IMAGE GENERATION

James Mann

CONTENTS

A. ANALYSIS

DEFINING THE PROBLEM

My project is an image generation utility, accessible through the web. Image generation will be handled by a conditioned Denoising Diffusion Probabilistic Model (DDPM), chosen for its flexibility and tendency toward fidelity. The user will not have direct access to the model, generation will be abstracted away into a platform offering some controlled control over 'hyperparameters' that guide the generation process. Chief among these hyperparameters will be the prompt, determining the 'what' of the generated image. More accurately, the prompt controls for what of the model's latent information is most impressed itself upon the image.

Once an image is generated there will be a decision point for the user as they choose between discarding downloading and saving the image. Prior to this decision the user will be presented with a preview of the generated image, within the webpage. Should the user elect to save the image, they will be required for a caption which, along with the image, are saved into a catalogue. This catalogue can also be accessed from within the website, through a page that will contain a grid of all the images and their captions. Images in the catalogue can be selected for a couple functions: renaming, downloading, and deletion. Each function will have its own routine. Images within the catalogue will be able to be searched and indexed. The most powerful of these indexes will be searching by caption, returning images with captions semantically similar to your search term even if not a direct match.

SOLVABLE BY A COMPUTATIONAL APPROACH

THINKING ABSTRACTLY AND VISUALISATION

Abstraction is the process of forming representations of reality that omits everything not directly related to what you're interested in. Apps frequently use abstraction to improve the user experience by hiding anything complex and only including important features.

Instances of abstraction in my image generator:

- The image generation function itself will be totally abstracted from the user. They do not need to understand or particularly interface with the DDPM to utilise the model which is an abstraction.
- When programming the DDPM I will be making use of libraries that abstract some of the lowest level maths, this speeds up my own writing and computing overhead as more time and work has been spent on optimising this code, in addition it allows me to interact with specialised hardware, like a GPU, to speed up training. This is abstraction because it is wrapping up a lot of complexity into a more manageable form.
- Hyperparameter setting will also be an abstraction, if the user chooses to generate something with a certain label, there is no way to pass that label as is to the model, instead it needs to go through preprocessing and transformation to turn it into something the model can actually use.

THINKING AHEAD

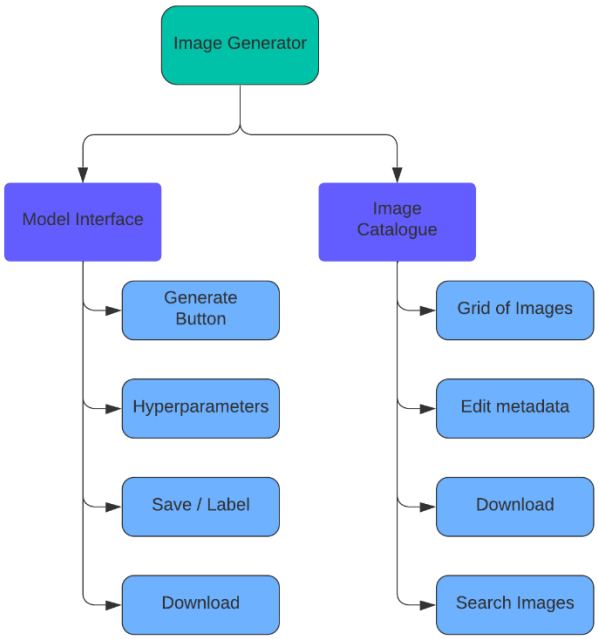
Thinking ahead is the process of identifying inputs and outputs to a solution before you have tackled the problem, you can then think of potential problems and then you will have time to think of how that problem could be solved.
These can then inform tests which serve as some predefined criteria for success.

Some of these criteria are:

- The image generation needs to be unbreakable, at its simplest it should just be a button that requires no understanding or thought to use, complexity is brought into this when considering hyperparameters, I will have to ensure that no combination of hyperparameters can lead to a breakage in the program itself.
- The quality of generated images is ultimately the most important part of this project, it is difficult to define success here as it is, to some extent, subjective, but I think that it will not be too challenging to tell when the model is outputting 'good' images just based on an instinctive feeling of 'passing'.
- Accessing and changing the details for previously generated images is another important feature, without an easy way to access previously generated images the app will not be as effective as intended as there will be no way to see all the images generated in one place. A simple, clean user interface which is easy to use and good at showing the images in a clear way is necessary to the overall strength of the project.

THINKING PROCEDURALLY & DECOMPOSITION

To help break down my problem I am using a top down modular design



This breaks my app down into two parts, this only covers the parts accessible to the user.

- Users have essentially two parts to their interaction with the app, generating images and accessing the catalogue of what's been previously generated. These two exist mostly detached from the other so when programming I can adopt a parallel approach rather than sequential.
- The model interface branch contains an abstracted representation of what the model interface should do, that is provide a way for the user to utilise the model. This hides a lot of the complexity of that task but the aim of this diagram is to provide a list of features, not be concerned with their implementation. Other features it is important to include are: some way to have control over the models output (hyperparameters), functionality for transferring images to the catalogue, and an option to immediately download the generated image
- The image catalogue branch lists some of the most necessary items that it would be important to add. A grid of images to visualise what is stored in the catalogue, some way to search them, and some control over individual images. Again, this abstracts the complexity of the database and user interface that I will need to design and implement.
- An omission from the diagram that is nonetheless pivotal to the project is the development of the model, however, as this exists totally outside of the user experience, I decided not to include it.

THINKING LOGICALLY

Thinking logically involves identifying the points in a solution where a decision has to be taken, determining the logical conditions that affect the outcome of that decision and how decisions affect the flow through a program.

My app will have a few decision points but much of it will be strictly sequential, when changing hyperparameters the program will have to be capable implementing the corresponding effects on the generation process, increasing the number of steps, for example, is one way that the user meets a decision point.

Because I am planning to use a website for my user interface there will be 'decision points' involved in that, with the user's navigation being, to some extent, one long series of decision points.

A large part of the DDPM's sampling process is to iteratively take 'noise' away from an image, this looping continues until the image is totally denoised.

THINKING CONCURRENTLY

Thinking concurrently is the process of implementing parts of a solution or program concurrently or side by side, e.g., by determining what parts of a problem can be solved at the same time.

My app can be coded concurrently because there is no serial order to most of the parts, one part is not relying on the completion of another. For example, I can implement a grid of images in html without needing the model to be working, similarly, I can generate images without needing anywhere to put them or I could work on the database for the images without connecting it to the website. This shows that there is a lot of opportunity to profit from concurrent development before combining them at a later stage.

Another way I will use concurrent programming is the instantiation of the website user interface, to be able to handle multiple users. All modern user interfaces behave like this, with the User Interface (UI) layer being on its own thread, and the business logic below happening on several other threads. There are several methodologies that already exist for this, which will be implemented in the project, such as the Model View Controller (MVC) and Model View View-Model (MVVM).

STAKEHOLDERS

STAKEHOLDERS

Stakeholders are those who hold a stake, financial emotional or otherwise. In my case, a financial stakeholder is unlikely, instead my stakeholder will be a 'client' who has an interest in the apps making. Image generation has great broad appeal with especial relevance to creatives. Conscious of this, I intend to make a simple and friendly app, maximising accessibility. Plain and effective, delivering on specific criteria and requirements will be the guiding principle.

My individual stakeholder is Charlie Pike, a 17-year-old student with an interest in computer science art and animals. He balances schoolwork with working on the farm and wants an enjoyable, creative

pastime. His art interest is stunted by a natural deficiency in the faculties on which art depends. For this reason, he leans toward computational work arounds, like image generation. He has, therefore, desired me to develop such an application for him. As a computer science student, he will be especially sensitive to issues that may arise during development and the importance of realistic expectations.

Charlie will use the app to generate images and explore how different prompts can impact what is generated. In addition, being someone interested in computer science and artificial intelligence, he will also use the app and the model to improve his own understanding of the diffusion process.

TARGET AUDIENCE

Image generation is a pretty uniquely flexible utility. There is no certain way to interact with it. A large part of working with text conditioned models is seeing how the model interprets language and probing this mapping to create better and better images. This creative freedom, expressed in domains as unbounded as language and image means there is no monolithic target audience. In the same way aesthetic preference varies from person to person, value to be found in the model varies from person to person.

The target audience for this app don't exist in one demographic. It could be people who are artistic, interested in the visual side of things, people interested in language and the systems interpretation of it, or people interested in the technical details of the technology. People from varied backgrounds with a common creativity and excitement for what is new.

Catering for such a broad demographic is difficult; simplicity, minimalism, and avoiding the introduction of complexity, at least outwardly is, therefore, imperative. Choosing a webapp for the app's interface is best aligned with this vision. Anyone can use a website. Instant access, no pause for download or set up, again maintains the ease-of-access I am intending. When it comes to designing the actual website, I will need to find the balance between simplicity and usability, too little and it is gimmicky, too much and it is torturous to engage with. Finding a way to not preclude access to the fine-grained controls, without introducing needless obstacle to the uninvested user will be a real concern and point of focus for development.

INTERVIEW

An interview is a structured conversation where one participant asks questions, and the other provides answers. I intend to ask my client the following questions to help with the design and creation of the Diving Calculator App.

QUESTIONS

Below are a series of questions I plan to ask my client along with reasons for each question. I hope these will help me determine what the App should accomplish and help with the design ideas.

Question	Reason
What is your name?	To determine the interviewees full name

What is your profession?	To see if the profession matches the need for the creation of the app
Why do you want this app creating?	To determine the need of the client
What apps/resources do you currently use to calculate the dives difficulty?	To ascertain what the client already uses to see if I can bring in any of what is already there to the app
Why do you want something new to do the same job?	To further get the need for the app
Who would the intended audience be for the app?	To ascertain the possible user base indicated by the client
What features would you want the app to have?	As a person who is interested in the creation of an app, see what they require it to do
What devices would you want the app to work on?	To get a list of possible devices requested by the client, they will have a better idea as to what they intend to use it on
Would you want the app to have a login?	This could determine how information is saved, stored, and accessed
How do you want the app to display the information?	This will help in determining the user interface features that will have to be designed and coded into the app
Would you want any export features?	To see if there is any results/data that the user may wish to output
How would you like the app to be installed?	To determine a mode of deployment and delivery of the created app
Do you want any colours/themes in the app?	This will help determine the look and feel of the app and a colour scheme from the offset
How much should the app be sold for?	To determine how much the client thinks the app should sell for

CONDUCTED INTERVIEW

What is your name?

Charlie Pike

What is your profession?

“I'm currently a student, focusing on digital media, computer graphics, and interactive design. I'm also deeply involved in personal projects related to digital art and computing.”

Why do you want this app creating?

"I'm excited about the potential of AI in art. I believe this app could open up new creative possibilities, allowing artists like me to experiment with AI-driven image generation and explore new frontiers in digital art."

Why do you want something new to do the same job?

"While there are tools available for digital art, an AI-driven app offers a unique approach. It's not just about doing the same job; it's about pushing boundaries and discovering new artistic expressions that might not be possible with traditional tools."

Who would the intended audience be for the app?

"The app would appeal to a wide range of users, from digital artists and designers to students and tech enthusiasts. Anyone interested in the intersection of art and technology would find this app intriguing."

What features would you want the app to have?

"I'd like to see features like customisable hyperparameters for image generation, a user-friendly interface for both beginners and advanced users, and a gallery or catalogue to store and view generated images."

What devices would you want the app to work on?

"Ideally, it should be accessible on both desktop and mobile devices. Desktops offer more control for detailed work, while mobile access would allow for on-the-go creativity and sharing."

Would you want the app to have a login?

"Yes, a login feature would be great for personalizing the experience and saving preferences and artworks. However, privacy and data security should be a priority."

How do you want the app to display the information?

"The app should display information in a clean, minimalistic layout that emphasizes the artwork. Easy navigation and a visually appealing interface are key."

Would you want any export features?

"Definitely. The ability to export images in various formats and resolutions would be essential for artists who want to use these images in different mediums or share them online."

How would you like the app to be installed?

"A straightforward web-based application would be ideal for accessibility. If it's a downloadable app, the installation process should be simple and user-friendly."

Do you want any colours/themes in the app?

"Aesthetically, a dark mode option would be great for long work sessions. The ability to customize the theme would be a nice touch, allowing users to personalize their workspace."

How much should the app be sold for?

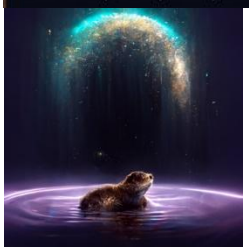
"It's hard to put a price on it without knowing all the features, but I believe in accessibility. Maybe a freemium model with basic features for free and advanced features available in a paid version. The price should be reasonable to attract students and emerging artists."

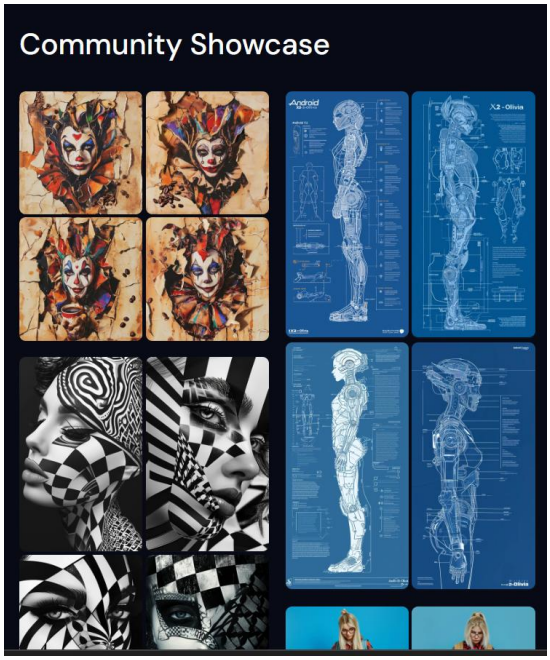
RESEARCH

EXISTING OR SIMILAR SOLUTIONS

MIDJOURNEY

Midjourney is the industry leading start up for image generation, at the time of writing they most recently released midjourney v6, their most powerful model yet. Midjourney's is a text-to-image model, meaning that it takes text as input and then attempts to generate an image that closely follows that text prompt. They are a paid service that operates on a third party (discord).





Above are three images I found in Midjourney's community showcase section, where users share artwork they have created. Attached to each image is its corresponding prompt, this shows how sensitive and flexible the midjourney model is as some of the prompts are complex and specific to such an extent even most humans would struggle to create a mental image of them. This showcase section serves as the inspiration for my own catalogue feature and is shown in the image to the left. Midjourney is an industry leader and it is clear why, the prompt fidelity and image richness combine to create extremely high quality art.

Advantages

The advantage of midjourney is the quality and flexibility of the model. Incorporating text in the generation process is a powerful addition that greatly enhances the user experience as it gives them more control. If possible, I would incorporate text prompting in my app for the user value it provides, however it would far more complex and challenging computationally than using labels or no classification input at all. Balancing expressivity with complexity will be an essential consideration.

Disadvantages

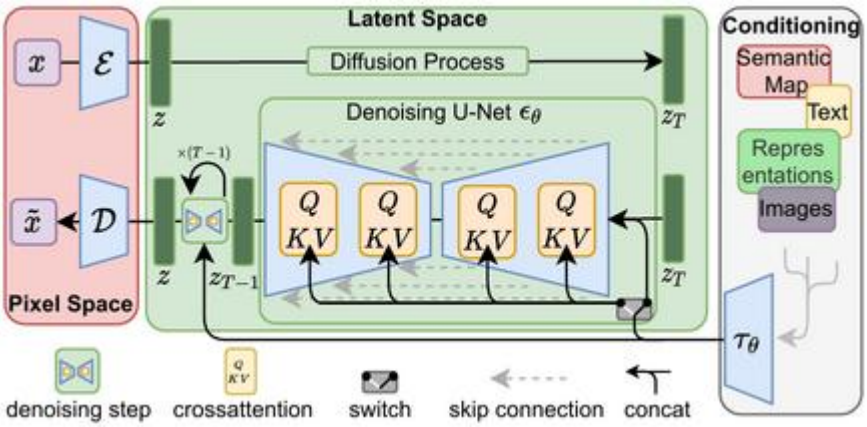
Midjourney functions through an intermediary third party, discord. This has a couple disadvantages, chiefly being the constraint on Midjourney's ability to optimise for its own purpose and functions. Discord is primarily a social chat site and midjourney is causing it to operate outside its remit. Furthermore, as opposed to an actual website, midjourney are limited to only the functionality discord chooses to implement. Another problem with using a third party is that you can only access midjourney through the third party, this means all users are forced to create a discord account. Finally, as a paid service, midjourney is exclusively available to users who are willing to pay what can be significant fees to use the service.

STABLE DIFFUSION

Stable Diffusion is an open-source model trained by Stability AI. The weights have been made publicly available so anyone is able to download and make use of them, this does require the knowledge to create and run the model. It is similar in functionality to midjourney, being text-to-image, however at a lower quality. There have been two versions of stable diffusion, with v2 building on and improving v1. Both operate with a similar architecture, ~860M parameter UNet, using some variation of Contrastive Language-Image Pre-Training (CLIP), to condition text information.

Depicted left is an image generated by stable diffusion, this was the output generated by the prompt “a photograph of an astronaut riding a horse”. As you can see it is a high quality, high expressivity, and high-fidelity model, that said it does still have limitations. All image generation models are prone to ‘absurdisms’ these are features in an image that, although prima facie plausible, upon later inspection make no sense. In this image the giveaway would be the shadows, look at them each individually and locate the light source that casts them, there are inconsistencies that act as a tell that the image artificial.

The diagram to the right is the architecture for the stable diffusion model. There are two significantly distinct parts, separated into red and green. The pixel space of the red refers to what you or I would consider meaningful images, the green latent space is of smaller resolution, decreasing the computational load. It is here the denoising takes place. In the latent space meaning does not necessarily need to be recognisable to humans. Information moves from the pixel space down into the latent space via a downsampling operation and moves without via a trainable upsampler.



Advantages

The advantages of stable diffusion are that it is open source and so very accessible, anyone with an internet connection can download the weights and use it for free. This lowers the barrier to entry for usage of image generation models, meaning more people can work with it. It is also a very powerful model in that it can generate high quality images from a text prompt, making it very versatile.

