Creating a Docker Environment for Jupyter Notebook-Based Machine Learning Projects

Authors

ALESSANDRO CABRAS[1]

[1]INAF - Osservatorio Astronomico di Cagliari

Reviewers

ANTONIO PODDIGHE
ANTONIETTA ANGELA RITA FARA

Contents

Abstract

This project proposes the configuration of a locally optimized development environment for Artificial Intelligence projects, leveraging containers and Jupyter notebooks. While services like Google Colab offer quick and convenient access to pre-configured cloud resources for machine learning, subscription costs and resource limitations may be restrictive for some projects. To overcome these challenges, the creation of a locally executable environment similar to Colab is suggested, but deployable on-premise on a local server. This approach allows for full hardware customization, including GPU selection, and eliminates subscription cost constraints. In the following chapters, the necessary steps to configure this local environment will be outlined, starting from hardware selection and proceeding with the installation of required dependencies and environment setup. By following these guidelines, users will be able to establish a local machine learning development environment that provides greater control and flexibility, while retaining the convenience and familiarity associated with Google Colab.

List of Acronyms

AI Artificial Intelligence

GPU Graphics Processing Unit

ECC Error-Correcting Code

HPC High Power Computing

CUDA Compute Unified Device Architecture

PCIe Peripheral Component Interconnect express

OAC Osservatorio Astronomico di Cagliari

TPU Tensor Processing Units

List of Figures

1
Introduction

In recent years, the field of machine learning has rapidly gained prominence, revolutionizing industries across the globe with its ability to derive insights, patterns, and predictions from vast amounts of data. From healthcare to finance, from transportation to entertainment, the applications of machine learning are virtually limitless, offering unparalleled opportunities for innovation and advancement.

Within this transformative landscape, the domain of radioastronomy stands out as a particularly compelling arena for the application of machine learning techniques. Radio telescopes capture immense volumes of data from the depths of space, unveiling the mysteries of the cosmos with unprecedented precision and detail. However, the sheer scale and complexity of these data sets present significant challenges for traditional analysis methods. Here, the power of machine learning emerges as a vital tool, enabling researchers to sift through terabytes of data, identify subtle patterns, and extract meaningful insights about the universe.

Yet, harnessing the full potential of machine learning in radio astronomy requires more than just sophisticated algorithms and innovative methodologies. It demands dedicated hardware infrastructure tailored to the specific demands of machine learning tasks. Building and deploying machine learning models, especially those operating on vast astronomical datasets, necessitate computational resources optimized for high-performance computing and parallel processing.

In response to these evolving needs, the development of dedicated machine-learning environments becomes imperative. These environments provide researchers and data scientists with the computational power and specialized hardware required to train, test, and deploy machine learning

models efficiently. By leveraging technologies such as Docker containers, which offer portability and scalability, and platforms like Google Colab, which provide interactive and collaborative environments, organizations can accelerate their machine learning initiatives and unlock new frontiers in radioastronomy and beyond.

In this document, we explore the significance of establishing dedicated machine-learning environments equipped with dedicated hardware.

1.1 Choosing the Right Hardware for Machine Learning Environments

When it comes to building dedicated hardware infrastructure for machine learning environments, one of the most crucial decisions revolves around the choice of GPUs. GPUs have emerged as one of the best choices for modern machine learning, offering parallel processing capabilities ideal for training deep neural networks and processing large datasets. However, selecting the appropriate GPU architecture requires careful consideration of factors such as performance, cost-effectiveness, and workload characteristics.

In the realm of GPUs, two primary categories dominate the landscape: gaming GPUs and server-grade GPUs. While both types share fundamental architectural similarities, they are optimized for distinct use cases and exhibit differences in terms of performance, reliability, and price.

Gaming GPUs, exemplified by popular consumer brands like NVIDIA GeForce and AMD Radeon, are designed primarily for gaming enthusiasts and PC gaming rigs. These GPUs prioritize features such as high clock speeds, fast memory bandwidth, and advanced graphical rendering capabilities to deliver immersive gaming experiences. While gaming GPUs can certainly be used for machine learning tasks, they may not offer the same level of reliability as their server-grade counterparts.

