Deep Learning: Project 4

Florence Hugh & August Jonasson

January 17, 2025

Introduction and dataset

In this project we explore and fit a diffusion model, namely the denoisning diffusion probabilistic model (DDPM) presented in C. F. Higham, D. J. Higham, and Grindrod 2023, or equivalently, in Ho, Jain, and Abbeel 2020, for the purpose of generating new (novel) images. The dataset of our choice contains ≈ 200 images of the fictional Pokémon creature Bulbasaur[1], and it is thus images of Bulbasaur that we seek to generate using the DDPM.

We chose this dataset as it is relatively small (easing the computational load), and because its individual images are of relatively low complexity (a drawing using only a few shapes and colors as opposed to, say, a photo taken of something in the real world), which we hope will make fitting a model easier. We also like Bulbasaurs, of which some examples from the dataset can be seen in Figure 1.

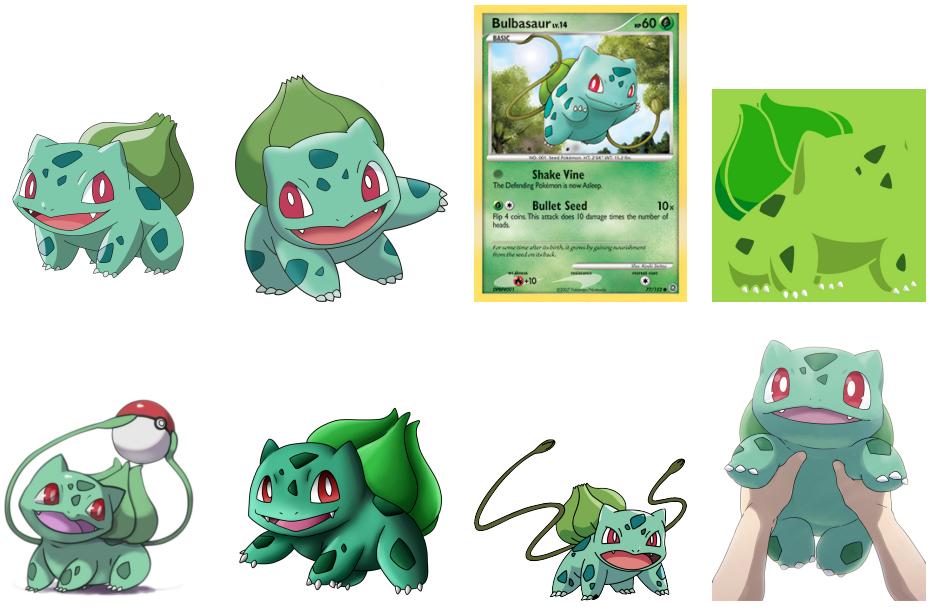

Figure 1: Example samples of images in the dataset.

[1] https://en.wikipedia.org/wiki/Bulbasaur

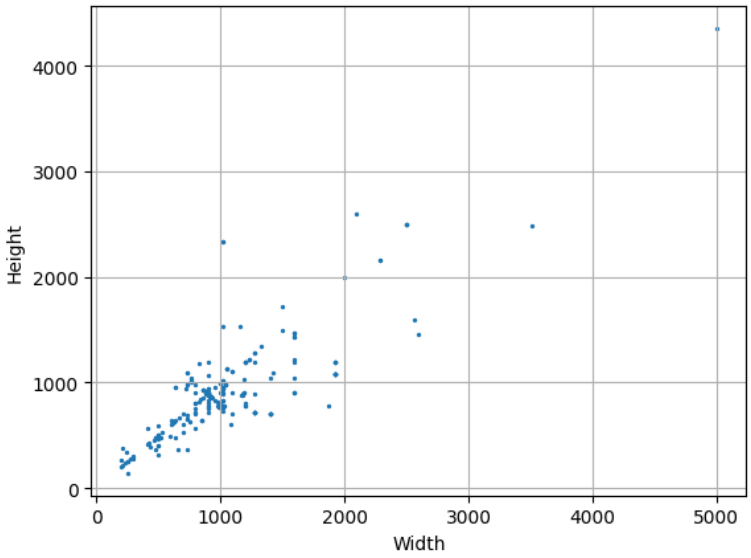

Figure 2: The dimensions of each Bulbasaur image.

Exploratory analysis and data preprocessing

As seen from Figure 2, the resolutions of the individual images within the dataset are not fixed and they also have a varying amount of channels. We thus require some pre-processing of the data before starting to model.

First, we identify along which axis (x or y) that the image is the smallest. We then, as centered as possible, crop out a square of that particular side length using the `image.crop_to_bounding_box`[2] function, leaving us with a square image. The next step is to resize the image to the desired resolution 64x64 (chosen w.r.t. compute constraints) using *bilinear interpolation*[3]. The `image.resize` function was used for this. Finally we rescale the pixel values to lie in the $[-1, 1]$ interval using the `image.clip_by_value` function. This is done in order to have the pixel values centered on the same value as the standard normal noise that we are to add in the diffusion process. Finally we have also used the `image.random_flip_left_right` function as a simple form of data augmentation. This is meant to both expand the dataset, but also prevent overfitting in the long run.

[2]all functions in this section come from tensorflow.image module https://www.tensorflow.org/api_docs/python/tf/image
[3]read more in this wikipedia article https://en.wikipedia.org/wiki/Bilinear_interpolation

Underlying theory of the DDPM

In order to implement a DDPM, it is important to be familiar with some of the underlying theory, as the hyperparameters aim to fulfill certain conditions, which we will lay bare in the following section. The notation follows C. F. Higham, D. J. Higham, and Grindrod 2023 precisely and the reader can refer to this article for all of the theoretical results that follow.

One crucial result from the probability theory underlying the DDPM is that

$$q(x_t|x_0) \to \mathcal{N}(x_t; 0, I), \quad \text{as} \quad t \to \infty, \tag{1}$$

where q is the transition density of the forward process, t is the time-step and $\mathcal{N}$ is a normal density - i.e. that given a starting image x_0, the diffused image at some late time-step T, x_T, should be indistinguishable from standard normal noise. Without this property, it wouldn't be reasonable to start from pure noise when generating new images from the trained model. Using the definition of the forward process together with the Markovian property, it follows that the transition density is defined as

$$q(x_t|x_0) = \mathcal{N}(x_t; \sqrt{\overline{\alpha}_t}x_0, \sqrt{1-\overline{\alpha}_t}\,\mathrm{I}), \tag{2}$$

where

$$\overline{\alpha}_t = \prod_{i=1}^{t} \alpha_i,$$

and $\alpha_t = 1 - \beta_t$ for monotonically increasing sequence $\{\beta_t \in (0,1)\}_{t=1}^{T}$. It is now evident from equations (1) and (2) that controlling the decay of $\sqrt{\overline{\alpha}_t}$ such that it tends to zero for some sufficiently large $t = T$ is of utmost importance when defining the diffusion process.

Setting the distribution scheme of the $\{\beta_t\}$ to a uniform spread on the $[0.0001, 0.02]$ interval (mimicking the authors Ho, Jain, and Abbeel 2020, of the original DDPM article), we can now focus entirely on varying T such that the decay becomes sufficient.

