6CCS3PRJ
Appendix

Final Project Appendix

Author: Abdurrahman Lleshi

Supervisor: Atsushi Suzuki

Student ID: K20014224

April 5, 2023

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Abdurrahman Lleshi

April 5, 2023

Contents

Appendix A

Extra Information

A.1 Mathematical Proofs for the forward & backward diffusion equation

These proofs where taken from Lilian Weng in heres paper What are Diffusion Models?v[1]

A.1.1 Forward diffusion process

$$q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I}) \quad q(\mathbf{x}_{1:T}|\mathbf{x}_0) = \prod_{t=1}^{T} q(\mathbf{x}_t|\mathbf{x}_{t-1})$$

We sample x_t at any arbitrary time step t in a closed form using a reparameterisation trick. Let $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$:

$$\begin{aligned}
\mathbf{x}_t &= \sqrt{\alpha_t}\mathbf{x}_{t-1} + \sqrt{1-\alpha_t}\boldsymbol{\epsilon}_{t-1} && \text{;where } \boldsymbol{\epsilon}_{t-1}, \boldsymbol{\epsilon}_{t-2}, \cdots \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \\
&= \sqrt{\alpha_t\alpha_{t-1}}\mathbf{x}_{t-2} + \sqrt{1-\alpha_t\alpha_{t-1}}\bar{\boldsymbol{\epsilon}}_{t-2} && \text{;where } \bar{\boldsymbol{\epsilon}}_{t-2} \text{ merges two Gaussians (*).} \\
&= \ldots \\
&= \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon} \\
q(\mathbf{x}_t|\mathbf{x}_0) &= \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I})
\end{aligned}$$

A.1.2 Reverse diffusion process

$$q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) = \mathcal{N}(\mathbf{x}_{t-1}; {\color{blue}\tilde{\boldsymbol{\mu}}(\mathbf{x}_t, \mathbf{x}_0)}, {\color{red}\tilde{\beta}_t\mathbf{I}})$$

Using Bayes' rule we achieve:

$$
\begin{aligned}
q(\mathbf{x}_{t-1}|\mathbf{x}_t,\mathbf{x}_0) &= q(\mathbf{x}_t|\mathbf{x}_{t-1},\mathbf{x}_0)\frac{q(\mathbf{x}_{t-1}|\mathbf{x}_0)}{q(\mathbf{x}_t|\mathbf{x}_0)} \\
&\propto \exp\Big(-\frac{1}{2}\big(\frac{(\mathbf{x}_t-\sqrt{\alpha_t}\mathbf{x}_{t-1})^2}{\beta_t}+\frac{(\mathbf{x}_{t-1}-\sqrt{\bar{\alpha}_{t-1}}\mathbf{x}_0)^2}{1-\bar{\alpha}_{t-1}}-\frac{(\mathbf{x}_t-\sqrt{\bar{\alpha}_t}\mathbf{x}_0)^2}{1-\bar{\alpha}_t}\big)\Big) \\
&= \exp\Big(-\frac{1}{2}\big(\frac{\mathbf{x}_t^2-2\sqrt{\alpha_t}\mathbf{x}_t\color{blue}{\mathbf{x}_{t-1}}\color{black}{+}\alpha_t\color{red}{\mathbf{x}_{t-1}^2}}{\beta_t}+\frac{\color{red}{\mathbf{x}_{t-1}^2}\color{black}{-2}\sqrt{\bar{\alpha}_{t-1}}\mathbf{x}_0\color{blue}{\mathbf{x}_{t-1}}\color{black}{+}\bar{\alpha}_{t-1}\mathbf{x}_0^2}{1-\bar{\alpha}_{t-1}}-\frac{(\mathbf{x}_t-\sqrt{\bar{\alpha}_t}\mathbf{x}_0)^2}{1-\bar{\alpha}_t}\big)\Big) \\
&= \exp\Big(-\frac{1}{2}\big(\color{red}{(\frac{\alpha_t}{\beta_t}+\frac{1}{1-\bar{\alpha}_{t-1}})\mathbf{x}_{t-1}^2}-\color{blue}{(\frac{2\sqrt{\alpha_t}}{\beta_t}\mathbf{x}_t+\frac{2\sqrt{\bar{\alpha}_{t-1}}}{1-\bar{\alpha}_{t-1}}\mathbf{x}_0)\mathbf{x}_{t-1}}\color{black}{+}C(\mathbf{x}_t,\mathbf{x}_0)\big)\Big)
\end{aligned}
$$

We then parameterised the mean and variance as follows:

$$
\begin{aligned}
\tilde{\beta}_t &= 1/(\frac{\alpha_t}{\beta_t}+\frac{1}{1-\bar{\alpha}_{t-1}}) = 1/(\frac{\alpha_t-\bar{\alpha}_t+\beta_t}{\beta_t(1-\bar{\alpha}_{t-1})}) = \color{green}{\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\cdot\beta_t} \\
\tilde{\boldsymbol{\mu}}_t(\mathbf{x}_t,\mathbf{x}_0) &= (\frac{\sqrt{\alpha_t}}{\beta_t}\mathbf{x}_t+\frac{\sqrt{\bar{\alpha}_{t-1}}}{1-\bar{\alpha}_{t-1}}\mathbf{x}_0)/(\frac{\alpha_t}{\beta_t}+\frac{1}{1-\bar{\alpha}_{t-1}}) \\
&= (\frac{\sqrt{\alpha_t}}{\beta_t}\mathbf{x}_t+\frac{\sqrt{\bar{\alpha}_{t-1}}}{1-\bar{\alpha}_{t-1}}\mathbf{x}_0)\color{green}{\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\cdot\beta_t} \\
&= \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\mathbf{x}_t+\frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}\mathbf{x}_0
\end{aligned}
$$

Appendix B

User Guide

B.1 Instructions

This is a step by step instructions to how to run the Jupyter notebook to test our project and how to setup the King's HPC servers.

B.2 Server Setup

The main step is to setup our King's Computational Research Engineering and Technology Environment (CREATE) High Performance Computing (HPC) server. Access is only provided via KCL's e-research team, contact them to grant access. This will allow to run the jupyter notebook without any issues into running out of memory or vRAM during run time.

We will refer to the setup guide provided at https://docs.er.kcl.ac.uk/CREATE/access/.

B.2.1 Server Setup - Creating SSH Keys & Logging in

1. First create a pair of Secure shell (SSH) keys use the following for your operating system: MacOS and Linux or Windows

2. Once the SSH keys have been created locate the id_rsa.pub which can be found in .ssh folder. Copy the contents of the is_rsa.pub to your Add new SSH key in e-Research Portal.

3. Open a terminal of choice and enter the following command:

```
ssh -m hmac-sha2-512 <your k-number>@hpc.create.kcl.ac.uk
```

4. Once entered for the first time you will be promoted to accept a multi factor authentication (MFA) request on Access Approvals tab of e-Research Portal, approve the request.

5. Finally, once in will be granted with a terminal as below:

B.2.2 Server Setup - Setting up virtual python environment to allow Jupyter to work

1. Running the following commands in sequence will create a virtual environment that will have jupyerlab installed.

```
module load python/3.8.12-gcc-9.4.0
virtualenv jvenv -p `which python`
source jvenv/bin/activate
pip install jupyterlab
```

B.2.3 Server Setup - Creating a script to be submitted using sbatch

1. In diffusion_model folder you will find a modified batch file named ops-jupyter.sh. Copy the content onto your clipboard.

2. Entering the terminal paste the content onto a file using a terminal editor like vi/vim, emacs, nano. For this example we will use nano:

```
nano ops-jupyter.sh
# Paste the contents via ctrl+v
ctrl+x
y
enter
```

3. Once the file has been saved and named you are ready to submit the script to a sbatch process:

```
sbatch ops-jupyter.sh
```

4. After you have submitted the script you can check once ready via the following commands:

```
squeue -u k{Number}
```

5. Checking ST (status) column you should see PD (PENDING), checking in a few minutes the status code should have changed to R (READY). Once ready should produce a slurm file.

6. Finally run ‘ls‘ command to check if a file ‘slurm-xxxxx.out‘ file was generated where ‘x...x‘ are the ID number of the batch job

7. You have successful created a batch job and will be running for the next 48 hours! (However upon our investigation the batch jobs do no run more than 24 hours.)

B.2.4 Server Setup - Connecting jupyter notebook

1. Nano into the slurm file. Where the zeros are the batch job ID.

```
nano slurm-0000000.sh
```

2. Next you should see outputted SSH tunnel as follows:

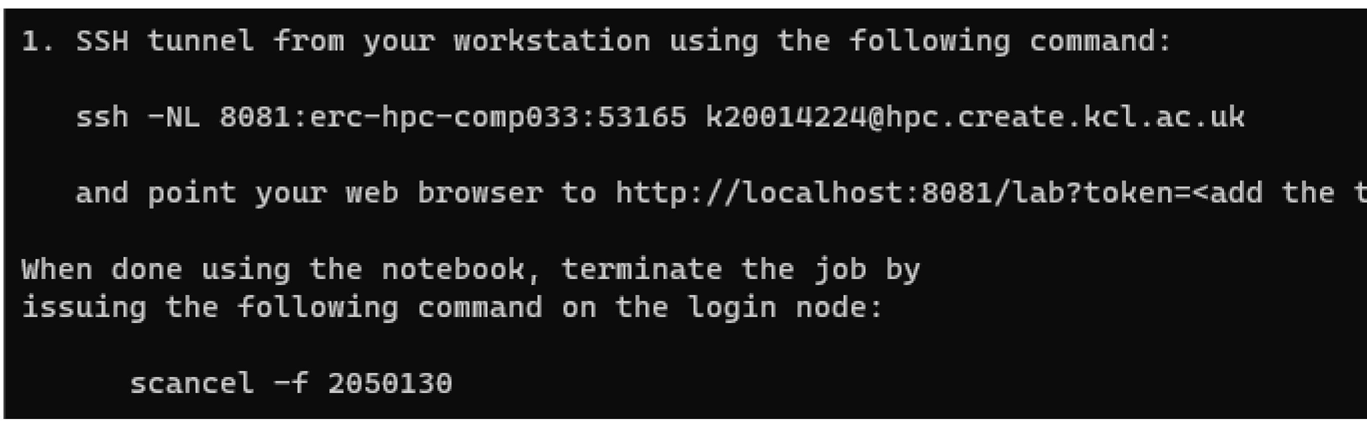


Figure B.1: Active Jupyter Notebook

3. Then copy the second line staring with ‘ssh -NL...‘ and open a new terminal. Paste the command into the new terminal. For windows you may need to include the following:

```
ssh -NL {PORT_1}:erc-hpc-comp{SERVER}:{PORT} {KNUMBER}@hpc.create.kcl.ac.uk
    -m hmac-sha2-512
```

Where PORT_1 would be 8888 if server-batch.sh is used and PORT would be assigned during the batch job and SERVER too.

4. Finally, back to the original terminal. After scrolling down the slurm file copy the token from any of the two URLs which has the ‘token=...‘ available. We do not need the URLs ip:port as we will be using localhost:8888.

B.3 Jupyter notebook - Setting up remote with VSCode

1. Install VSCode at https://code.visualstudio.com/ and open the application.
2. Next open up the project source code within VSCode.
3. Navigate to diffusion_model folder and open the main notebook called diffusion_model.ipynb.
4. VSCode will prompt to install Jupyter extensions. If not navigate to the extension menu and search for 'Jupyter', the plugin is published by Microsoft. Install the plugins.
5. Navigate to the top right of the Jupyter notebook where 'Select Kernel' is displaying and click.
6. Once the prompt opens up click on 'Select Another Kernel...'.
7. Then click on 'Existing Jupyter Server...'
8. Then select 'Enter the URL of the running Jupyter server'
9. This is where we input the following URL:

```
http://localhost:8888/lab?token=<COPIED FROM JUPYTER OUTPUT BELOW>
```

10. Then hitting enter key twice.
11. Finally, the option to select Python 3 (ipykernel) should become available.

It is key to only run one notebook at a time. This is due to the Jupyter batch job not able to to handle more than one notebook. To overcome this you can run more batch jobs. However, it is important to change the port in the opsjupyter.sh from 8888 to any other 8XXX port where XXX are random 3 digit numbers between 0-9, as well as renaming the opsjupyter.sh file into a new name like opsjupyter-8000.sh.

B.4 Jupyter notebooks available to be ran

It is important to install the python requirements withing the diffusion_model.ipynb notebook inside the diffusion_model folder.

The following notebooks are available to run:

- diffusion_model.ipynb (This is the main file and should be ran first due to having all the python installation requirements. This is also where the variables and other parameters where changed to allow for our investigation)

- diffusion_model_results.ipynb (We used to generate results for our evaulation and results)
- DM_adamax.ipynb (This notebook contains the adamax optimiser)
- DM_lion.ipynb (This notebook contains the lion optimiser)
- DM_pruning.ipynb (This notebook contains three types of pruning for the model. After running the UNet model, select ONLY ONE pruning type to run.)
- DM_test_unit.ipynb (Contains the test unit for our UNet)

B.5 Ending process

Once experimenting or running the Jupyter notebooks, the batch job can remain running. To save electricity, not waste computing power allowing other users to use the GPUs and free up ports it is important to manual end the batch job.

A simple command and end the batch job:

```
scancel -f BATCH_JOB_ID
```

References

[1] Lilian Weng. What are diffusion models? *lilianweng.github.io*, Jul 2021.

Appendix C

Source Code

Requirements installation

In [ ]:
```
!pip3 install torch torchvision torchaudio
!pip install lion-pytorch
!pip install matplotlib
!pip install numpy
```

Test environment config.

In [ ]:
```
import sys
print(sys.executable)
import torch
print(torch.__file__)
print(torch.cuda.is_available())
from torch.utils import collect_env
print(collect_env.main())
```

Check if the environment has access to the NVIDIA A100 GPU.

In [ ]:
```
!nvidia-smi
```

Diffusion Model

A simple implementation of the diffusion model in PyTorch without text decoder and encoder for a full text-to-image generation pipeline.

