in your Dockerfile. This can be achieved by adding the following line:

```
CMD ["python", "./tutorial_script.py"]
```

Everything put together; your Dockerfile should look as follows (note that you can add comments by starting your line with a #):

```
# Use basic Python 3 image as launching point
FROM python:3-slim
# Add script
ADD tutorial_code/tutorial_script.py /
# Install dependencies
RUN pip install numpy
# Execute the script
CMD ["python", "./tutorial_script.py"]
```

Summary:

- **FROM**: specifies which image you want to base your image on (e.g., Ubuntu or here Python 3).
- **ADD**: is used to include files like code you want to execute or anything else that should be available when running the image.
- **RUN**: specifies which commands should be executed when building your image.
- **CMD**: will execute the command specified in the brackets when the image is run.

7.3 Building and running the image

You are now ready to build an image from your Dockerfile. For this, open a terminal inside your *docker_tutorial* folder and run the following command:

```
docker build -t docker_python_tutorial .
```

The dot indicates that Docker should look for your file called Dockerfile in the current directory. The *-t* is used to tag your image (here, *docker_python_tutorial*).

We can now run the image as a container and test it out. You can first run the command *docker images* in your terminal. If the previous build was successful, you should see an entry *"docker_python_tutorial"*. As explained in the "Submission Rules & Instructions", we want to

[21] https://hub.docker.com/_/python

include the evaluation tasks outside our container. Here we will take the publicly available ones, which you can download on our website [22]. Then put the data folder into the directory where we have our Dockerfile and rename the data folder to *secret_data/* (that's where we will put the private test set in the folder *evaluation/* after you submitted your solution).

Now you are ready to run our newly created Docker image by passing it the path to our task data:

```
docker run --mount type=bind,source="$(pwd)"/secret_data,target=/data docker_python_tutorial
```

This (or a similar one) is a command for the terminal which is needed for submission together with your image. Using the mount command, you included the folder containing the tasks downloaded before in your docker container, located in your current directory in the data folder. Finally, you use the name of the previously created image to tell which image you want to use.

The terminal output should now look akin to this:

```
(base)   docker run --mount type=bind,source="$(pwd)"/secret_data,target=/data
docker_python_tutorial
Generated solution for 50 of 400 test examples
Generated solution for 100 of 400 test examples
Generated solution for 150 of 400 test examples
Generated solution for 200 of 400 test examples
Generated solution for 250 of 400 test examples
Generated solution for 300 of 400 test examples
Generated solution for 350 of 400 test examples
Generated solution for 400 of 400 test examples
Program has finished!
```

And you should find a file called *solution_lab42.json* in the *secret_data/solution/* folder.

8 Google cloud Tutorial

In this section we will briefly explain you how you can set up your own virtual machine that has the same specifications as the one we use to evaluate your submissions. In that way you can test out your solutions directly before sending them to us. Note also that every new user gets free credits on Google Cloud which you can use for this purpose.

These are the first steps we suggest:

1. Create a Google Cloud account: If you don't already have a Google Cloud account, sign up for one at https://cloud.google.com.
2. Go to the Google Cloud Console: Once you have a Google Cloud account, log in to the Google Cloud Console [23].
3. Create a new project: In the Google Cloud Console, click the project drop-down menu and select or create the project that you want to use for the virtual machine.
4. Go to the Compute Engine section: In the Google Cloud Console, navigate to the Compute Engine section by clicking on the navigation menu and selecting Compute Engine > VM instances [24].
5. Create a new instance: To create a new instance, click the "Create" button.
6. Configure the instance: In the "Create an instance" form, you will be prompted to configure the instance. Choose a name for the instance, select a zone, choose the machine type and the boot image. The configurations we chose for ARCathon are described in the next subsection.
7. Create the instance: When you're ready, click the "Create" button to create the instance.
8. Connect to the instance: Once the instance is created, you can connect to it by clicking the "SSH" button in the Google Cloud Console. This will open a terminal window and connect you to the instance.