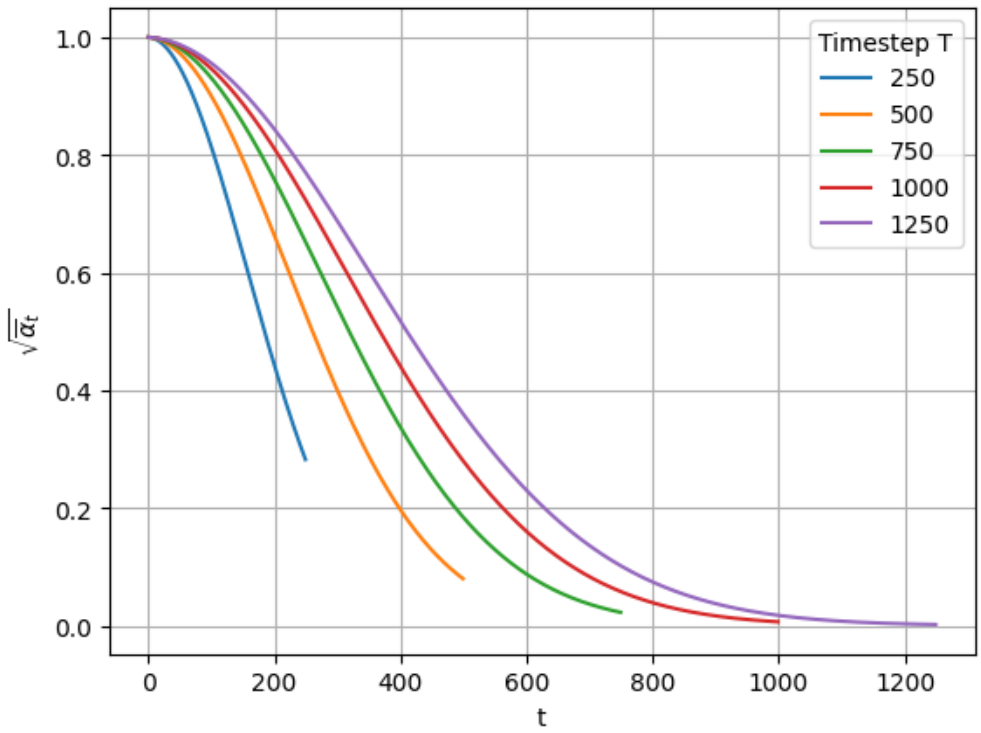

Figure 3: Decays of $\sqrt{\overline{\alpha}_t}$ for varying number of noise-adding steps T.

Based on the results of Figure 3, we decide to go with $T = 1000$, as this yields $\sqrt{\overline{\alpha}_T} \approx 10^{-5}$ which we deem to be sufficiently close to zero (this is also what the authors Ho, Jain, and Abbeel 2020 went with, which we have now confirmed the reasons for).

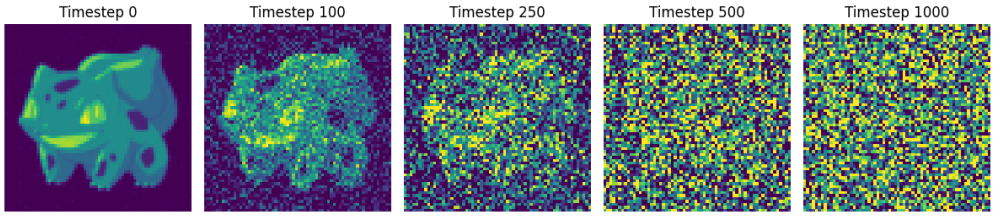

Figure 4: Result of forward process over time on an example of a cropped, rescaled and resized image of Bulbasaur using $T = 1000$.

In Figure 4 we see an example of the forward process applied to a pre-processed image of our dataset, at a select few timesteps. Looking carefully, one might still be able to glimpse some semblance of a structure (not purely noise) at $t = 500$. However faint the structure looks to our eyes, the computer will see it more clearly.

Architecture

Having motivated our decision for the hyperparameters of the diffusion process, we will in this section briefly present the architecture of the neural network that we use to fit the weights and biases of the DDPM.

The U-net architecture, widely used for implementing diffusion models, is effective due to its capability to handle image segmentation tasks (Williams et al. 2023). As our purpose is to generate images of Bulbasaurs, U-Nets would be capable of capturing important features, like the red eyes and its leaf-like shell. Therefore we have used a U-Net based architecture with residual and attention blocks following the code from Keras official website[4].

The encoder part of a U-Net processes the noisy image through a series of convolutional operations, similarly to how a standard convolution neural network works. The difference comes in the decoder part of the network which reconstructs the image by transitioning back to the original resolution. This process takes help from the skip connections which incorporate features from corresponding levels in the encoder to preserve important spatial information (Williams et al. 2023).

The encoder part of the network we used for training consists of two 3x3 convolution layers with a sigmoid linear (SiLu) activation function with added padding to match output pixels with input pixels. During this part, the pixels will be reduced while the channels will increase. The channels are increased to store information from the decreased pixels, this ensures no information gets lost during training.

The network consists of four levels in which the last two levels have an attention block added. These are added to pick up salient inputs. At the bottleneck, convolution layers and attention blocks are applied to generate a feature map of size (8, 8, 256). Finally, the decoder part uses the feature map from the bottleneck to convert the image back to its original input size. This phase works similar as the encoder phase with the difference that it first up samples the images using interpolative resizing of type "nearest" to match the sizing to its opposite level and then follow with matching convolution layers and attention blocks. The final output layer will be of dimensions (64, 64, 3). This architecture landed on around 30 million trainable parameters.

Training and results

Finally, we can use the architecture and the chosen $\{\sqrt{\overline{\alpha}_t}\}$ with $T = 1000$ to train our model. Fitting the model is equivalent to a least-squares regression problem where the randomly sampled noise ϵ acts as response variable, and the timestep $t \in \{1, \ldots, T\}$ and image x_t, derived from applying the diffusion process up to time t to an image x_0 from the training data, act as a pair of predictors. It can be shown through the maximization of the ELBO(x) (along with some assumptions of a dominating term) that this really turns into a problem of least squares[5], hence regression and the minimization of

$$||\epsilon - \epsilon_\theta(\sqrt{\overline{\alpha}_t}x_0 + \sqrt{1 - \overline{\alpha}_t}\epsilon, t)||_2^2, \qquad \forall t \in \{1, \ldots, T\}.$$

[4]https://keras.io/examples/generative/ddpm/

[5]this was shown during a lecture of the course MT7042 at Stockholm University and we have no official source for this

It is this ϵ_θ that our neural network aims to model. In the training process one uniformly samples a t once per image and epoch. Therefore we believe the number of epochs needs to be larger than the number of time-steps T in order to cover most of the t's for an image. For more information regarding the training of the model and the actual generating of the images from the trained model, see **Algorithm 1 & 2** in C. F. Higham, D. J. Higham, and Grindrod 2023. We now move on to showing some of the results.

