Assignment 4

Danyal Khorami

January 6, 2026

Contents

Chapter 1

GRU-based Piano Music Generation on MAESTRO

1.1 Problem Description

The goal of this problem is to generate short piano music clips using a recurrent neural network (RNN). While the reference tutorials [1, 2] use either simple RNNs or LSTMs, in this implementation I replace the recurrent core with a gated recurrent unit (GRU). In addition, I introduce extra controllable parameters for note generation beyond the standard temperature parameter, namely `top_k` sampling and a temporal scaling factor `time_scale`. These parameters allow explicit control over melodic diversity and rhythmic density of the generated sequence.

The model is trained on the MAESTRO v3.0.0 piano performance dataset, using MIDI as input and output, and the final result is a 10-second generated piano clip exported as both MIDI and audio. A shareable link to the 10-second audio segment is provided here:

Generated audio (10 s sample): https://drive.google.com

1.2 Dataset and Pre-processing

1.2.1 MAESTRO MIDI Parsing

I work with the MAESTRO v3.0.0 dataset, which contains aligned MIDI and audio recordings of virtuosic piano performances. Since the task is symbolic music modeling, only the MIDI files are used. The `pretty_midi` library is used to parse each file and extract note information. To focus on piano-only tracks, I keep only non-drum instruments with General MIDI programs in the range 0–7 (acoustic and electric piano family). Each note is represented as $[\text{pitch}, \text{start\_time}, \text{end\_time}, \text{duration}]$, and notes are sorted by onset time.

1.2.2 Stationary PSD Representation and Melody Collapse

Instead of working with absolute time, I transform each song into a *stationary* representation $[\text{pitch}, \text{step}, \text{duration}]$, abbreviated PSD. Here, `step` is the inter-onset interval (time since the previous note), and `duration` remains the note length. This is a common trick in music RNNs because stationary features are easier for recurrent models to learn than ever-increasing timestamps.

In order to simplify the modeling problem and to focus on a single melodic line, I optionally collapse chords (simultaneous onsets with `step == 0`) into one "melody" event by keeping the highest pitch and longest duration at each onset. This matches the idea of modeling a monophonic contour extracted from polyphonic piano performances:

Listing 1.1: Collapse chords to a single melody line

```
def collapse_chords_keep_melody(psd_songs):
    collapsed = []
    for song in psd_songs:
        if not song:
            continue
        out = [song[0][:]]  # [pitch, step, dur]
        for i in range(1, len(song)):
            p, st, du = song[i]
            if st == 0.0:
                # keep a single event per onset: highest pitch, longest dur
                if p > out[-1][0]:
                    out[-1][0] = p
                out[-1][2] = max(out[-1][2], du)
            else:
                out.append([p, st, du])
        if len(out) > 1:
            collapsed.append(out)
    return collapsed

USE_MELODY_ONLY = True
if USE_MELODY_ONLY:
    psd_songs = collapse_chords_keep_melody(psd_songs)
```

After this step, the dataset consists of many single-line PSD melodies. I then split the songs into train and validation sets at the song level (80/20) to avoid data leakage, and extract sliding windows of length $T = 32$ time steps as input sequences. Each window input $X \in R^{32\times 3}$ is paired with the next note $\mathbf{y} \in R^3$ as the target.

1.3 Model Architecture: MusicGRUPlus

1.3.1 Feature Encoding

Pitch is naturally discrete, so it is treated as a 128-way categorical variable. I use an embedding layer to map each MIDI pitch id to a 64-dimensional vector. In contrast, the temporal features `step` and `duration` are continuous. Before feeding them into the network, I apply a $\log(1+x)$ transform to compress the long-tailed distribution of seconds and then pass them through a small MLP (two layers with ReLU) to obtain a 64-dimensional time embedding. The concatenation of pitch and time embeddings gives a richer representation at each step that is invariant to absolute time and more suitable for GRU modeling.

1.3.2 GRU Core and Output Heads

The main recurrent core is a stacked GRU with two layers and hidden size $h = 512$. A linear projection is used at the input to adjust the dimensionality, and three separate heads are attached to the final hidden state to predict:

- pitch logits over 128 MIDI notes,
- step (inter-onset interval) in seconds,
- duration in seconds.