In [ ]:
```
import torch
import torchvision
import matplotlib.pyplot as plt
import torch.nn.functional as F
from torchvision import datasets, transforms
# from torchvision.transforms import Compose, ToTensor, Lambda, Resize, CenterCr
from torch.utils.data import DataLoader
import numpy as np
from torch import nn
import math
```

In [ ]:
```
# Generates 150 samples of 25 columns x 10 rows of images
def show(dataset, num_sample=150, cols=25, rows=10):
    plt.figure(figsize=(15, 15))
    for i, img in enumerate(dataset):
        if i == num_sample:
            break
        plt.subplot(num_sample // rows + 1, cols, i + 1)
        plt.axis('off')
        plt.imshow(img[0])

# Download the dataset
# *WARNING:* This will take a while to download (depending on connection speed)
data = torchvision.datasets.CelebA(root='', split="train", download=True)
```

```
# Show the first 150 samples
show(data)
```

Step 1 - Forward Diffusion Process

The linear schedule used in the forward diffusion process to calculate the alphas, betas, diffusion and posterior.

In [ ]:

```
# A linear schedule as proposed in https://arxiv.org/pdf/2102.09672.pdf
def linear_beta_schedule(timesteps):
    beta_start = 0.0001
    beta_end = 0.02

    return torch.linspace(beta_start, beta_end, timesteps)

# A cosine schedule as proposed in https://arxiv.org/abs/2102.09672.pdf
def cosine_beta_schedule(timesteps, s=0.008):
    steps = timesteps + 1
    x = torch.linspace(0, timesteps, steps)
    alphas_cumprod = torch.cos(((x / timesteps) + s) / (1 + s) * torch.pi * 0.5)
    alphas_cumprod = alphas_cumprod / alphas_cumprod[0]
    betas = 1 - (alphas_cumprod[1:] / alphas_cumprod[:-1])

    return torch.clip(betas, 0.0001, 0.9999)

# A quadratic schedule
def quadratic_beta_schedule(timesteps):
    beta_start = 0.0001
    beta_end = 0.02

    return torch.linspace(beta_start**0.5, beta_end**0.5, timesteps) ** 2

# A sigmoid schedule
def sigmoid_beta_schedule(timesteps):
    beta_start = 0.0001
    beta_end = 0.02
    betas = torch.linspace(-6, 6, timesteps)

    return torch.sigmoid(betas) * (beta_end - beta_start) + beta_start

# Returns a specific index t of a passed list of values vals while considering t
def get_index_from_list(vals, t, x_shape):
    batch_size = t.shape[0]
    out = vals.gather(-1, t.cpu())

    return out.reshape(batch_size, *((1,) * (len(x_shape) - 1))).to(t.device)

# Returns the diffusion model's forward diffusion sample, taking an image x_0 an
def forward_diffusion_sample(x_0, t, device="cpu"):

    noise = torch.randn_like(x_0)

    sqrt_alphas_cumprod_t = get_index_from_list(sqrt_alphas_cumprod, t, x_0.shap

    sqrt_one_minus_alphas_cumprod_t = get_index_from_list(
        sqrt_one_minus_alphas_cumprod, t, x_0.shape
    )
```

```
    # mean + variance
    return sqrt_alphas_cumprod_t.to(device) * x_0.to(device) \
    + sqrt_one_minus_alphas_cumprod_t.to(device) * noise.to(device), noise.to(de


# Define beta schedule
T = 300
betas = linear_beta_schedule(timesteps=T)

# define alphas
alphas = 1. - betas
alphas_cumprod = torch.cumprod(alphas, axis=0)
alphas_cumprod_prev = F.pad(alphas_cumprod[:-1], (1, 0), value=1.0)
sqrt_recip_alphas = torch.sqrt(1.0 / alphas)

# Calculate for diffusion q(x_t | x_{t-1})
sqrt_alphas_cumprod = torch.sqrt(alphas_cumprod)
sqrt_one_minus_alphas_cumprod = torch.sqrt(1. - alphas_cumprod)

# Calculate for posterior q(x_{t-1} | x_t, x_0)
posterior_variance = betas * (1. - alphas_cumprod_prev) / (1. - alphas_cumprod)
```

Image Preprocessing Helper Functions

In [ ]:
```
# Parameters for the dataset with image size of 64x64, 128x128, 256x256
# These will be used to resize the images and test the models on different image
IMG_SIZE = 64
IMG_SIZE_128 = 128
IMG_SIZE_256 = 256

# Batch size for training and testing with 128 images per batch and 256 images p
BATCH_SIZE = 128
BATCH_SIZE_256 = 256
BATCH_SIZE_512 = 512


# The tensor transformer for the dataset
def load_transformed_dataset():
    transform = transforms.Compose([
        transforms.Resize((IMG_SIZE, IMG_SIZE)),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Lambda(lambda t: t * 2 - 1)
    ])

    data_transform = transform

    train = datasets.CelebA(root='', split="train", download=True, transform=dat

    test = datasets.CelebA(root='', split="test", download=True, transform=data_

    return torch.utils.data.ConcatDataset([train, test])


# Load the transformer dataset
data = load_transformed_dataset()

# Appends the data into a dataloader with a batch size of 128 or 256 depending c
```

```
dataloader = DataLoader(data, batch_size=BATCH_SIZE, shuffle=True, drop_last=Tru

# The reverse transformer for the dataset to show the images back to their origi
def reverse_tensor_img(image):
    reverse_transform = transforms.Compose([
        transforms.Lambda(lambda t: (t + 1) / 2),
        transforms.Lambda(lambda t: t.permute(1, 2, 0)),
        transforms.Lambda(lambda t: t*255),
        transforms.Lambda(lambda t: t.cpu().numpy().astype(np.uint8)),
        transforms.ToPILImage(),
    ])

    # Take first image of batch
    if len(image.shape) == 4:
        image = image[0, :, :, :]

    plt.imshow(reverse_transform(image))
```

Testing if the forward diffusion process is working correctly.

In [ ]:
```
# Convert reverse_tensor_img to a cuda tensor function
# reverse_tensor_img = torch.jit.script(reverse_tensor_img)

# Load a single image from the dataloader
image = next(iter(dataloader))[0]

# Add image dimensions for the graph, the amount of image steps and the step siz
plt.figure(figsize=(18, 18))
plt.axis('off')
num_images = 20
stepsize = int(T/num_images)

# Plot the image with the step size and show the image
for idx in range(0, T, stepsize):
    t = torch.Tensor([idx]).type(torch.int64)
    plt.subplot(1, num_images+1, (idx//stepsize) + 1)
    image, noise = forward_diffusion_sample(image, t)
    plt.axis('off')
    reverse_tensor_img(image)
```

Step 2 - Backward Diffusion Process (U-Net)

In [ ]:
```
# The convolutional block for the model
# The block consists of two convolutional layers with each one having its own ba
# The block also has a time embedding layer that is used to add the time embeddi
# The block also has skip connections using the time embedding layer and the con
class Block(nn.Module):
    def __init__(self, in_channel, out_channel, time_emb_dim, up=False):
        super().__init__()
        # Time embedding layer
        self.time_mlp =  nn.Linear(time_emb_dim, out_channel)

        # First convolutional layers
        # If up is true then add a convolutional transpose layer to upsample the
        if up:
            self.conv1 = nn.Conv2d(2*in_channel, out_channel, 3, padding=1)
            self.transform = nn.ConvTranspose2d(out_channel, out_channel, 4, 2,
        else:
```

```
            self.conv1 = nn.Conv2d(in_channel, out_channel, 3, padding=1)
            self.transform = nn.Conv2d(out_channel, out_channel, 4, 2, 1)

        # Second convolutional layer
        self.conv2 = nn.Conv2d(out_channel, out_channel, 3, padding=1)

        # Batch normalization layers for both convolutional layers
        self.bnorm1 = nn.BatchNorm2d(out_channel)
        self.bnorm2 = nn.BatchNorm2d(out_channel)

        # Relu activation function
        self.relu  = nn.ReLU()

    def forward(self, x, t, ):
        # First Conv
        h = self.bnorm1(self.relu(self.conv1(x)))
        # Time embedding
        time_emb = self.relu(self.time_mlp(t))
        # Extend last 2 dimensions
        time_emb = time_emb[(..., ) + (None, ) * 2]
        # Add time channel
        h = h + time_emb
        # Second Conv
        h = self.bnorm2(self.relu(self.conv2(h)))
        # Down or Upsample
        return self.transform(h)

# A sinusoidal time embedding layer as described in the paper https://arxiv.org/
class SinusoidalPositionEmbeddings(nn.Module):
    def __init__(self, dim):
        super().__init__()
        self.dim = dim

    def forward(self, time):
        device = time.device
        half_dim = self.dim // 2
        embeddings = math.log(10000) / (half_dim - 1)
        embeddings = torch.exp(torch.arange(half_dim, device=device) * -embeddir
        embeddings = time[:, None] * embeddings[None, :]
        embeddings = torch.cat((embeddings.sin(), embeddings.cos()), dim=-1)
        return embeddings

# A UNet architecture for the image denoising task with time embedding in each l
class SimpleUnet(nn.Module):
    def __init__(self):
        super().__init__()
        image_channels = 3 # RGB: 3 channels for RED, GREEN, BLUE
        down_channels = (64, 128, 256, 512, 1024) # Number of channels in each c
        up_channels = (1024, 512, 256, 128, 64) # Number of channels in each ups
        out_dim = 1 # 1x1 final of output channels
        time_emb_dim = 32 # Dimension of time embedding

        # Time embedding
        self.time_mlp = nn.Sequential(
                SinusoidalPositionEmbeddings(time_emb_dim),
                nn.Linear(time_emb_dim, time_emb_dim),
                nn.ReLU()
            )

        # Initial projection
```

```
        self.conv0 = nn.Conv2d(image_channels, down_channels[0], 3, padding=1)

        # Downsample
        self.downs = nn.ModuleList([Block(down_channels[i], down_channels[i+1],
                                    time_emb_dim) \
                    for i in range(len(down_channels)-1)])
        # Upsample
        self.ups = nn.ModuleList([Block(up_channels[i], up_channels[i+1], \
                                        time_emb_dim, up=True) \
                    for i in range(len(up_channels)-1)])

        # Final output 1x1 conv
        self.output = nn.Conv2d(up_channels[-1], 3, out_dim)

    def forward(self, x, timestep):
        # Embedd time
        t = self.time_mlp(timestep)
        # Initial conv
        x = self.conv0(x)
        # Unet
        residual_inputs = []
        for down in self.downs:
            x = down(x, t)
            residual_inputs.append(x)
        for up in self.ups:
            residual_x = residual_inputs.pop()
            # Add residual x as additional channels
            x = torch.cat((x, residual_x), dim=1)
            x = up(x, t)
        return self.output(x)

model = SimpleUnet()
print("Num params: ", sum(p.numel() for p in model.parameters()))
model
device = "cuda" if torch.cuda.is_available() else "cpu"
model.to(device=device)
```

Step 3 - Training, Lose, Sampling

In [ ]:

```
# A function to get the loss for the model given the input image and the timeste
def get_loss(model, x_0, t, type="l1"):
    if type == "l1":
        x_noisy, noise = forward_diffusion_sample(x_0, t, device)
        noise_pred = model(x_noisy, t)
        return F.l1_loss(noise, noise_pred)
    elif type == "l2":
        x_noisy, noise = forward_diffusion_sample(x_0, t, device)
        noise_pred = model(x_noisy, t)
        return F.mse_loss(noise, noise_pred)
    else:
        raise NotImplementedError()
```

Sampling

In [ ]:

```
# A sample that calls the model to predict the noise in the image and returns th
# Applies noise to this image, if we are not in the last step yet.
@torch.no_grad()
```

```
def sample_timestep(x, t):
    # Get noise from betas, timestep and image shape
    betas_t = get_index_from_list(betas, t, x.shape)
    sqrt_one_minus_alphas_cumprod_t = get_index_from_list(
        sqrt_one_minus_alphas_cumprod, t, x.shape
    )
    sqrt_recip_alphas_t = get_index_from_list(sqrt_recip_alphas, t, x.shape)

    # Call model (current image - noise prediction)
    model_mean = sqrt_recip_alphas_t * (
        x - betas_t * model(x, t) / sqrt_one_minus_alphas_cumprod_t
    )
    posterior_variance_t = get_index_from_list(posterior_variance, t, x.shape)

    if t == 0:
        return model_mean
    else:
        noise = torch.randn_like(x)
        return model_mean + torch.sqrt(posterior_variance_t) * noise

# A function to plot the denoised image at each timestep showing a 10 step diffu
@torch.no_grad()
def sample_plot_image():
    # Sample noise
    img_size = IMG_SIZE
    img = torch.randn((1, 3, img_size, img_size), device=device)
    plt.figure(figsize=(15,15))
    plt.axis('off')
    num_images = 10
    stepsize = int(T/num_images)

    for i in range(0,T)[::-1]:
        t = torch.full((1,), i, device=device, dtype=torch.long)
        img = sample_timestep(img, t)
        if i % stepsize == 0:
            plt.subplot(1, num_images, int(i/stepsize+1))
            reverse_tensor_img(img.detach().cpu())
    plt.show()

# A function to return an np array of the denoised image at each timestep showin
@torch.no_grad()
def sample_plot_FID():
    # Sample noise
    img_size = IMG_SIZE
    img = torch.randn((1, 3, img_size, img_size), device=device)
    plt.figure(figsize=(15,15))
    plt.axis('off')
    num_images = 10
    stepsize = int(T/num_images)

    for i in range(0,T)[::-1]:
        t = torch.full((1,), i, device=device, dtype=torch.long)
        img = sample_timestep(img, t)

    # return an np array of the image
    return img.detach().cpu().numpy()

# A function to plot our results for the model
@torch.no_grad()
def plot_results(results):
```

```python
    # Results is a list of tuples (loss, step) for each step
    loss, step = zip(*results)
    plt.plot(step, loss)
    # loss_step = np.array(results)
    # plt.plot(loss_step[:,1], loss_step[:,0])
    plt.xlabel("Training Step")
    plt.ylabel("Training Loss")
    plt.title("Loss per step")
    plt.savefig("adam_loss.png")
    plt.show()
```