Using a slightly lower than recommended learning rate ($= 2 \cdot 10^{-4}$), we land on running 2000 epochs of training (which took around 12 hours to run on our best CPU) using the Adam optimizer with default[6] settings, and the MSE loss function. The sampling process is then applied to generate 16 new images from this trained model, starting from standard normal noise.

Figure 5: 16 images sampled from a 2000 epoch model using a linear decay on the $\{\alpha_t\}$, $T = 1000$, learning rate$= 2 \cdot 10^{-4}$, the mean-squared-error loss function and a default Adam optimizer.

From Figure 5 we can see that the model did a fairly good job on generating Bulbasaurs. With minor deforming shapes, it did capture that it has two red eyes, some spots on its body and a leaf-like shell on its back. Since some of these images look a bit too much like the original images our training might have overfitted. Usually this would be the place for validation of the results, but, due to DDPMs being so new, agreed upon methods of validation are yet to be established. More on this in the next section.

However, there are ways to monitor the training of the model. If we split the data into a training set and a test set, we can still use a training/validation error plot in the usual sense. We do, after all, have a regression problem (even if diffusion models are generally considered as part of unsupervised learning), albeit a bit special, as we are using randomly sampled noise as our response variables. A validation of the training would then amount to, apart from the noise that we have fit the model to, randomly sampling an additional noise, and seeing how well that noise is predicted from the given test data.

[6]for the default settings, see https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam

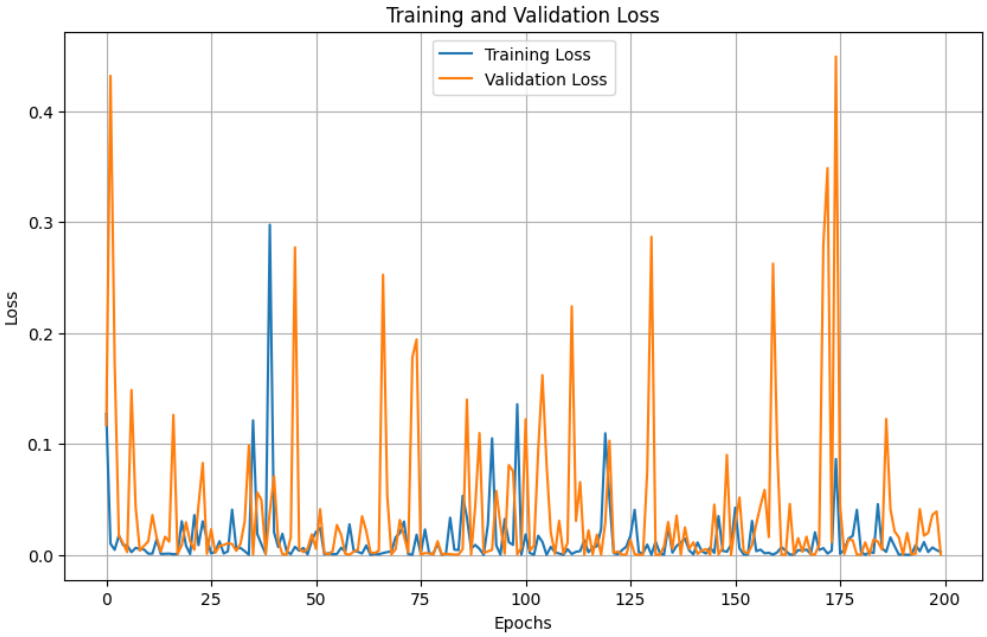

Figure 6: Training and validation error plot of the DDPM using a 80/20 split of the data. Epochs should be multiplied by ten.

Figure 6 represents a model that is different from the one that gave us the results in Figure 5 (which used the whole dataset for training only) in the sense that we have now split the data into a training set and a test set. As such, this should not be interpreted as a complete diagnosis of the training of the previous model, but instead serve as an illustratory example, or at best point out some potential issues with the previous model. The reason for not using this split data in the previous model is that our goal was to generate images of Bulbasaur. Of course we want these images to be as good as possible and therefore we should of course dedicate the entire dataset to training.

Interpreting Figure 6, we see that for the most part, both the training and the validation error is relatively low, the training more so than the validation, which is expected. Were it not for the excessive "spikyness" of the curves, we might have said that the training looks healthy. So where do these spikes come from, and why do they appear to have such a random distribution along the curves? One explanation, which as far as we can tell is also the best, is that as we are trying to fit a model to noise we should expect to see this noise in the model. Usually when performing regression, the predictors and the responses portray some linked pattern. One type of change in the predictor has the tendency for a particular type change in the response. For the DDPM this is not the case. Irregardless of the input, the response is still just standard normal noise. However, just because the response is noise, it doesn't mean that we don't have any trends. The standard normal distribution will of course produce more values close to zero, as this is where its mean lies. As such, it would be more beneficial for the model to predict more values close to zero and this is what we expect to see.

When studying these training/validation curves for the DDPM, it is thus not the shape of the curve, but the distribution of the errors that matter. For a healthy model, we expect the errors to behave like the square of a standard normal distribution (since we are using the MSE), i.e. like a chi-squared distribution. This could very well be the case for both the training and validation errors in Figure 6 and we should thus not write it off as a poor fit immediately (unfortunately we could not find any references for this - it might not be true at all even).

Possible improvements

Validation on result

Validating that our result is reasonable is a bit tricky as DDPM is a relatively new field in deep learning and therefore there are no well-established methods. What we want to do is decide the trade-off between plausibility and novelty. But as there is no real method for this we may simply compare the images with our own eyes. We saw that a few of the generated images looked exactly the same as the original images but the computer might detect some small difference that cannot be seen with our eyes only. Thus, this validation technique is not at all reliable.

Another validation could be to compute the Fréchet inception distance to decide the quality of the generated images. This would be done by comparing the distribution of two sets of images, one of the generated images and the other of the real images, to see if they match. This method uses an inception network[7] to compare the distance of the mean and standard deviation between the two image sets with the defined formula:

$$d_F(\mathcal{N}(\mu,\Sigma),\mathcal{N}(\mu',\Sigma'))^2 = ||\mu-\mu'||_2^2 + \mathrm{tr}\bigg(\Sigma+\Sigma'-2(\Sigma\Sigma')^{\frac{1}{2}}\bigg),$$

where higher values indicate poor generative images and lower values indicate good generated images.

Monitoring the training

There could be some improvements with our training/validation error. Seeing as our dataset was very small, we could have made a smaller split, in the ratio of 90/10 instead of 80/20 since this made our training set even smaller. However, a better alternative might have been to use a leave-one-out cross-validation, but because this would take far too long seeing as it already takes quite a lot of time to train the dataset already we opted to not apply this.

How to handle overfitting

DDPMs are designed to train noise and predict noise, hence overfitting these models would mean that our generated images are reproducing the exact copy of the original data. Our generated data seem to overfit, if we look at the top-right image in Figure 1 this image can be spotted in our generated images in Figure 5. Since our dataset was fairly small, one way to prevent overfitting would be to incorporate additional data augumentation.