Disadvantages

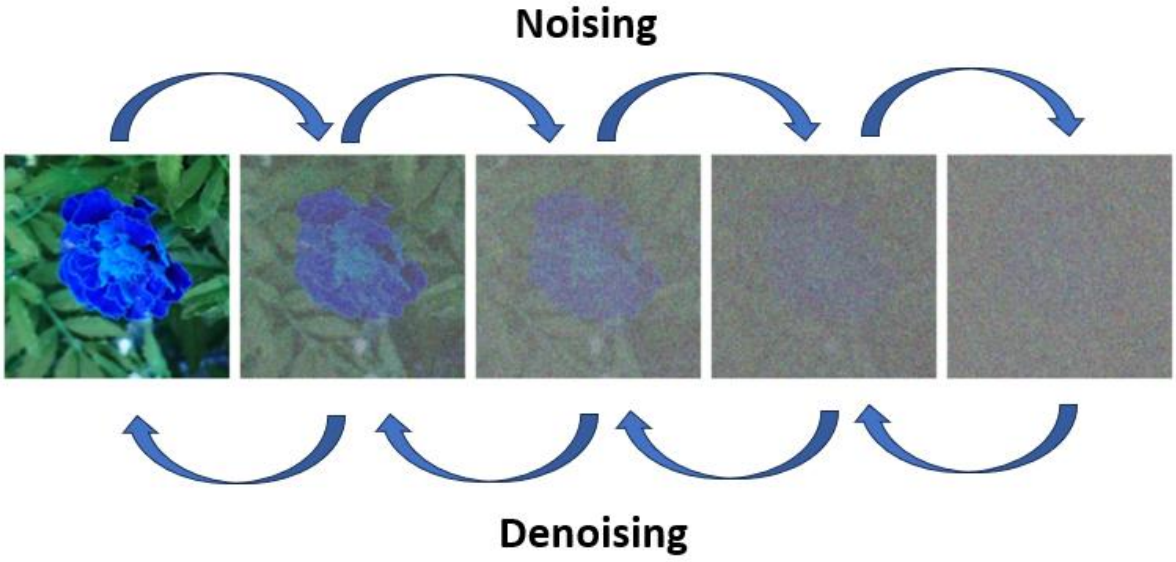
The disadvantages of stable diffusion are that even though it is open source, it is not necessarily easy for people to get started with it, without prior knowledge of how to set up systems like it. Another disadvantage is its computational weight, there is a trade-off between model ability and model size, but stable diffusion, with its 865M parameters requires a significant amount of memory. In addition, the nature of these models is that they require specialised hardware to run quickly, for example, it is recommended to use a GPU to get a reasonable time for generation.

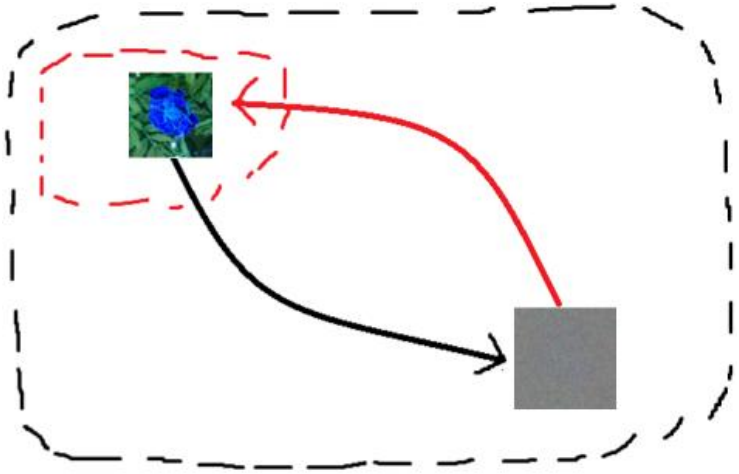
DENOISING DIFFUSION PROBABILISTIC MODELS

Based off this research I have decided the most suitable approach to my problem would be a denoising diffusion probabilistic model, the class of generators both Midjourney and SD would fall under. DDPMs are a modern approach to image generation that has shown impressive promise in the richness of the detail in the images it creates and the diversity in its sampling distribution. In this section I outline the key ideas behind DDPMs, covering what they do, how they do it, and some of the theory on maybe why they do it, and why they are such an improvement.

The guiding idea for DDPMs comes from two processes, noising, and its reverse, denoising. Any data contains, in some ratio, signal and noise. Signal refers to meaningful information, noise to meaningless. To noise, therefore, is to slide the scale against signal and towards noise, intuitively, this has the effect of destroying the meaning in data. To denoise on the other hand, is to go the other way, to reintroduce signal and meaning.

When the subject data is image there is an easy visualisation for both processes.





The noising / denoising processes can be more informatively visualised within an abstraction of pixel space. With the black box representing all possible pixel configurations, and the red the configurations we are interested in, those that resemble a flower. The black arrow is the noising process, taking points from the flower subset, and moving them outside of the set. The red arrow, then, is the denoising process, bringing a point from outside the subset, to within.

With this abstraction in mind, we can properly articulate the problem: to find an algorithm that accurately models the denoising process. DDPMs are an answer to that question.

DDPMs are neural networks trained to predict the noise added to an image. Once trained, they can then be sampled from to create novel images that could plausibly have come from the training set. Training and sampling algorithms are expressed succinctly by the authors below.

Algorithm 1 Training

1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \text{Uniform}(\{1, \dots, T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: Take gradient descent step on
$$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t) \right\|^2$$
6: **until** converged

Algorithm 2 Sampling

1: $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2: **for** $t = T, \dots, 1$ **do**
3: $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
4: $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t) \right) + \sigma_t \mathbf{z}$
5: **end for**
6: **return** $\mathbf{x}_0$

Line by line, what algorithm 1 means:

1. Iterate over the following steps.
2. Sample an image from the training set, this is something from the distribution you want the model to learn.
3. Sample a timestep from a uniform distribution over all the timesteps, this timestep dictates how noised the sampled image will be.
4. Sample gaussian noise with a zero mean and covariance matrix equal to the identity matrix, this serves as the source of the random noise.
5. Adjust the model to improve its guess for the noise in the noised sample it was shown. Backpropagate the square of the difference between the actual noise and the output of the model when given a faded version of the image, layered with a faded version of the noise, and the sampled timestep.
6. Repeat 2-5 until the model has converged at an acceptable denoising ability.

Line by line, what algorithm 2 means:

1. Sample from a gaussian distribution for pure noise.
2. Loop over all the timesteps, in reverse.
3. Sample more gaussian noise.
4. Subtract a scaled amount of the predicted noise from the model and add a correspondingly scaled version of the noise from 3.
5. Keep looping.
6. Return the denoised version of the noise from 1.

Both are very abstracted versions of the actual process but capture all the essential elements. They neglect complexities like the architecture of the model itself and the training process which both require a lot of thought and consideration.

ESSENTIAL FEATURES OF THE PROPOSED SOLUTION

INDEX PAGE

The index page of a website is the page that is returned on a user's immediate access. This is also often referred to as the 'home page'. Important considerations, therefore, include things like how immediately

aesthetically appealing the page is, as well as how usable and clean the interactivity is. Key features within the index page:

- The design itself; mostly blank being a white background.
- The nav bar; along the top of the page contains a centred title and links leading to other parts of the site.
- The generation functionality; an empty box in the middle of the blank background with a button below labelled ‘generate’. Between the sparseness of the rest of the page and the distinction of this bit, attention will be instantly drawn. On the button’s press the server samples from the model which returns an image the server can then display into the box.
- Options for generated images; once an image is generated there will be a couple options available to the user, either:
    - Delete the image.
    - Caption the image, to be saved into the catalogue.
    - Download the image.

CATALOGUE PAGE

The catalogue page will be where the user can control images stored on the server. These are previously generated images that the user chose to caption and save. Some key features of the catalogue page are:

- The grid; on opening the catalogue page the server will query the database for a random selection of x images, to be displayed. These images will be contained in their own cells within the grid.
- The images; each image will have its own functionality, on hovering over an image its caption will appear and should an image be selected there are a couple options for the user:
    - Delete the image.
    - Rename the image.
    - Download the image.
- Filtering; images in the catalogue can be filtered by date added newest to oldest, or vice versa. Another filtering option is to search by caption, this search will be smart so as to not need direct matches from search term to caption to return relevant results.

THE MODEL

The model is the most important part of this project. It would be difficult to create a good image generation app that has no ability to generate images. Fitting the model into this section however, is a challenge as there is only one feature to the model, its ability to generate images. Nevertheless, describing the images it should produce is somewhat value, therefore:

- Diverse; images should not look like one another. What this means is that even when two identical samplings are made of the model, the images should be easily distinguishable. To make this possible the model must learn a good representation for the form of an image, not the direct pixel values. This means that sampling is done from a larger distribution, which is what we really intend when we say diversity.
- Fidelity; when you tell the model to generate a flower you want it to generate a flower. Fidelity is how truly the model’s generation follows your prompt. High fidelity mean you access different parts of the model’s latent representation with a high degree of accuracy. The challenge of this is creating a latent representation good enough that there is stuff to be accessed. Typically, this would mean developing the model, this comes with its own problems.

- Speed; this is one of those aforementioned challenges, the model is a process of taking the signal in some data, and turning it into information on a related but different task. To do this the model contains parameters that define the transformation. More parameters means a better defined, and more complex, transform, the converse of this is that more parameters also means more computation. Finding the balance between these, therefore, is another consideration.

THE DATABASE

When a user elects to save an image, the database is how that image is saved; when a user then wants to retrieve that image, the database is required again. Managing all the images that have been saved in order to make them re-available to the user is an important part of the app.

When a user saves an image, the database receives a few things:

- The image itself; databases are not ideal to store the images themselves, instead the database will contain a reference to the image. This reference must be unique and reproducible, a good fit for these requirements would taking the hash of the image, saving the image with that same hash as the filename, and keeping that hash for the id in the database.
- Date saved; the date and time the image was added to the catalogue.
- Caption; the user-given caption, this will be of a fixed maximum size.

LIMITATIONS

I plan on identifying any possible limitations that I may encounter whilst creating my image generation app. I will justify and explain each.

Time

Time is a definite limitation. I will only have a set amount of time that I can spend on each stage. The final hand in will be in 9 months. I need to ensure that I devote a set amount of time each week to my project and have sufficient planning and testing in place before moving forward to my development. I know that my client will not always be able to answer questions in a timely manner, so need to build this into my planning. Also, while I do have some experience of building similar systems, this will be at a larger scale than anything I have done previously, in addition, the ideas behind diffusion are new to me so I need to ensure I develop a strong intuition for the process as a whole.

Knowledge

I must ensure that I learn how to build a DDPM. I know now that I will need to better my understanding of machine learning as most of the ideas I will be relying on come from this space. While I do know python, my knowledge of javascript is limited, I will need to learn how to use javascript to create an effective web application if I want my app to be usable for the user. The library pytorch will be the backbone of my development of the model so becoming comfortable and familiar with this is a must. This may be challenging as it has a tendency to be quite complex.

School Subjects

Another limitation is School. I will be busy with my subjects and have lots of other commitments that will take me away from my project. If I am careful with my planning and devote at least a few hours each week to my project this shouldn't affect the completion of the project on time.

School Computers

The model will inevitably be extremely computationally heavy, this means I will have to make considerations about the hardware I use. For developing the model I will be able to use a mixture of Google Colab and Kaggle, both platforms provide free GPU access, greatly speeding up my training times for the model. This will not be helpful when it comes to actually hosting the webapp as they do not support this, this means I will have to consider what I can host the server off. It cannot be a school computer as it has neither the necessary memory, nor the necessary libraries to even run the code. Instead, I will have to either host it in the cloud, or on my own devices. Ideally, it will run on my raspberry pi, however that will undoubtedly be a stretch for the amount of computation the pi is capable of.

REQUIREMENTS

HARDWARE

Item	**Justification**
Monitor	**Allows the user to see the website in order to interact with it, is a medium for displaying generated images.**
Keyboard	**Allows the user to enter information for the prompt.**
Mouse	**Allows the user to navigate the website, selecting buttons and commanding the server.**
Browser	**My app works through a website, a browser is therefore required to abstract the complexity of the web away into a simple interface for me to create and user to access.**
Networking ability	**Because I intend to run my app on a separate device (raspberry pi) it is necessary for devices to be able to connect over a network.**
Raspberry pi	**The raspberry pi runs the server, for this reason it must have some compute capacities. In my case, the device has 2gb of ram and 1.8Ghz**

For this section I will exhaustively outline what a user requires to use the solution:

- A browser.

Before detailing what is necessary to run and develop the solution:

- A raspberry pi; to run the server that hosts the website that makes up the solution.
- Pytorch; the library which all the machine learning will be ran through. It contains abstractions and facilitations that make development far quicker and easier. One of these is the autograd

functionality that can calculate gradients all through the model. This makes backpropagation doable.

- Kaggle; while sampling from the model is computationally intensive, that pales in comparison to training the model. To feasibly do this requires specialised hardware, specifically, a GPU, or some such hardware accelerator. Given their present pricing inaccessibility, free tools like Kaggle which provide access to GPUs are the only reason that this project can continue.
- Flask; a python library for website hosting, very lightweight and customisable. Can easily be ran within tight hardware constraints.
- Postgresql; the database I have chosen to use. Lightweight and fairly simple to use, good compatibility with python.
- Kaggle (again); training the model requires vast amounts of labelled data. From this data the model can build up its latent understanding of images. Kaggle also serves as a centralised location for such datasets.

SUCCESS CRITERIA

Below are the success criteria which determine what needs to be completed by the end of the project. These were got from research, questionnaires, and interviews with my client. Not all success criteria will be possible to accomplish by the end of the project, but it will be a good indicator to see what was accomplished.

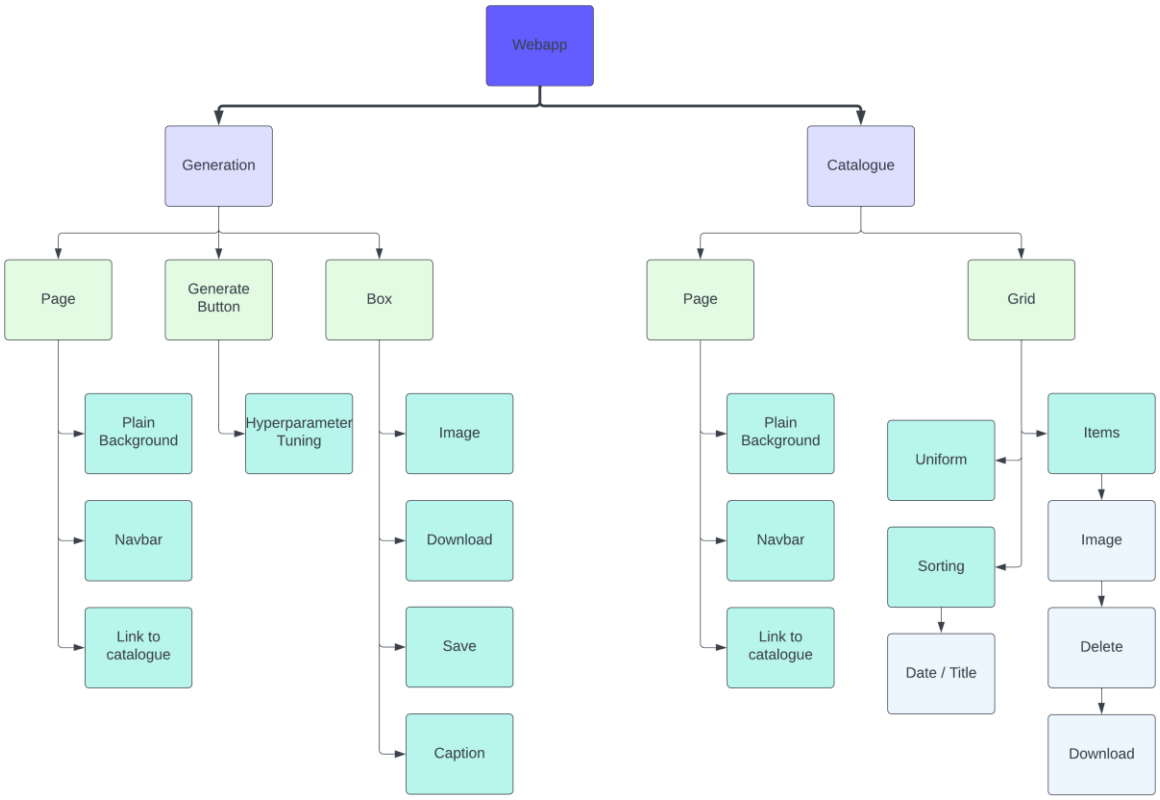
ID	Success Criteria	Reason
1	Works on browsers for desktop or mobile	Interview revealed that it is important for client to be able to access the app from many different devices. This is best achieved with a versatile website, rather than individual apps for each platform.
2	Website can be accessed from remote devices	The server should be able to be connected to from devices on the network. This is important so that different types of devices can use the app.
3	Index page of website contains a button to generate immediately	This is important both as it will be visually striking but also because it is forces a minimal design right from the outset. This appeals to the audience in the bottom end of technical proficiency.
4	Option menu to toggle the specific hyperparameters of the generation process	The flip side to having such a simple generate button catering to the least technical. Giving those who know, or want to learn, how to interact with the hyperparameters an ability to do so is in line with my research.
5	Generated image is displayed on the website	Visually striking but also compliments the user experience.
6	Generated image has an option for an instant, one click, download	Helpful for the user for things to be plain and accessible.
7	Generated image can be saved with a name to the database	Imperative to a working catalogue.
8	Navigation from the index page to the catalogue page	Traversing across the site should be easy.

9	Catalogue page immediately displays generated images in a grid like format	Aesthetically pleasing and most effective way to maximise amount of data on the page without becoming overwhelming.
10	Search bar at the top of the catalogue page can filter the images shown	Important functionality, the user should be able to search for specific images, but also images of a certain class.
11	Any catalogued image can be deleted	Generic CRUD.
12	Any catalogued image can be renamed	Generic CRUD.
13	Any catalogued image can be downloaded	Users should be able to download images to use elsewhere.
14	The generation process returns plausible images that match the prompt	This is vague as plausible means different things to different people, the doctrine I intend to follow is squint and guess.
15	The generation process runs in less than 30 seconds	The user may lose interest if it is any slower, preferably much faster than this.
16	The website interfaces are all aesthetically pleasing	Important to the user experience.

*All of the above contain implicit necessity that the function should not just exist, but also work. This means that for say criterion 3, passing is not just a case of the button existing, but that the button does actually trigger a sampling from the model.

B. DESIGN

SYSTEMS DIAGRAM AND SOLUTION STRUCTURE



Break down the systems diagram into each part, speak more about the backend and the real nitty gritty of implementation.

ALGORITHMS

Could do algorithms for:

- Generation Process
- Navigation process
- Catalogue process
- Saving process
-

USABILITY FEATURES AND STRUCTURE

While the diffusion process is complicated, I intend for the user experience to be simple. A user's interface with the system will be contained within two distinct modes, generation or the catalogue. The usability features I implement will be limited both by necessity and by a desire to retain a minimalist feel for the user. This means that every feature must be justified. Minimalism is also instructive in the aesthetic effect the user should come away with.

A website is effective against these criteria because of the flexibility it affords. HTML + CSS + JavaScript offers a fine-grained control that isn't available elsewhere. This means that I can make aesthetic choices now with little regard to actual practicability, as I can be confident that I can at least approximate my designs.

In this section I begin to give each planned feature countenance. To make the section more natural I will design and discuss features as I would expect them to be encountered by an actual user during actual usage.

THE HOME PAGE

This will be the first thing the user sees as they access the website, it should be clean and functional with an obvious layout that keeps it accessible to less experienced internet users. I will go over all of the elements it contains and the designs of each before integrating them into one final overall design for the home page.

1. **THE NAV BAR**

   The nav bar will be a band that runs along the top of the page and serves as a container for navigating around the site. In addition, it will include a title for the website as an aesthetic choice. The nav bar should be distinctive and memorable as well as easy to use. The nav bar contains a usability feature in the button that directs the user to the catalogue page. This should be an obvious thing to press.