[22] https://lab42.global/wp-content/uploads/2022/08/ARC-800-tasks.zip
[23] https://console.cloud.google.com
[24] https://console.cloud.google.com/compute

9. Start using the virtual machine and test your images: You can now use the virtual machine to install software, run applications, and perform other tasks as necessary. As we use Container Optimized OS, Docker will be preinstalled and you can directly pull your image from a registry or also upload it if you have it in form of an archive.

8.1 Virtual machine configuration (CPU only)

In step 6 of the list before you have to configure the VM such that it matches with the configuration we use. For that after you have chosen a name and a region in the Machine configuration section choose *GENERAL-PURPOSE* and then Series N1 and machine type n1-standard-4 (cf. Fig. 4).

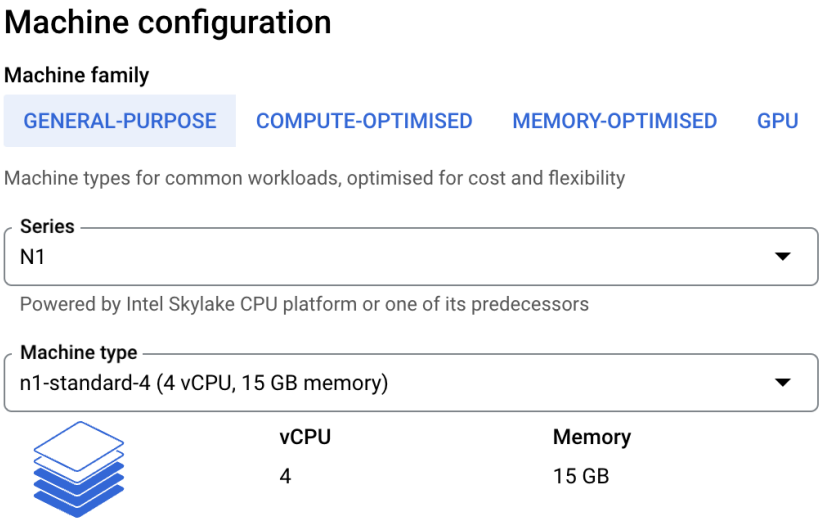


Figure 4: Machine configuration of the virtual machine without GPU.

In the Boot disk section click on *CHANGE* and choose Container Optimized OS and Container-Optimized OS 101-17162.40.34 LTS as the operating system and version. We take a balanced persistent disk as boot disk type. It should look as in Fig. 5

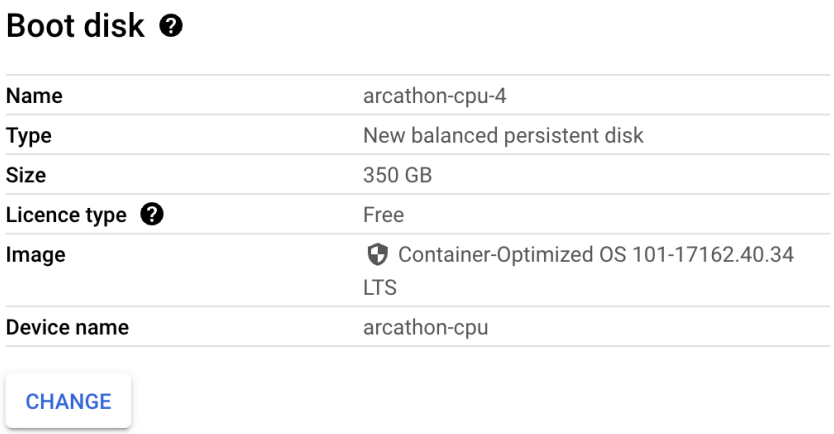


Figure 5: Boot disk configuration together with choise of operating system.

Apart from that we use the standard settings and you can click on *CREATE*.

8.2 Virtual machine configuration (GPU)

The only difference compared to the configuration of the VM without GPU is in the *Machine configuration* section where we add one Nvidia T4. It looks as in Fig. 6.