Furthermore, since the problem is regression one might be inclined to regularize using methods such as the L_2 penalty (ridge regression). Apparently, this is not effective when using adaptive learning algorithms (Loshchilov and Hutter 2019) such as Adam, and one would have switch to something like SGD with momentum instead (which apparently performs very well when paired with ridge). One could also try using the L_1 penalization (lasso). In general, most regularization strategies that are applicable to regression problems will be applicable here as well. Another option would be dropout. This seems especially feasible as there are a lot of trainable parameters in our particular U-net.

[7] https://en.wikipedia.org/wiki/Inception_(deep_learning_architecture)

Appendix A: training the model

Here, we show some of the generated images that were sampled as the training of the model went on.

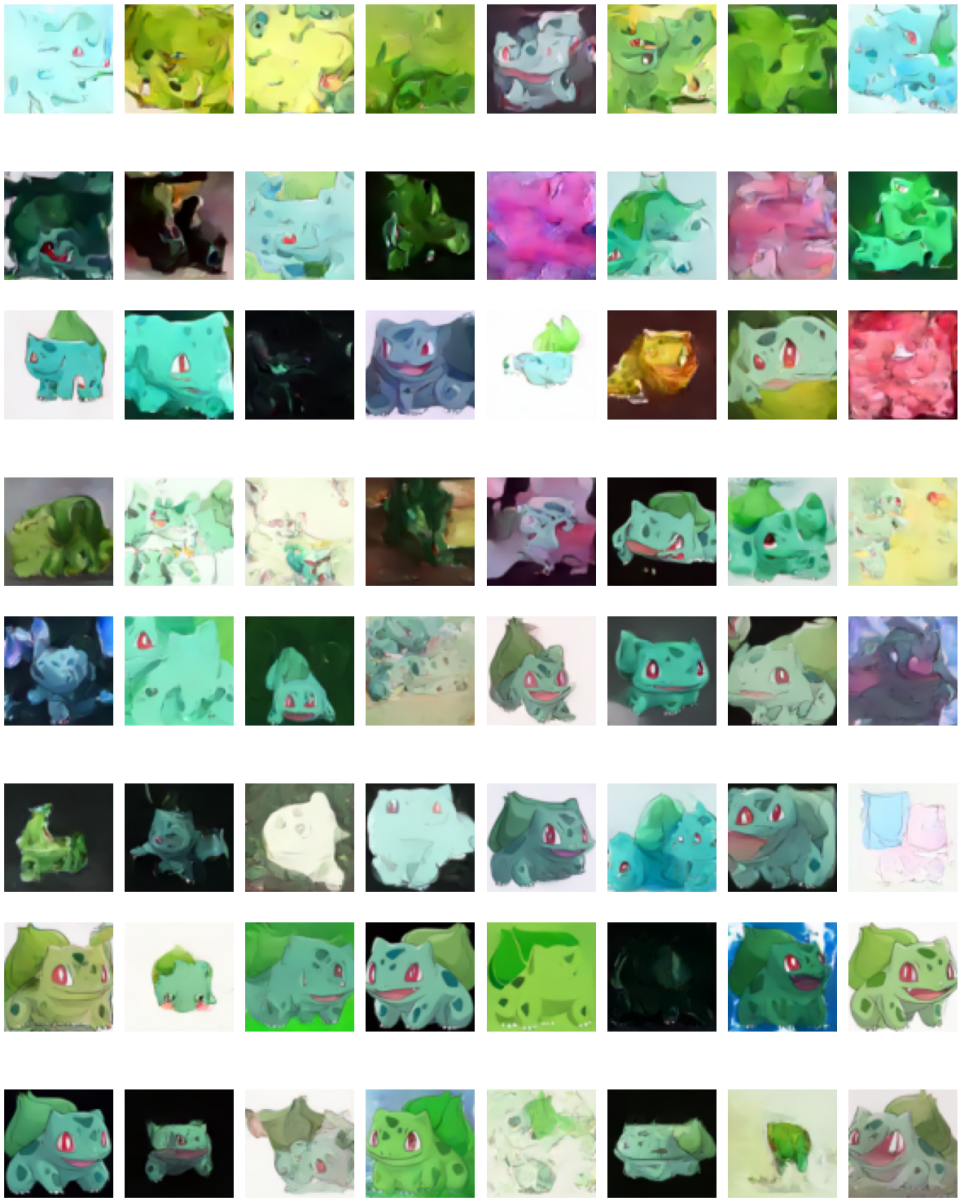

Figure 7: Generated images in sets of 16 from the training of the model. First set corresponds to the model after training for 200 epochs; second: 800 epochs; third: 1400 epochs; fourth: 2000 epochs.

Appendix B: Python Code

Most of the code was inspired by https://keras.io/examples/generative/ddpm/, and several functions were used without change.

```
#%%
import kagglehub

# Download latest version
path = kagglehub.dataset_download("thedagger/pokemon-generation-one")

print("Path to dataset files:", path)


#%%
import math
import matplotlib.pyplot as plt
import random

# Requires TensorFlow >=2.11 for the GroupNormalization layer.
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

#%%
from sklearn.model_selection import train_test_split
import numpy as np
from PIL import Image
import os


bulbasaur = os.path.join(os.getcwd(), "Bulbasaur")
file_paths = []
labels = []

for file_name in os.listdir(bulbasaur):
    file_path = os.path.join(bulbasaur, file_name)
    if os.path.isfile(file_path):
        file_paths.append(file_path)
        labels.append("bulbasaur")

train_images_ds = tf.data.Dataset.from_tensor_slices(file_paths)

X_train, X_test, y_train, y_test = train_test_split(file_paths, labels, test_size=0.1,
     random_state=42)

train_images_ds = tf.data.Dataset.from_tensor_slices(X_train)
test_images_ds = tf.data.Dataset.from_tensor_slices(X_test)


# %%
batch_size = 1
num_epochs = 2000  # Just for the sake of demonstration
total_timesteps = 1000
norm_groups = 8  # Number of groups used in GroupNormalization layer
learning_rate = 0.001

img_size = 64
img_channels = 3
clip_min = -1.0
clip_max = 1.0

first_conv_channels = 32
channel_multiplier = [1, 2, 4, 8]
```

```
widths = [first_conv_channels * mult for mult in channel_multiplier]
has_attention = [False, False, True, True]
num_res_blocks = 2  # Number of residual blocks

# %%
def augment(img):
    """Flips an image left/right randomly."""
    return tf.image.random_flip_left_right(img)
# tf.image.random_contrast(img)

def resize_and_rescale(img, size):
    """Resize the image to the desired size first and then
    rescale the pixel values in the range [-1.0, 1.0].

    Args:
        img: Image tensor
        size: Desired image size for resizing
    Returns:
        Resized and rescaled image tensor
    """

    height = tf.shape(img)[0]
    width = tf.shape(img)[1]
    crop_size = tf.minimum(height, width)

    img = tf.image.crop_to_bounding_box(
        img,
        (height - crop_size) // 2,
        (width - crop_size) // 2,
        crop_size,
        crop_size,
    )

    # Resize
    img = tf.cast(img, dtype=tf.float32)
    img = tf.image.resize(img, size=size, antialias=True)

    # Rescale the pixel values
    img = img / 127.5 - 1.0
    img = tf.clip_by_value(img, clip_min, clip_max)
    return img


def train_preprocessing(file_path):
    img = tf.io.read_file(file_path)  # Read the image file
    img = tf.image.decode_image(img, channels=img_channels)  # Decode as RGB
    img = resize_and_rescale(img, size=(img_size, img_size))
    img = augment(img)
    return img

train_ds = (
    train_images_ds.map(train_preprocessing, num_parallel_calls=tf.data.AUTOTUNE)
    .batch(batch_size, drop_remainder=True)
    .shuffle(batch_size * 2)
    .prefetch(tf.data.AUTOTUNE)
)

test_ds = (
    test_images_ds.map(train_preprocessing, num_parallel_calls=tf.data.AUTOTUNE)
    .batch(batch_size, drop_remainder=True)
    .shuffle(batch_size * 2)
    .prefetch(tf.data.AUTOTUNE)
)