   A potential design for the nav bar would be as below,

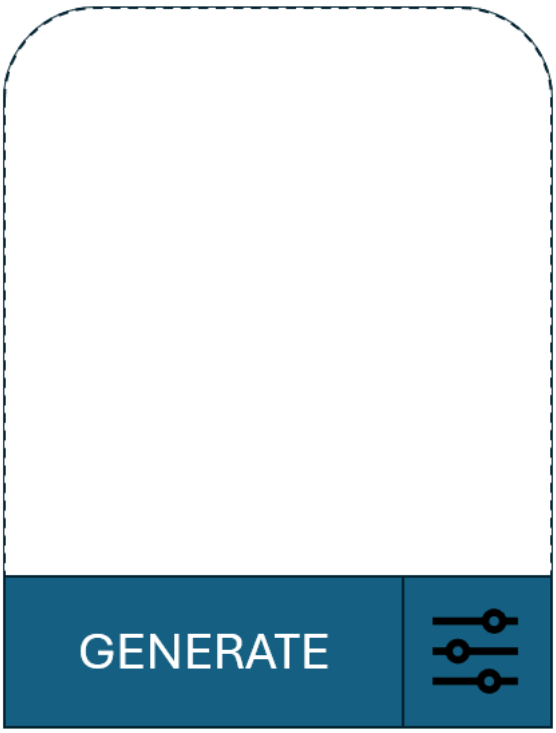
   This design is not final as colours, positionings etcetera can all be easily changed.

2. **THE BOX**

   The box will be the container for generated images and so a focus point for the user. Where the rest of the background is empty this will be an instant target for the user's attention. It will be located in the centre of the page. The box will be the container of all the functionality of the page and so will contain multiple subparts. Each subpart should be clear and obvious in its function, the subparts include:

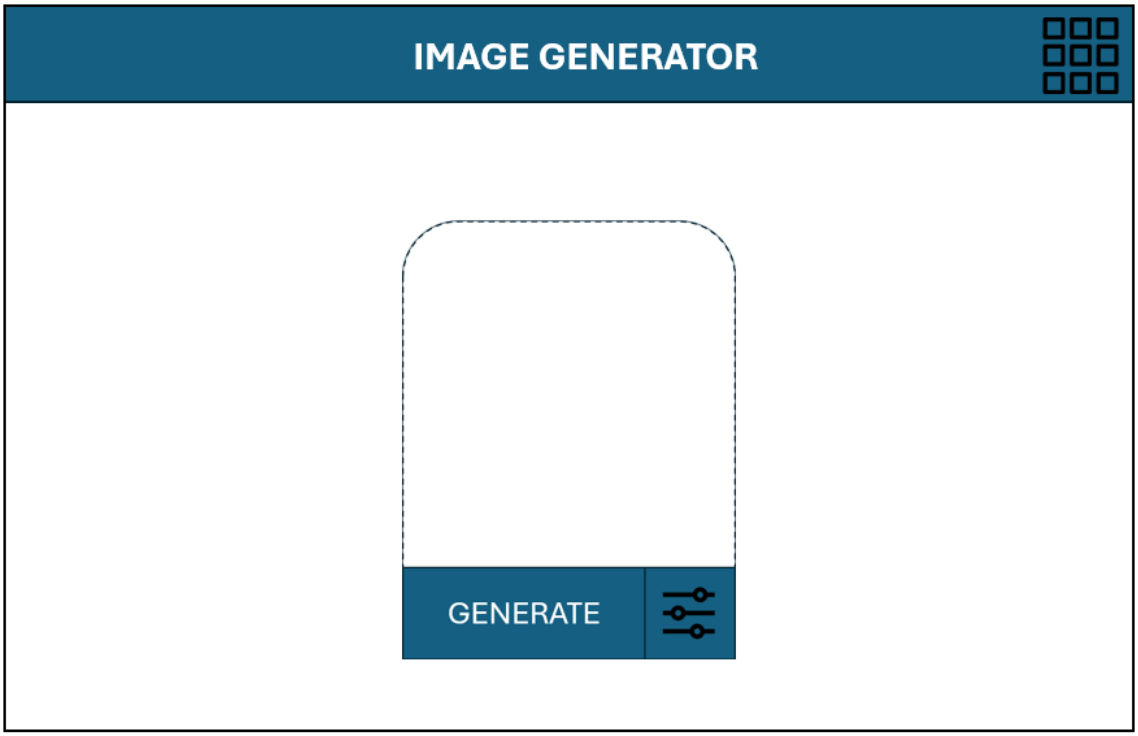
   a) The generate button, this should be big and immediately obvious as the thing to press to cause generation. As such I think a large colourful button displaying the text "GENERATE" should do amply.
   b) The image container, I want there to be consistency in layout both before and after generation as such before an image is generated the space it will occupy will be predefined in a hashed box that covers the majority of the "The Box". When generation is finished the image can then be displayed within it.
   c) Hyperparameter adjustment, while I don't want to confuse the less technical user, I also want to avoid handicapping the more proficient user by limiting the ways that they can interact with the model. As such I want to hide the hyperparameter settings from the user behind a button. On click, this button will expand to give a selection of alterations that can be made.

   My idea for the design of the box looks like this.



Once an image has been generated the hyperparameters option will be replaced with an option to save the generated image. On click this will, again, open a pop up menu containing some more inputs regarding the save such as caption. This save menu will contain a final ‘commit’ button that will send the image to the database.

With the elements of the page defined, I can now offer an example of what I would expect the page to look like.



This design achieves my goals, it is simple without compromising on function. While I don't expect to deviate from this design particularly, I am also not going to allow myself to become bound captive to arbitrary design choices. Such a fatalistic view is contrary to my intention for development, making rapid changes based on evidenced decisions that pertain to user experience and improved efficacy of the solution.

THE CATALOGUE

Once a user has selected to save an image it is committed to the database, the catalogue page will allow users to access their saved images. Because I have not planned for it there will be no discrimination in the database, and thus the catalogue page, in image ownership, generated images will belong to the website, and everyone will have access to everything. The catalogue page will, in the first instance, request n images, randomly sampled from the database. This ensures that on every refresh there will be a different array of images displayed, thus keeping it fresh for the user.

For the actual design of the catalogue page there will be a couple of features. This will be a more dense page than the generation page as there is more content to display, nevertheless I intend to stick with an efficient minimalism that maintains visual cleanliness. Before I explain the designs for the individual components of the page, I think it prudent to first reiterate the features of the page. For consistency across the site, I will retain the nav bar banner running across the top of the page, with the slight alteration of the grid icon that would've led to the catalogue page (which the user is now on), replaced with a different icon that leads back to the generation page. The next thing to note is that below the nav bar but above the image content will be some usability features, these will all offer some control over the images shown. This may include a search bar to return images with a caption similar to an entered search term, it will also include some sorting functionality, either to filter by date or by alphabetical order. These features are important when considering the site's longevity. If the site is frequently used and the database starts to become large enough that it is infeasible to simply traverse the images until a desired

one is found, then an ability to narrow down or accurately identify specific images becomes essential. This will ensure that the user isn't cobbled by the amount of generations they store and that the site is flexible and scalable.

VARIABLES / DATA STRUCTURES / CLASSES

TEST DATA

POST DEVELOPMENT

C. DEVELOPING THE CODED SOLUTION

The development of my app will follow a two-pronged, iterative approach, wherein once each iteration of a module has been produced, I will go back to my client Charlie Pike to discuss the project's direction. Feedback from this will guide development and ensure the app remains appealing to the target audience. His job as a stakeholder will be to comment on the current state of the project and suggest improvements for subsequent iterations.

This project is well suited for the agile SDLC paradigm, as it is highly compartmental and parallelisable. This is because most parts of the project are not interdependent. The most eminently obvious example of this is the model and the app, the quality of the model and its development does not depend, at all, on the status of the app, and vice-versa. For this reason, this section is broken into subsections on the development of each which will touch on each other only very lightly. This is only one example of how the project can be decomposed and dealt with in a 'divide and conquer' style. The agile methodology, with its iterative improvements and agnosticism about the order of development, is therefore particularly suitable for my case.

THE MODEL

This section covers the construction of the ddpm. As is often the case in machine learning, I will follow an iterative approach, attempting to find the best combination of architecture and hyperparameters for the task. As to the task itself, I'll explain more clearly the technical requirements as I now understand them, following more research.

Any picture is a sample, drawn from the distribution of all possible pictures. This distribution exists as a small subset of the distribution of all possible images. This is true, intuitively, when you consider that you can only take pictures of what exists – that is, what is there to be pictured - whereas an RGB image can be any set of values.

Image generation, then, is a misnomer. image generation is easy, provided you remain ambivalent about the meaning of the images. A more accurate rewording of the task would be to learn an internal representation of the picture distribution, which can be drawn from to produce images indistinguishable from that same distribution.

This way of conceptualising the task is laborious but also the flash of inspiration that guided diffusion models. Where previous approaches had aimed to learn the distribution directly, diffusion model's key insight was that rather than learning the features of an image that make it a picture, to instead learn the features of an image that make it a *not* picture.

The way that this is went about, in diffusion models, is to take an image from within the distribution, gradually transform it over a series of small steps, moving it out of the distribution, and then learn the reverse of that process.

BUILDING THE DIFFUSION CLASS

All code in this section will be from Kaggle, for the ease it brings when dealing with large datasets and the free access it gives to GPUs. The reason for the latter benefit will be apparent when it comes to training.

The first step is to define the noising process. Mathematically, it is given as below.

$$q(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I})$$

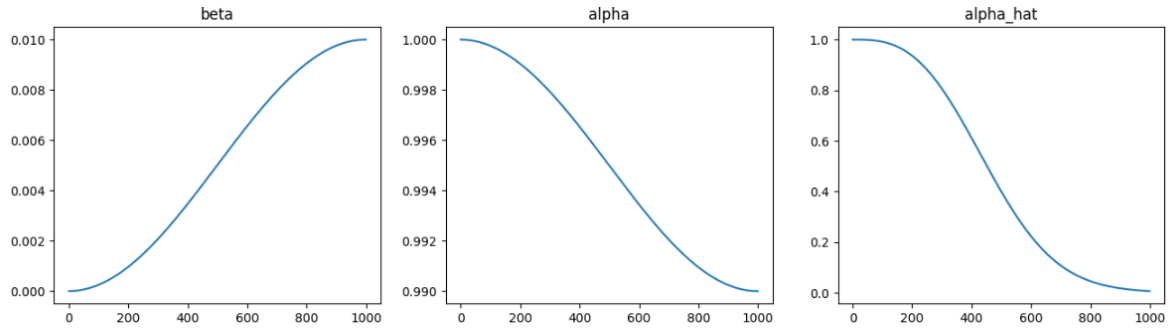
What this really means is that to get x_t, given x_0, you sample from a normal distribution with a mean set to a faded version of x_0, and a covariance set to a scaled version of the identity matrix. The fade and the scale are directly proportional, increasing with t. A greater t yields a more noised version of a more faded original image. Fading and scaling are defined by some predetermined values $\bar{\alpha}$. Putting this into code, in a 'Diffusion' class, looks like.

```
class Diffusion:
    def __init__(self, noise_steps=1000, beta_start=1e-4, beta_end=0.02):
        self.noise_steps = noise_steps
        self.beta_start = beta_start
        self.beta_end = beta_end

        self.beta = self.prepare_noise_schedule()
        self.alpha = 1. - self.beta
        self.alpha_hat = torch.cumprod(self.alpha, dim=0)

    def prepare_noise_schedule(self):
        lin = self.beta_end * (1 - torch.cos(torch.linspace(0, 1, self.noise_steps+1)[1:]*torch.pi)) / 2
        return lin

    def noise_images(self, x, t):
        sqrt_alpha_hat = torch.sqrt(self.alpha_hat[t])[:, None, None, None]
        sqrt_one_minus_alpha_hat = torch.sqrt(1 - self.alpha_hat[t])[:, None, None, None]
        ε = torch.randn_like(x)
        return sqrt_alpha_hat * x + sqrt_one_minus_alpha_hat * ε, ε

    def sample_timesteps(self, n):
        return torch.randint(low=1, high=self.noise_steps, size=(n,))
```

- The '__init__' method defines some constants that are used in the noising process. The values 'noise_steps', 'beta_start', and 'beta_end' are all hyperparameters that can be tweaked. The values they take now are just borrowed from the original DDPM paper.
- The 'prepare_noise_schedule' function defines the noise schedule which controls how much noise is added at each step.
- The 'noise_images' function codes up the process I mentioned above, of transforming an image into its noisy correspondent. This returns the noisy image itself, but also the noise that was added, to be used in the training process.
- The 'sample_timesteps' function is a helper function that returns n timesteps within the interval, to be used in the 'noise_image' method.

These values can then be plotted. Notice that from my use of the cosine schedule my 'beta' and 'alpha' values take on a sinusoidal shape. The reason I chose a cosine schedule is that it has been shown to

yield better results, as the model spends more time in timesteps devoted to removing miniscule amounts of noise, meaning it can concentrate on learning detail.



Before I demonstrate the noising function, I need to decide what it is I will actually be noising. This choice will decide what it is I train the prototype models on and so warrants some especial consideration. Taking a dataset like actual image-text pairs would be more aligned of the end goal of this project, however due to the size and complexity of the data it would make iterative development more difficult. This is because of the computation / ability trade-off. An 256x256 image is 8x larger in each direction than a 32x32, but 64x larger overall. This added computational load is felt through the entire process. It is infeasible to begin with data of such size and contrary to any idea of iterated improvement. Instead, using a less complex, smaller dataset like fashion-mnist makes more sense.

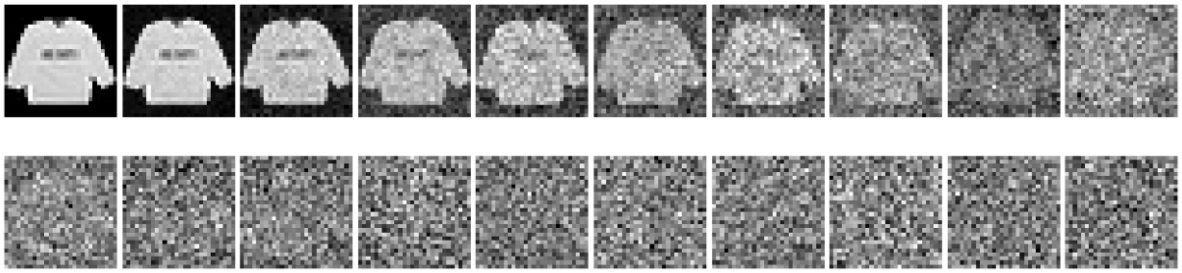
Fashion-mnist is a collection of 70,000 images split between ten classes. The images are grayscale and 28x28, meaning they are extremely light and easy to process. Faster processing means I can get in more iterations per unit time and achieve a better final solution. Once I have plateaued on progress with fashion- mnist, I will take the code, most of which is transferrable, and move onto other, more complex, datasets.

Fashion-mnist is stored in a csv which must be parsed to turn the data into something usable. This can be done with the library ‘pandas’ followed by some data processing.

```
df = pd.read_csv('/kaggle/input/fashionmnist/fashion-mnist_train.csv').to_numpy()

labels = torch.tensor(df[:,0])
images = torch.tensor(df[:,1:].reshape(60000, 1, 28, 28))

images = 2*((images - images.min()) / (images.max() - images.min())) - 1
```

In the first line of the above snippet I use pandas ‘read_csv’ function to extract the data stored in the csv file, I convert this to a more workable array using the method ‘to_numpy’. As it is the array contains both the image data and the labels, these are separated in the next two lines. Data is stored tabularly with each row containing 785 entries, 784 for the pixel values in the 28x28 image, and 1 for the integer corresponding to a label. Slicing the array into the first column and everything else solves this. Wrapping this splits with ‘torch.tensor’ turns them into pytorch tensors which is necessary for any later processing with the rest of the pytorch library. The ‘reshape’ operation turns the data into the 3d structure used for images. The final line normalises the values, which were [0, 255], to [-1, 1]. This done in machine learning for numerical stability and improvements in model fitting.

```python
diffusion = Diffusion()

fig, axes = plt.subplots(nrows=2, ncols=10, figsize=(16, 9))

noiseds = []
for i in range(0, 1000, 50):
    noised, _ = diffusion.noise_images(images[0], torch.tensor([i]))
    noiseds += [noised]

fig.subplots_adjust(hspace=-0.7, wspace=0.05)

for i, img in enumerate(noiseds):
  axes[i // 10, i % 10].imshow(normalise(img[0]).cpu().detach().numpy().swapaxes(0, 2).swapaxes(0, 1), cmap='gray')
  axes[i // 10, i % 10].axis('off')
```



The above snippet takes shows an image at the different noising steps. Using the ‘noise_images’ function from the diffusion class it loops over the timesteps in increments of 50, noising the image appropriately, and displays it in the grid. Altering the ‘beta_end’ attribute will control how quicky signal is destroyed over the noise schedule.

BUILDING THE FIRST MODEL

In a DDPM you train the model to predict the noise that has been added to an image. This is conventionally done with a UNet type architecture.

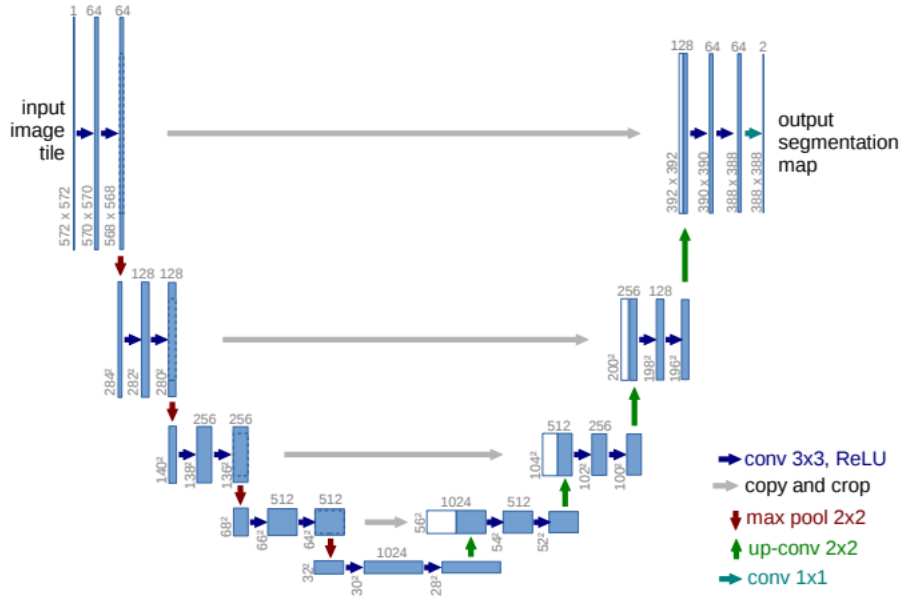


Fig. 1. U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

A UNet brings an image through a series of blocks that decrease its spatial size but preserve and operate on its signal. This repeats down to a minimum size called the bottleneck where the information passthrough is most restricted. With such limited volume the model is forced to compress the data down to its most semantically expressive form. The downsampling in a UNet is allegorical to lossy compression of data. What follows is a mirror inverse upsampling where the model restores the data back to its original dimensions. This upsampling is aided by a series of residual connections, indicated by the grey arrows, that act as skips for data flowthrough in the network. Residual connections take the activation of each downsampling step and copy it across to its corresponding upsampling step. This residual is concatenated with the previous upsample to form the input for each upsample block.

To create the model, I will be using the library pytorch for flexibility and its implementation of automatic differentiation which abstracts backpropagation into a single line. Furthermore, it has strong support for hardware acceleration. While coding, I intend to make considerations about future usability, writing in a modular self-contained manner. To do this I will be using object-oriented programming.

