Assignment Solution

Cristian Mora

JR_1038744 Machine Learning Engineer (m/f/x)

Part 1: The Use Case

The use case relates to the deployment of an API demo to allow image generation as part of POST request to it. The selected technology is known as ControlNet which is based on the generation of images by an AI model which in the context of this assignment, is expected to be consumed by end users as a service.

The first interesting point of this technique is the lack of training of large diffusion models, but rather an alternative way to fine tuned to specific applications. The main concept behind this technique is the control of pre-trained large diffusion models to support additional inputs parameters [1] controlled by so called external networks; much smaller scale models. This technique or strategy allows the development of models for specialized or particular tasks where the larger models might not perform as well. Starting by one image that serves as reference, the combination of external networks and diffusion models can help to generate similar ones or new ones with specific characteristics.
At a first glance, application can be targeted to simulate scenarios where obtaining training data (images) is not trivial, issues happen at a low rate or it is just highly costly as in production processes with low yield.
Following this line of generation of images, ControlNet can be used to generate synthetic data that closely follow the real scenarios. For instance, using prompts and reference images at the same time, one can create a parametrized set of images that include features such as variations on distance, size of elements, etc. these new images can be feed into other models for the detection of such issues; a direct application in the frame of object detection and recognition when little data is available.
By using external networks specialized at depth inference it could be possible to make reconstruction of 3d objects such as in detection of defects in machined components.
Building models that can restoration, could be used to enhance pictures taken under poor conditions.
However, the ability of having solutions that do not require a datacenter to train a single model, can generate the emergence of use cases (and models) to solve issues or problems that can be solved in a different and more efficient way. Perhaps, a camera with depth of view (stereo sensors) would be more efficient and precise than a model that extracts depth information from a picture; this is the scenario where a specialized device exists already and an AI model might not be the best solution.
The capability of generating parameterized realistic images, can lead to high amounts models developed under wrong expectations, which might end up being as costly as a bigger model and at the same time less useful. Use-cases still would need to be evaluated and considered.

Even when the only available tool is a hammer, not all problems are nails.

Part 2: Code Assessment

The code of the original project (https://github.com/lllyasviel/ControlNet) is an artifact of research. In this context, documentation of the code itself is scarce; while the documentation of the concept, technique and use-cases, along implementations, are abundant. There is a clear example on how the model can be trained, and how can be reused somewhere else (as plug-in for Automatic1111). From the usage perspective, the code is not ready to be used in production, but is clear enough to allow its enhancement. Most of the changes I would attempt on the code might end up being a cleanup of redundant functionalities, removing hard-coded parameters and streamlining the process to better standardize its usage. In general terms, the repository needs some refactoring. Using pipeline structures similar to torchvision transforms can lead to less duplication of code.

Most of the changes I would made to the code, relate to structure, separation of functionalities and packaging; there seems to be a considerable portion of it that is not used at all. The main idea being to reuse components as often as possible and avoid code duplication. The examples provided in the repository are a testament of how much code is duplicated with very little changes; most of the example contain some sort of web-ui that provides access to the models; most of the parameters are also very similar if not the same.

The workflows for development would be a nice addition to this repository and the general project, there is even a version 1.1 in a different repository instead of using tags or release branches. The differences between both are not clear to me, and it can not be grasped from reading the commit messages. Some high level practices that can be beneficial for this repository are:

- Attach to a version control workflow: trunk-based development might be a good idea at the beginning. Use feature branches instead of direct pushing to the main branch and integrate some discipline.
- Use clearer commit messages, explain what the changes are instead of repetitive or even one character commits.
- Take a look to suggestions in pull requests (embrace the community), so the usage of CPU and GPU is asier.
- Have some versioning scheme such as semantic versioning to track relevant changes in the code.

It is difficult to name more practices, without having the feeling of bashing on the project, which is not the idea. These type of repositories are meant for researchers; where development rules don’t need to be so strong. At a later stage the code can be brought to a more productive status by software and Machine learning engineers, so researches can focus in their area.