```

```python

# %%
class GaussianDiffusion:
    """Gaussian diffusion utility.

    Args:
        beta_start: Start value of the scheduled variance
        beta_end: End value of the scheduled variance
        timesteps: Number of time steps in the forward process
    """

    def __init__(
        self,
        beta_start=1e-4,
        beta_end=0.02,
        timesteps=500,
        clip_min=-1.0,
        clip_max=1.0,
    ):
        self.beta_start = beta_start
        self.beta_end = beta_end
        self.timesteps = timesteps
        self.clip_min = clip_min
        self.clip_max = clip_max

        # Define the linear variance schedule
        self.betas = betas = np.linspace(
            beta_start,
            beta_end,
            timesteps,
            dtype=np.float64,  # Using float64 for better precision
        )
        self.num_timesteps = int(timesteps)

        alphas = 1.0 - betas
        alphas_cumprod = np.cumprod(alphas, axis=0)
        alphas_cumprod_prev = np.append(1.0, alphas_cumprod[:-1])

        self.betas = tf.constant(betas, dtype=tf.float32)
        self.alphas_cumprod = tf.constant(alphas_cumprod, dtype=tf.float32)
        self.alphas_cumprod_prev = tf.constant(alphas_cumprod_prev, dtype=tf.float32)

        # Calculations for diffusion q(x_t | x_{t-1}) and others
        self.sqrt_alphas_cumprod = tf.constant(
            np.sqrt(alphas_cumprod), dtype=tf.float32
        )

        self.sqrt_one_minus_alphas_cumprod = tf.constant(
            np.sqrt(1.0 - alphas_cumprod), dtype=tf.float32
        )

        self.log_one_minus_alphas_cumprod = tf.constant(
            np.log(1.0 - alphas_cumprod), dtype=tf.float32
        )

        self.sqrt_recip_alphas_cumprod = tf.constant(
            np.sqrt(1.0 / alphas_cumprod), dtype=tf.float32
        )
        self.sqrt_recipm1_alphas_cumprod = tf.constant(
            np.sqrt(1.0 / alphas_cumprod - 1), dtype=tf.float32
        )

        # Calculations for posterior q(x_{t-1} | x_t, x_0)
        posterior_variance = (
```

```
            betas * (1.0 - alphas_cumprod_prev) / (1.0 - alphas_cumprod)
        )
        self.posterior_variance = tf.constant(posterior_variance, dtype=tf.float32)

        # Log calculation clipped because the posterior variance is 0 at the beginning
        # of the diffusion chain
        self.posterior_log_variance_clipped = tf.constant(
            np.log(np.maximum(posterior_variance, 1e-20)), dtype=tf.float32
        )

        self.posterior_mean_coef1 = tf.constant(
            betas * np.sqrt(alphas_cumprod_prev) / (1.0 - alphas_cumprod),
            dtype=tf.float32,
        )

        self.posterior_mean_coef2 = tf.constant(
            (1.0 - alphas_cumprod_prev) * np.sqrt(alphas) / (1.0 - alphas_cumprod),
            dtype=tf.float32,
        )

    def _extract(self, a, t, x_shape):
        """Extract some coefficients at specified timesteps,
        then reshape to [batch_size, 1, 1, 1, 1, ...] for broadcasting purposes.

        Args:
            a: Tensor to extract from
            t: Timestep for which the coefficients are to be extracted
            x_shape: Shape of the current batched samples
        """
        batch_size = x_shape[0]
        out = tf.gather(a, t)
        return tf.reshape(out, [batch_size, 1, 1, 1])

    def q_mean_variance(self, x_start, t):
        """Extracts the mean, and the variance at current timestep.

        Args:
            x_start: Initial sample (before the first diffusion step)
            t: Current timestep
        """
        x_start_shape = tf.shape(x_start)
        mean = self._extract(self.sqrt_alphas_cumprod, t, x_start_shape) * x_start
        variance = self._extract(1.0 - self.alphas_cumprod, t, x_start_shape)
        log_variance = self._extract(
            self.log_one_minus_alphas_cumprod, t, x_start_shape
        )
        return mean, variance, log_variance

    def q_sample(self, x_start, t, noise):
        """Diffuse the data.

        Args:
            x_start: Initial sample (before the first diffusion step)
            t: Current timestep
            noise: Gaussian noise to be added at the current timestep
        Returns:
            Diffused samples at timestep `t`
        """
        x_start_shape = tf.shape(x_start)
        return (
            self._extract(self.sqrt_alphas_cumprod, t, tf.shape(x_start)) * x_start
            + self._extract(self.sqrt_one_minus_alphas_cumprod, t, x_start_shape)
            * noise
        )
```

```
251
252     def predict_start_from_noise(self, x_t, t, noise):
253         x_t_shape = tf.shape(x_t)
254         return (
255             self._extract(self.sqrt_recip_alphas_cumprod, t, x_t_shape) * x_t
256             - self._extract(self.sqrt_recipm1_alphas_cumprod, t, x_t_shape) * noise
257         )
258
259     def q_posterior(self, x_start, x_t, t):
260         """Compute the mean and variance of the diffusion
261         posterior q(x_{t-1} | x_t, x_0).
262
263         Args:
264             x_start: Stating point(sample) for the posterior computation
265             x_t: Sample at timestep `t`
266             t: Current timestep
267         Returns:
268             Posterior mean and variance at current timestep
269         """
270
271         x_t_shape = tf.shape(x_t)
272         posterior_mean = (
273             self._extract(self.posterior_mean_coef1, t, x_t_shape) * x_start
274             + self._extract(self.posterior_mean_coef2, t, x_t_shape) * x_t
275         )
276         posterior_variance = self._extract(self.posterior_variance, t, x_t_shape)
277         posterior_log_variance_clipped = self._extract(
278             self.posterior_log_variance_clipped, t, x_t_shape
279         )
280         return posterior_mean, posterior_variance, posterior_log_variance_clipped
281
282     def p_mean_variance(self, pred_noise, x, t, clip_denoised=True):
283         x_recon = self.predict_start_from_noise(x, t=t, noise=pred_noise)
284         if clip_denoised:
285             x_recon = tf.clip_by_value(x_recon, self.clip_min, self.clip_max)
286
287         model_mean, posterior_variance, posterior_log_variance = self.q_posterior(
288             x_start=x_recon, x_t=x, t=t
289         )
290         return model_mean, posterior_variance, posterior_log_variance
291
292     def p_sample(self, pred_noise, x, t, clip_denoised=True):
293         """Sample from the diffusion model.
294
295         Args:
296             pred_noise: Noise predicted by the diffusion model
297             x: Samples at a given timestep for which the noise was predicted
298             t: Current timestep
299             clip_denoised (bool): Whether to clip the predicted noise
300                 within the specified range or not.
301         """
302         model_mean, _, model_log_variance = self.p_mean_variance(
303             pred_noise, x=x, t=t, clip_denoised=clip_denoised
304         )
305         noise = tf.random.normal(shape=x.shape, dtype=x.dtype)
306         # No noise when t == 0
307         nonzero_mask = tf.reshape(
308             1 - tf.cast(tf.equal(t, 0), tf.float32), [tf.shape(x)[0], 1, 1, 1]
309         )
310         return model_mean + nonzero_mask * tf.exp(0.5 * model_log_variance) * noise
311
312
313 # %%
314 # Kernel initializer to use
```

```python
def kernel_init(scale):
    scale = max(scale, 1e-10)
    return keras.initializers.VarianceScaling(
        scale, mode="fan_avg", distribution="uniform"
    )


class AttentionBlock(layers.Layer):
    """Applies self-attention.

    Args:
        units: Number of units in the dense layers
        groups: Number of groups to be used for GroupNormalization layer
    """

    def __init__(self, units, groups=8, **kwargs):
        self.units = units
        self.groups = groups
        super().__init__(**kwargs)

        self.norm = layers.GroupNormalization(groups=groups)
        self.query = layers.Dense(units, kernel_initializer=kernel_init(1.0))
        self.key = layers.Dense(units, kernel_initializer=kernel_init(1.0))
        self.value = layers.Dense(units, kernel_initializer=kernel_init(1.0))
        self.proj = layers.Dense(units, kernel_initializer=kernel_init(0.0))

    def call(self, inputs):
        batch_size = tf.shape(inputs)[0]
        height = tf.shape(inputs)[1]
        width = tf.shape(inputs)[2]
        scale = tf.cast(self.units, tf.float32) ** (-0.5)

        inputs = self.norm(inputs)
        q = self.query(inputs)
        k = self.key(inputs)
        v = self.value(inputs)

        attn_score = tf.einsum("bhwc, bHWc->bhwHW", q, k) * scale
        attn_score = tf.reshape(attn_score, [batch_size, height, width, height * width
    ])

        attn_score = tf.nn.softmax(attn_score, -1)
        attn_score = tf.reshape(attn_score, [batch_size, height, width, height, width
    ])

        proj = tf.einsum("bhwHW,bHWc->bhwc", attn_score, v)
        proj = self.proj(proj)
        return inputs + proj


class TimeEmbedding(layers.Layer):
    def __init__(self, dim, **kwargs):
        super().__init__(**kwargs)
        self.dim = dim
        self.half_dim = dim // 2
        self.emb = math.log(10000) / (self.half_dim - 1)
        self.emb = tf.exp(tf.range(self.half_dim, dtype=tf.float32) * -self.emb)

    def call(self, inputs):
        inputs = tf.cast(inputs, dtype=tf.float32)
        emb = inputs[:, None] * self.emb[None, :]
        emb = tf.concat([tf.sin(emb), tf.cos(emb)], axis=-1)
        return emb