This first iteration of my implementation of a UNet is shown and explained below.

```python
class double_conv(nn.Module):
    def __init__(self, inc, outc):
        super().__init__()

        self.c1 = nn.Conv2d(inc, outc, 3, 1, 1)
        self.c2 = nn.Conv2d(outc, outc, 3, 1, 1)

    def forward(self, x):
        h = self.c1(x)
        h = self.c2(h)

        return h
```

- Define 'double_conv' class, inherits from 'nn.Module'
- Initialised with two arguments, input and output channels.
- Initialise the pytorch superclass.
- Define two convolutional layers to transform the data.
- Define the forward function to take an input image 'x' and pass it through the two transformations.
- Return the transformed 'h'

```python
class downconv(nn.Module):
    def __init__(self, inc, outc):
        super().__init__()
        self.dconv = double_conv(inc, outc)
        self.down = nn.MaxPool2d(2)

    def forward(self, x):
        x = self.dconv(x)
        h = self.down(x)

        return h, x
```

- Define 'downconv' class, inherits from 'nn.Module'.
- Initialised with two arguments, input and output channels.
- Initialise the pytorch superclass.
- Define a 'double_conv' transformation.
- Define a maxpool downsampling operation.
- Define the forward function to take an input image 'x' and pass it through the two transformations.
- Return the transformed 'h' as well as 'x', the result of the double_conv with no downsampling applied.

```python
class bottleneck(nn.Module):
    def __init__(self, inc, outc):
        super().__init__()
        self.dconv = double_conv(inc, outc)
        self.up = nn.ConvTranspose2d(outc, inc, 2, 2, 0, 1)

    def forward(self, x):
        h = self.dconv(x)
        h = self.up(h)

        return h
```

- Define 'bottleneck' class, inherits from 'nn.Module'
- Initialised with two arguments, input and output channels.
- Initialise the pytorch superclass.
- Define a 'double_conv' transformation.
- Define a transposed convolutional layer for upsampling.
- Define a maxpool downsampling operation.
- Define the forward function to take an input image 'x' and pass it through the two transformations.
- Return the transformed 'h'

```python
class upconv(nn.Module):
    def __init__(self, inc, outc, up=True):
        super().__init__()
        self.dconv = double_conv(2*inc, inc)
        self.up = nn.ConvTranspose2d(inc, outc, 2, 2) if up else nn.Identity()

    def forward(self, x, res_x):
        h = self.dconv(torch.cat([x, res_x], axis=1))
        h = self.up(h)

        return h
```

Very similar to the 'downconv' but the maxpool operation that decreases spatial size is substituted for a transposed convolutional layer that operates as its ~inverse, restoring spatial size. The 'up' argument controls whether the upsampling operation is applied. This is done by defining 'self.up' as either 'ConvTranspose' or 'Identity', which returns its input.
In the forward method two arguments are taken, x and res_x. Res_x refers to the residual connection denoted in the diagram by the grey arrows. The inputs are concatenated along the channel axis before being passed to the double_conv.

```python
class unet(nn.Module):
    def __init__(self, d=16):
        super().__init__()

        self.downs = [
            downconv(1, d),
            downconv(d, 2*d),
            downconv(2*d, 4*d)
        ]
        self.bottleneck = bottleneck(4*d, 8*d)
        self.ups = [
            upconv(4*d, 2*d),
            upconv(2*d, d),
            upconv(d, d, up=False)
        ]
        self.outconv = nn.Conv2d(d, 1, 1)

    def forward(self, x):
        residuals = []

        for down in self.downs:
            x, res = down(x)
            residuals += [res]

        x = self.bottleneck(x)

        for index, up in enumerate(self.ups):
            x = up(x, residuals[::-1][index])

        return self.outconv(x)
```

- The ‘unet’ class is the culmination of all the modules.
- Initialised with a key word argument ‘d’ that defines the channel depth used by the network.
- Define the sequence of down operations in a list to improve code structure. ‘d’ increases as the model gets deeper.
- Define the bottleneck layer.
- Only two real ‘up’ operations, the third is only there for code cohesion.
- A final, dimension restoring, output convolution to return the data to the correct number of channels.

This section requires more detail than bullet points.
The forward method takes as input noised images and creates an empty list ‘residuals’ to store any skipping data, this is instead of creating various variable names.
Loop over the down operations and unpack their return values into an ‘x’, input to the next layer, and a ‘res’, added to ‘residuals’ and used in the latter half of the network.
Pass the final downsampled ‘x’ through the bottleneck.

Loop over the upsampling operations, using python’s inbuilt ‘enumerate’ function to keep track of the working index, pass into the current ‘up’ the output of its direct predecessor as well as its skip connection.

Passes the data through a final output convolution to return to the appropriate channel depth.

With the basic functions of the class implemented as modular blocks, changes made in subsequent iterations will be far more facile. This current model state can be tested to ensure it works by passing through noiseless images from the dataset.

```python
f = unet()

f(images[:10]).shape
```

```
torch.Size([10, 1, 28, 28])
```

Define an instance of the unet class

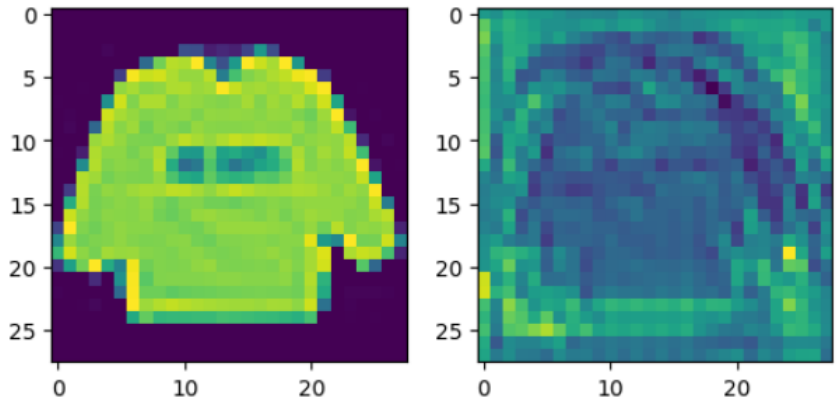
Pass through the first ten images in the dataset and inspect their shape

This test shows the unet can take fashion-mnist images and return outputs of the same shape.

Visualising these current input output mapping of the model as below.

```
fig, axes = plt.subplots(nrows=1, ncols=2)

axes[0].imshow(images[0].detach().numpy().swapaxes(0, 2).swapaxes(0, 1))
axes[1].imshow(yhat[0].detach().numpy().swapaxes(0, 2).swapaxes(0, 1))
```

```
<matplotlib.image.AxesImage at 0x7c9fc457f9a0>
```



Some information is preserved (probably because of the residual connections), but there is no real perception. This is to be expected, the parameters of the model are currently still in the same state as when they were randomly initialised. The only way to turn this model into anything moderately useful is to train it.

MODEL TRAINING

With the model defined, the next step is to train it. Pytorch abstracts much of the training process, nevertheless, it is still the most challenging part of the entire project. Returning to the training process from the paper, contained in the analysis, with a more technical lens I can begin to articulate exactly how to implement that density of theory.

Algorithm 1 Training

1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \text{Uniform}(\{1, \dots, T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: Take gradient descent step on
$$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t) \right\|^2$$
6: **until** converged

More verbosely what this algorithm is doing, and how I shall implement it, is:

1. Repeat.
2. Sample an image from the dataset x_0.
3. Sample a random number that falls within the valid range for a timestep t.
4. Sample some random noise, to be added to the image $\boldsymbol{\epsilon}$.
5. This line carries the bulk of weight of the algorithm and needs further breaking down:
    a. ‘Take gradient descent step on’ means adjust the network to decrease the following.

b. ∇_θ refers to the gradient of the following, with respect to the parameters.
c. $\epsilon_\theta\left(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t\right)$ is the output of the model, given a faded version of x_0 added element-wise with a correspondingly faded version of $\boldsymbol{\epsilon}$.
d. $\left\|\boldsymbol{\epsilon} - \epsilon_\theta\left(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t\right)\right\|^2$ is, therefore, the square of the difference between the output of the model and the noise added to the image.

6. Convergence in the context of training a machine learning model refers to when the parameters hit some kind of minima wherein further updates would yield either negligible improvements or regression, in model performance.

Converting this into code requires some additional thought. Firstly, I need to make define some objects that are essential to the training process.

```
diffusion = Diffusion()
model = unet()
loss_fn = nn.MSELoss()
opt = torch.optim.Adam(model.parameters(), lr=3e-4)
print(sum(p.numel() for p in model.parameters()))
```

```
254289
```

The first step is to define the diffusion class, although this has previously been defined, repeating doing so here makes the code cleaner, putting interconnected objects next to each other. Next, I define an instance of the 'unet' class as the model that is to be trained. The loss function I use, 'MSELoss', is the mean squared error loss, this finds the mean squared difference of the pixel values between two images. The optimiser 'Adam' is what manages the changes in the weights as a result of the loss function. When I define it, I pass the parameters of the model and a value for the learning rate, where the optimiser governs in what direction changes are made, the LR controls for how large that change is. My selection for the learning rate is a fairly typical 0.0003, this can be changed as a hyperparameter while optimising the training process. The final line is a sanity check that tells me how many parameters there are in the model, this will be essential when comparing models. At present ~250 000 parameters places this model on the smaller end of the spectrum.

I can now move onto writing the training loop for the present iteration of the model. In the following section I will not describe each individual line, save for the ones that are actually important

```
losses = []

plt.ion()  # Turn on interactive mode
fig, ax = plt.subplots()
batch = 500

for epoch in range(10):
    for i in tqdm(range(0, len(images), batch)):
        x = images[i:i+batch]

        t = diffusion.sample_timesteps(batch)
        noised, noise = diffusion.noise_images(x, t)

        opt.zero_grad()

        yhat = model(noised)
        loss = loss_fn(yhat, noise)

        losses += [loss.cpu().detach().numpy()]

        loss.backward()
        opt.step()

    ax.clear()
    ax.plot([-np.log(np.mean(losses[i:i+50])) for i in range(0, len(losses), 50)])
    ax.set_title("Training Loss")
    ax.set_xlabel("Iterations")
    ax.set_ylabel("Loss")
    display.clear_output(wait=True)
    display.display(plt.gcf())

    print(epoch, np.mean(losses[-1000:]))
```

- The section begins with some definitions, a list to store all the losses of the model, an interactive graph to visualise those losses, and a batch number. Batch number is a hyperparameter that controls how large the input data is, most of machine learning is matrices, matrices have suffered brutal optimisation at the hands of machine learning engineers to make them very fast. This means that pushing through more samples in the same forward pass is faster than splitting them up and pushing them through separately. This holds true up to the limit of your compute, when increasing any further creates operations that outsize the total memory of a device. The next lines are a double loop, first over all the epochs (an epoch is a run over the entire dataset) and second over the dataset as split into batches. Within these loops comes the actual meat of the process.
- A slice of the dataset is taken out into ‘x’ to be used as the current batch, some timesteps are randomly sampled as ‘t’, and they are both passed to the ‘noise_images’ function of the ‘diffusion’ class to produce the inputs and ground truths for the model. What comes next is a call on the optimiser to return all gradients to 0 as we are on a new batch. The model is then called on the ‘noised’ images and the output of that compared to the ground truth (the actual noise added to the images). The loss calculated from this is appended to the list. Pytorch’s inbuilt automatic differentiation then determines how much each parameter is responsible for that loss, and the optimiser adjusts all those parameters to make them ~less responsible.
- This following section is a plotting of the losses, with some smoothing added. Working with such inherently random data this smoothing helps make any trends more visible by minimising the jumps between batches.

This above code works, but fails to take advantage of hardware acceleration to hasten training. The parallelisation available on GPUs complements the highly optimised algorithms for machine learning to yield speed ups of orders of magnitude. Making training faster allows me to increase my iterations per unit time, resulting in a better convergence to optimal architecture and hyperparameters.

In my case, it is Kaggle's offer of 30 hours a week of free access to a GPU (Nvidia's p100) that I shall be capitalising upon. Converting the code to work with this GPU is made easy by Pytorch's heavily abstracted API, appending an instruction directing the relevant objects to the accelerating device is sufficient. I will outline the changes below.

```
device = 'cuda:0'
```

Define the device as 'cuda:0', this is interpreted by pytorch as the first accelerating device available to the user.

```
df = pd.read_csv('/kaggle/input/fashionmnist/fashion-mnist_train.csv').to_numpy()

labels = torch.tensor(df[:,0]).to(device)
images = torch.tensor(df[:,1:].reshape(60000, 1, 28, 28)).to(device)

images = 2*((images - images.min()) / (images.max() - images.min())) - 1
```

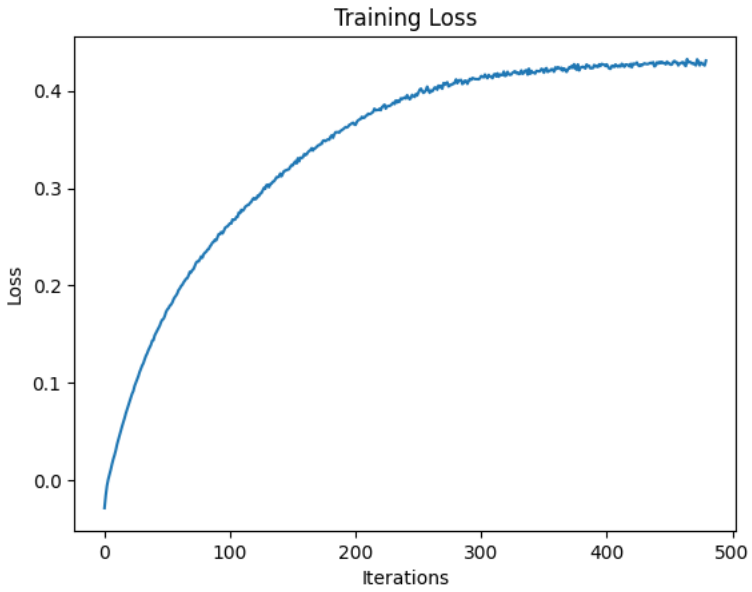
I append the declarations of 'images' and 'labels', directing them to the predefined device.

```
class Diffusion:
    def __init__(self, noise_steps=1000, beta_start=1e-4, beta_end=0.02, device='cuda:0'):
        self.noise_steps = noise_steps
        self.device = device
        self.beta_start = beta_start
        self.beta_end = beta_end

        self.beta = self.prepare_noise_schedule()
        self.alpha = 1. - self.beta
        self.alpha_hat = torch.cumprod(self.alpha, dim=0).to(self.device)

    def prepare_noise_schedule(self):
        lin = self.beta_end * (1 - torch.cos(torch.linspace(0, 1, self.noise_steps+1)[1:]*torch.pi)) / 2
        return lin.to(self.device)

    def noise_images(self, x, t):
        sqrt_alpha_hat = torch.sqrt(self.alpha_hat[t])[:, None, None, None]
        sqrt_one_minus_alpha_hat = torch.sqrt(1 - self.alpha_hat[t])[:, None, None, None]
        ε = torch.randn_like(x).to(device)
        return sqrt_alpha_hat * x + sqrt_one_minus_alpha_hat * ε, ε

    def sample_timesteps(self, n):
        return torch.randint(low=1, high=self.noise_steps, size=(n,))
```

There are a couple instances within the diffusion class where I direct different objects to the device. Anything that is to be seen by the model is moved across, as a general rule. Within the training loop itself it is very much more of the same, ensuring all data is on the same device.

```
losses = []

plt.ion()  # Turn on interactive mode
fig, ax = plt.subplots()
batch = 500

for epoch in range(1):
    for i in tqdm(range(0, len(images), batch)):
        x = images[i:i+batch]

        t = diffusion.sample_timesteps(batch)
        noised, noise = diffusion.noise_images(x, t)
        noised, noise = noised.to(device), noise.to(device)

        opt.zero_grad()

        yhat = model(noised)
        loss = loss_fn(yhat, noise)

        losses += [loss.cpu().detach().numpy()]

        loss.backward()
        opt.step()

    ax.clear()
    ax.plot([-np.log(np.mean(losses[i:i+50])) for i in range(0, len(losses), 50)])
    ax.set_title("Training Loss")
    ax.set_xlabel("Iterations")
    ax.set_ylabel("Loss")
    display.clear_output(wait=True)
    display.display(plt.gcf())

    print(epoch, np.mean(losses[-1000:]))
```

Running the above for 200 epochs at around 5s per epoch gives a total training time ≈20 minutes. This is for a particularly small model, absent many of the most computationally heavy features that could be potentially included. This model gets to a mean squared error of ~0.65 where it plateaus, indicating no further improvement. Note, this is different to the number indicated in the below graph because I have chosen to use the negative log loss for visualisation, when you are dealing with very small numbers, changes become imperceptible, using logs exaggerates differences within the range I am interested in.



Another thing I'd like to point out is that this will be typical of the shape I expect from my loss curves. Very rapid improvement that tails off into a plateau as the model exhausts any avenues for potential gains.

VISUALISING THE PERFORMANCE OF A MODEL

Any model's only as useful as it's deemed. It would be improper to make any kind of categorisation based on anything but the actual metric of interest, image generation. It is, therefore, insufficient to rely on our loss curve alone to quantise the performance of the model. Instead, visualising the model's propensity for the tasks of interest is an essential step in understanding it's ability. This can be decomposed further into two related but distinct abilities: generating, the process of moving from randomly generated noise to plausible images; and denoising, the proxy problem the model 'sees', reverting noised images.

Each occupies its own unique position in building up an understanding of the model and as such warrants its own consideration and thought on proper visualisation.

The algorithm for generation is defined in the paper as on the right. Intuitively, this acts as a random walk, an indirect path, moving pure noise to something recognisable. The process begins with sampled gaussian noise (i.e. $t = T$) and then iterates downwards through the timesteps, subtracting a scaled version of the predicted noise in each.

Algorithm 2 Sampling

1: $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2: **for** $t = T, \dots, 1$ **do**
3: $\quad \mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
4: $\quad \mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t) \right) + \sigma_t \mathbf{z}$
5: **end for**
6: **return** $\mathbf{x}_0$

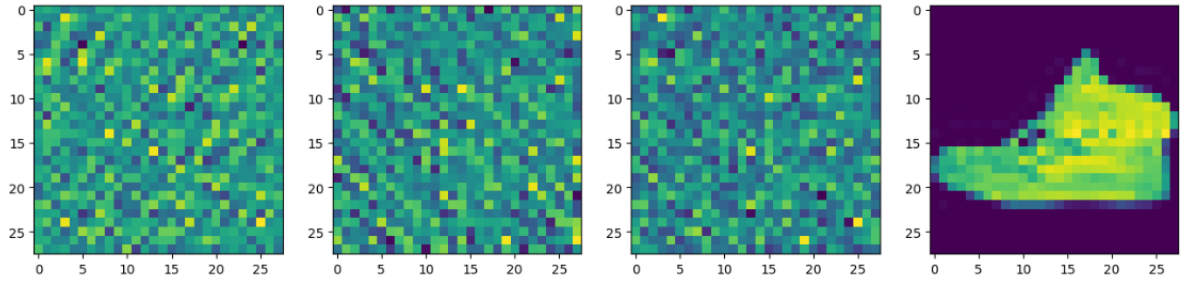
Denoising is mostly identical, save for replacing x_T with an actual noised image and adjusting the iterations accordingly.

Another informative visualisation, I find, is *zero-shot denoising*. Where both other processes involve iteration and scaling to transform a sample, it is possible to make a single step defined by a single prediction. This has the advantage of testing a different faculty of the model. The algorithm for this is not so well defined but can be reached with some rearrangement of the noise addition formula.