FID SCORING - REDUNDANT DONT RUN

In [ ]:
```python
# # Implementations of an FID score for the UNet model process using the dataset
# from pytorch_fid.inception import InceptionV3
# from pytorch_fid.fid_score import calculate_frechet_distance
# from scipy import linalg


DEVICE = 'cuda' if torch.cuda.is_available() else 'cpu'

@torch.no_grad()
def cal_activation_statistics(images, pred, batch_size=BATCH_SIZE, dims=1024):
    # model.eval()
    act = np.empty((len(images), dims))

    if device == 'cuda':
        batch = images.cuda()
    else:
        batch = images

    # pred = model(batch)[0]

    if pred.size(2) != 1 or pred.size(3) != 1:
        pred = F.adaptive_avg_pool2d(pred, output_size=(1, 1))

    act = pred.cpu().data.numpy().reshape(pred.size(0), -1)
    mu = np.mean(act, axis=0)
    sigma = np.cov(act, rowvar=False)
    return mu, sigma

def calculate_frechet_distance(mu1, sigma1, mu2, sigma2, eps=1e-6):
    diff = mu1 - mu2
    covmean, _ = linalg.sqrtm(sigma1.dot(sigma2), disp=False)
    if not np.isfinite(covmean).all():
        msg = ('fid calculation produces singular product; '
               'adding %s to diagonal of cov estimates') % eps
        print(msg)
        offset = np.eye(sigma1.shape[0]) * eps
        covmean = linalg.sqrtm((sigma1 + offset).dot(sigma2 + offset))
    score = (diff.dot(diff) + np.trace(sigma1) + np.trace(sigma2) - 2 * np.trace
    return score

def fid_score(model, fk_img, t):
    # # Load the Inception v3 model
    # inception_model = torch.hub.load('pytorch/vision', 'inception_v3', pretrai
    # inception_model.eval()
    # Get the real image from the dataset
```

```
    # data = datasets.CelebA('data', split='test', download=True, target_type='a
    # dataloader = DataLoader(data, batch_size=1, shuffle=True)
    # image_real, _ = next(iter(dataloader))
    # # convert image_real to a np array
    # image_real = image_real.numpy()

    # fk_img = sample_plot_FID()

    # Grab the first image from sample_plot_FID() from np array
    first_img = fk_img[0]
    second_img = fk_img[1]

    # Calculate the mean and covariance for the real images
    real_mu, real_sigma = cal_activation_statistics(image_real, inception_model(

    x_noisy, noise = forward_diffusion_sample(fk_img, t, device)
    # Calculate the mean and covariance for the fake images
    fake_mu, fake_sigma = cal_activation_statistics(fk_img, inception_model())

    # Calculate the FID score
    fid = calculate_frechet_distance(real_mu, real_sigma, fake_mu, fake_sigma)

    # Print the FID score
    print('FID: %.3f' % fid)
```

Training

In [ ]:
```
# device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"CUDA Avaliable: {torch.cuda.is_available()}")

# Output the amount of parameters in the model and aviailable cuda devices
print("Num params: ", sum(p.numel() for p in model.parameters()))
print("Num devices: ", torch.cuda.device_count())
print(f"Device name: {torch.cuda.get_device_name(0)}")
print(f"Device CUDA capability: {torch.cuda.get_device_capability(0)}")
### Results
# The number of prameters in the model is outputted.
# The model is trained for 5 epochs at 1475 steps.
# The model is trained on a single GPU (NVIDIA A100 40GB).
```

In [ ]:
```
from torch.optim import Adam

model.to(device)

optimiser = Adam(model.parameters(), lr=0.001)
epochs = 4

loss_step = []

for epoch in range(epochs):
    print(f"Epoch {epoch}")
    print(f"Amount of steps in dataloader: {len(dataloader)}")
    print(f"Amount of batches in dataloader: {len(dataloader.dataset)}")
    print(f"Batch size: {dataloader.batch_size}")
    running_loss = 0.0
    for step, batch in enumerate(dataloader):
        optimiser.zero_grad()
```

```
        t = torch.randint(0, T, (BATCH_SIZE,), device=device).long()
        loss = get_loss(model, batch[0], t, "l1")
        loss.backward()
        optimiser.step()
        # fid_score(model, batch[0], t)
        # Print the loss every 150 steps
        if step % 10 == 0:
            # fid_score(model, batch[0], t)
            # Append the loss to a list with loss and step
            loss_step.append([loss.item(), step])
            # running_loss += loss.item() *
            print(f"Epoch {epoch} | step {step:03d} Loss: {loss.item()} ")

        # # Print the loss and image every 250 steps
        # if step % 250 == 0:
        #     # fid_score(model, batch[0], t)

        #     print(f"Epoch {epoch} | step {step:03d} Loss: {loss.item()} ")
        #     print(f"Done with epoch {epoch} and step {step:03d}")
        #     # plot_results(loss_step)
        #     sample_plot_image()


        if step == 1427 or step == 1426 or step == 1428 or step == 356:
            """ Final Output """
            print(f"The final epoch is {epoch} and the final step is {step}")
            print(F"The final loss is {loss.item()}")

            sample_plot_image()

        # # Save loss and step to a csv file called adam_loss.csv
        # with open("adam_loss.csv", "w") as f:
        #   writer = csv.writer(f)
        #   writer.writerows(loss_step)

        # # Make a plot from the loss and step
        # plot_results(loss_step)
    if epoch == 0:
        print(loss_step)
    if epoch == 1:
        print(loss_step)
    if epoch == 2:
        print(loss_step)

    # Once 100 epochs are done, save the model
    if epoch == 3:
        print(loss_step)
        torch.save(model.state_dict(), "model-adam.pt")
        print("Model saved!")
```

Training

In [ ]:
```
# device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"CUDA Avaliable: {torch.cuda.is_available()}")

# Output the amount of parameters in the model and aviailable cuda devices
print("Num params: ", sum(p.numel() for p in model.parameters()))
print("Num devices: ", torch.cuda.device_count())
print(f"Device name: {torch.cuda.get_device_name(0)}")
print(f"Device CUDA capability: {torch.cuda.get_device_capability(0)}")
### Results
# The number of prameters in the model is outputted.
# The model is trained for 5 epochs at 1475 steps.
# The model is trained on a single GPU (NVIDIA A100 40GB).
```

In [ ]:
```
from torch.optim import Adamax

model.to(device)

optimiser = Adamax(model.parameters(), lr=0.001)
epochs = 4

loss_step = []

for epoch in range(epochs):
    print(f"Epoch {epoch}")
    print(f"Amount of steps in dataloader: {len(dataloader)}")
    print(f"Amount of batches in dataloader: {len(dataloader.dataset)}")
    print(f"Batch size: {dataloader.batch_size}")
    running_loss = 0.0
    for step, batch in enumerate(dataloader):
        optimiser.zero_grad()

        t = torch.randint(0, T, (BATCH_SIZE,), device=device).long()
        loss = get_loss(model, batch[0], t, "l1")
        loss.backward()
        optimiser.step()
        # fid_score(model, batch[0], t)
        # Print the loss every 150 steps
        if step % 10 == 0:
            # fid_score(model, batch[0], t)
            # Append the loss to a list with loss and step
            loss_step.append([loss.item(), step])
            # running_loss += loss.item() *
            print(f"Epoch {epoch} | step {step:03d} Loss: {loss.item()} ")

        # Print the loss and image every 250 steps
        if step % 250 == 0:
            # fid_score(model, batch[0], t)

            print(f"Epoch {epoch} | step {step:03d} Loss: {loss.item()} ")
            print(f"Done with epoch {epoch} and step {step:03d}")
            # plot_results(loss_step)
            sample_plot_image()


        if step == 1427 or step == 1426 or step == 1428:
            """ Final Output """
```

```
            print(f"The final epoch is {epoch} and the final step is {step}")
            print(F"The final loss is {loss.item()}")

            sample_plot_image()

        # # Save loss and step to a csv file called adam_loss.csv
        # with open("adam_loss.csv", "w") as f:
        #   writer = csv.writer(f)
        #   writer.writerows(loss_step)

        # # Make a plot from the loss and step
        # plot_results(loss_step)
    if epoch == 0:
        print(loss_step)
    if epoch == 1:
        print(loss_step)
    if epoch == 2:
        print(loss_step)

    # Once 100 epochs are done, save the model
    if epoch == 3:
        print(loss_step)
        torch.save(model.state_dict(), "model-adamax.pt")
        print("Model saved!")
```

Training

In [ ]:
```
# device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"CUDA Avaliable: {torch.cuda.is_available()}")

# Output the amount of parameters in the model and aviailable cuda devices
print("Num params: ", sum(p.numel() for p in model.parameters()))
print("Num devices: ", torch.cuda.device_count())
print(f"Device name: {torch.cuda.get_device_name(0)}")
print(f"Device CUDA capability: {torch.cuda.get_device_capability(0)}")
### Results
# The number of prameters in the model is outputted.
# The model is trained for 5 epochs at 1475 steps.
# The model is trained on a single GPU (NVIDIA A100 40GB).
```

In [ ]:
```
from lion_pytorch import Lion

model.to(device)

optimiser = Lion(model.parameters(), lr=1e-4, weight_decay=1e-2)
epochs = 4

loss_step = []

for epoch in range(epochs):
    print(f"Epoch {epoch}")
    print(f"Amount of steps in dataloader: {len(dataloader)}")
    print(f"Amount of batches in dataloader: {len(dataloader.dataset)}")
    print(f"Batch size: {dataloader.batch_size}")
    running_loss = 0.0
    for step, batch in enumerate(dataloader):
        optimiser.zero_grad()

        t = torch.randint(0, T, (BATCH_SIZE,), device=device).long()
        loss = get_loss(model, batch[0], t, "l1")
        loss.backward()
        optimiser.step()
        # fid_score(model, batch[0], t)
        # Print the loss every 150 steps
        if step % 10 == 0:
            # fid_score(model, batch[0], t)
            # Append the loss to a list with loss and step
            loss_step.append([loss.item(), step])
            # running_loss += loss.item() *
            print(f"Epoch {epoch} | step {step:03d} Loss: {loss.item()} ")

        # Print the loss and image every 250 steps
        if step % 250 == 0:
            # fid_score(model, batch[0], t)

            print(f"Epoch {epoch} | step {step:03d} Loss: {loss.item()} ")
            print(f"Done with epoch {epoch} and step {step:03d}")
            # plot_results(loss_step)
            sample_plot_image()


        if step == 1427 or step == 1426 or step == 1428:
            """ Final Output """
```

```
            print(f"The final epoch is {epoch} and the final step is {step}")
            print(F"The final loss is {loss.item()}")

            sample_plot_image()

        # # Save loss and step to a csv file called adam_loss.csv
        # with open("adam_loss.csv", "w") as f:
        #   writer = csv.writer(f)
        #   writer.writerows(loss_step)

        # # Make a plot from the loss and step
        # plot_results(loss_step)
    if epoch == 0:
        print(loss_step)
    if epoch == 1:
        print(loss_step)
    if epoch == 2:
        print(loss_step)

    # Once 100 epochs are done, save the model
    if epoch == 3:
        print(loss_step)
        torch.save(model.state_dict(), "model-lion.pt")
        print("Model saved!")
```

Test environment config.

```
In [ ]: import sys
print(sys.executable)
import torch
print(torch.__file__)
print(torch.cuda.is_available())
from torch.utils import collect_env
print(collect_env.main())
```

Check if the environment has access to the NVIDIA A100 GPU.

```
In [ ]: !nvidia-smi
```

Diffusion Model - Pruning

A simple implementation of the diffusion model in PyTorch without text decoder and encoder for a full text-to-image generation pipeline.

```
In [ ]: import torch
import torchvision
import matplotlib.pyplot as plt
import torch.nn.functional as F
from torchvision import datasets, transforms
# from torchvision.transforms import Compose, ToTensor, Lambda, Resize, CenterCr
from torch.utils.data import DataLoader
import numpy as np
from torch import nn
import math
```

```
In [ ]: # Generates 150 samples of 25 columns x 10 rows of images
def show(dataset, num_sample=150, cols=25, rows=10):
    plt.figure(figsize=(15, 15))
    for i, img in enumerate(dataset):
        if i == num_sample:
            break
        plt.subplot(num_sample // rows + 1, cols, i + 1)
        plt.axis('off')
        plt.imshow(img[0])

# Download the dataset
# *WARNING:* This will take a while to download (depending on connection speed)
data = torchvision.datasets.CelebA(root='', split="train", download=True)

# Show the first 150 samples
show(data)
```

Step 1 - Forward Diffusion Process

Step 1.1 - The linear schedule used in the forward diffusion process to calculate the alphas, betas, diffusion and

posterior.

```
In [ ]: # A linear schedule as proposed in https://arxiv.org/pdf/2102.09672.pdf
def linear_beta_schedule(timesteps):
    beta_start = 0.0001
    beta_end = 0.02

    return torch.linspace(beta_start, beta_end, timesteps)

# A cosine schedule as proposed in https://arxiv.org/abs/2102.09672.pdf
def cosine_beta_schedule(timesteps, s=0.008):
    steps = timesteps + 1
    x = torch.linspace(0, timesteps, steps)
    alphas_cumprod = torch.cos(((x / timesteps) + s) / (1 + s) * torch.pi * 0.5)
    alphas_cumprod = alphas_cumprod / alphas_cumprod[0]
    betas = 1 - (alphas_cumprod[1:] / alphas_cumprod[:-1])

    return torch.clip(betas, 0.0001, 0.9999)

# A quadratic schedule
def quadratic_beta_schedule(timesteps):
    beta_start = 0.0001
    beta_end = 0.02

    return torch.linspace(beta_start**0.5, beta_end**0.5, timesteps) ** 2

# A sigmoid schedule
def sigmoid_beta_schedule(timesteps):
    beta_start = 0.0001
    beta_end = 0.02
    betas = torch.linspace(-6, 6, timesteps)

    return torch.sigmoid(betas) * (beta_end - beta_start) + beta_start

# Returns a specific index t of a passed list of values vals while considering t
def get_index_from_list(vals, t, x_shape):
    batch_size = t.shape[0]
    out = vals.gather(-1, t.cpu())

    return out.reshape(batch_size, *((1,) * (len(x_shape) - 1))).to(t.device)

# Returns the diffusion model's forward diffusion sample, taking an image x_0 an
def forward_diffusion_sample(x_0, t, device="cpu"):

    noise = torch.randn_like(x_0)

    sqrt_alphas_cumprod_t = get_index_from_list(sqrt_alphas_cumprod, t, x_0.shap

    sqrt_one_minus_alphas_cumprod_t = get_index_from_list(
        sqrt_one_minus_alphas_cumprod, t, x_0.shape
    )

    # mean + variance
    return sqrt_alphas_cumprod_t.to(device) * x_0.to(device) \
    + sqrt_one_minus_alphas_cumprod_t.to(device) * noise.to(device), noise.to(de


# Define beta schedule
T = 300
```

```
betas = linear_beta_schedule(timesteps=T)

# define alphas
alphas = 1. - betas
alphas_cumprod = torch.cumprod(alphas, axis=0)
alphas_cumprod_prev = F.pad(alphas_cumprod[:-1], (1, 0), value=1.0)
sqrt_recip_alphas = torch.sqrt(1.0 / alphas)

# Calculate for diffusion q(x_t | x_{t-1})
sqrt_alphas_cumprod = torch.sqrt(alphas_cumprod)
sqrt_one_minus_alphas_cumprod = torch.sqrt(1. - alphas_cumprod)

# Calculate for posterior q(x_{t-1} | x_t, x_0)
posterior_variance = betas * (1. - alphas_cumprod_prev) / (1. - alphas_cumprod)
```

