Zora Watters & Jake Andrus

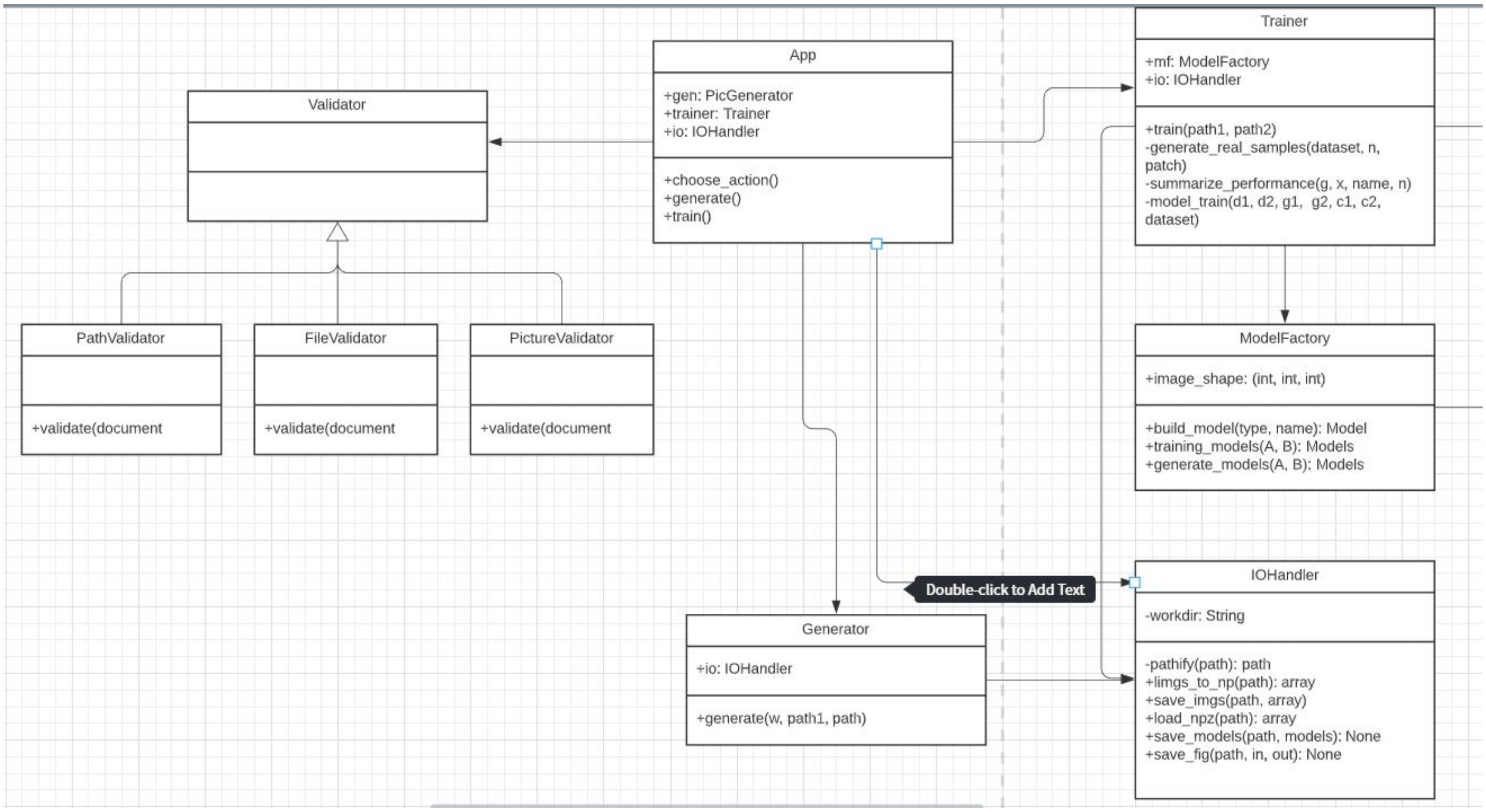
ImpressiGAN

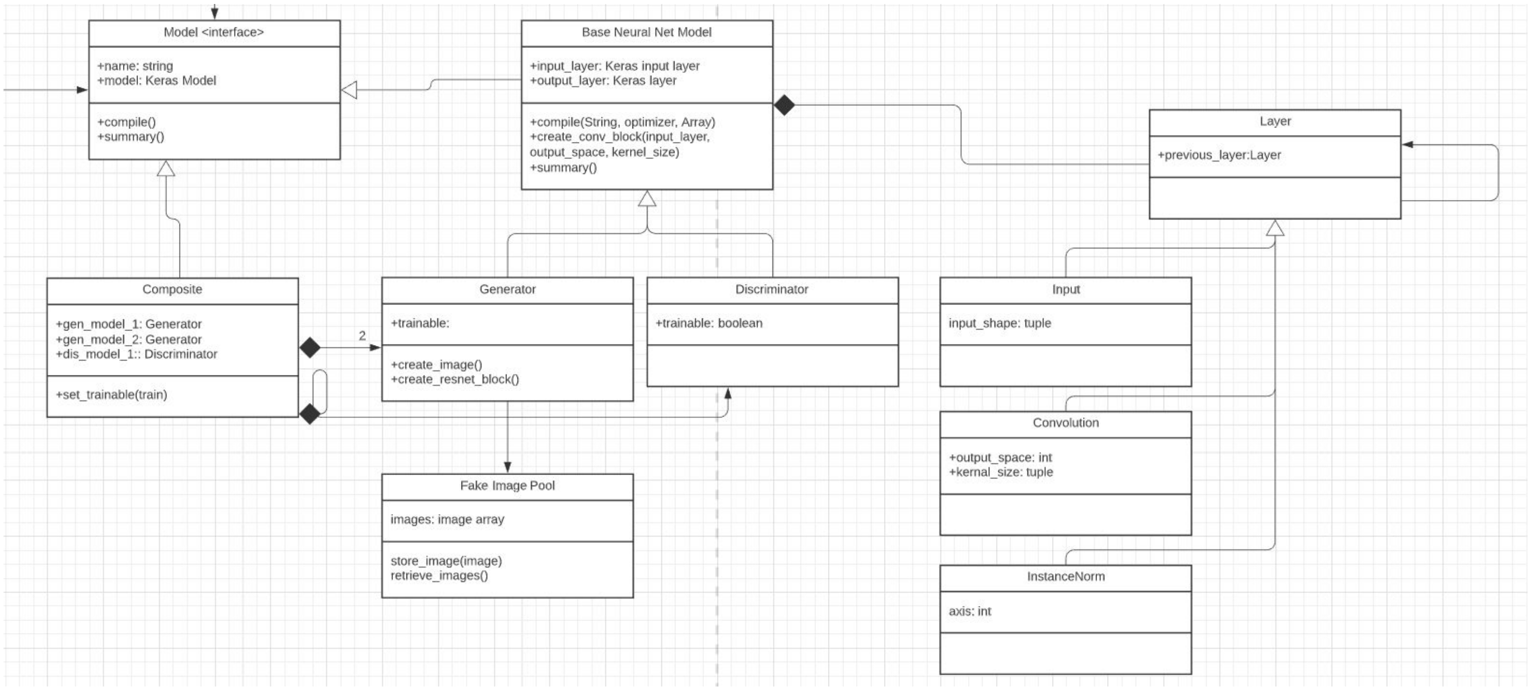
Final Report

Final State of System

Our system is a command-line application which implements a process that trains a neural model capable of generating impressionist art from random noise as well as from a starter image by utilizing a GAN architecture. The train feature takes in two paths to two folders and checks if the folders contain at least 1000 images each; one folder is the actual photos to convert, and the other is the artwork to stylize those conversions. Once the folders are checked, our CLI facade starts the training process. Another feature we have is to convert an image of the user's choosing into an impressionist painting. To do this, the user enters a file path to the image, as well as a target path to the folder they want the converted image to live, and the facade sends the request to our generator class. We didn't end up implementing a GUI, for mostly time constraints but also to keep the complexity of our system to a minimum. We also didn't implement tuneable hyperparameters due to the time it took to train models.