1. The noise addition formula is defined as $N = \boldsymbol{x}\sqrt{\bar{\alpha}} + \boldsymbol{\varepsilon}\sqrt{1-\bar{\alpha}}$, where N denotes the noised image.
2. This can be rearranged for $\boldsymbol{x} = \frac{N - \sqrt{1-\bar{\alpha}}\boldsymbol{\varepsilon}}{\sqrt{\bar{\alpha}}}$
3. With this rearrangement you can visualise the direction of the model, without needing to sit around and wait for the generation process.

```
fig, axes = plt.subplots(nrows=1, ncols=4, figsize=(16, 9))
i = 3
sf = (1-diffusion.alpha_hat[t[i].to(torch.long)]).sqrt()

axes[0].imshow(normalise(yhat[i]).cpu().detach().numpy().swapaxes(0, 2).swapaxes(0, 1))
axes[1].imshow(normalise(noised[i] - yhat[i]*sf).cpu().detach().numpy().swapaxes(0, 2).swapaxes(0, 1))
axes[2].imshow(normalise(noised[i]).cpu().detach().numpy().swapaxes(0, 2).swapaxes(0, 1))
axes[3].imshow(normalise(noised[i] - noise[i]*sf).cpu().detach().numpy().swapaxes(0, 2).swapaxes(0, 1))
```

```
<matplotlib.image.AxesImage at 0x79c5959825c0>
```



+ Code + Markdown

The above code works (*'works'*) in the sense that it is an accurate implementation of the *zero-shot generation* process; regrettably, that is the only sense. The code displays, from right to left, the noise predicted by the model, that same noise scaled and subtracted from the noised sample, the noised sample itself, and then the noised sample less the scaled version of the noise added (i.e. the noiseless sample). It is clear that the model is poor in that there is no obvious signal added. That is not to say the model is not working at all, it may be that the model is working, just not well enough to surpass the human brain's barrier to notice. As the model is iterated upon this should become a more discerning source of information, hopefully.

The other visualisations can both be implemented in a single function, added to the diffusion class. Keeping related things in a single class is good for code cleanliness and makes subsequent iterations faster. This function is an implementation of the process as defined in the paper so I shall be sparing when referring to specific operations. The majority of this code is changing matrices into correct shapes, or normalising.

```
    def denoise(self, model, noised, t, c, jumps=50):
        denoiseds = []
        with torch.no_grad():
            x = noised
            for i in tqdm(reversed(range(1, t)), position=0):
                t = (torch.ones(1) * i).long().to(self.device)
                predicted_noise = model(x, t, c)
                alpha = self.alpha[t][:, None, None, None]
                alpha_hat = self.alpha_hat[t][:, None, None, None]
                beta = self.beta[t][:, None, None, None]
                if i > 1:
                    noise = torch.randn_like(x)
                else:
                    noise = torch.zeros_like(x)
                x = 1 / torch.sqrt(alpha) * (x - ((1 - alpha) / (torch.sqrt(1 - alpha_hat))) * predicted_noise) + torch.sqrt(beta) * noise

                if i % jumps == 1:
                  denoiseds += [x]
                  denoiseds[-1] = (denoiseds[-1].clamp(-1, 1) + 1) / 2
                  denoiseds[-1] = (denoiseds[-1] * 255).type(torch.uint8)

        x = (x.clamp(-1, 1) + 1) / 2
        x = (x * 255).type(torch.uint8)
        denoiseds += [x]

        return denoiseds
```

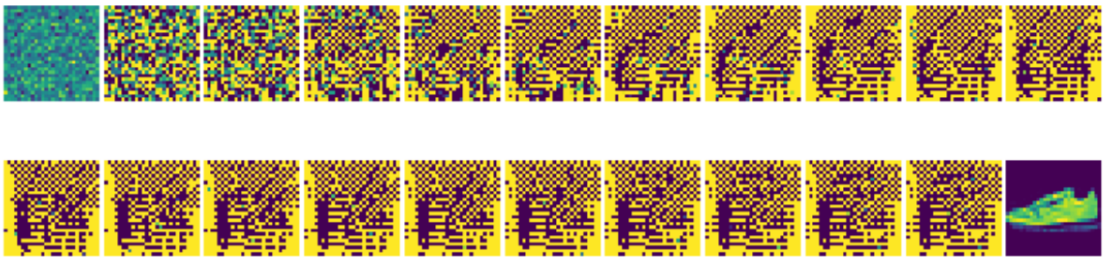
A consideration I made when writing this class was that there may be some value in seeing the process at multiple steps. This is what the ‘denoiseds’ list contains, stored versions of the image at various steps, interspaced by ‘jumps. The ‘torch.no_grad()’ calling tells the model not to keep track of gradients. This is best practice as it makes the model quicker.

The function returns ‘denoiseds’.

This function can then be called to denoise an actual sample. Within the red bar is information that this was a sample noised to $t = 995$, pretty noisy. The subsequent denoising is less than ideal, I don't believe that this current iteration of the model can be considered to be denoising at all. What occurs instead I have no plausible explanation for, and attribute, as all else, to divine omen.

```
i = 2

denoiseds = [noised[i:i+1]] + diffusion.denoise(model, noised[i:i+1], t[i:i+1], jumps=50) + [x[i:i+1]]
fig, axes = plt.subplots(nrows=2, ncols=12, figsize=(16, 9))

fig.subplots_adjust(hspace=-0.7, wspace=0.05)

for i, img in enumerate(denoiseds):
  axes[i // 12, i % 12].imshow(normalise(img[0]).cpu().detach().numpy().swapaxes(0, 2).swapaxes(0, 1))
  axes[i // 12, i % 12].axis('off')

for axes in axes.flatten()[len(denoiseds):]:
  axes.set_visible(False)
```

```
994it [00:01, 577.82it/s]
```



PLANS FOR IMPROVEMENT

I think it is now well and documentarily established that the model leaves some to be desired. It is clear that there is room for improvement and the question left is how to attain such improvement. This section begins with some explanation for why the model is so bad, using these weaknesses as a springboard to potential avenues to improvement.

When conceiving of ways to improve the model a sensible beginning is to imagine what it's missing. Abstracted to ultimacy the model is operations on data, any improvement must target one of these two. The job of the model is to predict noise, something non detrimental to this end may be some conception of what timestep the entered data is at. When considering how much noise has been added, the timestep must be essential, in fact, although I cannot prove it I think it highly likely that somewhere within the model are circuits aimed at guessing the timestep, for use in later operations. The ring-fencing of such circuits from the already limited pool hurts the model's ability to learn the complexity of the data. Therefore, the injection of timestep data is likely to proffer an increase in ability to the model.

Another possible improvement to the data would be to include conditional information. Presently, with fashion-mnist, each sample must belong to one of ten classes. Informing the model as to which could help its efforts. Conceptually this looks like giving the model a more specific area of pixel space to be aiming for. This and timestep data could be injected together.

Improvements to the model are abundant but mostly group around complexity, giving the model the ability to fit to a greater degree of complexity. Plain convolutional layers, as I have presently, are

incapable of learning non-linearity, I imagine it a safe assumption to guess that the function I am aiming to learn is indeed non-linear, for this reason I must introduce non-linearities like activation functions. An activation function follows a layer and applies some kind of non-linear transformation to the output. This enables a neural network to become a universal approximator.

Another weakness of the model is an inability to learn long-range dependencies. What I mean by this is that convolutional layers are acting over a fixed kernel of $m \times n$, this means that anything beyond the kernel *doesn't exist*, and thus cannot inform the operation. This makes sense as a limitation when you consider the importance of being able to attend to different parts of an image to develop an understanding, where I to only show you say 3x3 pixels at a time, you too would struggle. An obvious fix to this is to increase the kernel size, while this works it comes with its own drawbacks like heavier computation. Another, different, approach to solving this problem is to use attention over the entire image. Attention was initially designed to be used in natural language processing, mimicking the human ability to attend to words in the context of their surroundings. It has since found success most everywhere. Implementing attention is challenging and throws compute considerations out of the window. Being $O(n^2)$ attention's compute demands increase quadratically with sequence length, given image size also increases quadratically, this is unideal without a GPU supercluster. GPU poverty aside however, a limited implementation would be effective.

IMPROVEMENTS TO THE MODEL I: ARCHITECTURE

In this section I make several improvements to the architecture, developing its ability to learn the nuances of the underlying distribution. Specifically, I add ResNet and Self-Attention blocks throughout the model and introduce conditioning on both the timestep.

```python
def pos_encoding(t, channels=64, device='cuda:0'):
        inv_freq = 1.0 / (
            10000
            ** (torch.arange(0, channels, 2, device=device).float() / channels)
        )
        pos_enc_a = torch.sin(t.repeat(1, channels // 2) * inv_freq)
        pos_enc_b = torch.cos(t.repeat(1, channels // 2) * inv_freq)
        pos_enc = torch.cat([pos_enc_a, pos_enc_b], dim=-1)
        return pos_enc
```

This function is how timestep information can be encoded into something workable for the model. This method is known as sinusoidal embedding for its use of trigonometry to produce set patterns as outputs. This works via elementwise addition with the data to introduce slight perturbations, recognised as the model as information pertinent to the current timestep.

```python
class ResNet(nn.Module):
    def __init__(self, inc, outc, embdim=64, device='cuda:0'):
        super().__init__()

        self.crossemb = nn.Sequential(
            nn.Linear(2*embdim, embdim),
            nn.SiLU(),
            nn.Linear(embdim, embdim),
            nn.SiLU()
        )
        self.inj1 = nn.Linear(embdim, outc)
        self.inj2 = nn.Linear(embdim, outc)

        self.gn1 = nn.GroupNorm(1, inc)
        self.c1 = nn.Conv2d(inc, outc, 3, 1, 1)

        self.gn2 = nn.GroupNorm(1, outc)
        self.c2 = nn.Conv2d(outc, outc, 3, 1, 1)

        self = self.to(device)

    def forward(self, x, t, c):
        t = pos_encoding(t)
        cond = self.crossemb(torch.cat([t, c], axis=-1))

        h = F.silu(self.gn1(x))
        x = self.c1(h) + self.inj1(cond)[:,:,None,None]
        h = F.silu(self.gn2(x))
        h = self.c2(h) + self.inj2(cond)[:,:,None,None]

        return h + x
```

This is the successor to the double conv block from the previous iteration. It differs from its predecessor in four key ways.

- **Activations:** Throughout the model I add an activation function following certain layers. This introduces non-linearity to the network, allowing it to learn a more complex underlying distribution. The activation I have chosen is the sigmoid linear unit (SiLU).
- **Embedding injection:** in the __init__ method I define a crossemb module. This takes as input a concatenation of the class embedding with the timestep encoding, pushes them through some activated linear transformations into a representation that then forks into two separate, further linear transformations. The transformations are injected via element-wise addition into two distinct places within the ResNet, after each of the two convolutions.
- **Normalisation:** Prior to each convolution I push the data through a normalising operation. This shifts and scales the data to stay clustered around 0 and has been shown to greatly stabilise, thus accelerating, training.
- **Residual Connections:** In the forward pass I use a mixture of **x** and **h** to create a ‘skip connection’ for the data, circumventing the second convolution block. This is a second pathway for the data and allows gradients to reach the earliest parts of the model for updates.

```python
class SelfAttention(nn.Module):
    def __init__(self, s, c, device='cuda:0'):
        self.s, self.c = s, c
        super().__init__()

        self.gn1 = nn.GroupNorm(1, s**2)
        self.q, self.k, self.v = nn.Linear(c, c), nn.Linear(c, c), nn.Linear(c, c)
        self.mha = nn.MultiheadAttention(c, 4, batch_first=True)
        self.fff = nn.Sequential(
            nn.GroupNorm(1, s**2),
            nn.Linear(c, 4*c),
            nn.SiLU(),
            nn.Linear(4*c, c)
        )

        self = self.to(device)

    def forward(self, x):
        x = x.view(-1, self.c, self.s**2).swapaxes(1, 2)
        x = self.gn1(x)
        Q, K, V = self.q(x), self.k(x), self.v(x)
        attn, _ = self.mha(Q, K, V)
        attn = self.fff(attn) + x

        attn = attn.swapaxes(1, 2)
        return attn.view(-1, self.c, self.s, self.s)
```

The inclusion of attention in the model allows considerations to be made to farther range dependencies over the image. This is analogous to conferring the faculty of skimming to a reader, indeed, it was for text that self-attention of the sort I make use of was developed. Briefly, the function of this layer is as follows

1. Take an input image of shape (channels, size, size) and flatten it to an array (channels, size^2). This is equivalent to taking off a rows of pixels and laying them in a long line.
2. Swap the axes of the flattened image for (size^2, channels). This is necessary to make the model attend spatially.
3. Using three linear transformations create three distinct representations of the image.
4. Pass these representations through multiheaded attention.
5. Pass the output of this attention through a full feedforward block to perceive high-level features in the attention map.
6. Residually connect this FFF output with the image input.
7. Swap the axes back and return the shape to its appropriate dimensions for output.

```python
class Inception(nn.Module):
    def __init__(self, cin, d, k=[3, 5, 7]):
        super().__init__()

        self.c1 = nn.Conv2d(cin, int(d/4), k[0], 1, k[0] // 2)
        self.c2 = nn.Conv2d(cin, int(d/2), k[1], 1, k[1] // 2)
        self.c3 = nn.Conv2d(cin, int(d/4), k[2], 1, k[2] // 2)

    def forward(self, x):
        c1, c2, c3 = self.c1(x), self.c2(x), self.c3(x)

        return torch.cat((c1, c2, c3), axis=1)
```

An inception layer is many convolutional layers in a trench coat. Each convolution operates at different kernel sizes, searching for features of different sizes. These features maps are then concatenated to produce an output. Adding an inception layer at the start of the model has the effect of enabling it to glean more features from the input to be put through the Unet. Thus increasing the complexity of the distribution that the latter parts of the model can learn.

Before moving on to a description of the updated model's training and subsequent efficacy I must mention a non-obvious serious bug I found during this process and discuss its causes, its effects, how I came across it, and finally its resolution.

The bug exists because I have tried to be clever. In defining my model, instead of verbosely defining each layer as its own variable I store them in lists. This works well, making now's code cleaner and future development easier however, in abstracting away the actual sequence of operations I have inadvertently handicapped the model. Pytorch works as an automatic differentiation engine, a large part of this remit is keeping track of what affects what, to this end all pytorch objects, unless expressly told not to, feed into a computational graph. When I blurred the lines in my **forward** method of the **unet** class by mixing pytorch classes with python's inbuilt list function I obscured a majority of the model from this computational graph. Any parameters contained by these obscured classes were, thereby, neglected when the model came to self-adjusting. This explains the pitiful performance of the prototype model, not only was it limited architecturally, of what it did have I had shackled the overwhelming majority.

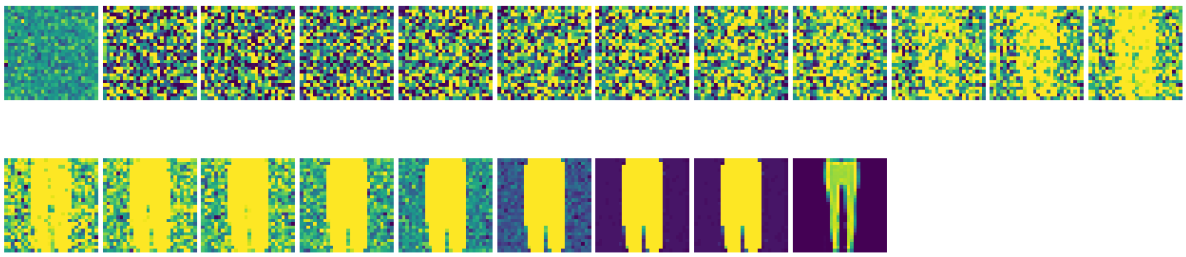
This bug was not obvious and passed undetected for a great period of time. There were a couple factors in this, chief being that in spite of the glaring error, *it ran*. To any observer not intimately acquainted with the code it would appear the model was 'working' in the sense it mapped inputs to outputs without complaint. Only on a very low-level inspection of specific parameters would the error have become glaring. The tip off for me was indirect, my code contains a line that outputs the parameter count of the model immediately after it is defined. I noticed that even though I was making significant changes to the model, none of them were being reflected in the parameter count. Upon inspection I found that the function I used to enumerate the parameters **model.parameters()** was absent the two lists of upward and downward operations. This was not critical to the model as omission from **model.parameters()** does not negate a layers existence and functionality. What it does mean is that when I passed **model.parameters()** into the optimiser I was, in effect, telling it not to touch any of the parameters that formed the bulk of the model. It is worth noting that at that point said parameters were randomly initialised. This meant that when I was training the model it was under constant barrage from utter meaningless while at the same time being told that it was some abstract latent representation. The model had no chance.

For the brutal detriment it served to the model, resolving the issue was elementary. Pytorch have evidently considered that users such as myself will find themselves asking for a use case such as mine. Pytorch have therefore developed a simple class **nn.ModuleList** that converts a python list into something that retains all its original functionality, whilst being visible to **model.paramters()**. Wrapping the lists with this was a complete fix.

With the bug resolved and a more complex architecture in place I could move onto training and evaluating my progress. Once again, the key metric is an unquantifiable *feeling*, about the images generated by the sampling process. An example of the model's output is featured below.

```
i = 4

denoiseds = [noised[i:i+1]] + diffusion.denoise(model, noised[i:i+1], t[i:i+1], c[i:i+1], jumps=50) + [x[i:i+1]]
fig, axes = plt.subplots(nrows=2, ncols=12, figsize=(16, 9))

fig.subplots_adjust(hspace=-0.7, wspace=0.05)

for i, img in enumerate(denoiseds):
  axes[i // 12, i % 12].imshow(normalise(img[0]).cpu().detach().numpy().swapaxes(0, 2).swapaxes(0, 1))
  axes[i // 12, i % 12].axis('off')

for axes in axes.flatten()[len(denoiseds):]:
  axes.set_visible(False)
```

```
884it [00:09, 91.71it/s]
```



This is clearly far more successful than the prototype. Noise is directed into a ~plausible sample, this is of course not optimal, but it was never meant to be. This was for architecture development and to better my understanding of the process. Nevertheless, it is gratifying to see function, especially when compared to previous iterations. Some additional comments I would make about this model is it has failed to learn subtlety. The generated image is of trousers, but it is not necessarily as they appear in the training data. The strong yellow suggests that while the model has learnt shape and noise it is has not quite caught the intricacies of texture in the images. This is not a concern yet as it is likely this is a result of undertraining.

With a model working for fashion-mnist I feel equipped to transition to a more difficult dataset: Cifar100. The following section discovers this transition.

TRANSLATION TO CIFAR

Cifar100 is a dataset containing 50000, 32x32 RGB images. They capture 100 types of items in a wide range of scenes and as such are more interesting to train diffusion models over. Images generated by such a model are more likely to hold a more general appeal as it represents a divergence from purely academic, proof of concept pursuits, to something the average person would consider interesting. The complexity of CIFAR as a dataset will be reflected in the difficulty of training the model. Being closer to reality the model has to learn more and more detail to compete with similarly nuanced training data. Porting the code directly is unlikely to be instantly successful, however it provides a strong, scalable backbone to improve upon.