Image Preprocessing Helper Functions

In [ ]:

```
# Parameters for the dataset with image size of 64x64, 128x128, 256x256
# These will be used to resize the images and test the models on different image
IMG_SIZE = 64
IMG_SIZE_128 = 128
IMG_SIZE_256 = 256

# Batch size for training and testing with 128 images per batch and 256 images p
BATCH_SIZE = 128
BATCH_SIZE_256 = 256

# The tensor transformer for the dataset
def load_transformed_dataset():
    transform = transforms.Compose([
        transforms.Resize((IMG_SIZE, IMG_SIZE)),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Lambda(lambda t: t * 2 - 1)
    ])

    data_transform = transform

    train = datasets.CelebA(root='', split="train", download=True, transform=dat

    test = datasets.CelebA(root='', split="test", download=True, transform=data_

    return torch.utils.data.ConcatDataset([train, test])


# Load the transformer dataset
data = load_transformed_dataset()

# Appends the data into a dataloader with a batch size of 128 or 256 depending c
dataloader = DataLoader(data, batch_size=BATCH_SIZE, shuffle=True, drop_last=Tru

# The reverse transformer for the dataset to show the images back to their origi
def reverse_tensor_img(image):
    reverse_transform = transforms.Compose([
        transforms.Lambda(lambda t: (t + 1) / 2),
        transforms.Lambda(lambda t: t.permute(1, 2, 0)),
        transforms.Lambda(lambda t: t*255),
        transforms.Lambda(lambda t: t.cpu().numpy().astype(np.uint8)),
        transforms.ToPILImage(),
```

```
    ])

    # Take first image of batch
    if len(image.shape) == 4:
        image = image[0, :, :, :]

    plt.imshow(reverse_transform(image))
```

Testing if the forward diffusion process is working correctly.

In [ ]:
```
# Convert reverse_tensor_img to a cuda tensor function
# reverse_tensor_img = torch.jit.script(reverse_tensor_img)

# Load a single image from the dataloader
image = next(iter(dataloader))[0]

# Add image dimensions for the graph, the amount of image steps and the step siz
plt.figure(figsize=(18, 18))
plt.axis('off')
num_images = 20
stepsize = int(T/num_images)

# Plot the image with the step size and show the image
for idx in range(0, T, stepsize):
    t = torch.Tensor([idx]).type(torch.int64)
    plt.subplot(1, num_images+1, (idx//stepsize) + 1)
    image, noise = forward_diffusion_sample(image, t)
    plt.axis('off')
    reverse_tensor_img(image)
```

Step 2 - Backward Diffusion Process (U-Net)

In [ ]:
```
# The convolutional block for the model
# The block consists of two convolutional layers with each one having its own ba
# The block also has a time embedding layer that is used to add the time embeddi
# The block also has skip connections using the time embedding layer and the con
class Block(nn.Module):
    def __init__(self, in_channel, out_channel, time_emb_dim, up=False):
        super().__init__()
        # Time embedding layer
        self.time_mlp =  nn.Linear(time_emb_dim, out_channel)

        # First convolutional layers
        # If up is true then add a convolutional transpose layer to upsample the
        if up:
            self.conv1 = nn.Conv2d(2*in_channel, out_channel, 3, padding=1)
            self.transform = nn.ConvTranspose2d(out_channel, out_channel, 4, 2,
        else:
            self.conv1 = nn.Conv2d(in_channel, out_channel, 3, padding=1)
            self.transform = nn.Conv2d(out_channel, out_channel, 4, 2, 1)

        # Second convolutional layer
        self.conv2 = nn.Conv2d(out_channel, out_channel, 3, padding=1)

        # Batch normalization layers for both convolutional layers
        self.bnorm1 = nn.BatchNorm2d(out_channel)
        self.bnorm2 = nn.BatchNorm2d(out_channel)
```

```
        # Relu activation function
        self.relu  = nn.ReLU()

    def forward(self, x, t, ):
        # First Conv
        h = self.bnorm1(self.relu(self.conv1(x)))
        # Time embedding
        time_emb = self.relu(self.time_mlp(t))
        # Extend last 2 dimensions
        time_emb = time_emb[(..., ) + (None, ) * 2]
        # Add time channel
        h = h + time_emb
        # Second Conv
        h = self.bnorm2(self.relu(self.conv2(h)))
        # Down or Upsample
        return self.transform(h)

# A sinusoidal time embedding layer as described in the paper https://arxiv.org/
class SinusoidalPositionEmbeddings(nn.Module):
    def __init__(self, dim):
        super().__init__()
        self.dim = dim

    def forward(self, time):
        device = time.device
        half_dim = self.dim // 2
        embeddings = math.log(10000) / (half_dim - 1)
        embeddings = torch.exp(torch.arange(half_dim, device=device) * -embeddir
        embeddings = time[:, None] * embeddings[None, :]
        embeddings = torch.cat((embeddings.sin(), embeddings.cos()), dim=-1)
        return embeddings

# A UNet architecture for the image denoising task with time embedding in each l
class SimpleUnet(nn.Module):
    def __init__(self):
        super().__init__()
        image_channels = 3 # RGB: 3 channels for RED, GREEN, BLUE
        down_channels = (64, 128, 256, 512, 1024) # Number of channels in each c
        up_channels = (1024, 512, 256, 128, 64) # Number of channels in each ups
        out_dim = 1 # 1x1 final of output channels
        time_emb_dim = 32 # Dimension of time embedding

        # Time embedding
        self.time_mlp = nn.Sequential(
                SinusoidalPositionEmbeddings(time_emb_dim),
                nn.Linear(time_emb_dim, time_emb_dim),
                nn.ReLU()
            )

        # Initial projection
        self.conv0 = nn.Conv2d(image_channels, down_channels[0], 3, padding=1)

        # Downsample
        self.downs = nn.ModuleList([Block(down_channels[i], down_channels[i+1],
                                    time_emb_dim) \
                    for i in range(len(down_channels)-1)])
        # Upsample
        self.ups = nn.ModuleList([Block(up_channels[i], up_channels[i+1], \
                                        time_emb_dim, up=True) \
                    for i in range(len(up_channels)-1)])
```

```
        # Final output 1x1 conv
        self.output = nn.Conv2d(up_channels[-1], 3, out_dim)

    def forward(self, x, timestep):
        # Embedd time
        t = self.time_mlp(timestep)
        # Initial conv
        x = self.conv0(x)
        # Unet
        residual_inputs = []
        for down in self.downs:
            x = down(x, t)
            residual_inputs.append(x)
        for up in self.ups:
            residual_x = residual_inputs.pop()
            # Add residual x as additional channels
            x = torch.cat((x, residual_x), dim=1)
            x = up(x, t)
        return self.output(x)

model = SimpleUnet()
print("Num params: ", sum(p.numel() for p in model.parameters()))
model
device = "cuda" if torch.cuda.is_available() else "cpu"
model.to(device=device)
```

Pruining the UNet

In [ ]:
```
import torch.nn.utils.prune as prune
```

In [ ]:
```
# Local parameters to prune
module_initial_conv = model.conv0
# print(list(module_initial_conv.named_parameters()))
# print(list(module_initial_conv.named_buffers()))

module_downs = model.downs
# print(list(module_downs.named_parameters()))
# print(list(module_downs.named_buffers()))

module_ups = model.ups
# print(list(module_ups.named_parameters()))
# print(list(module_ups.named_buffers()))

module_output = model.output
# print(list(module_output.named_parameters()))
# print(list(module_output.named_buffers()))
```

In [ ]:
```
global_weight_parameters_to_prune = (
    (model.conv0, 'weight'),
    (model.output, 'weight'),
)

global_weight_parameters_to_prune_ups_downs = [
    (model, "weight") for model in filter(lambda m: type(m) == torch.nn.Conv2d,
]
```

```
print("Global weight parameters to prune: ", global_weight_parameters_to_prune)
print("Global weight parameters to prune ups downs: ", global_weight_parameters_
```

In [ ]:
```
global_bias_parameters_to_prune = (
    (model.conv0, 'bias'),
    (model.output, 'bias'),
)

global_bias_parameters_to_prune_ups_downs = [
    (model, "bias") for model in filter(lambda m: type(m) == torch.nn.Conv2d, mo
]

print("Global bias parameters to prune: ", global_bias_parameters_to_prune)
print("Global bias parameters to prune ups downs: ", global_bias_parameters_to_p
```

Unstructured Pruning

Random Unstructured Pruning

In [ ]:
```
# Here we prune 50% of the parameters in each of the modules of our UNet
# We prune the weights and biases separately to allow for local pruning tests
# We comment out the local pruning tests to avoid running them every time

# Weight pruning
prune.random_unstructured(module_initial_conv, name="weight", amount=0.5)
prune.random_unstructured(module_output, name="weight", amount=0.5)

for module in module_downs:
    prune.random_unstructured(module.conv1, name="weight", amount=0.5)
    prune.random_unstructured(module.conv2, name="weight", amount=0.5)
for module in module_ups:
    prune.random_unstructured(module.conv1, name="weight", amount=0.5)
    prune.random_unstructured(module.conv2, name="weight", amount=0.5)

# Bias pruning
prune.random_unstructured(module_initial_conv, name="bias", amount=0.5)
prune.random_unstructured(module_output, name="bias", amount=0.5)


for module in module_downs:
    prune.random_unstructured(module.conv1, name="bias", amount=0.5)
    prune.random_unstructured(module.conv2, name="bias", amount=0.5)
for module in module_ups:
    prune.random_unstructured(module.conv1, name="bias", amount=0.5)
    prune.random_unstructured(module.conv2, name="bias", amount=0.5)
```

L1 Unstructured Pruning

In [ ]:
```
# Weight pruning
prune.l1_unstructured(module_initial_conv, name="weight", amount=0.5)
prune.l1_unstructured(module_output, name="weight", amount=0.5)


for module in module_downs:
    prune.l1_unstructured(module.conv1, name="weight", amount=0.5)
    prune.l1_unstructured(module.conv2, name="weight", amount=0.5)
for module in module_ups:
```

```
    prune.l1_unstructured(module.conv1, name="weight", amount=0.5)
    prune.l1_unstructured(module.conv2, name="weight", amount=0.5)

# Bias pruning
prune.l1_unstructured(module_initial_conv, name="bias", amount=0.5)
prune.l1_unstructured(module_output, name="bias", amount=0.5)


for module in module_downs:
    prune.l1_unstructured(module.conv1, name="bias", amount=0.5)
    prune.l1_unstructured(module.conv2, name="bias", amount=0.5)
for module in module_ups:
    prune.l1_unstructured(module.conv1, name="bias", amount=0.5)
    prune.l1_unstructured(module.conv2, name="bias", amount=0.5)
```

Global Unstructured Pruning

In [ ]:
```
# prune.global_unstructured(
#     global_weight_parameters_to_prune,
#     pruning_method=prune.L1Unstructured,
#     amount=0.2,
# )

# prune.global_unstructured(
#     global_weight_parameters_to_prune_ups_downs,
#     pruning_method=prune.L1Unstructured,
#     amount=0.2,
# )

# # print(list(module_downs.named_parameters()))
# # print(list(module.named_buffers()))
# print(model.state_dict().keys())
```

Structured Pruning

In [ ]:
```
prune.ln_structured(module_initial_conv, name="weight", amount=0.5, n=1, dim=0)
prune.ln_structured(module_output, name="weight", amount=0.5, n=1, dim=0)

for module in module_downs:
    prune.ln_structured(module.conv1, name="weight", amount=0.5, n=1, dim=0)
    prune.ln_structured(module.conv2, name="weight", amount=0.5, n=1, dim=0)
for module in module_ups:
    prune.ln_structured(module.conv1, name="weight", amount=0.5, n=1, dim=0)
    prune.ln_structured(module.conv2, name="weight", amount=0.5, n=1, dim=0)
```

In [ ]:
```
# Only run to get the parameters Pruned
# WARNING: This will generate a lot of output
print(list(module_initial_conv.named_parameters()))
print(list(module_downs.named_buffers()))
print(list(module_ups.named_buffers()))
print(list(module_output.named_parameters()))
```