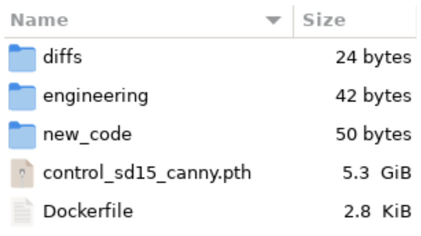
On a more performance related topic, one of the fascinating features of this technique, is that models can be trained in consumer hardware. The authors claim that the training was performed in NVIDIA RTX 3090 Ti (consumer graphics card) over the span of one week. While larger models (Stable Diffusion) was trained in NVIDIA A100 (enterprise hardware) running for around 2000 hours of GPU time. Since the model is basically fine tuned, in terms of costs is much convenient that training a more complex model [1]. This characteristic, allows the development of many more specialized models at lower costs.

Part 3: Coding Challenge

This part of the assignment is what took most of the time, and I have to admit, it took more than 6 hour by itself, mostly due to some technical challenges discussed at the end of this section.

The solution to the code challenge consists of an API that can make use of the requested model in order to generate images. The code is divided thematically into folders located along this document. The directory named “code” contains the main elements to create a Docker image and deploy a container with the application.
Before attempting to build the docker image, it is required to download the same model requested in the enunciate of the challenge: “control_sd15_canny.pth”. This model can be downloaded under this link:
https://huggingface.co/lllyasviel/ControlNet/blob/main/models/control_sd15_canny.pth
The downloaded file must have the original name (control_sd15_canny.pth) must be placed along the Dockerfile as shown in the following image:

Dockerfile and API:



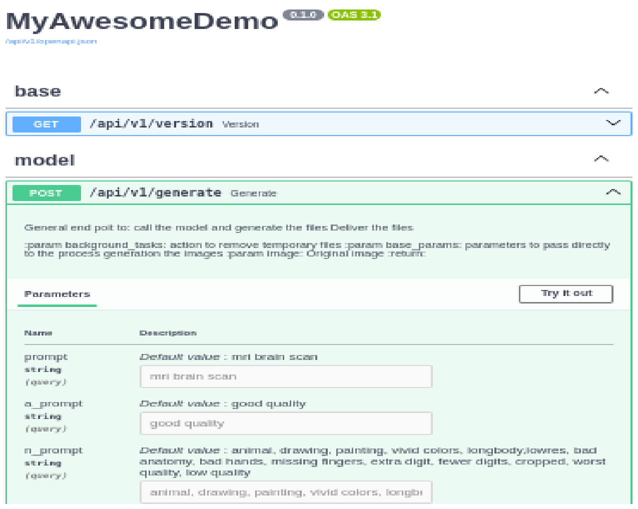
Name	Size
diffs	24 bytes
engineering	42 bytes
new_code	50 bytes
control_sd15_canny.pth	5.3 GiB
Dockerfile	2.8 KiB

Please note that the size of the file is beyond 5GiB and it was not sent by email. There are other files that are required to be downloaded in order to run the model and the API (external networks); those are downloaded automatically as part of the building process since they are requested by the code. The reason to not to include the model when building the docker image, is two fold. One one hand, the model is not downloaded automatically as part of the provided code (while other small ones do get downloaded). On the other hand, repetitive builds of the image in order to adjust, would generate long waiting periods due to the massive downloads and might even end up blocking the access to the page temporarily (ban).

Before building the image, consider that depending on the machine it is run and its certificates, it can be that issues regarding certificates are raised when files are downloaded using python and the request library (hidden in the main code), by coding in a network different of Zeiss' in my private machine, I avoided those issues. Back to the image.

Once the model is downloaded, the first step is to build the image, the name is not important, but it needs to be the same in the next step. This command needs to be executed where the Dockerfile is located.

```
docker build . -t mywesomeapi
```



MyAwesomeDemo 0.1.0 OAS 3.1
/api/v1/openapi.json

base

GET /api/v1/version Version

model

POST /api/v1/generate Generate

General end poit to: call the model and generate the files Deliver the files

:param background_tasks: action to remove temporary files :param base_params: parameters to pass directly to the process generation the images :param image: Original image :return:

Parameters | Try it out

Name	Description
prompt string (query)	*Default value* : mri brain scan mri brain scan
a_prompt string (query)	*Default value* : good quality good quality
n_prompt string (query)	*Default value* : animal, drawing, painting, vivid colors, longbody,lowres, bad anatomy, bad hands, missing fingers, extra digit, fewer digits, cropped, worst quality, low quality animal, drawing, painting, vivid colors, longb