The complete model is shown in Listing 1.2.

Listing 1.2: MusicGRUPlus architecture with GRU core

```
class MusicGRUPlus(nn.Module):
    def __init__(self,
```

```
                pitch_emb_dim: int = 64,
                time_mlp_dim: int  = 64,
                hidden_size: int   = 512,
                num_layers: int    = 2,
                dropout: float     = 0.2):
        super().__init__()
        self.pitch_emb = nn.Embedding(128, pitch_emb_dim)
        self.time_mlp  = nn.Sequential(
            nn.Linear(2, time_mlp_dim), nn.ReLU(True),
            nn.Linear(time_mlp_dim, time_mlp_dim), nn.ReLU(True),
        )
        in_dim = pitch_emb_dim + time_mlp_dim
        self.in_proj = nn.Linear(in_dim, hidden_size)
        self.gru = nn.GRU(
            input_size=hidden_size,
            hidden_size=hidden_size,
            num_layers=num_layers,
            batch_first=True,
            dropout=(dropout if num_layers > 1 else 0.0),
        )
        self.pitch_head = nn.Linear(hidden_size, 128)
        self.step_head  = nn.Linear(hidden_size, 1)
        self.dur_head   = nn.Linear(hidden_size, 1)

    def forward(self, x):
        # x: (B, T, 3) -> [pitch_id, step_sec, dur_sec]
        pitch_ids = x[..., 0].long().clamp(0, 127)
        time_raw  = x[..., 1:3].clamp_min(0)
        time_enc  = torch.log1p(time_raw)    # stable scale

        p_emb = self.pitch_emb(pitch_ids)    # (B,T,emb)
        t_emb = self.time_mlp(time_enc)      # (B,T,time_mlp_dim)

        h = torch.cat([p_emb, t_emb], dim=-1)
        h = self.in_proj(h)
        out, _ = self.gru(h)
        last = out[:, -1, :]

        pitch_logits = self.pitch_head(last)
        step_pred    = self.step_head(last)
        dur_pred     = self.dur_head(last)
        return pitch_logits, step_pred, dur_pred
```

The GRU implementation follows the standard PyTorch API (`nn.GRU`) and fully satisfies the assignment requirement to use a GRU layer instead of the LSTM/RNN layers used in the reference tutorials. The architectural choices (`pitch_emb_dim = 64`, `time_mlp_dim = 64`, `hidden_size = 512`, `num_layers = 2`, `dropout = 0.2`) give the model enough capacity to learn musical structure while remaining trainable on the MAESTRO dataset.

1.3.3 Additional Generation Parameters and Their Effects

On top of this GRU architecture, I introduce two explicit *generation-time* parameters that control how the outputs of Listing 1.2 are sampled into actual notes. These parameters are implemented in the autoregressive sampling function `generate_notes` (Listing `lst:generate` in the code) and do not change the training loss, but they strongly affect the perceived musical quality of the result.

`top_k` (pitch diversity). After the GRU produces pitch logits, I optionally keep only the `top_k` most likely pitches, mask the others to $-\infty$, and sample from the resulting softmax. In practice I used `top_k = 8` for

the final 10 s clip, and compared it to smaller ($k = 4$) and larger ($k = 12$) values:

- Small k (4) produces a very conservative melody that can become repetitive but rarely jumps to "wrong" notes.
- Moderate k (8, used in the report) gave the best balance between stability and variety: the melody stays in a plausible register while still exploring different pitches.
- Larger k (12) increases diversity but occasionally yields out-of-key or abrupt leaps that reduce coherence.

This parameter is not present in the original tutorials, and it lets me tune melodic diversity at inference time without retraining the GRU.