Boards like NVIDIA Tesla and AMD Instinct series, are purpose-built for High Power Computing (HPC) and data center applications. These GPUs are engineered to deliver exceptional computational power, reliability, and scalability, making them ideal for intensive machine-learning workloads. Server-grade GPUs often feature larger memory capacities, Error-Correcting Code (ECC) memory, ensuring robust performance and stability under high demanding conditions.

The choice between gaming GPUs and server-grade GPUs depends largely on the specific re-

quirements and constraints of the machine learning environment. For small-scale projects, research labs, or individual developers on a budget, gaming GPUs may offer a cost-effective solution with sufficient performance for training and experimentation. However, for enterprise-grade deployments, production environments, or applications demanding maximum performance and reliability, server-grade GPUs are typically the preferred choice.

In scenarios where scalability, reliability, and support for enterprise-grade features are paramount, server-grade GPUs shine. These include mission-critical applications in industries such as healthcare, finance, and autonomous vehicles, where downtime and performance bottlenecks can have significant consequences. Additionally, tasks involving large-scale data processing, complex simulations, and real-time inference benefit greatly from the raw computational power and optimized drivers of server-grade GPUs.

In contrast, gaming GPUs may find favor in non-production environments, academic settings, or projects with modest computational requirements and budget constraints. Gaming GPUs offer competitive performance at a lower price point, making them accessible to a broader audience of researchers, hobbyists, and enthusiasts.

In the context of the project outlined, the decision has been made to utilize gaming GPUs. Given the project's scope, budget considerations, and computational requirements, gaming GPUs present a compelling choice that strikes a balance between performance and cost-effectiveness. By leveraging gaming GPUs, the project team can access substantial computational power at a more affordable price point, enabling efficient training and experimentation with machine learning models. This decision underscores the importance of aligning hardware choices with project objectives, workload characteristics, and budgetary considerations, ensuring optimal performance and resource utilization throughout the development lifecycle.

1.2 Nvidia GeForce RTX 4090: A low budget solution

The Nvidia GeForce RTX 4090 stands out as a high-end graphics card, boasting an advanced architecture and formidable processing capabilities perfectly suited for machine learning tasks. Its Ada Lovelace architecture represents a significant advancement over the Ampere architecture, delivering notable improvements in performance and power efficiency. Enhanced by 4th generation Tensor Cores for AI operations and 3rd generation RT Cores for ray tracing, the RTX 4090 offers exceptional computational power with its 16384 Compute Unified Device Architecture (CUDA)

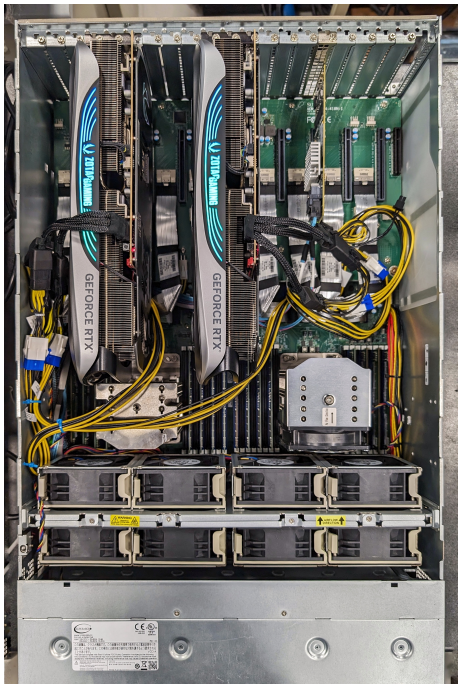

Figure 1.1: Two GPU NVIDIA RTX 4090, installed in a Supermicro server at the OAC.