In [ ]:
```
# Calculate the sparsity of the model after pruning

print("Sparsity in conv0.weight: {:.2f}%".format(
    100. * float(torch.sum(model.conv0.weight == 0))
    / float(model.conv0.weight.nelement())
```

```
))

print("Sparsity in output.weight: {:.2f}%".format(
    100. * float(torch.sum(model.output.weight == 0))
    / float(model.output.weight.nelement())
))

for module in module_downs:
    print("Sparsity in conv1.weight down channels: {:.2f}%".format(
        100. * float(torch.sum(module.conv1.weight == 0))
        / float(module.conv1.weight.nelement())
    ))
    print("Sparsity in conv2.weight down channels: {:.2f}%".format(
        100. * float(torch.sum(module.conv2.weight == 0))
        / float(module.conv2.weight.nelement())
    ))

for module in module_ups:
    print("Sparsity in conv1.weight up channels: {:.2f}%".format(
        100. * float(torch.sum(module.conv1.weight == 0))
        / float(module.conv1.weight.nelement())
    ))
    print("Sparsity in conv2.weight up channels: {:.2f}%".format(
        100. * float(torch.sum(module.conv2.weight == 0))
        / float(module.conv2.weight.nelement())
    ))

# Global pruning sparsity for weights
sum_down = 0
sum_up = 0
for module in module_downs:
    sum_down += torch.sum(module.conv1.weight == 0)
    sum_down += torch.sum(module.conv2.weight == 0)
for module in module_ups:
    sum_up += torch.sum(module.conv1.weight == 0)
    sum_up += torch.sum(module.conv2.weight == 0)

# nelement for down and up convs
nelement_down = 0
nelement_up = 0
for module in module_downs:
    nelement_down += module.conv1.weight.nelement()
    nelement_down += module.conv2.weight.nelement()
for module in module_ups:
    nelement_up += module.conv1.weight.nelement()
    nelement_up += module.conv2.weight.nelement()

print(
    "Global sparsity: {:.2f}%".format(
        100. * float(
            torch.sum(model.conv0.weight == 0)
            + torch.sum(model.output.weight == 0)
            + sum_down
            + sum_up
        )
        / float(
            model.conv0.weight.nelement()
            + model.output.weight.nelement()
            + nelement_down
            + nelement_up
```

```
        )
    )
)
```

Step 3 - Training, Lose, Sampling

In [ ]:
```
# A function to get the loss for the model given the input image and the timeste
def get_loss(model, x_0, t, type="l1"):
    if type == "l1":
        x_noisy, noise = forward_diffusion_sample(x_0, t, device)
        noise_pred = model(x_noisy, t)
        return F.l1_loss(noise, noise_pred)
    elif type == "l2":
        x_noisy, noise = forward_diffusion_sample(x_0, t, device)
        noise_pred = model(x_noisy, t)
        return F.mse_loss(noise, noise_pred)
    else:
        raise NotImplementedError()
```

Sampling

In [ ]:
```
# A sample that calls the model to predict the noise in the image and returns th
# Applies noise to this image, if we are not in the last step yet.
@torch.no_grad()
def sample_timestep(x, t):
    # Get noise from betas, timestep and image shape
    betas_t = get_index_from_list(betas, t, x.shape)
    sqrt_one_minus_alphas_cumprod_t = get_index_from_list(
        sqrt_one_minus_alphas_cumprod, t, x.shape
    )
    sqrt_recip_alphas_t = get_index_from_list(sqrt_recip_alphas, t, x.shape)

    # Call model (current image - noise prediction)
    model_mean = sqrt_recip_alphas_t * (
        x - betas_t * model(x, t) / sqrt_one_minus_alphas_cumprod_t
    )
    posterior_variance_t = get_index_from_list(posterior_variance, t, x.shape)

    if t == 0:
        return model_mean
    else:
        noise = torch.randn_like(x)
        return model_mean + torch.sqrt(posterior_variance_t) * noise

# A function to plot the denoised image at each timestep showing a 10 step diffu
@torch.no_grad()
def sample_plot_image():
    # Sample noise
    img_size = IMG_SIZE
    img = torch.randn((1, 3, img_size, img_size), device=device)
    plt.figure(figsize=(15,15))
    plt.axis('off')
    num_images = 10
    stepsize = int(T/num_images)

    for i in range(0,T)[::-1]:
        t = torch.full((1,), i, device=device, dtype=torch.long)
```

```
        img = sample_timestep(img, t)
        if i % stepsize == 0:
            plt.subplot(1, num_images, int(i/stepsize+1))
            reverse_tensor_img(img.detach().cpu())
    plt.show()

# A function to return an np array of the denoised image at each timestep showin
@torch.no_grad()
def sample_plot_FID():
    # Sample noise
    img_size = IMG_SIZE
    img = torch.randn((1, 3, img_size, img_size), device=device)
    plt.figure(figsize=(15,15))
    plt.axis('off')
    num_images = 10
    stepsize = int(T/num_images)

    for i in range(0,T)[::-1]:
        t = torch.full((1,), i, device=device, dtype=torch.long)
        img = sample_timestep(img, t)

    # return an np array of the image
    return img.detach().cpu().numpy()

# A function to plot our results for the model
@torch.no_grad()
def plot_results(results):

    # Results is a list of tuples (loss, step) for each step
    loss, step = zip(*results)
    plt.plot(step, loss)
    # loss_step = np.array(results)
    # plt.plot(loss_step[:,1], loss_step[:,0])
    plt.xlabel("Training Step")
    plt.ylabel("Training Loss")
    plt.title("Loss per step")
    plt.savefig("adam_loss.png")
    plt.show()
```

Training

In [ ]:
```
# device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"CUDA Avaliable: {torch.cuda.is_available()}")

# Output the amount of parameters in the model and aviailable cuda devices
print("Num params: ", sum(p.numel() for p in model.parameters()))
print("Num devices: ", torch.cuda.device_count())
print(f"Device name: {torch.cuda.get_device_name(0)}")
print(f"Device CUDA capability: {torch.cuda.get_device_capability(0)}")
### Results
# The number of prameters in the model is outputted.
# The model is trained for 5 epochs at 1475 steps.
# The model is trained on a single GPU (NVIDIA A100 40GB).
```

In [ ]:
```
from torch.optim import Adam

model.to(device)
```

```python
optimiser = Adam(model.parameters(), lr=0.001)
epochs = 1

loss_step = []

for epoch in range(epochs):
    print(f"Epoch {epoch}")
    print(f"Amount of steps in dataloader: {len(dataloader)}")
    print(f"Amount of batches in dataloader: {len(dataloader.dataset)}")
    print(f"Batch size: {dataloader.batch_size}")
    running_loss = 0.0
    for step, batch in enumerate(dataloader):
        optimiser.zero_grad()

        t = torch.randint(0, T, (BATCH_SIZE,), device=device).long()
        loss = get_loss(model, batch[0], t, "l1")
        loss.backward()
        optimiser.step()
        # fid_score(model, batch[0], t)
        # Print the loss every 150 steps
        if step % 10 == 0:
            # fid_score(model, batch[0], t)
            # Append the loss to a list with loss and step
            loss_step.append([loss.item(), step])
            # running_loss += loss.item() *
            print(f"Epoch {epoch} | step {step:03d} Loss: {loss.item()} ")

        # Print the loss and image every 250 steps
        if step % 250 == 0:
            # fid_score(model, batch[0], t)

            print(f"Epoch {epoch} | step {step:03d} Loss: {loss.item()} ")
            print(f"Done with epoch {epoch} and step {step:03d}")
            # plot_results(loss_step)
            sample_plot_image()


        if step == 1427 or step == 1426 or step == 1428:
            """ Final Output """
            print(f"The final epoch is {epoch} and the final step is {step}")
            print(F"The final loss is {loss.item()}")

            sample_plot_image()

        # # Save loss and step to a csv file called adam_loss.csv
        # with open("adam_loss.csv", "w") as f:
        #   writer = csv.writer(f)
        #   writer.writerows(loss_step)

        # # Make a plot from the loss and step
        # plot_results(loss_step)
    if epoch == 0:
        print(loss_step)
    loss_step = []
    if epoch == 1:
        print(loss_step)
    loss_step = []
    if epoch == 2:
        print(loss_step)
    loss_step = []
```

```
# Once 100 epochs are done, save the model
if epoch == 1:
    print(loss_step)
    torch.save(model.state_dict(), "model-adam.pt")
    print("Model saved!")
```

Test environment config.

In [ ]:
```
import sys
print(sys.executable)
import torch
print(torch.__file__)
print(torch.cuda.is_available())
from torch.utils import collect_env
print(collect_env.main())
```

Check if the environment has access to the NVIDIA A100 GPU.

In [ ]:
```
!nvidia-smi
```

Diffusion Model

A simple implementation of the diffusion model in PyTorch without text decoder and encoder for a full text-to-image generation pipeline.

In [1]:
```
import torch
import torchvision
import matplotlib.pyplot as plt
import torch.nn.functional as F
from torchvision import datasets, transforms
# from torchvision.transforms import Compose, ToTensor, Lambda, Resize, CenterCr
from torch.utils.data import DataLoader
import numpy as np
from torch import nn
import math
```

```
e:\6CCS3\PRJ\codebase\6CCS2PRJ\.venv\lib\site-packages\tqdm\auto.py:22: TqdmWar
ning: IProgress not found. Please update jupyter and ipywidgets. See https://ip
ywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

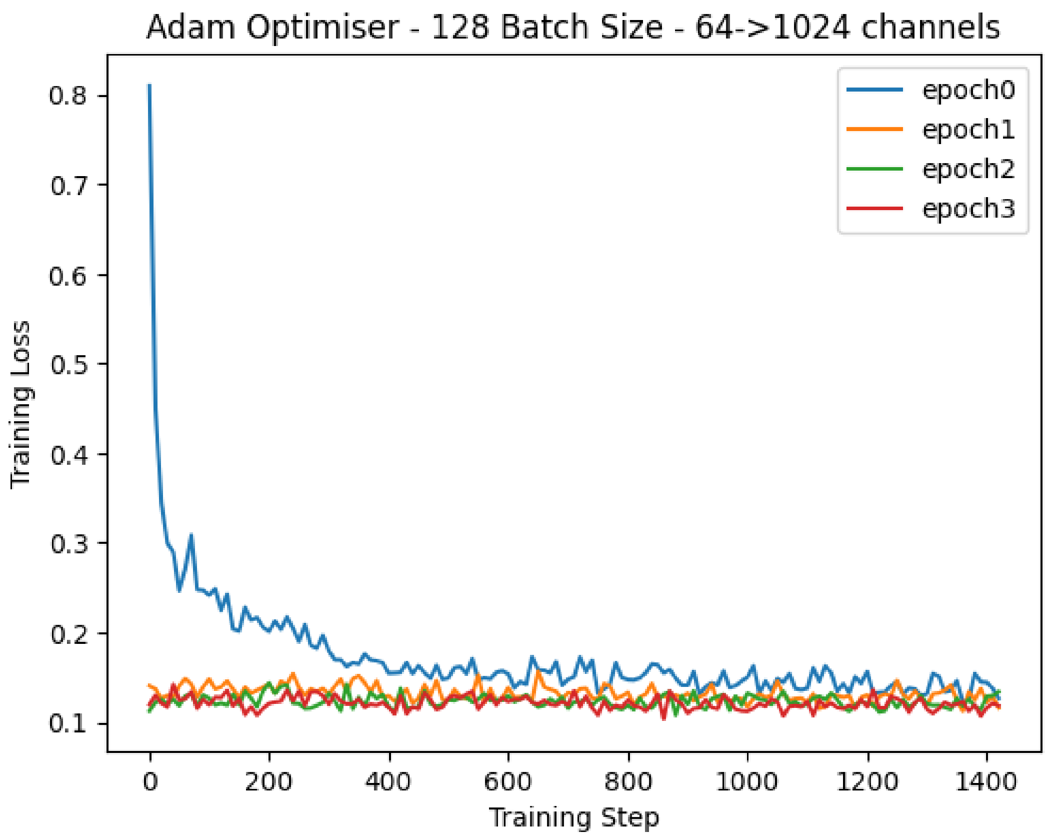
Results

Adam optimizer with learning rate 0.0001 and 3 epochs.

In [32]:
```
epoch0 = [[0.8100863695144653, 0], [0.4510430693626404, 10], [0.3447503745555877
epoch1 = [[0.14068108797073364, 0], [0.13694602251052856, 10], [0.12637534737586
epoch2 = [[0.1122107058763504, 0], [0.12206485122442245, 10], [0.125309467315673
epoch3 = [[0.11938553303480148, 0], [0.12963160872459412, 10], [0.12099280208349
# Convert 2d array to 1d array x and y for each epoch
# for i in range(0, 4):
#     loss, step = zip(*epoch[i
loss, step = zip(*epoch0)
loss1, step1 = zip(*epoch1)
loss2, step2 = zip(*epoch2)
loss3, step3 = zip(*epoch3)
plt.plot(step, loss, label='epoch0')
plt.plot(step1, loss1, label='epoch1')
```

```
plt.plot(step2, loss2, label='epoch2')
plt.plot(step3, loss3, label='epoch3')
plt.xlabel("Training Step")
plt.ylabel("Training Loss")
plt.title("Adam Optimiser - 128 Batch Size - 64->1024 channels")
plt.legend()
```

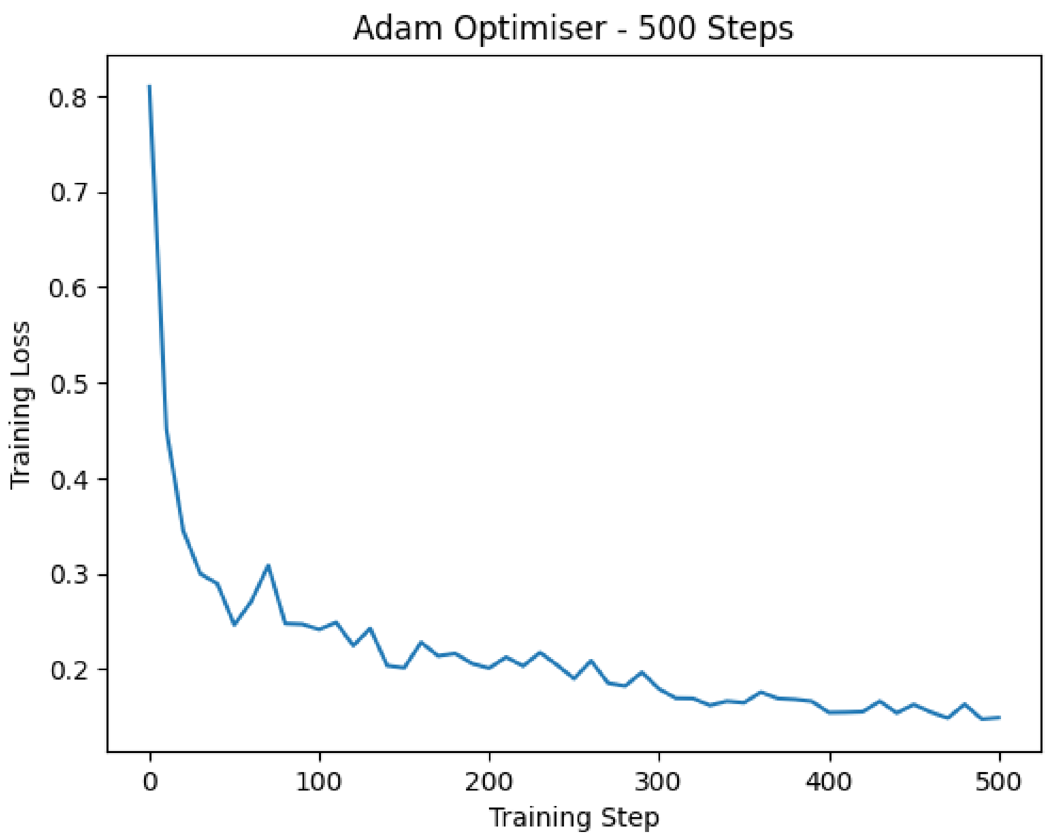
Out[32]: <matplotlib.legend.Legend at 0x1fd04e5b700>



In [35]:

```
epoch500steps = [[0.8100863695144653, 0], [0.4510430693626404, 10], [0.344750374

loss500, step500 = zip(*epoch500steps)
plt.plot(step500, loss500)
plt.xlabel("Training Step")
plt.ylabel("Training Loss")
plt.title("Adam Optimiser - 500 Steps")
```

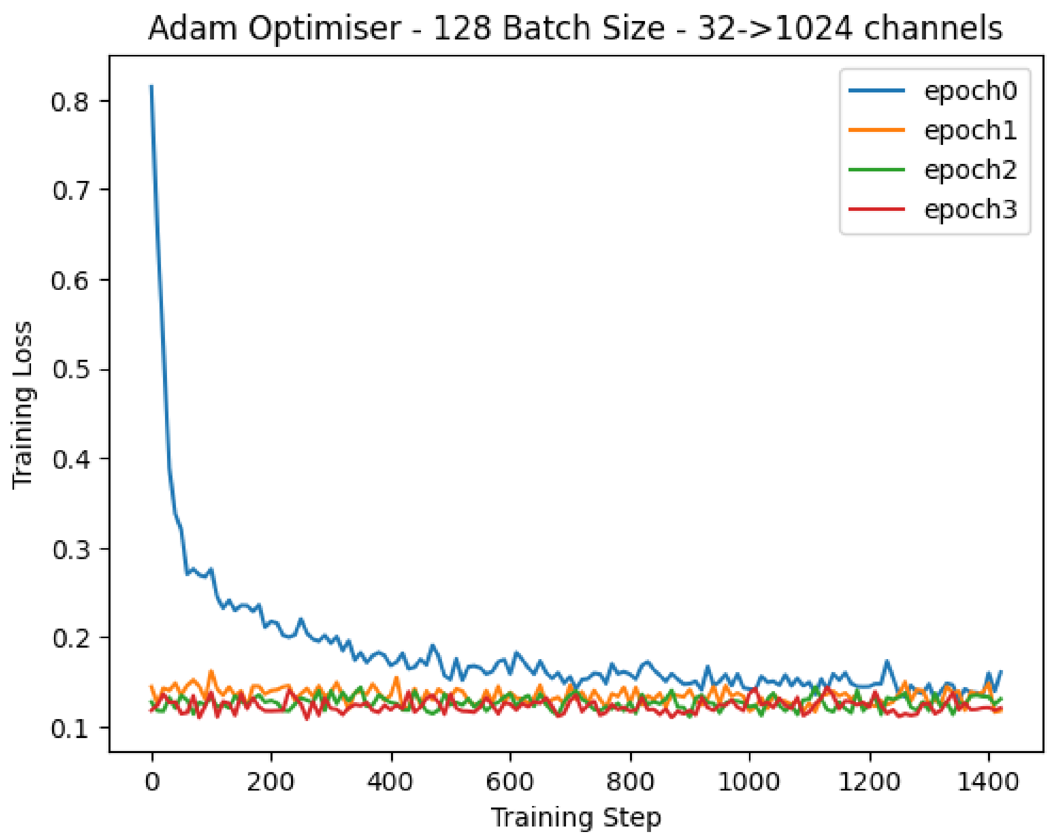
Out[35]: Text(0.5, 1.0, 'Adam Optimiser - 500 Steps')



In [25]:

```
epoch0 = [[0.814482569694519, 0], [0.653007984161377, 10], [0.5247682332992554, 
epoch1 = [[0.14496386051177979, 0], [0.12626831233501434, 10], [0.14321219921112
epoch2 = [[0.12785810232162476, 0], [0.11803515255451202, 10], [0.11763021349906
epoch3 = [[0.11829980462789536, 0], [0.12433420121669769, 10], [0.13653741776943


# Convert 2d array to 1d array x and y for each epoch
# for i in range(0, 4):
#     loss, step = zip(*epoch[i
loss, step = zip(*epoch0)
loss1, step1 = zip(*epoch1)
loss2, step2 = zip(*epoch2)
loss3, step3 = zip(*epoch3)
plt.plot(step, loss, label='epoch0')
plt.plot(step1, loss1, label='epoch1')
plt.plot(step2, loss2, label='epoch2')
plt.plot(step3, loss3, label='epoch3')
plt.xlabel("Training Step")
plt.ylabel("Training Loss")
plt.title("Adam Optimiser - 128 Batch Size - 32->1024 channels")
plt.legend()
```

Out[25]: <matplotlib.legend.Legend at 0x1fd0369c100>

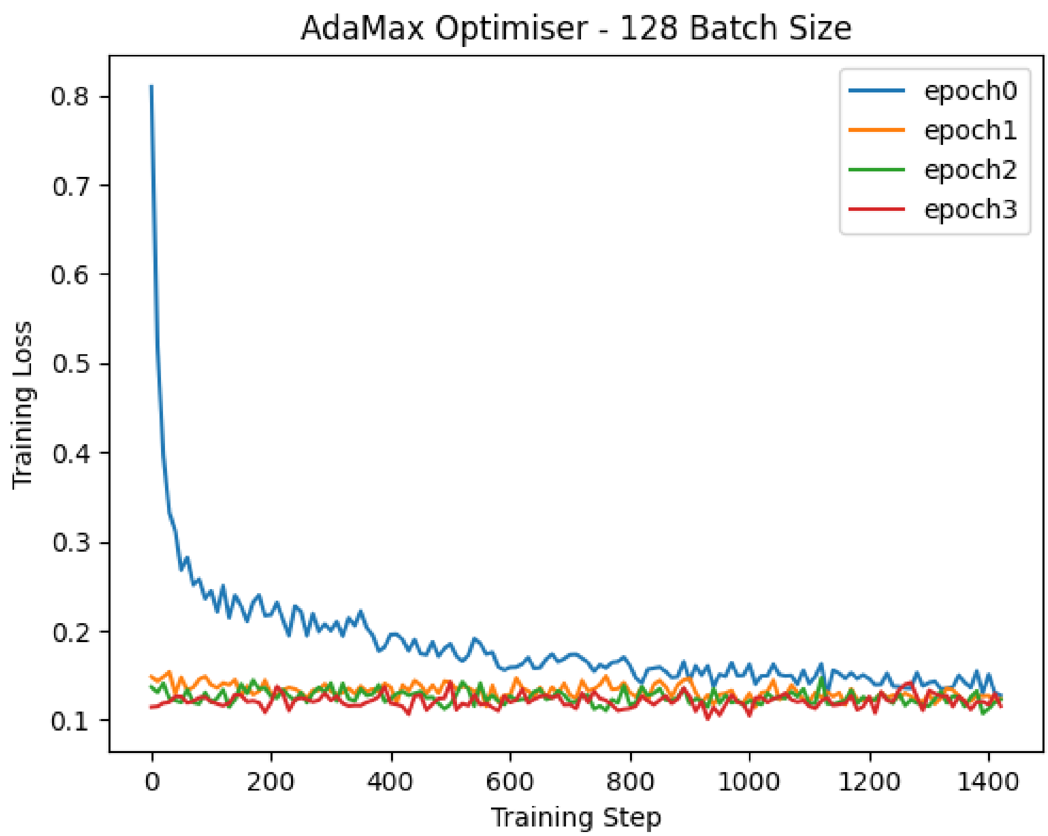


AdamMax optimizer with learning rate 0.0001 and 3 epochs.

In [24]:

```
epoch0 = [[0.8097755312919617, 0], [0.519804835319519, 10], [0.3965773284435272,
epoch1 = [[0.14834339916706085, 0], [0.14337162673473358, 10], [0.14803630113601
epoch2 = [[0.13701391220092773, 0], [0.1308240294456482, 10], [0.14086627960205
epoch3 = [[0.1142469048500061, 0], [0.11499416083097458, 10], [0.118826374411582

loss, step = zip(*epoch0)
loss1, step1 = zip(*epoch1)
loss2, step2 = zip(*epoch2)
loss3, step3 = zip(*epoch3)
plt.plot(step, loss, label='epoch0')
plt.plot(step1, loss1, label='epoch1')
plt.plot(step2, loss2, label='epoch2')
plt.plot(step3, loss3, label='epoch3')
plt.xlabel("Training Step")
plt.ylabel("Training Loss")
plt.title("AdaMax Optimiser - 128 Batch Size")
plt.legend()
```

Out[24]: <matplotlib.legend.Legend at 0x1fd036fbf70>

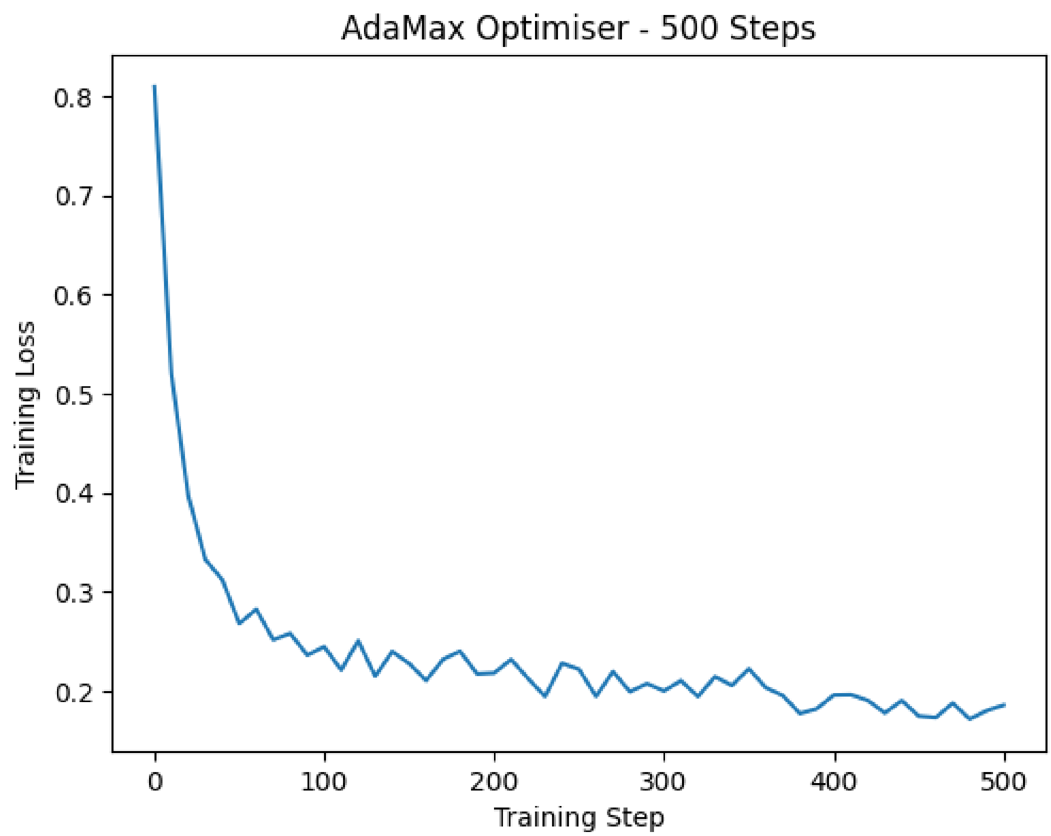


In [36]:

```
epoch500steps = [[0.8097755312919617, 0], [0.519804835319519, 10], [0.3965773284

loss500, step500 = zip(*epoch500steps)
plt.plot(step500, loss500)
plt.xlabel("Training Step")
plt.ylabel("Training Loss")
plt.title("AdaMax Optimiser - 500 Steps")
```

Out[36]:

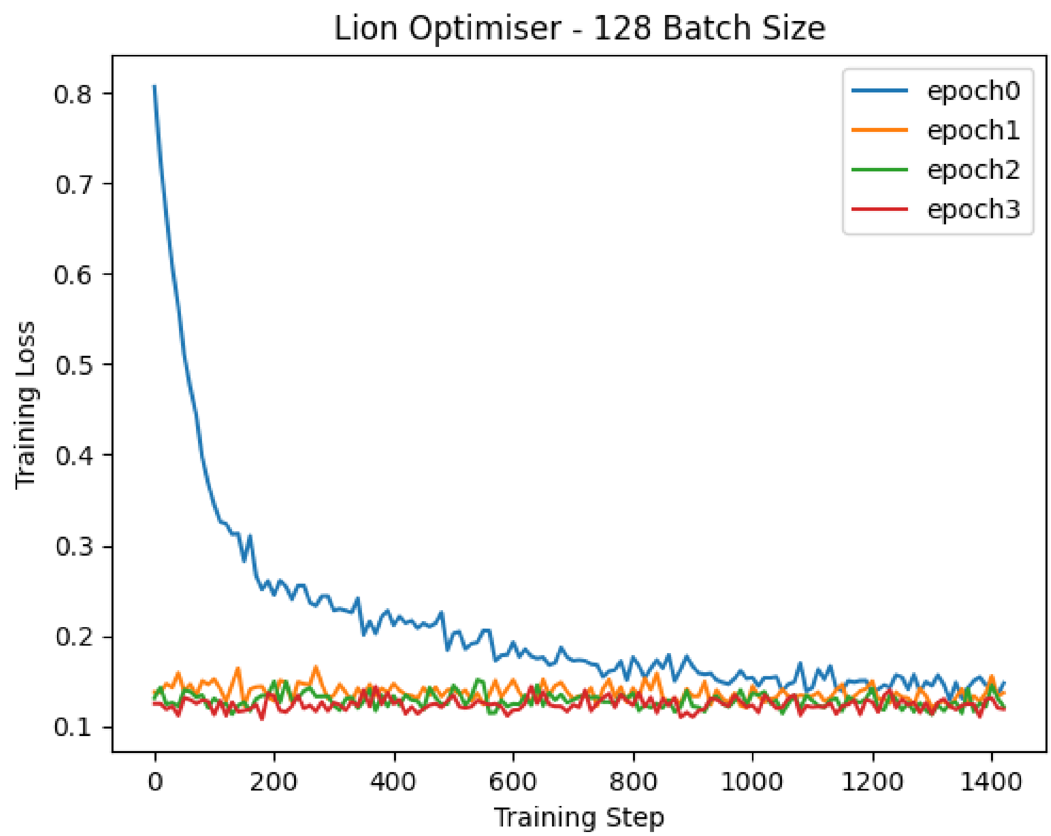
```
Text(0.5, 1.0, 'AdaMax Optimiser - 500 Steps')
```



Lion optimizer with learning rate 0.0001 and 3 epochs.

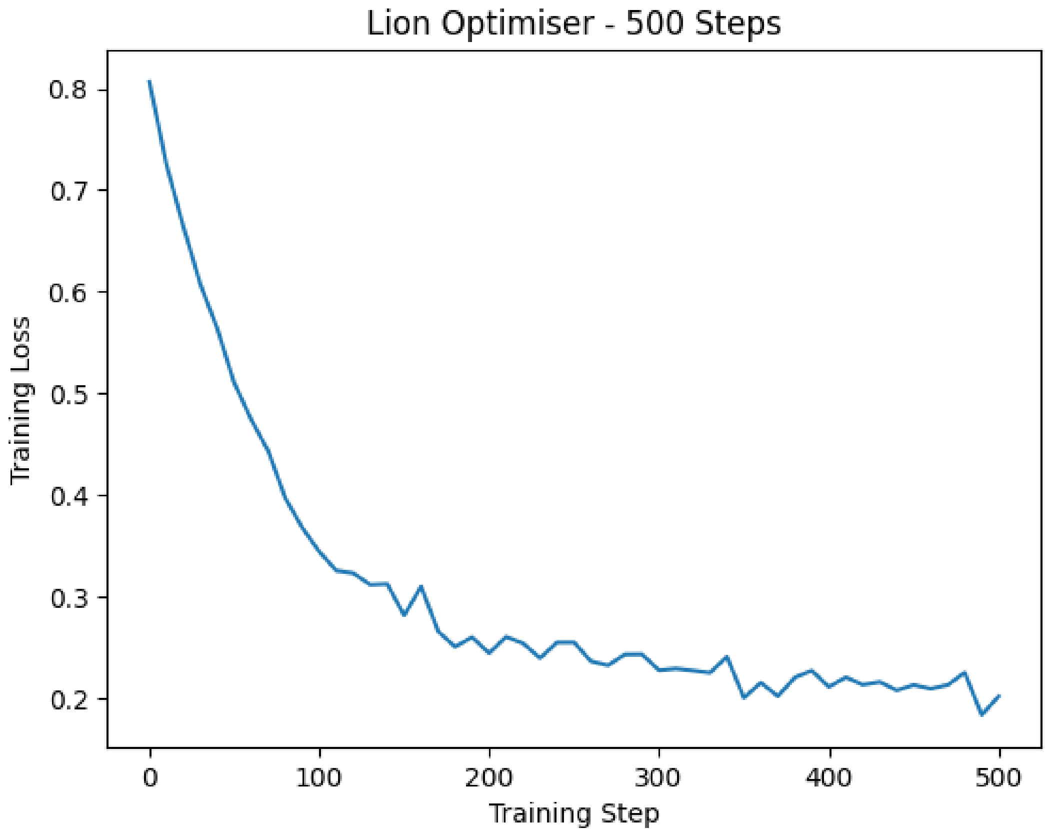
In [19]:
```
epoch0 = [[0.8066650629043579, 0], [0.725506067276001, 10], [0.6643359661102295,
epoch1 = [[0.13762417435646057, 0], [0.1372123658657074, 10], [0.146783202886581
epoch2 = [[0.13108249008655548, 0], [0.14272645115852356, 10], [0.12427704781293
epoch3 = [[0.12485311925411224, 0], [0.12479355186223984, 10], [0.11852773278951

loss, step = zip(*epoch0)
loss1, step1 = zip(*epoch1)
loss2, step2 = zip(*epoch2)
loss3, step3 = zip(*epoch3)
plt.plot(step, loss, label='epoch0')
plt.plot(step1, loss1, label='epoch1')
plt.plot(step2, loss2, label='epoch2')
plt.plot(step3, loss3, label='epoch3')
plt.xlabel("Training Step")
plt.ylabel("Training Loss")
plt.title("Lion Optimiser - 128 Batch Size")
plt.legend()
```

Out[19]:
```
<matplotlib.legend.Legend at 0x1fd7f2075e0>
```



In [37]:
```
epoch500steps = [[0.8066650629043579, 0], [0.725506067276001, 10], [0.6643359661

loss500, step500 = zip(*epoch500steps)
plt.plot(step500, loss500)
plt.xlabel("Training Step")
plt.ylabel("Training Loss")
plt.title("Lion Optimiser - 500 Steps")
```

Out[37]:
```
Text(0.5, 1.0, 'Lion Optimiser - 500 Steps')
```


Lion Optimiser - 500 Steps
Training Loss
Training Step
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
100
200
300
400
500

Test environment config.

In [1]:
```
import sys
print(sys.executable)
import torch
print(torch.__file__)
print(torch.cuda.is_available())
from torch.utils import collect_env
print(collect_env.main())
```

```
/scratch/users/k20014224/jvenv/bin/python
/scratch/users/k20014224/jvenv/lib/python3.8/site-packages/torch/__init__.py
True
Collecting environment information...
PyTorch version: 1.13.1+cu117
Is debug build: False
CUDA used to build PyTorch: 11.7
ROCM used to build PyTorch: N/A

OS: Ubuntu 20.04.6 LTS (x86_64)
GCC version: (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0
Clang version: Could not collect
CMake version: Could not collect
Libc version: glibc-2.31

Python version: 3.8.12 (default, Apr  5 2022, 19:30:22)  [GCC 9.4.0] (64-bit ru
ntime)
Python platform: Linux-5.15.0-67-generic-x86_64-with-glibc2.2.5
Is CUDA available: True
CUDA runtime version: 10.1.243
CUDA_MODULE_LOADING set to: LAZY
GPU models and configuration: GPU 0: NVIDIA A100-SXM4-40GB
Nvidia driver version: 510.108.03
cuDNN version: Could not collect
HIP runtime version: N/A
MIOpen runtime version: N/A
Is XNNPACK available: True

Versions of relevant libraries:
[pip3] numpy==1.24.2
[pip3] pytorch-fid==0.3.0
[pip3] torch==1.13.1
[pip3] torchaudio==0.13.1
[pip3] torchmetrics==0.11.1
[pip3] torchvision==0.14.1
[conda] Could not collect
None
```

Check if the environment has access to the NVIDIA A100 GPU.

In [2]:
```
!nvidia-smi
```

```
Sun Mar 26 12:25:30 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 510.108.03   Driver Version: 510.108.03   CUDA Version: 11.6     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA A100-SXM...  On   | 00000000:B1:00.0 Off |                    0 |
| N/A   32C    P0    56W / 400W |      3MiB / 40960MiB |      0%      Default |
|                               |                      |             Disabled |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Diffusion Model - Tests for the UNet model

A simple implementation of the diffusion model in PyTorch without text decoder and encoder for a full text-to-image generation pipeline.

In [1]:
```
import torch
import torchvision
import matplotlib.pyplot as plt
import torch.nn.functional as F
from torchvision import datasets, transforms
# from torchvision.transforms import Compose, ToTensor, Lambda, Resize, CenterCr
from torch.utils.data import DataLoader
import numpy as np
from torch import nn
import math
```

Step 1 - Forward Diffusion Process

Step 1.1 - The linear schedule used in the forward diffusion process to calculate the alphas, betas, diffusion and posterior.

In [2]:
```
# A linear schedule as proposed in https://arxiv.org/pdf/2102.09672.pdf
def linear_beta_schedule(timesteps):
    beta_start = 0.0001
    beta_end = 0.02

    return torch.linspace(beta_start, beta_end, timesteps)

# A cosine schedule as proposed in https://arxiv.org/abs/2102.09672.pdf
def cosine_beta_schedule(timesteps, s=0.008):
    steps = timesteps + 1
```

```
    x = torch.linspace(0, timesteps, steps)
    alphas_cumprod = torch.cos(((x / timesteps) + s) / (1 + s) * torch.pi * 0.5)
    alphas_cumprod = alphas_cumprod / alphas_cumprod[0]
    betas = 1 - (alphas_cumprod[1:] / alphas_cumprod[:-1])

    return torch.clip(betas, 0.0001, 0.9999)

# A quadratic schedule
def quadratic_beta_schedule(timesteps):
    beta_start = 0.0001
    beta_end = 0.02

    return torch.linspace(beta_start**0.5, beta_end**0.5, timesteps) ** 2

# A sigmoid schedule
def sigmoid_beta_schedule(timesteps):
    beta_start = 0.0001
    beta_end = 0.02
    betas = torch.linspace(-6, 6, timesteps)

    return torch.sigmoid(betas) * (beta_end - beta_start) + beta_start

# Returns a specific index t of a passed list of values vals while considering t
def get_index_from_list(vals, t, x_shape):
    batch_size = t.shape[0]
    out = vals.gather(-1, t.cpu())

    return out.reshape(batch_size, *((1,) * (len(x_shape) - 1))).to(t.device)

# Returns the diffusion model's forward diffusion sample, taking an image x_0 an
def forward_diffusion_sample(x_0, t, device="cpu"):

    noise = torch.randn_like(x_0)

    sqrt_alphas_cumprod_t = get_index_from_list(sqrt_alphas_cumprod, t, x_0.shap

    sqrt_one_minus_alphas_cumprod_t = get_index_from_list(
        sqrt_one_minus_alphas_cumprod, t, x_0.shape
    )

    # mean + variance
    return sqrt_alphas_cumprod_t.to(device) * x_0.to(device) \
    + sqrt_one_minus_alphas_cumprod_t.to(device) * noise.to(device), noise.to(de


# Define beta schedule
T = 300
betas = linear_beta_schedule(timesteps=T)

# define alphas
alphas = 1. - betas
alphas_cumprod = torch.cumprod(alphas, axis=0)
alphas_cumprod_prev = F.pad(alphas_cumprod[:-1], (1, 0), value=1.0)
sqrt_recip_alphas = torch.sqrt(1.0 / alphas)

# Calculate for diffusion q(x_t | x_{t-1})
sqrt_alphas_cumprod = torch.sqrt(alphas_cumprod)
sqrt_one_minus_alphas_cumprod = torch.sqrt(1. - alphas_cumprod)
```

```
# Calculate for posterior q(x_{t-1} | x_t, x_0)
posterior_variance = betas * (1. - alphas_cumprod_prev) / (1. - alphas_cumprod)
```

Image Preprocessing Helper Functions

In [3]:
```
# Parameters for the dataset with image size of 64x64, 128x128, 256x256
# These will be used to resize the images and test the models on different image
IMG_SIZE = 64
IMG_SIZE_128 = 128
IMG_SIZE_256 = 256

# Batch size for training and testing with 128 images per batch and 256 images p
BATCH_SIZE = 128
BATCH_SIZE_256 = 256

# The tensor transformer for the dataset
def load_transformed_dataset():
    transform = transforms.Compose([
        transforms.Resize((IMG_SIZE, IMG_SIZE)),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Lambda(lambda t: t * 2 - 1)
    ])

    data_transform = transform

    train = datasets.CelebA(root='', split="train", download=True, transform=dat

    test = datasets.CelebA(root='', split="test", download=True, transform=data_

    return torch.utils.data.ConcatDataset([train, test])


# Load the transformer dataset
data = load_transformed_dataset()

# Appends the data into a dataloader with a batch size of 128 or 256 depending o
dataloader = DataLoader(data, batch_size=BATCH_SIZE, shuffle=True, drop_last=Tru

# The reverse transformer for the dataset to show the images back to their origi
def reverse_tensor_img(image):
    reverse_transform = transforms.Compose([
        transforms.Lambda(lambda t: (t + 1) / 2),
        transforms.Lambda(lambda t: t.permute(1, 2, 0)),
        transforms.Lambda(lambda t: t*255),
        transforms.Lambda(lambda t: t.cpu().numpy().astype(np.uint8)),
        transforms.ToPILImage(),
    ])

    # Take first image of batch
    if len(image.shape) == 4:
        image = image[0, :, :, :]

    plt.imshow(reverse_transform(image))
```

```
Files already downloaded and verified
Files already downloaded and verified
```

Testing if the forward diffusion process is working correctly.