```

```
377
378 def ResidualBlock(width, groups=8, activation_fn=keras.activations.swish):
379     def apply(inputs):
380         x, t = inputs
381         input_width = x.shape[3]
382
383         if input_width == width:
384             residual = x
385         else:
386             residual = layers.Conv2D(
387                 width, kernel_size=1, kernel_initializer=kernel_init(1.0)
388             )(x)
389
390         temb = activation_fn(t)
391         temb = layers.Dense(width, kernel_initializer=kernel_init(1.0))(temb)[
392             :, :
393         ]
394
395         x = layers.GroupNormalization(groups=groups)(x)
396         x = activation_fn(x)
397         x = layers.Conv2D(
398             width, kernel_size=3, padding="same", kernel_initializer=kernel_init(1.0)
399         )(x)
400
401         x = layers.Add()([x, temb])
402         x = layers.GroupNormalization(groups=groups)(x)
403         x = activation_fn(x)
404
405         x = layers.Conv2D(
406             width, kernel_size=3, padding="same", kernel_initializer=kernel_init(0.0)
407         )(x)
408         x = layers.Add()([x, residual])
409         return x
410
411     return apply
412
413
414 def DownSample(width):
415     def apply(x):
416         x = layers.Conv2D(
417             width,
418             kernel_size=3,
419             strides=2,
420             padding="same",
421             kernel_initializer=kernel_init(1.0),
422         )(x)
423         return x
424
425     return apply
426
427
428 def UpSample(width, interpolation="nearest"):
429     def apply(x):
430         x = layers.UpSampling2D(size=2, interpolation=interpolation)(x)
431         x = layers.Conv2D(
432             width, kernel_size=3, padding="same", kernel_initializer=kernel_init(1.0)
433         )(x)
434         return x
435
436     return apply
437
438
439 def TimeMLP(units, activation_fn=keras.activations.swish):
440     def apply(inputs):
```

```
        temb = layers.Dense(
            units, activation=activation_fn, kernel_initializer=kernel_init(1.0)
        )(inputs)
        temb = layers.Dense(units, kernel_initializer=kernel_init(1.0))(temb)
        return temb

    return apply

# %%
def build_model(
    img_size,
    img_channels,
    widths,
    has_attention,
    num_res_blocks=2,
    norm_groups=8,
    interpolation="nearest",
    activation_fn=keras.activations.swish,
):
    image_input = layers.Input(
        shape=(img_size, img_size, img_channels), name="image_input"
    )
    time_input = keras.Input(shape=(), dtype=tf.int64, name="time_input")

    x = layers.Conv2D(
        first_conv_channels,
        kernel_size=(3, 3),
        padding="same",
        kernel_initializer=kernel_init(1.0),
    )(image_input)

    temb = TimeEmbedding(dim=first_conv_channels * 4)(time_input)
    temb = TimeMLP(units=first_conv_channels * 4, activation_fn=activation_fn)(temb)

    skips = [x]

    # DownBlock
    for i in range(len(widths)):
        for _ in range(num_res_blocks):
            x = ResidualBlock(
                widths[i], groups=norm_groups, activation_fn=activation_fn
            )([x, temb])
            if has_attention[i]:
                x = AttentionBlock(widths[i], groups=norm_groups)(x)
            skips.append(x)

        if widths[i] != widths[-1]:
            x = DownSample(widths[i])(x)
            skips.append(x)

    # MiddleBlock
    x = ResidualBlock(widths[-1], groups=norm_groups, activation_fn=activation_fn)(
        [x, temb]
    )
    x = AttentionBlock(widths[-1], groups=norm_groups)(x)
    x = ResidualBlock(widths[-1], groups=norm_groups, activation_fn=activation_fn)(
        [x, temb]
    )

    # UpBlock
    for i in reversed(range(len(widths))):
        for _ in range(num_res_blocks + 1):
            x = layers.Concatenate(axis=-1)([x, skips.pop()])
            x = ResidualBlock(
```

```
                widths[i], groups=norm_groups, activation_fn=activation_fn
            )([x, temb])
            if has_attention[i]:
                x = AttentionBlock(widths[i], groups=norm_groups)(x)

        if i != 0:
            x = UpSample(widths[i], interpolation=interpolation)(x)

    # End block
    x = layers.GroupNormalization(groups=norm_groups)(x)
    x = activation_fn(x)
    x = layers.Conv2D(3, (3, 3), padding="same", kernel_initializer=kernel_init(0.0))(
    x)
    return keras.Model([image_input, time_input], x, name="unet")

# %%
class DiffusionModel(keras.Model):
    def __init__(self, network, ema_network, timesteps, gdf_util, ema=0.999):
        super().__init__()
        self.network = network
        self.ema_network = ema_network
        self.timesteps = timesteps
        self.gdf_util = gdf_util
        self.ema = ema

    def train_step(self, images):
        # 1. Get the batch size
        batch_size = tf.shape(images)[0]

        # 2. Sample timesteps uniformly
        t = tf.random.uniform(
            minval=0, maxval=self.timesteps, shape=(batch_size,), dtype=tf.int64
        )

        with tf.GradientTape() as tape:
            # 3. Sample random noise to be added to the images in the batch
            noise_train = tf.random.normal(shape=tf.shape(images), dtype=images.dtype)

            # 4. Diffuse the images with noise
            images_t = self.gdf_util.q_sample(images, t, noise_train)

            # 5. Pass the diffused images and time steps to the network
            pred_noise = self.network([images_t, t], training=True)

            # 6. Calculate the loss
            loss = self.loss(noise_train, pred_noise)

        # 7. Get the gradients
        gradients = tape.gradient(loss, self.network.trainable_weights)