Figure 1.2: ZOTAC GAMING GeForce RTX 4090 Trinity graphics card [1]

cores and a boost clock of 2.52 GHz. Furthermore, its 24 GB GDDR6X memory, featuring a bandwidth >1000 GB/s, ensures rapid and efficient processing of large machine-learning models. Facilitating faster data transfer, the Peripheral Component Interconnect express (PCIe) 5.0 interface doubles the bandwidth compared to PCIe 4.0. Despite its high 450 W TDP, indicative of its ability to handle demanding applications, the RTX 4090 maintains compatibility with major machine learning frameworks such as TensorFlow, PyTorch, and CUDA, while boasting improved power efficiency thanks to its architecture. When selecting a graphics card for machine learning projects, considerations including model type, dataset size, and system resources must be weighed carefully against the capabilities of the GeForce RTX 4090 to ensure optimal performance and efficiency.

The RTX 4090 falls within a performance range similar to that of Ampere GPUs such as the

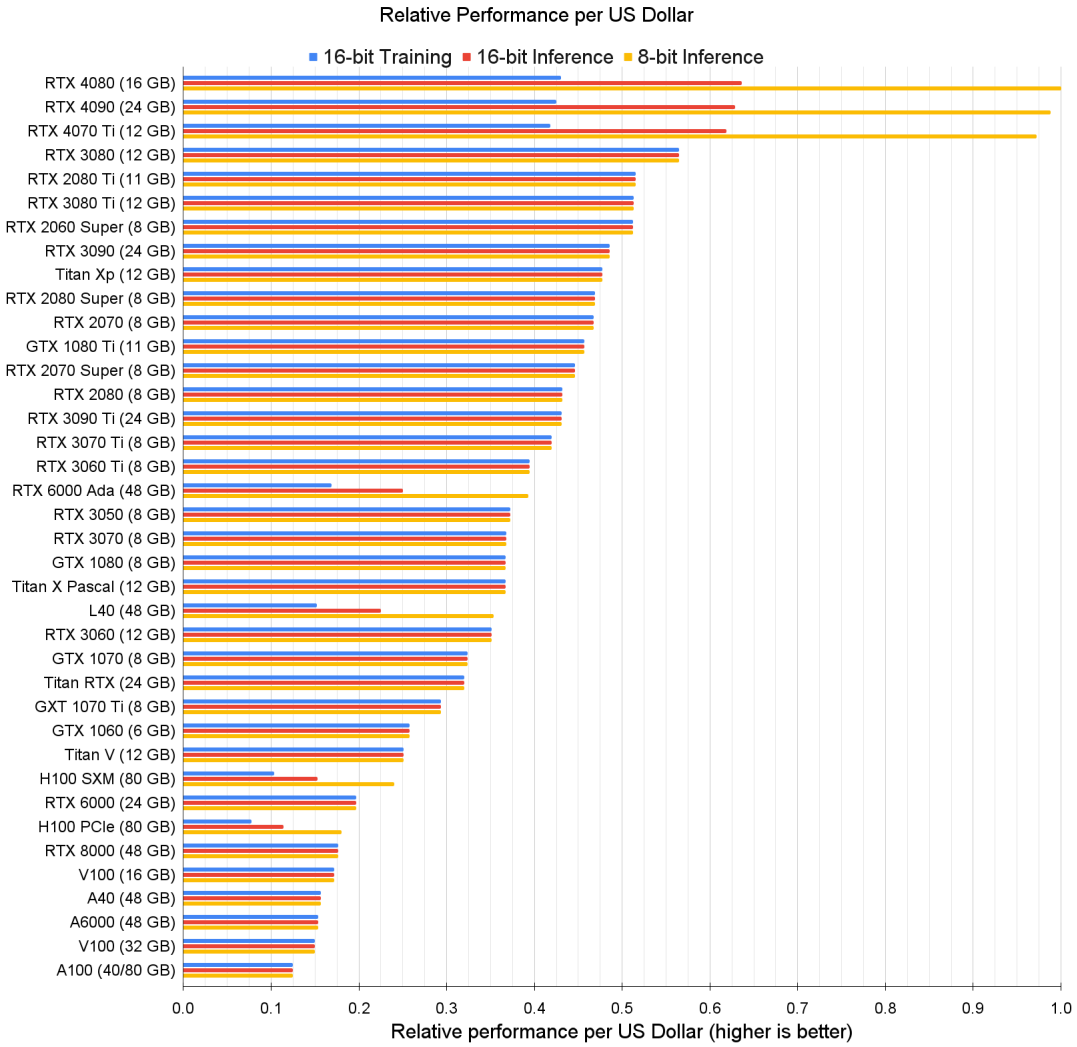

Figure 1.3: One of the graphs extracted from the web comparing GPU performance in the market relative to their price. [2]