Final Class Diagram

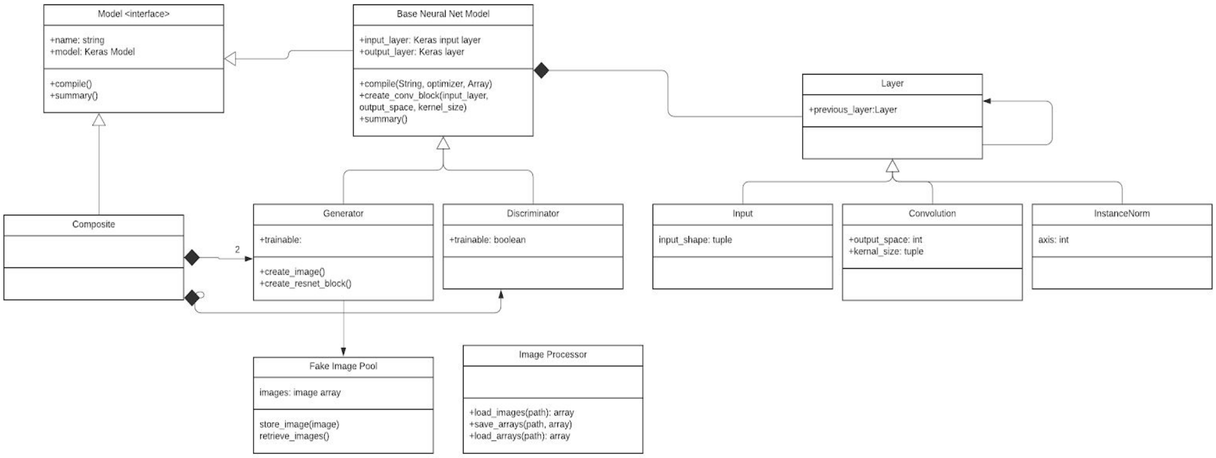




(Model is connected to model factory and trainer)

Comparison Statement

Diagram from project 4:



A key addition we made was clearly our UI, which functioned as the client to our entire system. It contained PicGenerator and Trainer abstractions for completing complex tasks. We also added a factory for handling file system processes like weights, image access, image creation, and path creation, as well as an IO class for interactions with the filesystem.

Third-Party Code

The CLI was implemented using a library, PyInquirer, that handles command line input in a clean way and stylizes your questions through types and style statements, which is how we were able to show multiple types of inputs like checkboxes and write-ins. We found out about this library and pulled inspiration from this article:
https://codeburst.io/building-beautiful-command-line-interfaces-with-python-26c7e1bb54df
And delved further into how to use user inputs within functions with this code example from the PyInquirer github: https://github.com/CITGuru/PyInquirer/blob/master/examples/hierarchical.py
However, we made a class for the apps, which is different than any example we came across, and used them in our own functions. Additionally the validator classes were heavily modified, using only the reset code from the CodeBurst article, which is accessed when the user gives an incorrect input.

This project is based around the academic paper titled *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks* by Jun-Yan Zhu, Taesung Park, Phillip Isola, Alexei A. Efros (https://arxiv.org/abs/1703.10593). It describes the architecture of the CycleGAN to convert images from one domain to another. *How to Develop a CycleGAN for Image-to-Image Translation with Keras*
by Jason Brownlee also provided some pieces of code involved with the implementation of the CycleGAN (https://machinelearningmastery.com/cyclegan-tutorial-with-keras/).

Overall

In doing this project, we really seemed to have trouble thinking about the structure first and then implementing code; of course the base items in our system were there, but the way they were going to fit together still ended up being a challenge we faced.

Another design element we experienced was deployment level design decisions, as we didn’t end up picking the type of UI we were going to use until after we were done with the backend code.

Typically, many machine learning papers and applications are done in a procedural manner in the form of a single python script or a jupyter notebook, which is great for research and exploration purposes but not for application purposes. It was a bit of a challenge adapting the typical procedures into an object-oriented design, but it ended up translating reasonably well.

Additionally, training procedures are extremely computation intensive, our models trained for about 5 hours and they could have been trained for much longer in order to achieve better results. This made testing difficult, and model optimizations slow.