The following details a setup I arrived at after dozens of experiments and adjustments, I omit details from each of those iterations for brevity. For this final architecture I've arrived at I will highlight places I have made refinement within the code, without repasting the entire body, given that changes are surprisingly few.

```python
def forward(self, x, t, c):
    t = pos_encoding(t)
    cond = self.crossemb(torch.cat([t, c], axis=-1))

    h = F.silu(self.gn1(x))
    x = self.c1(h)  + self.inj1(cond)[:,:,None,None]
    h = F.silu(self.gn2(x))
    h = self.c2(h)

    return h + x
```

In the forward pass of the **ResNet** block I only make use of a single injection. I reached this following experimentation with a few different ideas for conditioning. While it is very difficult to express exactly why this is better, I believe it is due to the second injection interfering with the residual connection and thus the output of the layer.

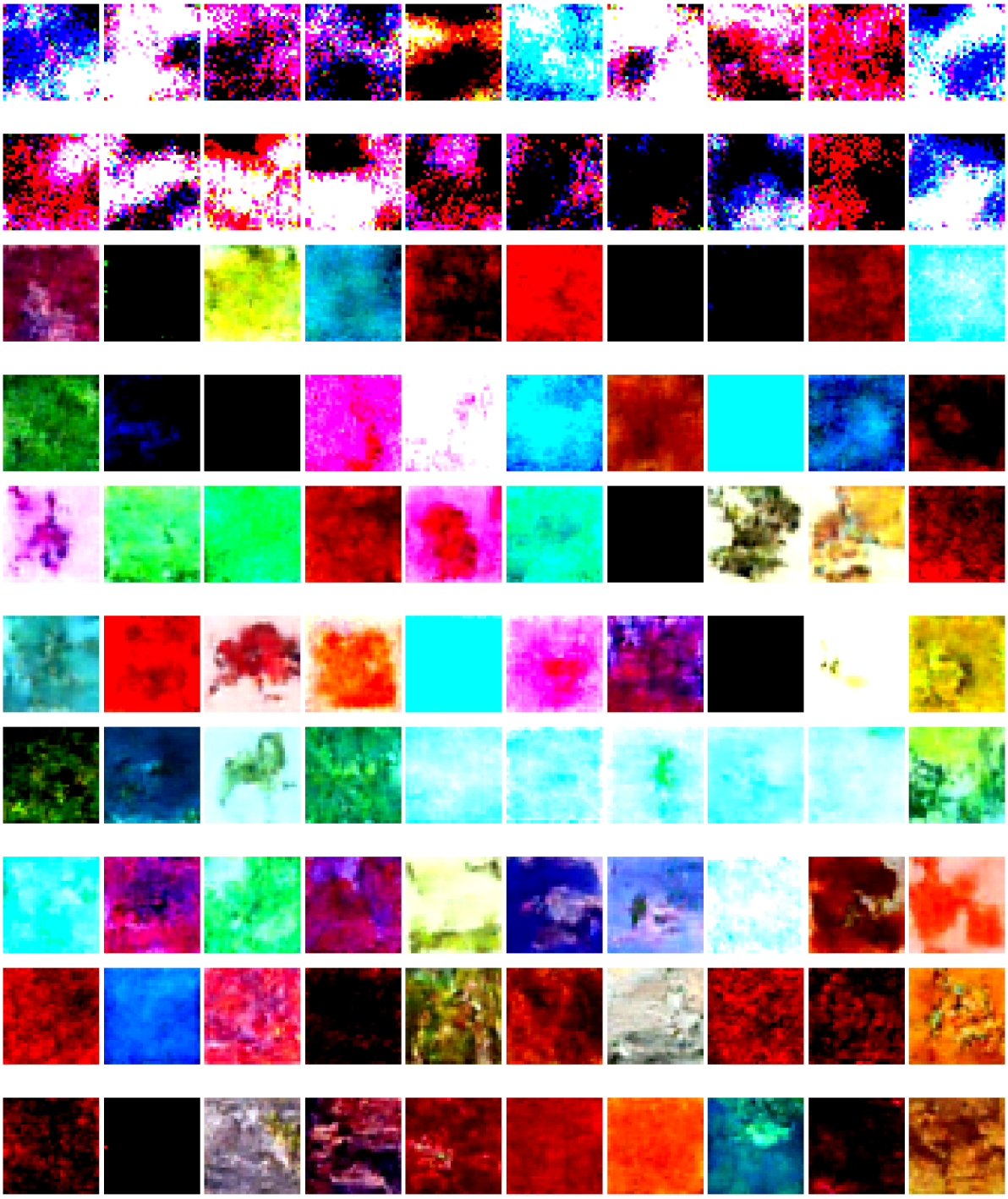
```python
self.incep = Inception(3, d)
self.downs = nn.ModuleList([
    downconv(d, 2*d),
    downconv(2*d, 4*d, attn=True, s=8),
    downconv(4*d, 8*d, attn=True, s=4)
])
self.bottleneck = bottleneck(8*d)
self.ups = nn.ModuleList([
    upconv(8*d, 4*d, 8),
    upconv(4*d, 2*d, 16),
    upconv(2*d, d, 32, up=False)
])
self.outconv = nn.Sequential(nn.Conv2d(2*d, 3, 1))
```

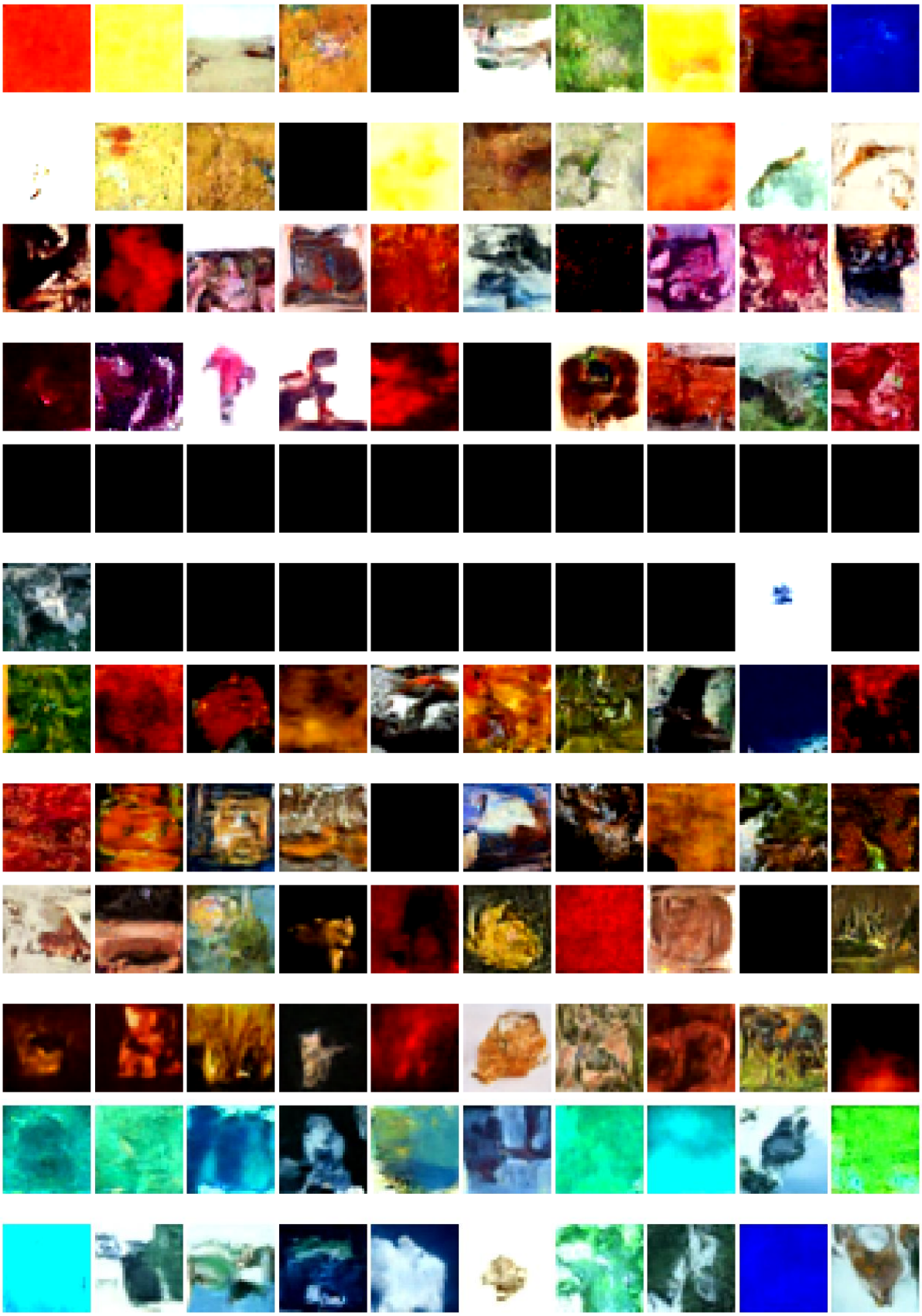
In the definition of the **Unet** I make changes to dimensionalities to accommodate the increased complexity of the new data. By doubling the capacity of the model all the way down I effectively scale the model, enabling it to learn the more complex and detailed patterns it will now be exposed to. I prepend the network with an **Inception** layer for the reasons aforementioned.

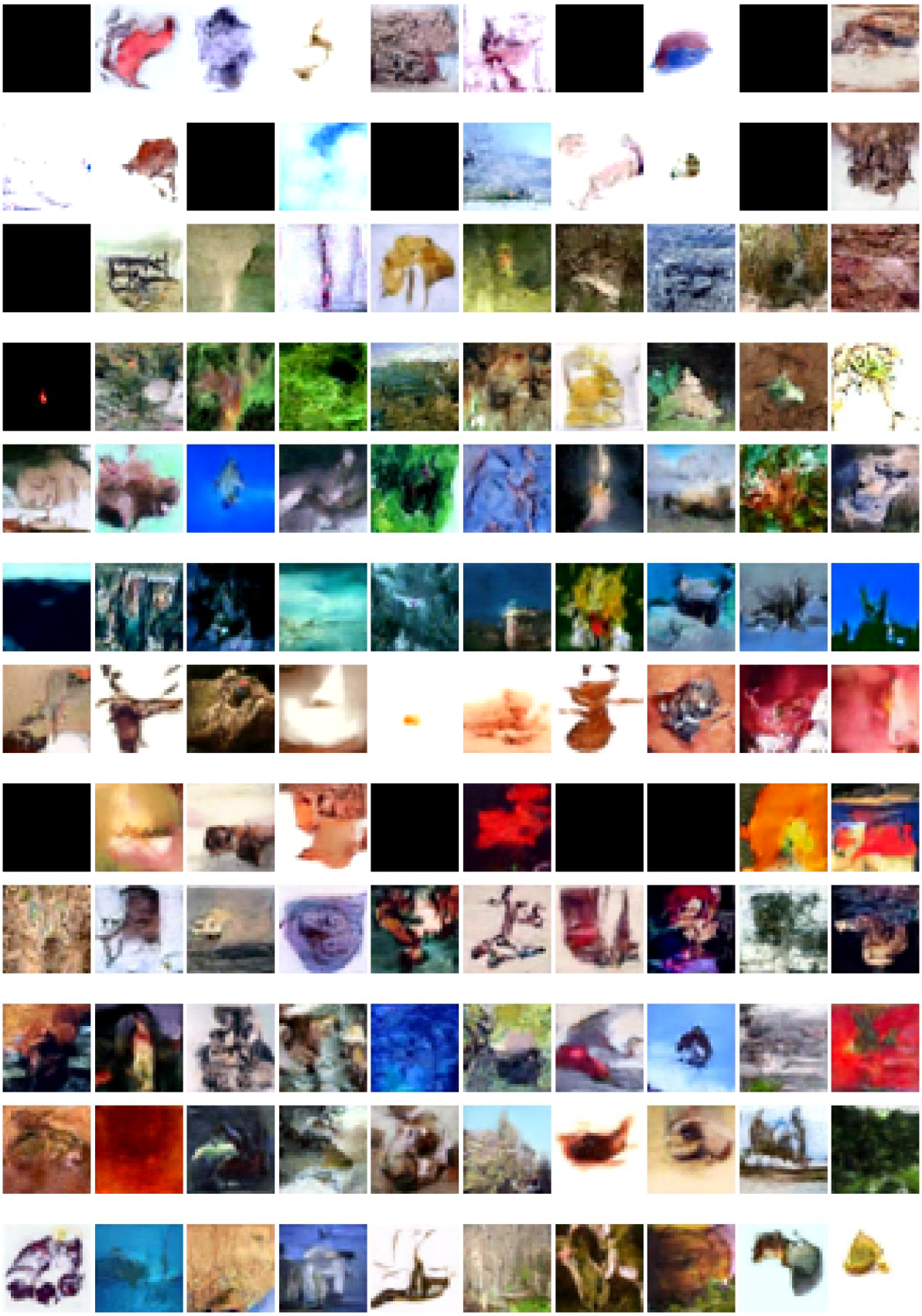
```python
batch = 125

for epoch in range(101):
    for i in tqdm(range(0, len(images), batch), ):
        x, c = images[i:i+batch], labels[i:i+batch]

        t = diffusion.sample_timesteps(batch).to(device)
        noised, noise = diffusion.noise_images(x, t)
        noised, noise = noised.to(device), noise.to(device)

        opt.zero_grad()
        t = t.unsqueeze(-1)

        yhat = model(noised, t, c)
        loss = loss_fn(yhat, noise)

        losses += [loss.cpu().detach().numpy()]

        loss.backward()
        opt.step()

    if epoch % 5 == 0:
        denoiseds = diffusion.denoise(model, torch.randn((1, 3, 32, 32)).to(device), torch.tensor([999]).to(device), c[:1], jumps=50)
        fig, axes = plt.subplots(nrows=2, ncols=12, figsize=(16, 9))

        fig.subplots_adjust(hspace=-0.7, wspace=0.05)
        print(metadata[b'fine_label_names'][c[0]])
        for i, img in enumerate(denoiseds):
            axes[i // 12, i % 12].imshow(normalise(img[0]).cpu().detach().numpy().swapaxes(0, 2).swapaxes(0, 1))
            axes[i // 12, i % 12].axis('off')

        for axes in axes.flatten()[len(denoiseds):]:
            axes.set_visible(False)

        plt.show()
```

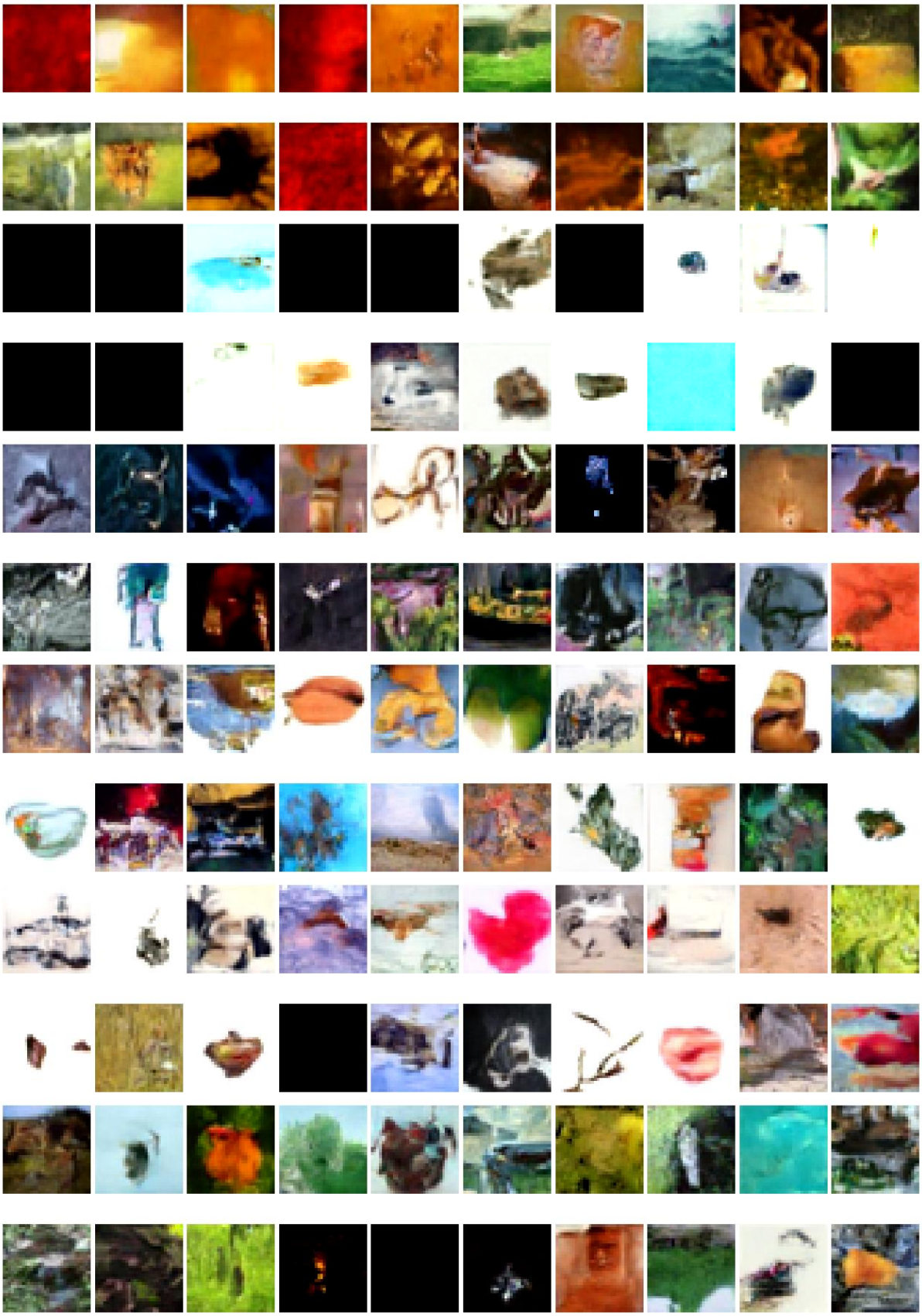
The above screenshot captures the updated training loop. The first thing to notice is the reduction in the batch size, in this my hand was forced by the increased depth and size of the data. The additional compute was too much for my GPU and I had to gradually decrease this hyperparameter until I found a value my system could manage with. The other major change is in how I monitor the progress of the network. Where previously was some functionality to create an updating loss value graph I have elected instead to replace it with sampling. This is the more informative of the pair I as the loss is only ever a proxy to the generation itself, viewing the samples themselves every n epochs is an easier way to monitor the progress of the model.