A100 and A40, providing an intriguing alternative for machine learning and artificial intelligence applications in desktop environments. The performance of the RTX 4090 also rivals with Tesla V100 GPUs, an earlier model still widely utilized in professional settings.

When considering both acquisition and maintenance costs, the situation changes, Figure 1.3 depicts the performance per US dollar for various GPUs, taking into account 8 or 16-bit representation of numbers during inference and training. Utilizing this chart to identify the most suitable GPU entails several key considerations. First and foremost is determining the required GPU memory capacity, with a rule of thumb suggesting a minimum of 12 GB for image generation and 24 GB for transformer work. Additionally, users should prioritize GPUs offering the highest relative performance-to-cost ratio based on their chosen metric while ensuring the selected GPU meets

their memory requirements.

While the RTX 4070 Ti and RTX 3080 emerge as the most cost-effective options for 8-bit and 16-bit inference respectively, it's important to note that these GPUs may lack sufficient memory for certain use cases. Nevertheless, they could serve as excellent entry-level cards for deep learning enthusiasts, particularly for academic applications where model size may be less critical.

The chart also takes into account the electricity cost of ownership over five years, assuming a 15% GPU utilization rate and an electricity price of $0.175 per kWh. Notably, the electricity cost for an RTX 4090 amounts to approximately $100 per year. Interpreting the chart reveals that a desktop computer equipped with RTX 4070 Ti GPUs delivers roughly twice the 8-bit inference performance per dollar compared to a system featuring an RTX 3090 GPU, highlighting the cost-effectiveness of the former option.

2

Popular tools for ML Development

In Artificial Intelligence (AI) projects development, access to powerful tools and environments is paramount to streamline workflows, facilitate collaboration, and accelerate experimentation. The next sections describe some of the essential tools used in the machine learning landscape, highlighting the significance of virtualized environments for code execution and collaboration.

2.0.1 Jupyter Notebooks

Jupyter Notebooks offers an interactive computing environment that mixes code, visualizations, and narrative text in a single document. With support for multiple programming languages, including Python, R, and Julia. By interleaving code cells with descriptive text and visualizations, practitioners can document their thought processes, experimental setups, and analysis results within a single, coherent narrative. This not only facilitates understanding for others but also enhances the reproducibility of experiments, as each step can be explicitly documented and executed sequentially.

Additionally, the ability to execute code cells interactively permits one to quickly prototype algorithms, visualize intermediate results, and iterate on different approaches within the same document. This iterative workflow encourages experimentation and facilitates the discovery of optimal solutions to complex machine-learning problems.

Furthermore, the collaborative features of Jupyter Notebooks make them well-suited for team-based research and development projects. Teams can easily share notebooks, collaborate on code

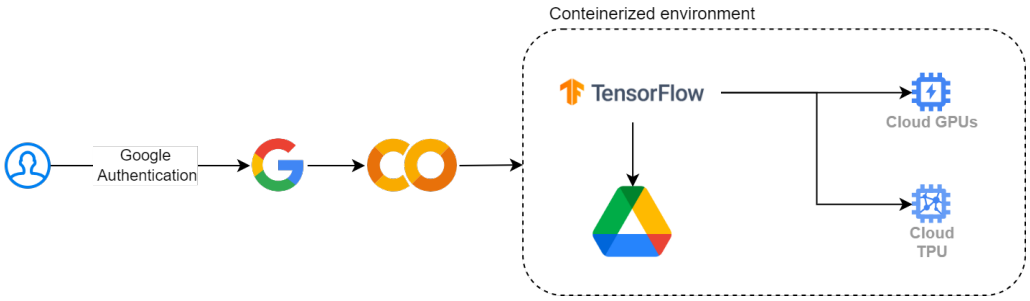

Figure 2.1: Descriptive diagram of Google Colab, from user authentication to cloud resource allocation.

and analysis, and provide feedback in real time. This collaborative environment promotes knowledge sharing, accelerates the pace of innovation, and fosters a culture of teamwork within machine learning teams.