        # 8. Update the weights of the network
        self.optimizer.apply_gradients(zip(gradients, self.network.trainable_weights))

        # 9. Updates the weight values for the network with EMA weights
        for weight, ema_weight in zip(self.network.weights, self.ema_network.weights):
            ema_weight.assign(self.ema * ema_weight + (1 - self.ema) * weight)

        # 10. Return loss values
        return {"training_loss": loss}

    def test_step(self, images):

        batch_size = tf.shape(images)[0]
        t = tf.random.uniform(
```

```
            minval=0, maxval=self.timesteps, shape=(batch_size,), dtype=tf.int64
        )

        noise_train = tf.random.normal(shape=tf.shape(images), dtype=images.dtype)
        images_t = self.gdf_util.q_sample(images, t, noise_train)
        pred_noise = self.network([images_t, t], training=False)
        loss = self.loss(noise_train, pred_noise)

        return {"validation_loss": loss}

    def generate_images(self, num_images=16, initial_noise=None):

        if initial_noise is None:
            # 1. Randomly sample noise (starting point for reverse process)
            samples = tf.random.normal(
                shape=(num_images, img_size, img_size, img_channels), dtype=tf.float32
            )
            initial_noise = samples
        else:
            samples = initial_noise

        # 2. Sample from the model iteratively
        for t in reversed(range(0, self.timesteps)):
            tt = tf.cast(tf.fill(num_images, t), dtype=tf.int64)
            pred_noise = self.ema_network.predict(
                [samples, tt], verbose=0, batch_size=num_images
            )
            samples = self.gdf_util.p_sample(
                pred_noise, samples, tt, clip_denoised=True
            )
        # 3. Return generated samples
        return samples, initial_noise

    def plot_images(
        self, epoch=None, logs=None, num_rows=2, num_cols=8, figsize=(12, 5), div_int
    =5, initial_noise=None
    ):
        """Utility to plot images using the diffusion model during training."""

        if (epoch+1) % div_int == 0:

            generated_samples, _ = self.generate_images(num_images=num_rows * num_cols
    ,  initial_noise=initial_noise)
            generated_samples = (
                tf.clip_by_value(generated_samples * 127.5 + 127.5, 0.0, 255.0)
                .numpy()
                .astype(np.uint8)
            )

            _, ax = plt.subplots(num_rows, num_cols, figsize=figsize)
            if num_rows == 1 and num_cols == 1:
                ax.imshow(generated_samples[0])
                ax.axis("off")
            else:
                ax = np.array(ax).reshape(-1)
                for i, image in enumerate(generated_samples):
                    ax[i].imshow(image)
                    ax[i].axis("off")

            plt.tight_layout()
            plt.show()

# %%
# Build the unet model
```

```
network = build_model(
    img_size=img_size,
    img_channels=img_channels,
    widths=widths,
    has_attention=has_attention,
    num_res_blocks=num_res_blocks,
    norm_groups=norm_groups,
    activation_fn=keras.activations.swish,
)

ema_network = build_model(
    img_size=img_size,
    img_channels=img_channels,
    widths=widths,
    has_attention=has_attention,
    num_res_blocks=num_res_blocks,
    norm_groups=norm_groups,
    activation_fn=keras.activations.swish,
)
ema_network.set_weights(network.get_weights())  # Initially the weights are the same

# Get an instance of the Gaussian Diffusion utilities
gdf_util = GaussianDiffusion(timesteps=total_timesteps)


# %%
# Get the model
model_3 = DiffusionModel(
    network=network,
    ema_network=ema_network,
    gdf_util=gdf_util,
    timesteps=total_timesteps,
)

# Compile the model
model_3.compile(
    loss=keras.losses.MeanSquaredError(),
    optimizer=keras.optimizers.Adam(learning_rate=learning_rate),
)

# Train the model
history = model_3.fit(
    train_ds,
    validation_data = test_ds,
    epochs=num_epochs,
    batch_size=batch_size,
    callbacks=[keras.callbacks.LambdaCallback(on_epoch_end=model_1.plot_images)],
)

# %%
training_loss = history.history['training_loss']
validation_loss = history.history['val_validation_loss']

hej = [  training_loss[i] for i in range(len(training_loss)) if i % 10 == 0 ]
hi = [  validation_loss[i] for i in range(len(training_loss)) if i % 10 == 0 ]

plt.figure(figsize=(10, 6))
plt.plot(hej, label="Training Loss")
plt.plot(hi, label="Validation Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Training and Validation Loss")
plt.legend()
plt.grid(True)
```

```
plt.show()

# %%
history2 = model_1.fit(
    train_ds,
    validation_data = test_ds,
    epochs=1000,
    batch_size=batch_size,
    callbacks=[keras.callbacks.LambdaCallback(on_epoch_end=model_1.plot_images)],
)

# %%
training_loss_2 = history2.history['training_loss']
validation_loss_2 = history2.history['val_validation_loss']

hej_2 = [  training_loss_2[i] for i in range(len(training_loss_2)) if i % 10 == 0 ]
hi_2 = [  validation_loss_2[i] for i in range(len(training_loss_2)) if i % 10 == 0 ]

training_loss_tot = hej + hej_2
validation_loss_tot = hi + hi_2

# %%
plt.figure(figsize=(10, 6))
plt.plot(training_loss_tot, label="Training Loss")
plt.plot(validation_loss_tot, label="Validation Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Training and Validation Loss")
plt.legend()
plt.grid(True)
plt.show()

# %%
# the noise to generate image from
num_images = 1
initial_noise = tf.random.normal(
                shape=(num_images, img_size, img_size, img_channels), dtype=tf.float32