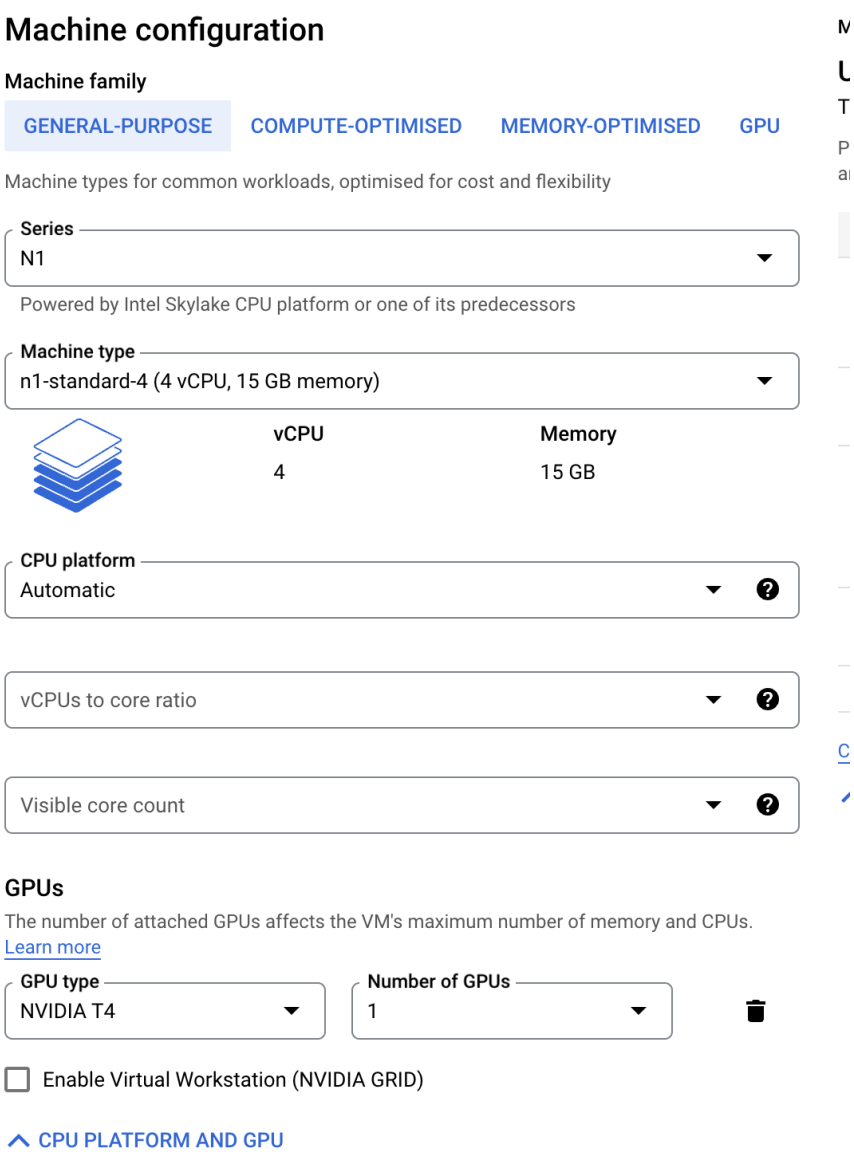


Figure 6: Machine configuration of the virtual machine with GPU.

Note that when using a GPU before being able to use CUDA you have to install the necessary drivers. They are provided by Google in case of the operating system we have chosen and you can find the explanation how you can install these drivers in the Documentation provided by Google [25]

References

[1] François Chollet. "On the measure of intelligence". In: *arXiv preprint arXiv:1911.01547* (2019). URL: https://arxiv.org/pdf/1911.01547.pdf.

[2] Mehran Kazeminia. "A Commentary on the Abstraction and Reasoning Challenge — Kaggle Competition". In: (). URL: https://medium.com/swlh/a-commentary-on-the-abstraction-and-reasoning-challenge-kaggle-competition-16ba30fac0ec.

[25] https://cloud.google.com/container-optimized-os/docs/how-to/run-gpus#install