2.0.2 Google Colab

Google Colab extends the capabilities of Jupyter Notebooks by providing a cloud-based environment for machine learning experimentation. Leveraging Google's infrastructure, Colab offers free access to GPUs and Tensor Processing Units (TPU)s, enabling users to train complex models and process large datasets without the need for expensive hardware investments. Moreover, Colab facilitates seamless collaboration through real-time editing and sharing of notebooks, allowing multiple users to work together on the same project simultaneously. Its integration with Google Drive enables easy storage and access to datasets and trained models, further enhancing productivity and collaboration.

Google Colab provides a virtualized environment pre-configured with commonly used dependencies for machine learning, such as TensorFlow, NumPy, and Matplotlib. This environment allows users to access and utilize these libraries without the need for manual installation or configuration. As depicted in Figure 2.1 each time computational power is requested, Google Colab allocates a fresh, clean instance of the virtual environment. This ensures consistency and reproducibility in experiments, as users can start with a clean slate for each session, free from any residual artifacts or modifications from previous runs.

Google Colab, while offering powerful capabilities for ML experimentation, does come with certain limitations and considerations. Firstly, it's important to note that Google Colab provides a free tier with access to GPUs and TPUs, making it an attractive option for many developers. However, this free tier has limitations in terms of resource availability and usage. For instance,

there are restrictions on the duration of each session, typically capped at 12 hours, after which the session is terminated. Additionally, there may be limits on the amount of GPU memory and storage space available for each user.

Furthermore, Colab offers a paid version, known as Colab Pro, which provides access to additional features and resources. These include longer session durations, priority access to GPUs, and increased memory and storage quotas. While Colab Pro offers enhanced capabilities, it is a subscription-based service, meaning users incur costs for accessing these premium features. The pricing model for Colab Pro is based on a monthly subscription fee, with costs varying depending on usage and resource requirements. It's also worth noting that while Google Colab provides access to powerful computing resources, these resources are shared among all users of the platform. As a result, users may experience variability in performance and availability, particularly during peak usage times when demand for resources is high.

In the current project, we aim to harness the advantages offered by Google Colab while opting for a local runtime environment. This approach involves configuring a dedicated machine that operates on the same principles as Google Colab but without the limitations associated with subscription-based access and with the added benefit of customizable hardware. By adopting this strategy, we seek to leverage the convenience and familiarity of the Google Colab environment while gaining greater control over hardware resources and eliminating dependencies on external services.

In the subsequent chapter, we will detail the steps required to set up a machine configured in this manner. This includes selecting suitable hardware components, installing the necessary software stack, and configuring the runtime environment to mirror the functionality and flexibility of Google Colab.

3

Setting Up a Local Environment

3.1 Docker configuration

To set up a local environment for running Google Colab-like runtimes on their machine, the user should follow the detailed steps provided below:

1. Install Docker: The user must ensure that Docker is installed on their local machine. Docker [3] is a platform that enables developers to build, package, and distribute applications as lightweight, portable containers. They can download and install Docker from the official website following the provided instructions.

2. Starting the Runtime: Once Docker is installed, the user can start a runtime using the following command:

```
docker run -p 127.0.0.1:9000:8080 us-docker.pkg.dev/colab-
    images/public/runtime
```

   This command initializes a Docker container using the Google Colab runtime image (us-docker.pkg.dev/colab-images/public/runtime). Docker images are lightweight, standalone, and executable packages that contain all the necessary software components, including the operating system, libraries, and dependencies, needed to run an application or service. It's worth noting that the Docker image is downloaded from the repository the first time it's used, and subsequently remains cached on the system, ensuring faster startup times for subsequent runs.