Step 2 - Backward Diffusion Process (U-Net)

In [48]:
```
# The convolutional block for the model
# The block consists of two convolutional layers with each one having its own ba
# The block also has a time embedding layer that is used to add the time embeddi
# The block also has skip connections using the time embedding layer and the con
class Block(nn.Module):
    def __init__(self, in_channel, out_channel, time_emb_dim, up=False):
        super().__init__()
        # Time embedding layer
        self.time_mlp =  nn.Linear(time_emb_dim, out_channel)

        # First convolutional layers
        # If up is true then add a convolutional transpose layer to upsample the
        if up:
            self.conv1 = nn.Conv2d(2*in_channel, out_channel, 3, padding=1)
            self.transform = nn.ConvTranspose2d(out_channel, out_channel, 4, 2,
        else:
            self.conv1 = nn.Conv2d(in_channel, out_channel, 3, padding=1)
            self.transform = nn.Conv2d(out_channel, out_channel, 4, 2, 1)

        # Second convolutional layer
        self.conv2 = nn.Conv2d(out_channel, out_channel, 3, padding=1)

        # Batch normalization layers for both convolutional layers
        self.bnorm1 = nn.BatchNorm2d(out_channel)
        self.bnorm2 = nn.BatchNorm2d(out_channel)

        # Relu activation function
        self.relu  = nn.ReLU()

    def forward(self, x, t, ):
        # First Conv
        h = self.bnorm1(self.relu(self.conv1(x)))
        # Time embedding
        time_emb = self.relu(self.time_mlp(t))
        # Extend last 2 dimensions
        time_emb = time_emb[(..., ) + (None, ) * 2]
        # Add time channel
        h = h + time_emb
        # Second Conv
        h = self.bnorm2(self.relu(self.conv2(h)))
        # Down or Upsample
        return self.transform(h)

# A sinusoidal time embedding layer as described in the paper https://arxiv.org/
class SinusoidalPositionEmbeddings(nn.Module):
    def __init__(self, dim):
        super().__init__()
        self.dim = dim

    def forward(self, time):
        device = time.device
        half_dim = self.dim // 2
        embeddings = math.log(10000) / (half_dim - 1)
```

```
        embeddings = torch.exp(torch.arange(half_dim, device=device) * -embeddir
        embeddings = time[:, None] * embeddings[None, :]
        embeddings = torch.cat((embeddings.sin(), embeddings.cos()), dim=-1)
        return embeddings

# A UNet architecture for the image denoising task with time embedding in each l
class SimpleUnet(nn.Module):
    def __init__(self):
        super().__init__()
        image_channels = 3 # RGB: 3 channels for RED, GREEN, BLUE
        down_channels = (64, 128, 256, 512, 1024) # Number of channels in each c
        up_channels = (1024, 512, 256, 128, 64) # Number of channels in each ups
        out_dim = 1 # 1x1 final of output channels
        time_emb_dim = 32 # Dimension of time embedding

        # Time embedding
        self.time_mlp = nn.Sequential(
                SinusoidalPositionEmbeddings(time_emb_dim),
                nn.Linear(time_emb_dim, time_emb_dim),
                nn.ReLU()
            )

        # Initial projection
        self.conv0 = nn.Conv2d(image_channels, down_channels[0], 3, padding=1)

        # Downsample
        self.downs = nn.ModuleList([Block(down_channels[i], down_channels[i+1],
                                    time_emb_dim) \
                    for i in range(len(down_channels)-1)])
        # Upsample
        self.ups = nn.ModuleList([Block(up_channels[i], up_channels[i+1], \
                                        time_emb_dim, up=True) \
                    for i in range(len(up_channels)-1)])

        # Final output 1x1 conv
        self.output = nn.Conv2d(up_channels[-1], 3, out_dim)

    def forward(self, x, timestep):
        # Embedd time
        t = self.time_mlp(timestep)
        # Initial conv
        x = self.conv0(x)
        # Unet
        residual_inputs = []
        for down in self.downs:
            x = down(x, t)
            residual_inputs.append(x)
        for up in self.ups:
            residual_x = residual_inputs.pop()
            # Add residual x as additional channels
            x = torch.cat((x, residual_x), dim=1)
            x = up(x, t)
        return self.output(x)

model = SimpleUnet()
print("Num params: ", sum(p.numel() for p in model.parameters()))
model
device = "cuda" if torch.cuda.is_available() else "cpu"
model.to(device=device)
```

```
Num params:  62438883
```

Out[48]:
```
SimpleUnet(
  (time_mlp): Sequential(
    (0): SinusoidalPositionEmbeddings()
    (1): Linear(in_features=32, out_features=32, bias=True)
    (2): ReLU()
  )
  (conv0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (downs): ModuleList(
    (0): Block(
      (time_mlp): Linear(in_features=32, out_features=128, bias=True)
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
      (transform): Conv2d(128, 128, kernel_size=(4, 4), stride=(2, 2), padding=
(1, 1))
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
      (bnorm1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_ru
nning_stats=True)
      (bnorm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_ru
nning_stats=True)
      (relu): ReLU()
    )
    (1): Block(
      (time_mlp): Linear(in_features=32, out_features=256, bias=True)
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
      (transform): Conv2d(256, 256, kernel_size=(4, 4), stride=(2, 2), padding=
(1, 1))
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
      (bnorm1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_ru
nning_stats=True)
      (bnorm2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_ru
nning_stats=True)
      (relu): ReLU()
    )
    (2): Block(
      (time_mlp): Linear(in_features=32, out_features=512, bias=True)
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
      (transform): Conv2d(512, 512, kernel_size=(4, 4), stride=(2, 2), padding=
(1, 1))
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
      (bnorm1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_ru
nning_stats=True)
      (bnorm2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_ru
nning_stats=True)
      (relu): ReLU()
    )
    (3): Block(
      (time_mlp): Linear(in_features=32, out_features=1024, bias=True)
      (conv1): Conv2d(512, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
      (transform): Conv2d(1024, 1024, kernel_size=(4, 4), stride=(2, 2), paddin
g=(1, 1))
      (conv2): Conv2d(1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=
(1, 1))
      (bnorm1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_r
unning_stats=True)
```

```
      (bnorm2): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_r
unning_stats=True)
      (relu): ReLU()
    )
  )
  (ups): ModuleList(
    (0): Block(
      (time_mlp): Linear(in_features=32, out_features=512, bias=True)
      (conv1): Conv2d(2048, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
      (transform): ConvTranspose2d(512, 512, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
      (bnorm1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_ru
nning_stats=True)
      (bnorm2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_ru
nning_stats=True)
      (relu): ReLU()
    )
    (1): Block(
      (time_mlp): Linear(in_features=32, out_features=256, bias=True)
      (conv1): Conv2d(1024, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
      (transform): ConvTranspose2d(256, 256, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
      (bnorm1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_ru
nning_stats=True)
      (bnorm2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_ru
nning_stats=True)
      (relu): ReLU()
    )
    (2): Block(
      (time_mlp): Linear(in_features=32, out_features=128, bias=True)
      (conv1): Conv2d(512, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
      (transform): ConvTranspose2d(128, 128, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
      (bnorm1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_ru
nning_stats=True)
      (bnorm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_ru
nning_stats=True)
      (relu): ReLU()
    )
    (3): Block(
      (time_mlp): Linear(in_features=32, out_features=64, bias=True)
      (conv1): Conv2d(256, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
      (transform): ConvTranspose2d(64, 64, kernel_size=(4, 4), stride=(2, 2), p
adding=(1, 1))
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
      (bnorm1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_run
ning_stats=True)
      (bnorm2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_run
ning_stats=True)
```

```
      (relu): ReLU()
    )
  )
  (output): Conv2d(64, 3, kernel_size=(1, 1), stride=(1, 1))
)
```

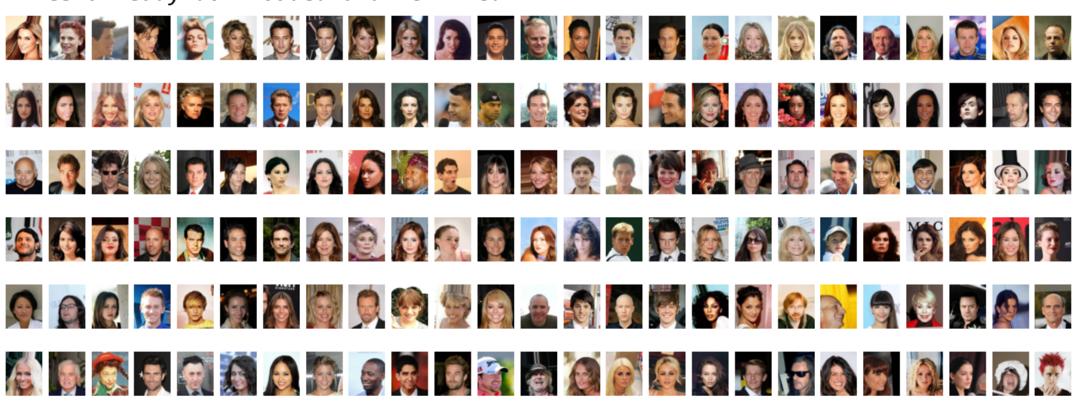
Test unit suite

Image testing.

In [6]:

```python
# Generates 150 samples of 25 columns x 10 rows of images
def show(dataset, num_sample=150, cols=25, rows=10):
    plt.figure(figsize=(15, 15))
    for i, img in enumerate(dataset):
        if i == num_sample:
            break
        plt.subplot(num_sample // rows + 1, cols, i + 1)
        plt.axis('off')
        plt.imshow(img[0])

# Download the dataset
# *WARNING:* This will take a while to download (depending on connection speed)
data = torchvision.datasets.CelebA(root='', split="train", download=True)

# Show the first 150 samples
show(data)
```

```
Files already downloaded and verified
```

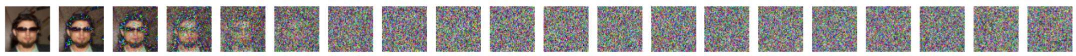


In [7]:

```python
# Convert reverse_tensor_img to a cuda tensor function
# reverse_tensor_img = torch.jit.script(reverse_tensor_img)

# Load a single image from the dataloader
image = next(iter(dataloader))[0]

# Add image dimensions for the graph, the amount of image steps and the step siz
plt.figure(figsize=(18, 18))
plt.axis('off')
num_images = 20
stepsize = int(T/num_images)

# Plot the image with the step size and show the image
for idx in range(0, T, stepsize):
    t = torch.Tensor([idx]).type(torch.int64)
    plt.subplot(1, num_images+1, (idx//stepsize) + 1)
```

```python
    image, noise = forward_diffusion_sample(image, t)
    plt.axis('off')
    reverse_tensor_img(image)
```

```
/tmp/slurm-tmp.2050130/ipykernel_208526/1265622936.py:16: MatplotlibDeprecation
Warning: Auto-removal of overlapping axes is deprecated since 3.6 and will be r
emoved two minor releases later; explicitly call ax.remove() as needed.
  plt.subplot(1, num_images+1, (idx//stepsize) + 1)
```



Simple test for the UNet model.

In [83]:
```python
import unittest

class TestImageSizes(unittest.TestCase):

    def test_unet_64(self):
        # Test the unet architecture on 64x64 images
        model = SimpleUnet()
        image = torch.randn(1, 3, IMG_SIZE, IMG_SIZE)
        t = torch.Tensor([0]).type(torch.int64)
        y = model(image, t)
        assert y.shape == torch.Size([1, 3, 64, 64])

    def test_unet_128(self):
        # Test the unet architecture on 128x128 images
        model = SimpleUnet()
        image = torch.randn(1, 3, IMG_SIZE_128, IMG_SIZE_128)
        t = torch.Tensor([0]).type(torch.int64)
        y = model(image, t)
        assert y.shape == torch.Size([1, 3, 128, 128])

    def test_unet_256(self):
        # Test the unet architecture on 256x256 images
        model = SimpleUnet()
        image = torch.randn(1, 3, IMG_SIZE_256, IMG_SIZE_256)
        t = torch.Tensor([0]).type(torch.int64)
        y = model(image, t)
        assert y.shape == torch.Size([1, 3, 256, 256])

class TestConvolutions(unittest.TestCase):

    def test_conv2d_64(self):
        # Test the conv2d layer on 64x64 images
        model = SimpleUnet()
        model_conv2d = model.conv0
        image = torch.randn(1, 3, IMG_SIZE, IMG_SIZE)
        t = torch.Tensor([0]).type(torch.int64)
        model_conv2d = model(image, t)
        assert model_conv2d.shape == torch.Size([1, 3, 64, 64])

    def test_output_64(self):
        # Test the output layer on 64x64 images
        model = SimpleUnet()
        model_output = model.output
        image = torch.randn(1, 3, IMG_SIZE, IMG_SIZE)
        t = torch.Tensor([0]).type(torch.int64)
        model_output = model(image, t)
        assert model_output.shape == torch.Size([1, 3, 64, 64])
```

```python
    def test_downs_64(self):
        # Test the downsampling layers on 64x64 images
        model = SimpleUnet()
        model_downs = model.downs
        image = torch.randn(1, 3, IMG_SIZE, IMG_SIZE)
        t = torch.Tensor([0]).type(torch.int64)
        model_downs = model(image, t)
        assert model_downs.shape == torch.Size([1, 3, 64, 64])

    def test_ups_64(self):
        # Test the upsampling layers on 64x64 images
        model = SimpleUnet()
        model_ups = model.ups
        image = torch.randn(1, 3, IMG_SIZE, IMG_SIZE)
        t = torch.Tensor([0]).type(torch.int64)
        model_downs = model(image, t)
        assert model_downs.shape == torch.Size([1, 3, 64, 64])

class TestUNetArc(unittest.TestCase):

    def test_architecture_time_emb(self):
        # Test the UNet architecture is correctly set up
        model = SimpleUnet()
        image = torch.randn(1, 3, IMG_SIZE, IMG_SIZE)
        t = torch.Tensor([0]).type(torch.int64)
        y = model(image, t)

        # Test time embedding
        assert model.time_mlp(t).shape == torch.Size([1, 32])

# Test suit to allow to run all tests at once, remove specific tests if needed t
def suit():
    suite = unittest.TestSuite()
    suite.addTest(TestImageSizes('test_unet_64'))
    suite.addTest(TestImageSizes('test_unet_128'))
    suite.addTest(TestImageSizes('test_unet_256'))
    suite.addTest(TestConvolutions('test_conv2d_64'))
    suite.addTest(TestConvolutions('test_output_64'))
    suite.addTest(TestConvolutions('test_downs_64'))
    suite.addTest(TestConvolutions('test_ups_64'))
    suite.addTest(TestUNetArc('test_architecture_time_emb'))
    return suite

if __name__ == '__main__':
    runner = unittest.TextTestRunner()
    runner.run(suit())

# unittest.main(argv=['-t'], verbosity=2, exit=False)
```

```
........
----------------------------------------------------------------------
Ran 8 tests in 4.918s

OK
```