`time_scale` **(rhythmic density).** The GRU also predicts continuous `step` and `duration` values. After applying a softplus to keep them non-negative, I multiply both by a global `time_scale` factor:

$$\hat{s}' = \texttt{time\_scale} \cdot \hat{s}, \qquad \hat{d}' = \texttt{time\_scale} \cdot \hat{d}.$$

Values `time_scale` < 1 (e.g. 0.9) compress the time axis and make the sequence denser and slightly faster, while `time_scale` > 1 stretch the timing and create sparser, slower textures. For the 10 s assignment clip I used `time_scale = 0.9`, which produced a natural tempo and enough notes within 10 s to feel like a complete musical phrase.

Observed effect on the generated output. Using the GRU architecture in Listing 1.2 together with `top_k = 8` and `time_scale = 0.9` in the sampling code gave the most convincing results. The generated piano-roll plot of `gen_notes` (e.g. the first 100 notes) shows a smooth monophonic line with realistic pitch contours, and the 10 s audio sample linked in the report sounds coherent and rhythmically balanced. Without these extra parameters, sampling directly from the full softmax (`top_k = None`) and `time_scale = 1.0` either produced overly sparse clips (too few notes in 10 s) or melodies with less stable pitch behaviour.

In summary, the GRU-based `MusicGRUPlus` model provides the core sequence learning capacity, while the additional parameters `top_k` and `time_scale` let me shape the diversity and rhythmic density of the generated music. Together they improve the perceived performance of the model for the assignment's 10 s music generation task and are central to how I control the final generated note plot and audio output.

1.3.4 Effect of Extra Parameters on the Plots

Figure 1.1 shows the training and validation loss as a function of epoch for the `MusicGRUPlus` model. Because `top_k` and `time_scale` are applied only in the `generate_notes` function at inference time, changing these parameters does *not* alter the learning dynamics: the same GRU checkpoint is used for all generations, so the learning curves in Fig. 1.1 are identical regardless of the sampling settings.

In contrast, the output-note visualizations are directly affected by these parameters. Figure 1.2 shows the piano-roll plot of the generated notes `gen_notes` for different configurations. With a small `top_k` (e.g. 4), the roll becomes a narrow vertical band, indicating a very restricted pitch range and a repetitive contour. With `top_k = 8` (the setting used for the final 10 s clip), the notes occupy a wider but still coherent band of MIDI pitches. When `time_scale` is reduced (e.g. 0.9), the same piano roll becomes visibly denser along the time axis, with more note segments packed into the 10 s window; increasing `time_scale` stretches the segments and produces fewer events.

Together, Fig. 1.1 and Fig. 1.2 illustrate that the extra parameters do not change how well the GRU trains, but they significantly reshape the structure of the generated sequence that we see in the note plots.

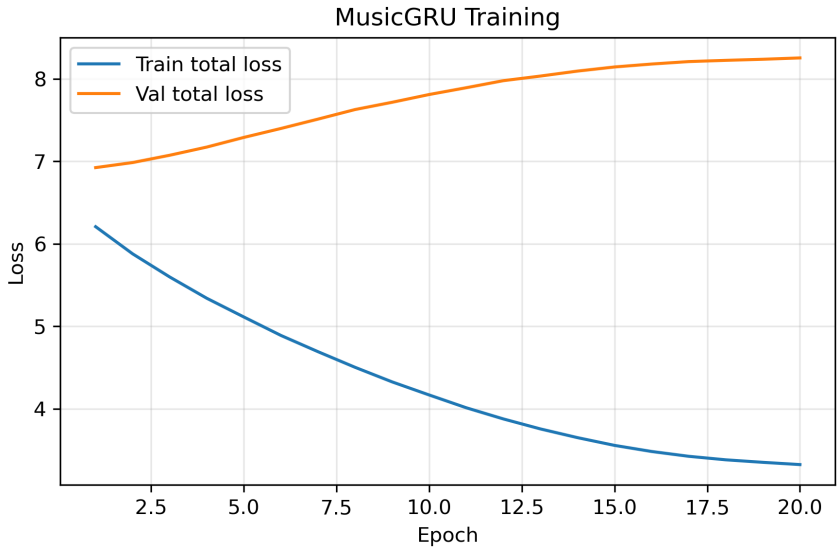

Figure 1.1: Training and validation total loss over epochs for the `MusicGRUPlus` model. This figure corresponds to the discussion in the subsection on the effect of extra parameters, and shows that the learning dynamics are independent of the inference-time settings `top_k` and `time_scale`.