            )

# Generate and plot some samples from same noise
for i in range(10):
    model_1.plot_images(num_rows=1, num_cols=1, div_int=1, epoch=0, initial_noise=
    initial_noise)

# %%
## generate model with more pokemons

bulbasaur = os.path.join(os.getcwd(), "Bulbasaur")
pikachu = os.path.join(os.getcwd(), "Pikachu")
squirtle = os.path.join(os.getcwd(), "Squirtle")

file_paths = []
labels = []

for file_name in os.listdir(bulbasaur):
    file_path = os.path.join(bulbasaur, file_name)
    if os.path.isfile(file_path):
        file_paths.append(file_path)
        labels.append("bulbasaur")

train_images_ds_new = tf.data.Dataset.from_tensor_slices(file_paths)

train_ds_new = (
    train_images_ds_new.map(train_preprocessing, num_parallel_calls=tf.data.AUTOTUNE)
```

```
    .batch(batch_size, drop_remainder=True)
    .shuffle(batch_size * 2)
    .prefetch(tf.data.AUTOTUNE)
)

# %%
betas_250 = np.linspace(
    0.0001,
    0.02,
    250,
    dtype=np.float64,  # Using float64 for better precision
)
alphas_250 = 1.0 - betas_250
alphas_cumprod_250 = np.cumprod(alphas_250, axis=0)

betas_500 = np.linspace(
    0.0001,
    0.02,
    500,
    dtype=np.float64,  # Using float64 for better precision
)
alphas_500 = 1.0 - betas_500
alphas_cumprod_500 = np.cumprod(alphas_500, axis=0)


betas_750 = np.linspace(
    0.0001,
    0.02,
    750,
    dtype=np.float64,  # Using float64 for better precision
)
alphas_750 = 1.0 - betas_750
alphas_cumprod_750 = np.cumprod(alphas_750, axis=0)

betas_1000 = np.linspace(
    0.0001,
    0.02,
    1000,
    dtype=np.float64,  # Using float64 for better precision
)
alphas_1000 = 1.0 - betas_1000
alphas_cumprod_1000 = np.cumprod(alphas_1000, axis=0)

betas_1250 = np.linspace(
    0.0001,
    0.02,
    1250,
    dtype=np.float64,  # Using float64 for better precision
)
alphas_1250 = 1.0 - betas_1250
alphas_cumprod_1250 = np.cumprod(alphas_1250, axis=0)


# %%
plt.plot(np.sqrt(alphas_cumprod_250), label="250")
plt.plot(np.sqrt(alphas_cumprod_500), label="500")
plt.plot(np.sqrt(alphas_cumprod_750), label="750")
plt.plot(np.sqrt(alphas_cumprod_1000), label="1000")
plt.plot(np.sqrt(alphas_cumprod_1250), label="1250")
plt.legend(title="Timestep T")
plt.grid(True)
plt.xlabel("t")
plt.ylabel("$\sqrt{\overline{\u03B1}_t}$")
# plt.xlim(200, 1300)
```

```
# plt.ylim(-0.01, 0.1)
plt.show()


# %%
# for plotting the forward process
# Instantiate the GaussianDiffusion class
diffusion = GaussianDiffusion(beta_start=1e-4, beta_end=0.02, timesteps=1001)

# Load or create an initial image (x_start)
# Example: Generate a simple synthetic image (grayscale gradient)
x_start = train_preprocessing("bulbasaur_example.png")

# %%
# Normalize and expand dimensions to match expected shape [batch_size, height, width, channels]
x_start = np.expand_dims(np.expand_dims(x_start, axis=0), axis=-1)  # Shape: [1, 64, 64, 1]

# Initialize a random number generator for noise
rng = np.random.default_rng(seed=42)

# Timesteps to visualize (e.g., start, middle, and final timesteps)
timesteps_to_visualize = [0, 100, 250, 500, 1000]

# Plot the diffusion process
plt.figure(figsize=(12, 6))
for idx, t in enumerate(timesteps_to_visualize):
    # Add noise progressively
    t_tensor = tf.constant([t], dtype=tf.int32)  # Current timestep
    noise = rng.normal(size=x_start.shape, scale=1.0).astype(np.float32)  # Gaussian noise
    x_t = diffusion.q_sample(x_start, t_tensor, noise)  # Diffused image

    # Clip values to [-1, 1] range and rescale to [0, 1] for visualization
    x_t_clipped = np.clip(x_t.numpy(), -1, 1)
    x_t_rescaled = (x_t_clipped + 1) / 2.0  # Rescale to [0, 1]

    # Plot the noisy image
    plt.subplot(1, len(timesteps_to_visualize), idx + 1)
    plt.imshow(x_t_rescaled[0, :, :, 0])
    plt.title(f"Timestep {t}")
    plt.axis("off")

plt.tight_layout()
plt.show()


# Exploratory analysis plot
# %%
def get_dims(filepath):
    img = Image.open(filepath)
    arr = np.array(img)

    if arr.ndim == 2:  # Grayscale image
        height, width = arr.shape
    elif arr.ndim == 3:  # Color image
        height, width, _ = arr.shape
    else:
        raise ValueError(f"Unexpected image dimensions: {arr.shape}")

    return height, width

bulbasaur = os.path.join(os.getcwd(), "Bulbasaur")
```

```
882
883 filepaths = []
884 for file_name in os.listdir(bulbasaur):
885     filepath = os.path.join(bulbasaur, file_name)
886     if os.path.isfile(filepath):
887         filepaths.append(filepath)
888
889 image_sizes = {}
890 for filepath in filepaths:
891     height, width = get_dims(filepath)
892     image_sizes[filepath] = (height, width)
893
894 height = []
895 width = []
896 for filepath, size in image_sizes.items():
897     height.append(size[0])
898     width.append(size[1])
899
900
901 # %%
902 plt.scatter(width, height, s=2)
903 plt.xlabel("Width")
904 plt.ylabel("Height")
905 plt.grid(True)
906 plt.show()
```

References

Loshchilov, Ilya and Frank Hutter (2019). *Decoupled Weight Decay Regularization*. arXiv: 1711.05101 [cs.LG]. URL: https://arxiv.org/abs/1711.05101.

Ho, Jonathan, Ajay Jain, and Pieter Abbeel (2020). *Denoising Diffusion Probabilistic Models*. arXiv: 2006.11239 [cs.LG]. URL: https://arxiv.org/abs/2006.11239.

Higham, Catherine F, Desmond J Higham, and Peter Grindrod (2023). "Diffusion Models for Generative Artificial Intelligence: An Introduction for Applied Mathematicians". In: *arXiv preprint arXiv:2312.14977*.

Williams, Christopher et al. (2023). "A unified framework for U-Net design and analysis". In: *Advances in Neural Information Processing Systems* 36, pp. 27745–27782.