The -p flag specifies port mapping, where 127.0.0.1:9000 on the host machine is mapped to port 8080 inside the container. This port mapping allows for communication between the host and the Docker container.

It's important to highlight that the Docker container provides an isolated and reproducible environment, much like the virtualized environment used in Google Colab. This means that all the required libraries, dependencies, and configurations, including those typically found in a Colab environment, are pre-installed within the Docker image. This ensures consistency and compatibility across different computing environments, allowing users to seamlessly transition between local development and cloud-based workflows.

3. GPU Support: If GPU support is desired, the user should ensure that NVIDIA drivers and the NVIDIA container toolkit are installed. To start the runtime with GPU support, the following command should be executed:

```
docker run --gpus=all -p 127.0.0.1:9000:8080 us-docker.pkg.dev
    /colab-images/public/runtime
```

This command is similar to the previous one but includes the –gpus=all flag, which instructs Docker to allocate all available GPUs to the container.

4. Accessing the Backend: Once the container is started, a message containing the initial backend URL used for authentication will be printed. The URL will be in the format "'http://127.0.0.1:9000/?token=…".

5. Enabling SSH Tunnel: If the user's remote machine is accessed via SSH it is advisable to create an SSH tunnel. This can be done using a command similar to the following:

```
ssh -N -f -L 9000:localhost:9000 user@remote_machine_ip
```

This command creates an SSH tunnel from the user's local device to the remote machine using the same port forwarded to the container.

6. Connecting to the Local Runtime from Colab: Lastly, the user should access Google Colab website and click on the "Connect" button. Select "Connect to a local runtime…" and enter the backend URL obtained in the previous steps in the dialog box. Click "Connect," and the connection to the local runtime should be successfully established.

3.2 Link Google Drive as volume

One of Colab's advantages over its competitors is its ability to provide easy access to Google Drive by integrating it into the filesystem of the containerized environment. One solution is to mount Google Drive on the Linux system using Rclone [4]. Users can follow these steps:

1. Installation of Rclone: Rclone must first be installed on the Linux system. Users can refer to the official Rclone documentation for installation instructions or utilize their system's package manager.

2. Configuration of Rclone for Google Drive: After installing Rclone, users need to configure it for accessing Google Drive. They can initiate the configuration process by executing the following command in a terminal:

   ```
   rclone config
   ```

   Following the guided prompts, users can add Google Drive as a new remote, obtain an access token, assign a name to the remote (e.g., "remote"), and complete the configuration.

3. Mounting Google Drive: Once Rclone is configured, users can mount their Google Drive on the Linux system using the following command:

   ```
   rclone mount gdrive: /path/on/the/system --allow-other --dir-
       perms 775 --daemon
   ```

   Users should replace "/path/on/the/system" with the desired destination directory on the Linux system. Upon executing this command, Google Drive will be mounted and accessible as a local directory on the system.

4. File Management: With Google Drive mounted on the Linux system, users can navigate, read, write, and modify files just like any other local file. All changes made to the files will be automatically synced with Google Drive and vice versa.

5. Unmounting Google Drive: When users finish working with Google Drive, they can unmount it from the Linux system using the following command:

   ```
   fusermount -u /path/on/the/system
   ```

   They should replace "/path/on/the/system" with the path to the mount directory of Google Drive on the Linux system.

By following these steps, users can seamlessly mount Google Drive on their Linux system using Rclone and efficiently manage their files between the system and Google Drive. This setup provides a convenient solution for accessing and working with files, including large datasets, within a Linux environment. At this point, it is possible to link the folder as a volume to the Docker container using the -v option, enabling seamless navigation and interaction with the files within the container environment.

```
docker run --gpus=all -p 127.0.0.1:9000:8080 -v /path/on/the/system:/
    path/on/the/container us-docker.pkg.dev/colab-images/public/runtime
```