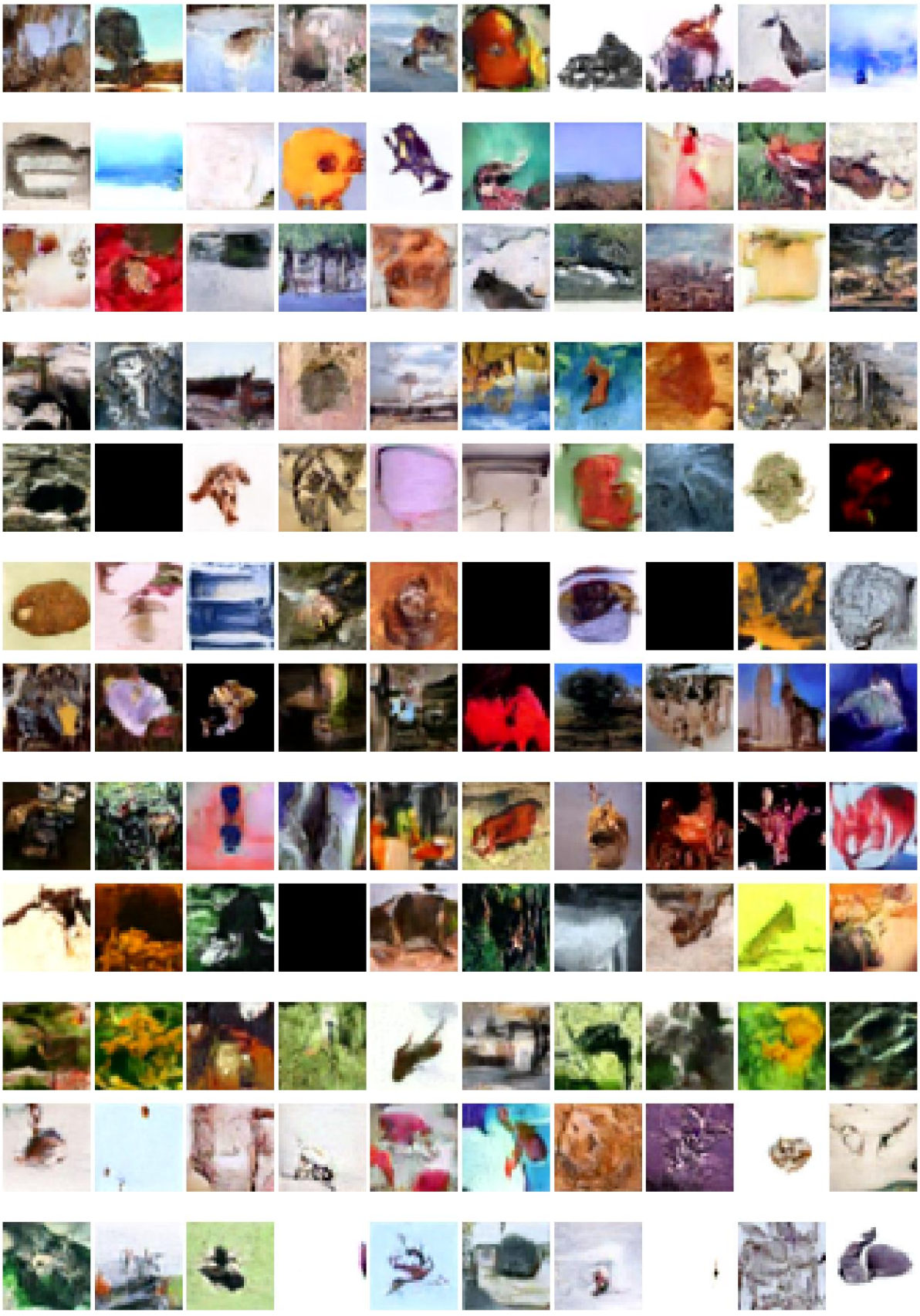
Below is the product of this visualisation. After every 5 epochs I sample 20 images from the model and display them in a 2x10 grid, the images are generated from gaussian noise with an initial timestep of 1000 and randomly selected conditioning information. I repeat this for 151 epochs which, on the P100, takes roughly three hours.

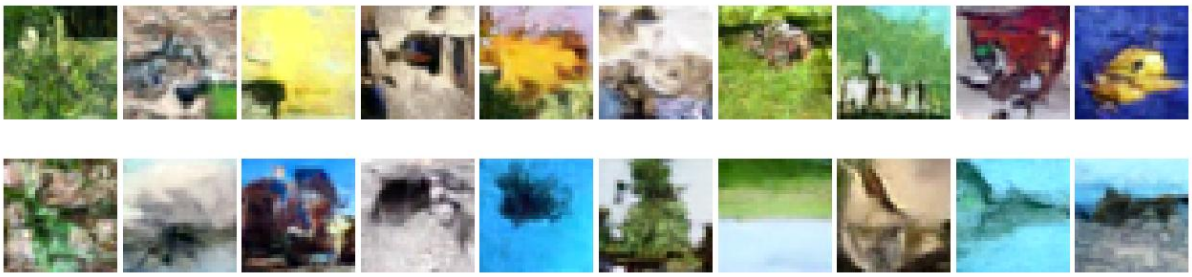












Some interesting things I note:

1. An explicable *vibe* to many of the images, especially in the latter stages of training, that I can only describe as akin to lucid dreaming. A first look plausibility followed by a confused closer inspection. This suggests that the model is not optimal, it is learning the distribution but only -ish. It knows what say a dog does not look like, but not well enough to produce an actual dog.
2. Mode collapse is a problem, this occurs when the model has no idea what it's doing and just reverts to a damage limiting, plain, monochromatic slab. In the pathfinding analogy of the model, this is equivalent to the model getting lost. This is more prevalent in earlier epochs as the model is a less good ~pathfinder generally, but still occurs right up to the end. Resolving this is most likely a sampling problem as in every architecture I tried I found similar issues.
3. The model sometimes struggles with orientation, for an example take the image pair below. The image on the left is as it was sampled, the image on the right is a rotated version. From the image on the right I think it is reasonable to assume this is a generation of the class 'sky' but if that is so how come as it initially came out the sky is beneath the earth. The reason for this is a technique I added in training, to prevent the model overfitting and artificially inflate the size of the dataset I added transforms to the data. This means that a random portion of the data will be rotated before being shown to the model. This is most likely what causes such an unrealistic feature.

CONCLUSIONS FROM THE MODEL

I tentatively leave the model here, while I do believe I could improve the model I find that it would cost time I could more profitably spend on other aspects of this project. This is not a perfect model, not by some distance, however it is passable, if I find its poor quality irksome I will likely return and, hopefully, improve it.

IMPROVEMENTS TO THE MODEL II:

IMPROVEMENTS TO THE MODEL III

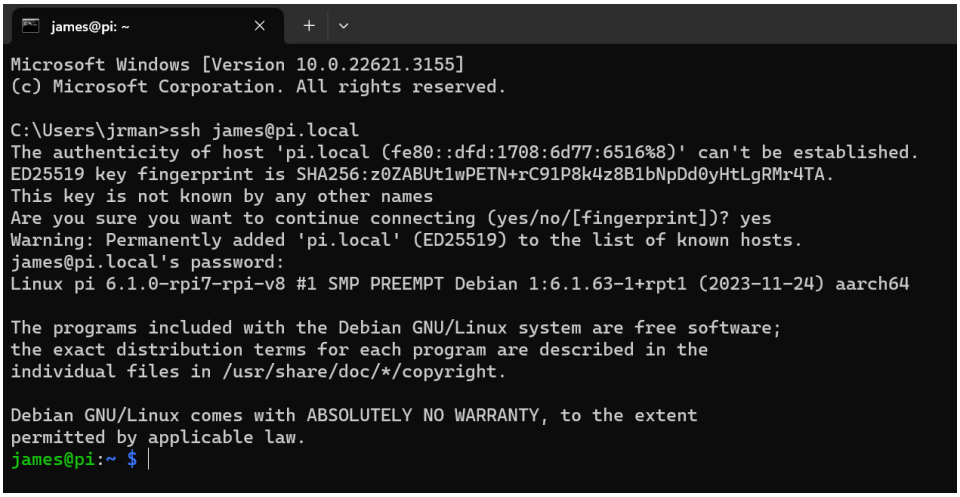
THE APP

The app ideally runs on its own hardware, for this reason I am going to be developing it on a raspberry pi 4, networked onto my Wi-Fi, to be accessed from my laptop. This has the advantage of increasing how robust my app is, in that the case where the app runs on my device, but no one else's, is removed. Instead, I can be fairly sure that if it works on my laptop it will work on anyone's, as there is no obvious cross contamination of dependencies. In compartmentalizing the two, client and server, I remove the single point of failure, or at least make it easier to pin down.

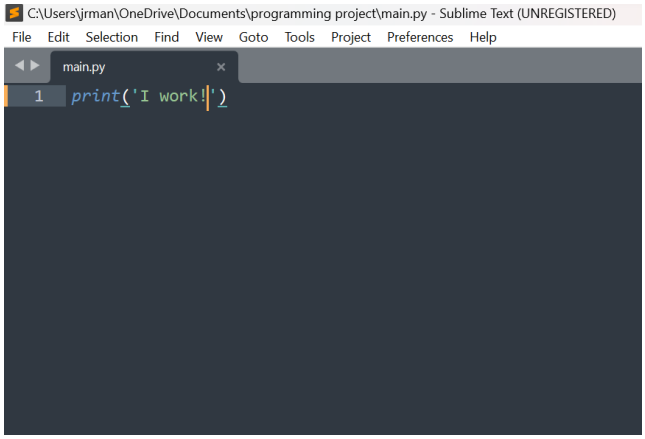
SETUP

In this section I set up the raspberry pi and outline my plan for how I will develop the app over two devices.

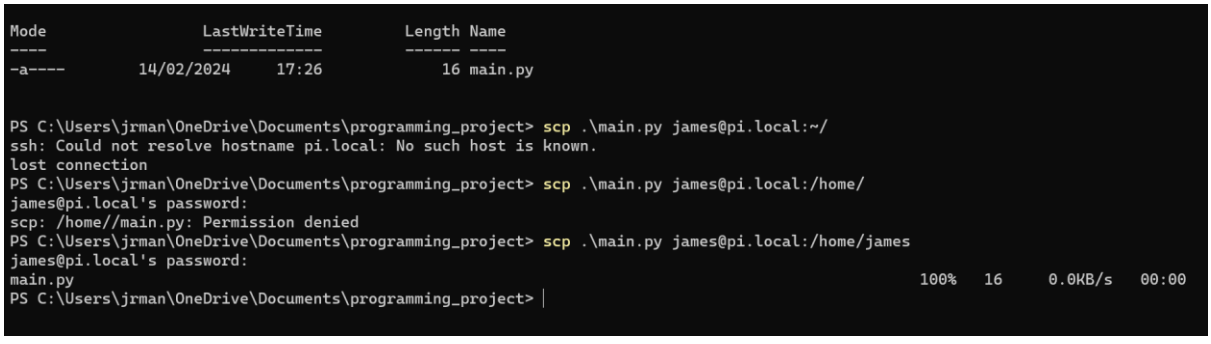
First things first, I set up the raspberry pi with a Raspbian lite, an OS with no desktop and minimal peripheral software. I then ssh into this raspberry pi from my laptop.



Good, it works. This means that I can run commands on my pi, from my laptop. Now, when it comes to creating those commands and actually writing code, ssh leaves a lot to be desired. Instead, I will use a text editor on my laptop, 'sublime', to write code. To test this and make sure the process works I will write a short script on my laptop and run it on my pi.



The script, written in sublime on my latop.



Seen above is the command shell where I use 'scp' (secure copy protocol) to transfer the file from my laptop onto the pi. Also in the screenshot is the command not working, seen in the response 'Permission denied', this was caused by me trying to copy into a directory I did not have privileges for. After rectifying that error and running it again, I can then check if it came through on the pi and still runs.

```
james@pi:~ $ ls
main.py
james@pi:~ $ python3 main.py
I work!
james@pi:~ $
```

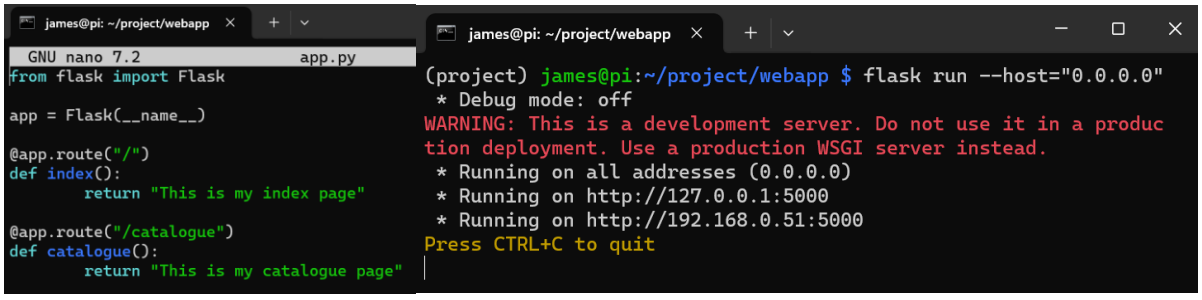
So it works. This means I can work faster on my laptop and transfer any files onto the pi. LGTM.

BASICS OF THE WEBAPP

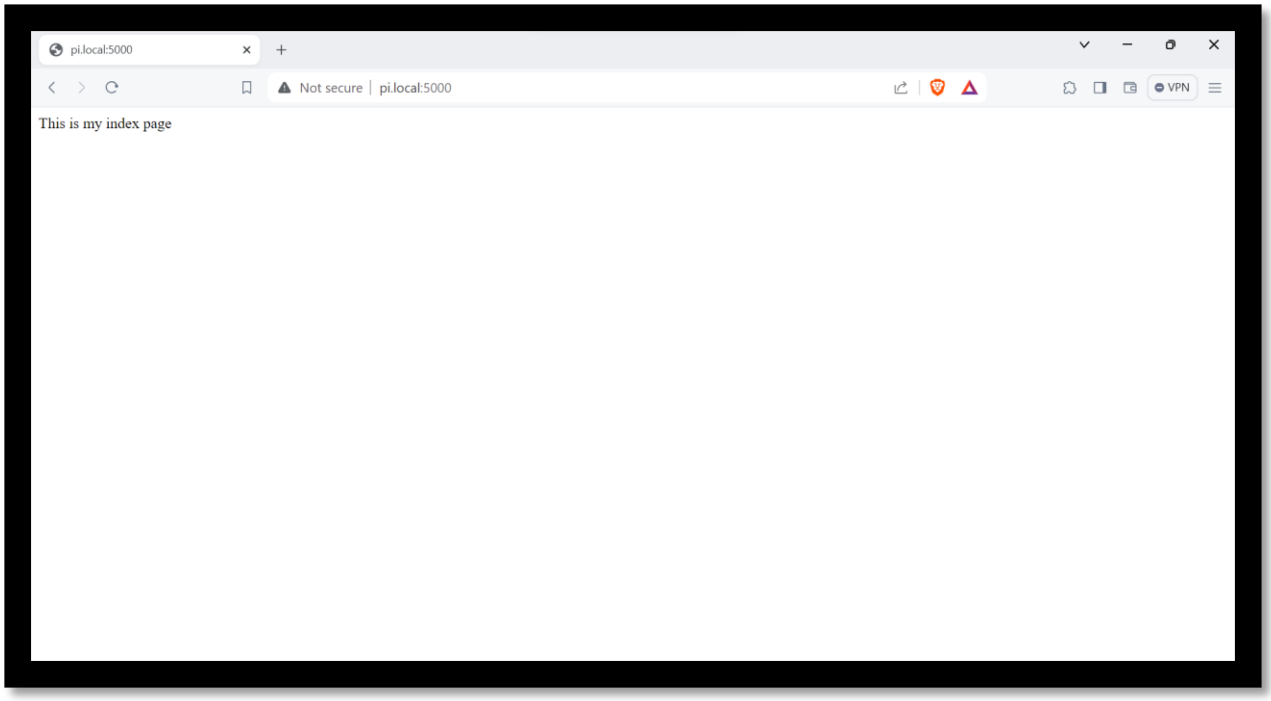
Now the pi is functional, and I have a direct, working link into it, I can commence actual development of the app. My intention is to create a flask webapp using jinja2 for web templating. These are libraries not included in the python install so I must pip install them onto my device. This will be done inside of a virtual environment as coding best practice and because it makes things cleaner.

```
(project) james@pi:~/project/bin $ pip install FLask
Looking in indexes: https://pypi.org/simple, https://www.piwheels.org/simple
Collecting FLask
  Downloading https://www.piwheels.org/simple/flask/flask-3.0.2-py3-none-any.whl (101 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 101.3/101.3 kB 1.1 MB/s eta 0:00:00
Collecting Werkzeug>=3.0.0 (from FLask)
  Downloading https://www.piwheels.org/simple/werkzeug/werkzeug-3.0.1-py3-none-any.whl (226 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 226.7/226.7 kB 3.1 MB/s eta 0:00:00
Collecting Jinja2>=3.1.2 (from FLask)
  Downloading https://www.piwheels.org/simple/jinja2/Jinja2-3.1.3-py3-none-any.whl (133 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 133.2/133.2 kB 3.1 MB/s eta 0:00:00
Collecting itsdangerous>=2.1.2 (from FLask)
  Downloading https://www.piwheels.org/simple/itsdangerous/itsdangerous-2.1.2-py3-none-any.whl (15 kB)
Collecting click>=8.1.3 (from FLask)
  Downloading https://www.piwheels.org/simple/click/click-8.1.7-py3-none-any.whl (97 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 97.9/97.9 kB 2.2 MB/s eta 0:00:00
Collecting blinker>=1.6.2 (from FLask)
  Downloading https://www.piwheels.org/simple/blinker/blinker-1.7.0-py3-none-any.whl (13 kB)
Collecting MarkupSafe>=2.0 (from Jinja2>=3.1.2->FLask)
  Downloading MarkupSafe-2.1.5-cp311-cp311-manylinux_2_17_aarch64.manylinux2014_aarch64.whl.metadata (3.0 kB)
Downloading MarkupSafe-2.1.5-cp311-cp311-manylinux_2_17_aarch64.manylinux2014_aarch64.whl (29 kB)
Installing collected packages: MarkupSafe, itsdangerous, click, blinker, Werkzeug, Jinja2, FLask
Successfully installed FLask-3.0.2 Jinja2-3.1.3 MarkupSafe-2.1.5 Werkzeug-3.0.1 blinker-1.7.0 click-8.1.7 itsdangerous-2
.1.2
(project) james@pi:~/project/bin $
```

With this done I can set up a simple webapp to ensure everything is in order, and start executing jobs for the project. I contain the code for the server in a *app.py* and run it.



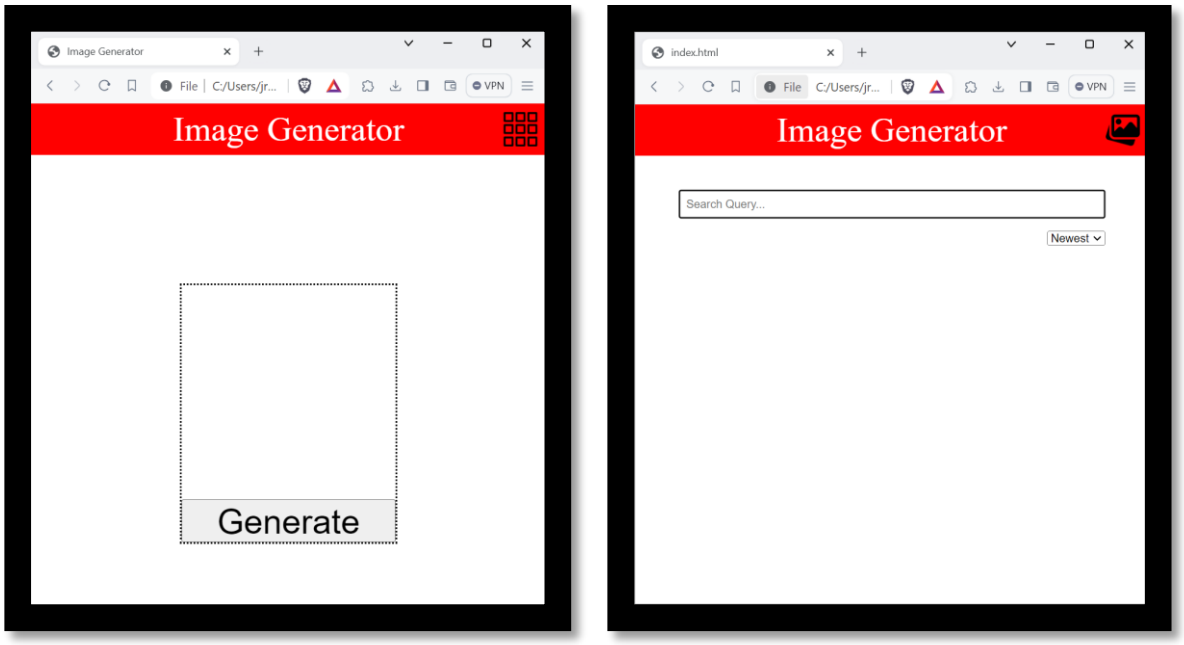
I can then check it is accessible from my laptop's browser.