In the process, an image with 15 layers called "mywesomeapi" will be built, the amount of layers in this case is a feature used to follow discrete steps, not required for production; it can be reduced. This might be time to go for a coffee, it might take a while due to downloads and caching. This will clone the project repository (ControlNet) as part of the process, create the conda environment, install dependencies and patch the repository to be run with the CPU (there is black magic coming on its way).
The Dockerfile also contains a small description of what it is intended to do at each step. Once the building process finishes, to run the container, run the following command (use the same name as in the previous step):

```
docker run -p 8080:8080 mywesomeapi
```

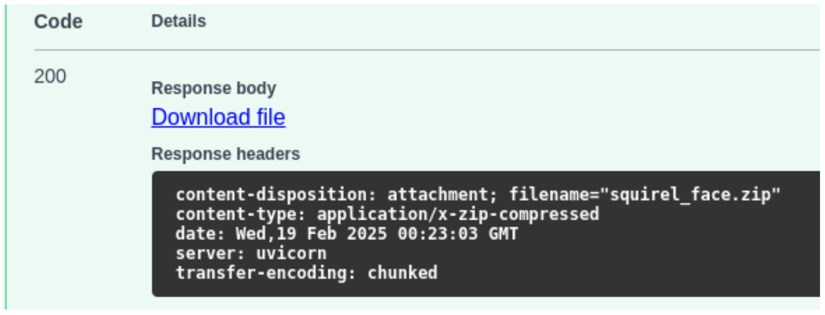
It is important to consider the port forwarding, the container exposes port 8080 for the API. Then, it the host machine, access the URL: http://localhost:8080/docs. This URL will provide access tot he SWAGGER documentation of the endpoints (auto-generated by FASTAPI), this describes the inputs and constraints of the inputs. The URL of this endpoint is *http://localhost:8080/api/v1/generate,* it only allows POST requests and all fields are mandatory. At the end of the form (not shown in the screenshot), one can upload an image as input to the model.

Some limitations:

- Due to the API running on CPU mode (see further discussion about challenges below), the default resolution for the resulting images is kept at 256 pixels and just one image is requested by default. These values can be increased at the cost of higher resource usage.

- Just images with formats “png”, “jpeg” and “jpg” are accepted. The format itself is not validated. However, the filename is expected to contain those extensions. Other extension will raise an exception and an error will be obtained in return.
- No interactive parameter selection is offered, just inputs that are expected to be filled with simple text fields. The wrong type or of inputs or out of boundaries would also raise errors.
- Due to time constraints: there is no data persistence (no database, no logs beyond those shown in the terminal, no storage of images at the application side, there is just delivered results and interim files are deleted after each request).
- There is no asynchronous delivery of results. Users need to wait for the request to be processed, depending on the input parameters this can range from seconds to hours.
- When using the above mention page, download is not automatic, a link is provided.
- Just one image can be processed at the time (just one input for images).

After a successful request and a waiting period, the results will be delivered as a link with a compressed file (zip) as shown in the left image below:


Code Details

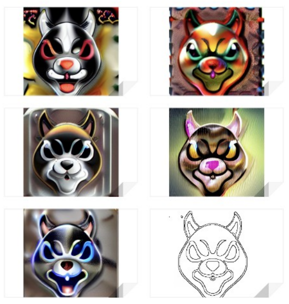
200

Response body

Download file

Response headers

```
content-disposition: attachment; filename="squirel_face.zip"
content-type: application/x-zip-compressed
date: Wed,19 Feb 2025 00:23:03 GMT
server: uvicorn
transfer-encoding: chunked
```



The compressed file (named after the original file with a different extension) contains a number of images and metadata that include:

- a yaml file with all input parameters provided to the API, without the original image,
- the detected mapping used by the algorithm and
- the requested number of images with consecutive names.

Code and structure:

Concept:

The main concept behind this solution is the implementation of a simple API that provides access to the model and manages the inputs from the users; the idea is to keep the code responsible of the model and the API as encapsulated as possible.