3.3 Automatize the process

The bash script described below automates the operations outlined in the preceding sections. It establishes the connection with the remote machine, initiates the container, and returns the backend URL for integration with Google Colab.

```
#!/bin/bash

# IP of the remote machine
REMOTE_IP="192.168.145.80"
# Username for accessing the remote machine
REMOTE_USER="neurabeast"

# Check if SSH tunnel is already active
if ! pgrep -f "ssh -N -f -L 9000:localhost:9000 $REMOTE_USER@$REMOTE_IP
    " > /dev/null; then
    # If SSH tunnel is not active, enable SSH tunnel
    ssh -N -f -L 9000:localhost:9000 $REMOTE_USER@$REMOTE_IP > /dev/
        null 2>&1
fi

# Check if the container is already running
container_id=$(ssh -o ConnectTimeout=10 $REMOTE_USER@$REMOTE_IP "docker
    ps -q -f 'ancestor=us-docker.pkg.dev/colab-images/public/runtime'"
```

```
    )

# If the container is running, stop it
if [ -n "$container_id" ]; then
    ssh $REMOTE_USER@$REMOTE_IP "docker stop $container_id"
fi

# Start a new container
remote_path=$(ssh -f -o ConnectTimeout=10 $REMOTE_USER@$REMOTE_IP "
    docker run --quiet --gpus=all -p 127.0.0.1:9000:8080 -v /home/
    neurabeast/data/:/content us-docker.pkg.dev/colab-images/public/
    runtime" | grep -o -m 1 'http://127.0.0.1:9000/?token=[a-zA-Z0-9]*'
    )

# Print the extracted path in green
echo -e "\e[32mBackend URL: $remote_path\e[0m"
```

To optimize the functioning of this script, it is useful to perform key exchange between the two machines to avoid SSH password prompts, and Docker must be configured to run without requiring sudo privileges. Below is a breakdown of its functionality:

1. The script verifies the status of the SSH tunnel by searching for the relevant process. If the tunnel is inactive, it establishes a new SSH tunnel: utilizing the ‘pgrep‘ command to identify the SSH tunnel process. In the absence of the process, the script initializes the tunnel with the ‘ssh‘ command, employing options such as ‘-i‘ to specify the private key, ‘-N‘ to prevent remote command execution, and ‘-f‘ to run SSH in the background. This newly created tunnel forwards port 9000 from the local machine to the remote machine, facilitating secure communication between the two endpoints.

2. The script checks if a container with the Colab image is already running on a remote host. If such a container is found, it is stopped to ensure a clean state.

3. Subsequently, the script proceeds to launch the Docker container on the remote machine through the ‘docker run‘ command: leveraging the ‘ssh‘ command to execute the container remotely. Additionally, the ‘–quiet‘ option is employed to silence non-critical output from the

Docker container, ensuring a streamlined execution process. It uses the ‘grep‘ command to search for a specific pattern in the Docker container logs, extracting the URL that matches the pattern.

4. Finally, the script prints the extracted backend URL.

Conclusions and Future Steps

In conclusion, this project introduces a pragmatic solution to the challenges faced by AI practitioners in accessing and utilizing cloud-based resources for machine learning development. By suggesting to make a development environment similar to Google Colab but usable on a local server, this project deals with worries about subscription costs and limited resources that could slow down some projects. This approach not only offers users full control over hardware customization, including GPU selection but also eliminates the financial constraints associated with subscription-based services. In the future, it would be advantageous to establish similar environments for projects extending beyond machine learning, which require proper isolation and the setup of pre-configured environments. This approach helps mitigate potential risks arising from incorrect package or library installations, as well as from shared resource utilization.

Bibliography

[1] *Zotac gaming geforce rtx 4090 trinity*, https://www.zotac.com/it/product/graphics_card/zotac-gaming-geforce-rtx-4090-trinity-0, Accessed: 2024-03-06.

[2] *Which gpu(s) to get for deep learning: My experience and advice for using gpus in deep learning*, https://timdettmers.com/2023/01/30/which-gpu-for-deep-learning/, Accessed: 2024-03-06.

[3] *Docker documentation*, https://docs.docker.com/manuals/, Accessed: 2024-03-06.

[4] *Rclone documentation*, https://rclone.org/docs/, Accessed: 2024-03-06.