I can access it from my laptop. With this confirmed actual development can begin.

BASIC STYLING FOR THE INDEX PAGE

With the technical setup completed I can move on to creating the basic shaping for the index page, as described in my design. This will be done in a combination of html, css and javascript. Html to control the content in the page, css to stylise that content, and javascript to introduce some interactivity.



Above are the preliminary ideas for the index page (left) and catalogue page (right). Colours and design may be improved in later iterations however the fundamental layout won't change much. The choices to

use a consistent nav bar makes transition between pages smoother and to place the link to the other page behind a correspondent image, whose location doesn't move, all add to the aesthetics and ease-of-use of the page. The simplicity of the generate function is very much in line with decisions made in the analysis and design portions of the project. The prominence of the search and sorting functions really preclude any misunderstanding on how to use the interface.

For brevity, at this point I will only include and elaborate on the code for the index page.

```css
body {
    margin: 0px;
    padding: 0px;
}

nav {
    height: 60px;
    width: 100%;
    background: red;
    position: relative;
}
header {
    text-align: center;
    font-size: 40px;
    line-height: 60px;
    color: white;
}

#grid {
    position: absolute;
    height: 40px;
    width: 40px;
}
.catalogue_link {
    top: 10px;
    right: 50px;
    position: absolute;
}

.box {
    height: 300px;
    width: 250px;
    border: 2px black dotted;
    margin: 150px auto 0px auto;
    position: relative;
}

#gen_button {
    height: 50px;
    width: 250px;
    bottom: 0px;
    position: absolute;
    font-size: 40px;
}
```

```html
<!DOCTYPE html>
<html>

<head>
    <link rel="stylesheet" type="text/css" href="static/indexstyles.css">
    <title>Image Generator</title>
</head>

<body>
    <nav>
        <header>Image Generator</header>
        <div class='catalogue_link'>
            <a href="catalogue/index.html">
                <img id='grid' src="static/grid.png">
            </a>
        </div>
    </nav>
    <div class="box">
        <div class="img_container">
            <img id='image' src="">
        </div>
        <button id='gen_button'>Generate</button>
    </div>
</body>

</html>
```

The HTML (left) and the CSS (above), at this point, are not particularly complex or specific. At this moment a majority of it is fairly boilerplate. I define the content of the page and then wiggle it around, is a reasonable summary. I make use of html 'divs' as they help keep everything self-contained and improve the accuracy of my CSS. As for the CSS, most of the code such as anything referring to 'height', 'width', 'top', or 'bottom' is manipulating the size or position of elements.

Noticeable absentees from this prototype, aside from stylising and prettifying the interface, include the absence of a hyperparameter settings option in the index page, and the absence of a catalogue from the catalogue page. The reasons for these omissions are that this is a very early prototype of the design and as such I am unsure how exactly the final hyperparameters for the model will look, and that when I came to design the grid, I realised that I was not working in an optimal order. To explain what I mean, the grid function is one part with an especial dependence on another, the database. The catalogue is a visual representation of the contents of the database, because of this it is necessary for me to construct the database.

CONSTRUCTING THE DATABASE

At this stage I return to the pi to set up the database. As described in the analysis and design I am using PostgreSQL for the databasing, this is because it is open source and doesn't require a license, but also

the strength of its documentation, something I will need to rely on should I become, at any point, stuck. Although simple, I reiterate the design of the database to bring clarity to the actions I take in this section.

First, an entity description for the database.

Images (HashRef, DateCreated, Prompt)

I can then go into the server and run this within the postgres database I have set up

```
projectdb=# CREATE TABLE images (
image_id serial PRIMARY KEY,
hashRef VARCHAR (32) UNIQUE NOT NULL,
dateCreated DATE NOT NULL DEFAULT CURRENT_DATE,
prompt TEXT NOT NULL
);
CREATE TABLE
```

- Define the table 'images'
- Automatically set its primary key sequentially
- 'hashRef' points to filename, must be unique
- 'dateCreated' set to date added to database
- 'prompt', associated caption

With the database defined, the next step is to connect it to the webapp. Naturally, there are no actual generated images on the server or in the database, so I must add some fake data. My plan for these next stages is as follows.

1. Find ~100 images online and download them onto my laptop.
2. Transfer these images onto the pi, into a temporary directory.
3. Write a script that will transform them into the correct format for the app, this involves giving them hashed filenames, adding their information to the database, and moving them into a 'processed' directory.
4. Go into the webapp file and add functionality to query the database.
5. Write the html / css for the grid
6. Write code to sample from the database and, using Jinja, plug this into the webpage.

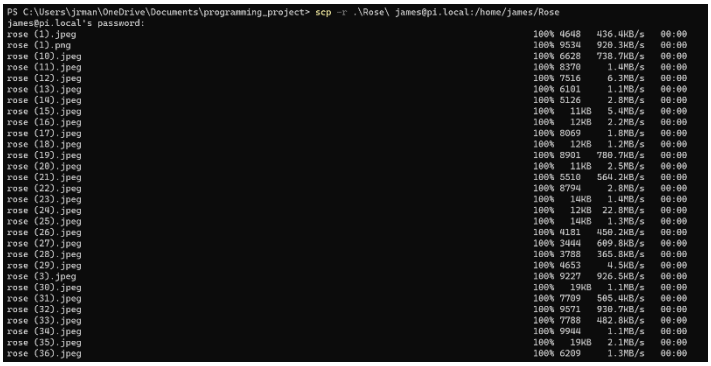
In addition to helping ensure that the database is working, in writing code for these tasks I will get a better feel for the processes and stronger understanding of what is technically required for the solution. Code written may also prove to be reusable in the actual central implementation.

1)

From Kaggle I have downloaded a dataset of images of flowers, taking one subset of this dataset I have a directory containing ~100 images of flowers.

2)

I can use SCP again with the '-r' parameter indicating I want it to recurse through an entire directory, copying across the entire contents. The image to the right shows the terminal as I do this. Listing each image as it transfers them one by one.



3)

With the files on the pi, the next step is to bring them into the correct format.

```
import os, cv2

path = '/home/james/project/webapp/Rose'

for i in os.listdir(path):
        img = cv2.imread(f'{path}/{i}')
        hashedfilename = str(abs(hash(str(img))))
        os.system(f'mv "{path}/{i}" /home/james/project/webapp/images/{hashedfilename}.jpeg')
```

The code above imports two modules, ‘os’ and ‘cv2’, responsible for controlling the operating system and opening the images, respectively. The ‘path’ variable is then declared, pointing to the location of directory containing the images. I loop over the filenames contained in the directory at ‘path’, loading the image at each into the variable ‘img’. Unwrapping the functions into the order in which they are executed: I convert ‘img’ into a string and get the hash for that string, which in python is an integer, take the absolute value of that integer as it is sometimes negative, and the negative breaks file naming, then convert that, now positive, integer into a string. The final line is instructing the operating system to move the file in the original directory into a new directory, with its hashed value as a filename.

```
import psycopg2 as psy
import os
import random
import string

try:
        conn = psy.connect("dbname='projectdb' user='james' host='localhost' password='database'")
        curs = conn.cursor()
        conn.autocommit = True
except:
        print("bugger")

for fname in os.listdir('images'):
        prompt = ''.join(random.sample(string.ascii_lowercase, 20))
        curs.execute(f"""INSERT INTO images(hashref, prompt)
                    VALUES('{fname}', '{prompt}');""")

conn.close()
```

The next stage brings in the database, the library psycopg2 is the postgres database adapter for python, with it I can push commands to the database. The try except statement attempts to create a connection to the database, using predefined details about the user and database name. The for loop iterates over all the hashed filenames in the new directory, creates a random ‘prompt’, then makes a new entry the database, of that filename and prompt. Below is what the database now looks like. The first two entries are a result of an initial manual insertion, and some omitted quotation marks respectively. They are now deleted.

image_id	hashref	datecreated	prompt
1	abcdef	2024-02-15	this is a prompt
2	{fname}	2024-02-15	{prompt}
4	974975887320425538.jpeg	2024-02-15	ipkybuogtzejhcmvxsad
5	83728759473951438 59.jpeg	2024-02-15	cbfwjnloqshkgmidyteu
6	67291651487763946 89.jpeg	2024-02-15	yletvocnjbuaqzshmrwf
7	88816584670359556 93.jpeg	2024-02-15	ixjuvfbdqntamzkeorwc
8	82502198914673317 97.jpeg	2024-02-15	jydgxmsvfholkzwqunrc
9	55456907212717621 03.jpeg	2024-02-15	dwqobmkivjxtlheufryp
10	928406475287683 41.jpeg	2024-02-15	ilmjpchtvaorsxqkfybg
11	86563913092274977 56.jpeg	2024-02-15	ejmqwhfslncapduzxkit
12	14530978134613466 12.jpeg	2024-02-15	fkcdhlaroypgjiswmenu
13	18192845321106458 25.jpeg	2024-02-15	lyvaqkxpbegdruzofsnm
14	46840250236058749 50.jpeg	2024-02-15	yvmiwsblfueraknoxqjg
15	76679221615030372 49.jpeg	2024-02-15	ifamogrkcvbhxynwuelj
16	11635178602591178 68.jpeg	2024-02-15	gpexwsvbdlyqhfcumrkt
17	57871205007066311 16.jpeg	2024-02-15	jbpqutydlxnmkasovgzc
18	59425233179426974 08.jpeg	2024-02-15	hxzdtonvfjlgmskqyuar
19	7864899411531947 66.jpeg	2024-02-15	cfdjgkhmnqwvxoritplz
20	68294789675189013 71.jpeg	2024-02-15	eptxogrnmkslqvjuhabd
21	86853994174896520 77.jpeg	2024-02-15	czetnrohwbvaudplmqyk
22	27270366166054097 24.jpeg	2024-02-15	xwsoqanhrfbucpvygdke
23	64062739349353909 04.jpeg	2024-02-15	zpjncfxglwmtdbiyvrqk
24	71841658967124721 45.jpeg	2024-02-15	jtucmixeokgndwrzshya
25	22235234864830578 94.jpeg	2024-02-15	rozeqlcmwfupgijvxbsa
26	50160294974443954 11.jpeg	2024-02-15	siatmnyogplbhxerwkvu
27	16976582799116277 30.jpeg	2024-02-15	imrlpsxtfkezgjhdnybw
28	82536254161254044 13.jpeg	2024-02-15	kvanqwslxgerfhtdupby
29	55002006081112381 65.jpeg	2024-02-15	rlxpugdobkmyetfwvhaj

CONNECTING THE DATABASE TO THE CATALOGUE PAGE

With the database full of dummy data, the next step is displaying it onto the catalogue page. This is challenging to do in a dynamic way. A predefined grid looks hacky when it is not filled, instead there needs to be a robust implementation that does not rely on the database being in some certain state. My plan for this is to develop logic to access the database, randomly sample up to n items from it and then fill in rows until I reach either n cells, or run out of images to fill them with. Any empty cells will not be displayed.

```python
from flask import Flask, render_template
from database import DataBase

db = DataBase()
app = Flask(__name__)

@app.route('/')
def index():
        return render_template('index.html')

@app.route('/catalogue')
def catalogue(ncols=3):
        items = db.get_n_items()
        nrows = len(items) // ncols
        rem = len(items) % ncols
        return render_template('catalogue/index.html', items=items, nrows=nrows, ncols=ncols, rem=rem)
```

The above is a screenshot of the current state of app.py, I have added a line that creates an instance of a database class object I have defined in a separate file ‘database.py’.

```python
import psycopg2 as psy
import random

class DataBase():
        def __init__(self):
                self.conn = psy.connect("dbname='projectdb' user='james' host='localhost' password='database'")
                self.curs = self.conn.cursor()
                self.conn.autocommit = True

        def get_n_items(self, n=30):
                self.curs.execute("SELECT * FROM images;")
                rows = self.curs.fetchall()
                n_selected = random.sample(rows, min(n, len(rows)))

                return n_selected
```

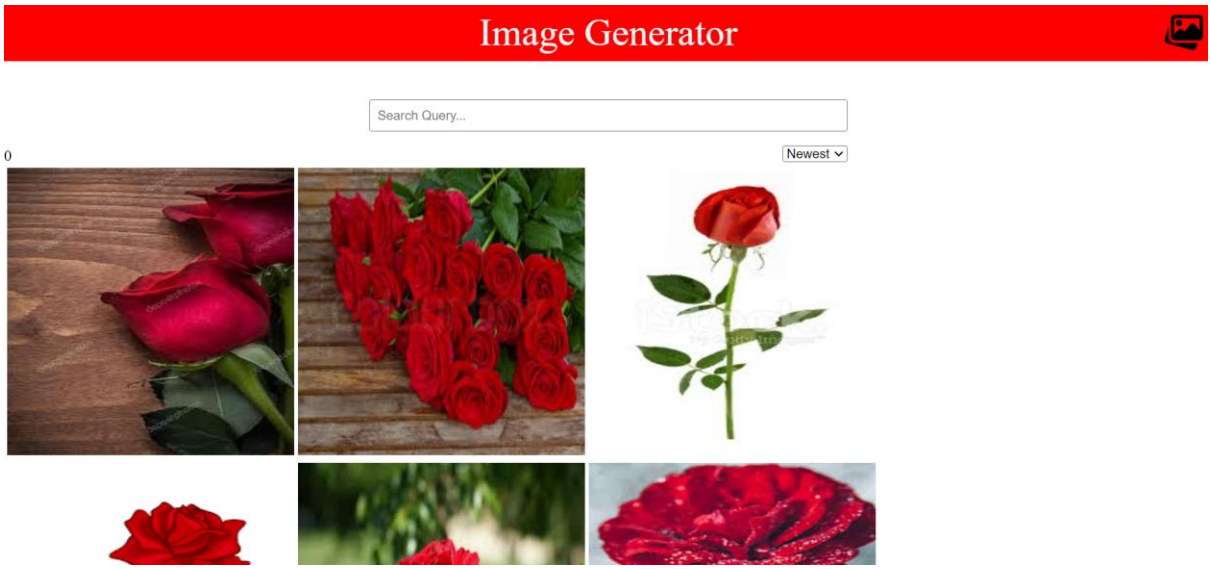
The bounded function ‘get_n_items’ queries the database and samples n items randomly from it, these n selected items are returned. It is worth noting that n items are only sampled in the case that there are $\geq$ n items in the database, otherwise it is simply permuting what there is.

Moving onto the changes made to ‘catalogue/index.html’.

```
<table>
        {% for i in range(nrows) %}
                <tr>
                        {% for n in range(ncols) %}
                                <td><img src="{{ url_for('static', filename='images/' + items[i*ncols + n][1]) }}"
                        {% endfor %}
                </tr>
        {% endfor %}
        {{ rem }}
</table>
```

The curly braces are where I am using jinja to introduce some flexibility to the table. Jinja allows for python ~execution, in html pages, using for loops allows me to vary the number of rows in my grid, based upon how many images have been returned. Another benefit of using for loops is that I can change variables like the number of columns ‘painlessly’ in the future, this means for later iterations, where I am being more design oriented, I could make such a change with minimal effort.

This comes together to produce a catalogue page looking like this.



FUNCTIONALITY FOR INPUTS

The next step is to develop some buttons and inputs available to the user. While they cannot be fully developed until I have made some headway into other parts of the project, laying the groundwork is productive.

COMBINING THE TWO

With the groundwork established on the webapp and the model at a place approaching passable I can begin to integrate the two.

HAVING THE MODEL ON THE PI

The first step is to get the model onto the pi and working. The first step for this is almost identical to the getting the dummy images on, I copy over the weights of the model to the pi.

```
PS C:\Users\jrman> scp "C:\Users\jrman\OneDrive\Documents\progra
mming_project\notebooks\cifar100-10-03" james@pi.local:/home/jam
es
james@pi.local's password:
cifar100-10-03                100% 8300KB   3.1MB/s   00:02
```

Now comes the more difficult part, actually running the model. The first issue is also the easiest to fix, I do not have the necessary libraries installed. I can import these as is necessary. The next step is to bring over the necessary classes in order to initialise the model. In its current form the model is just a saved binary of the weights, without giving it a skeleton to latch onto it is useless. For this reason, I will copy over all of the class definitions from the notebook into a single file which can then be imported. Keeping it all in one file like this is good for code cleanliness.

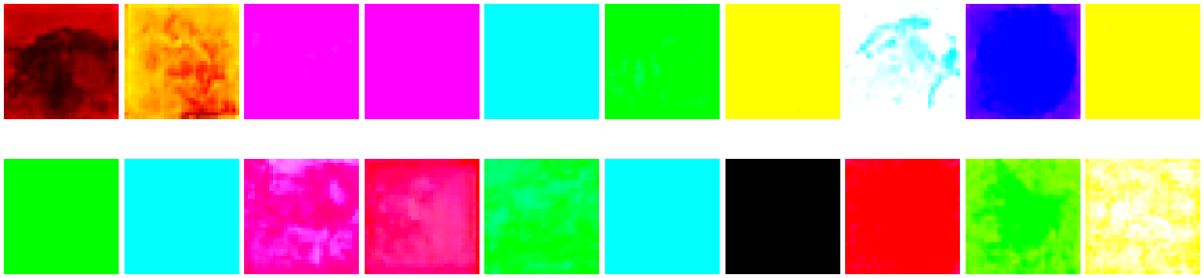
It is here I encounter the first of what will be a whole host of problems.

```
RuntimeError: Attempting to deserialize object on a CUDA device
but torch.cuda.is_available() is False. If you are running on a
CPU-only machine, please use torch.load with map_location=torch.
device('cpu') to map your storages to the CPU.
(project) james@pi:~/project/webapp $
```

When attempting to compile the model a runtime error is thrown complaining that the pi has no cuda device. Cuda refers to the proprietary software that runs on an accelerated device made by Nvidia. It is true, my pi possesses no such hardware, to rectify this I will go through every class and replace the **device** attributes that care currently set to **cuda:0** to **cpu**. I will also follow the suggestion in the error to use **torch.load** with **map_location** set appropriately. This solves the problem and I can now load an instance of the trained model on the pi.

With the instance loaded I can now begin to write the functionality for denoising. This is very transferrable from the notebook as I can simply copy over the **diffusion** class and it works out-of-the-box. I remove the methods I no longer need use of such as **noise** and **denoise**. After this it is as simple as passing in the loaded model as parameter to generate.

It is at this point that I find a catastrophic problem, the pi cannot run the model. The volume of compute required by the model is so substantial that the pi crashes whenever I try to run it. The solution to this is unclear. I expect the issue to reside in the **Self Attention** blocks but I cannot simply prune the model of them as they're indispensable to performance. This leaves me with little alternative but to retrain the model with a smaller depth and hope the reduction is enough to bring the compute required back within what can be offered by the pi.



This is with a reduction in the **d** parameter of the Unet by 50%, I gave the model ~2 hours to train and at no point did it generate any samples better than these. It is clear that reduction of the model size is infeasible, or at least, not so easy. This is a quite a severe setback for the project and there are no clear avenues to its resolution.