- The main functionalities are divided into two groups: a single endpoint that can be used to request processing of one image at the time, it deals with the inputs from a web service perspective: it processes inputs, calls the model functionality and delivers results.

- The second set of functionalities come from the model perspective. The wrappers and functions that implement the algorithm (taken form the original code + some cleanups) are focused on the “model pipeline”. Such functionalities include from loading the model to generating images. This part of the code can be considered as a dependency for the API and it could be split and abstracted though configurations. I did not divide it into more packages but that would be the route I would take in a production scenario.

In terms of serialization of the model there is not much to add. The model and external networks is already provided in a serialized manner in a “pth” file (serialized PyTorch state dictionary). There is no training required, so the model (large model + external network) is loaded once for the entire application and then just used to generate outputs. There is no model training as part of this solution, just inference.

Code:

The code is divided in the following folders and files:

- Dockerfile: file defining the image to be built and container to be deployed.

- diffs: This directory contains a single difference file intended to apply changes to the original repository. The main idea is to implement compatibility with CPU. Sadly, it was not possible to make it work with GPU in my case.

- engineering: this folder contains the original enunciate of the assignment, from where a single file is copied to the image (mri_brain.jpg). The image was used just to assign the proper paths to store when building the image. No critical use, but required to build the image.

- new_code: This directory contains:

    - *awesomer_demo.py*: reduced version of the provided “awesomedemo.py” whre code not used for inference was removed, some imports and functionalities where removed. This file is a dependency for the API. Different models, files and algorithms could be injected by using similar separation of concern in these kinf od files. As part of the building image, this file is executed triggering the download of further files by used by the main repository.

    - *myfastapi*: folder containing the code that defines the API that exposes the model. This is the implementation of a template from where I did stripped many capabilities and serves as boiler plate code. The repository can be found in https://github.com/zhiwei2017/FastAPI-Cookiecutter, and it **does not belong** to me, I just used it. I intentionally did not use any Zeiss built code in here.

    - The added code to this template, written by me, correspond to the following files:
        - **myfastapi/myawesomedemo/app/schemas/base.py**:
            - schema: *GenerationRequest*, defines the required inputs, all configurations to the inputs can be set here (it uses pydantic).

- **myfastapi/myawesomedemo/app/api/ml.py**:
  - Functionalities required to process the image and provide access to the mode. It could me moves somewhere else and better packaged.
- **myfastapi/myawesomedemo/app/api/model.py**
  - Endpoint definition. It works as a controller that uses functionalities provided by other portions of the code. It does not perform the inference, it calls a function that does it.

Encountered challenges:

The first challenge comes from the origin of this code which can not to be considered by any mean “production ready”; the code tends to be chaotic and repetitive. Its usage is in the context of research and it should not be directly used for deploying models to production. It is possible to use and it is possible to understand what is does. However, I would not suggest its usage in productive systems in its current state. Nevertheless, the repository contains all elements required to understand the process, algorithm and even demos from how to train the external networks to usage of the model within a web app. It is a very nice source of knowledge, fun at the same time frustrations.

The second challenge is related to hardware, I do not have access to a CUDA compatible graphics card, so I ended defaulting to CPU processing, which is much slower. Even tough the main library behind the code is PyTorch, which is compatible with pure CPU, the repository does not make its adoption easy. There are hard coded values that need to be changed in different places, conflicting at some times. That is the main reason behind the existence of the *diff* folder in the solution, which makes an aggressive modification of the original code in its last state. Along the same lines, for some reason the usage of the non ROCm version of PyTorch would en up in core dump errors, which are solved by installing the version compatible with ROCm. I did not find the reason, I just accepted it. That is the reaosn behind the pip installation of torch related packages inside the Dockerfile.
The solution ended being quite simple in its conception and coding, but took a considerable amount of time until was able to make it run, I do have access to an ROCm compatible card (AMD), trying to make it work there ended up being fruitless and I did not want to spent a longer time just adjusting drivers and OS dependencies; at the end the best solution was to abuse the repository and hard code the CPU usage as part of the solution…. And document it in this file.

References:

[1] Lvmin Zhang, Anyi Rao, and Maneesh Agrawala. Adding conditional control to text-to-image
diffusion models, 2023.