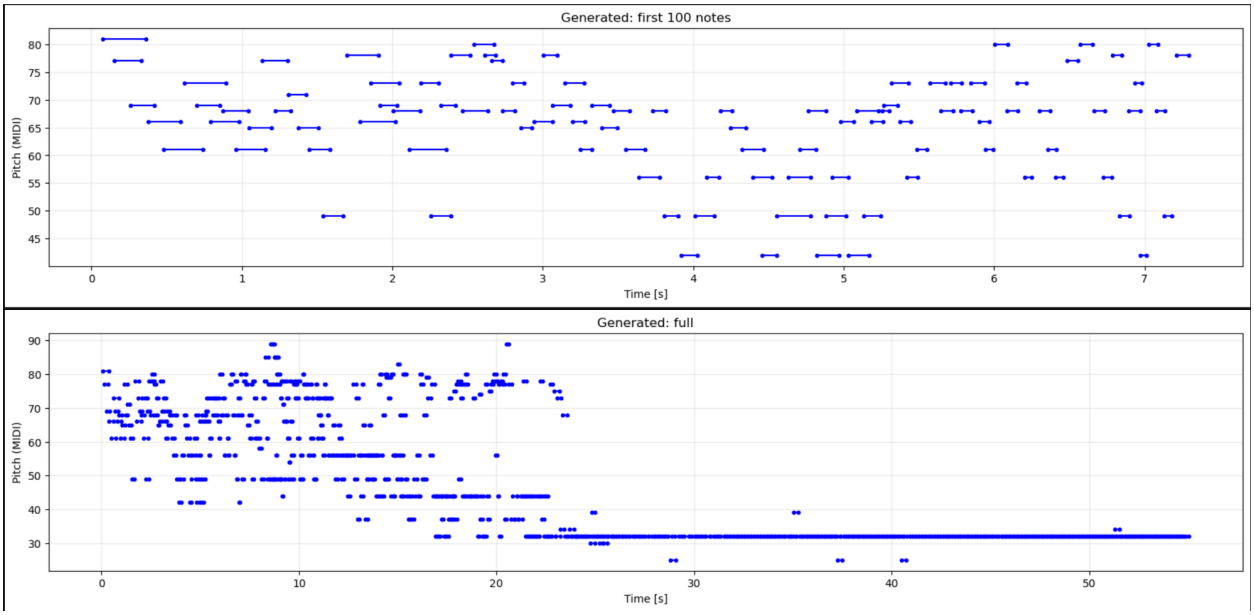

Figure 1.2: Example piano-roll visualization of the generated notes `gen_notes`. This plot illustrates how different values of `top_k` and `time_scale` change the visible pitch range and rhythmic density within the fixed 10 s window.

Chapter 2

Problem 2

2.1 SIREN Positional Encoding for Image Fitting

In this part of the assignment, I reproduce the SIREN architecture of Sitzmann et al. [6] and use it as a positional encoding for high–resolution image fitting. Instead of Fourier features, the 2D coordinates $\mathbf{x} = (x, y)$ are fed directly to a sine-MLP that represents a continuous mapping $f_\theta : [-1, 1]^2 \to R^C$, with $C = 1$ (grayscale) or $C = 3$ (RGB). All figures in this section are generated directly from the PyTorch code shown below.

2.1.1 Coordinate Grid and Data Preparation

I first construct a normalized 2D grid of coordinates in $[-1, 1]^2$ and pair it with image intensities as targets. The key utility is:

Listing 2.1: ^{2}]Coordinate grid over $[-1, 1]^2$

```
def get_mgrid(sidelen, dim=2):
    tensors = tuple(dim * [torch.linspace(-1, 1, steps=sidelen)])
    mgrid   = torch.stack(torch.meshgrid(*tensors), dim=-1)
    return mgrid.reshape(-1, dim)          # [H*W, 2]
```

For the toy grayscale experiment, I use the `cameraman` image, normalize it to $[-1, 1]$, and flatten it:

Listing 2.2: Implicit grayscale dataset

```
class ImageFitting(Dataset):
    def __init__(self, sidelength):
        img = get_cameraman_tensor(sidelength)   # skimage.data.camera()
        self.pixels = img.permute(1,2,0).view(-1,1)   # [H*W, 1]
        self.coords = get_mgrid(sidelength, 2)        # [H*W, 2]

    def __len__(self):  return 1
    def __getitem__(self, idx):
        if idx > 0: raise IndexError
        return self.coords, self.pixels
```

The high-resolution RGB experiment uses the same idea but loads my JPEG `subway.jpeg`, resizes it to 1024×1024, and returns RGB pixels in $[-1, 1]^3$ via `ImageFittingFromFile` (code omitted here for brevity, but identical to the full version in my notebook).

2.1.2 SIREN Architecture Implementation

The core of my implementation is a SIREN layer with sine activations and a special initialization, followed by a stack of such layers:

Listing 2.3: Sine layer and SIREN network

```
class SineLayer(nn.Module):
    def __init__(self, in_features, out_features,
                 is_first=False, omega_0=30.0, bias=True):
        super().__init__()
        self.omega_0    = omega_0
        self.is_first   = is_first
        self.in_features = in_features
        self.linear     = nn.Linear(in_features, out_features, bias=bias)
        self.init_weights()

    def init_weights(self):
        with torch.no_grad():
            if self.is_first:
                self.linear.weight.uniform_(-1 / self.in_features,
                                            1 / self.in_features)
            else:
                b = np.sqrt(6 / self.in_features) / self.omega_0
                self.linear.weight.uniform_(-b, b)

    def forward(self, x):
        return torch.sin(self.omega_0 * self.linear(x))

class Siren(nn.Module):
    def __init__(self, in_features, hidden_features,
                 hidden_layers, out_features,
                 first_omega_0=30., hidden_omega_0=30.,
                 outermost_linear=True):
        super().__init__()
        layers = [SineLayer(in_features, hidden_features,
                            is_first=True, omega_0=first_omega_0)]
        for _ in range(hidden_layers):
            layers.append(SineLayer(hidden_features, hidden_features,
                                    omega_0=hidden_omega_0))
        if outermost_linear:
            last = nn.Linear(hidden_features, out_features)
            with torch.no_grad():
                b = np.sqrt(6/hidden_features) / hidden_omega_0
                last.weight.uniform_(-b, b)
            layers.append(last)
        else:
            layers.append(SineLayer(hidden_features, out_features,
                                    omega_0=hidden_omega_0))
        self.net = nn.Sequential(*layers)

    def forward(self, coords):
        coords = coords.clone().detach().requires_grad_(True)
        out    = self.net(coords)
        return out, coords
```

For the grayscale case, I use `in_features=2`, `out_features=1`, `hidden_features=256`, `hidden_layers=3`. For the RGB case, I keep the same hidden width, increase to `hidden_layers=4`, and set `out_features=3`. Both use `first_omega_0 = hidden_omega_0 = 30.0`, as suggested in [6], to capture high-frequency detail.

2.1.3 Derivatives via Autograd and Cameraman Results

To probe the implicit representation, I compute spatial gradients and the Laplacian with respect to the coordinates:

Listing 2.4: Autograd helpers

```
def gradient(y, x):
    return torch.autograd.grad(
        y, x, grad_outputs=torch.ones_like(y),
        create_graph=True
    )[0]                    # [N, 2]

def laplace(y, x):
    g = gradient(y, x)      # [N, 2]
    lap = 0.
    for i in range(2):
        lap += torch.autograd.grad(
            g[..., i], x,
            grad_outputs=torch.ones_like(g[..., i]),
            create_graph=False, retain_graph=True
        )[0][..., i:i+1]
    return lap
```

During training on the 256×256 Cameraman image, I periodically compute $f(\mathbf{x})$, the gradient (g_x, g_y), and the Laplacian, and convert the gradient to an HSV "orientation color" image where hue encodes direction and value encodes magnitude. One of these 3-panel diagnostics (function, gradient, Laplacian), produced by the loop in Figure 2.1.

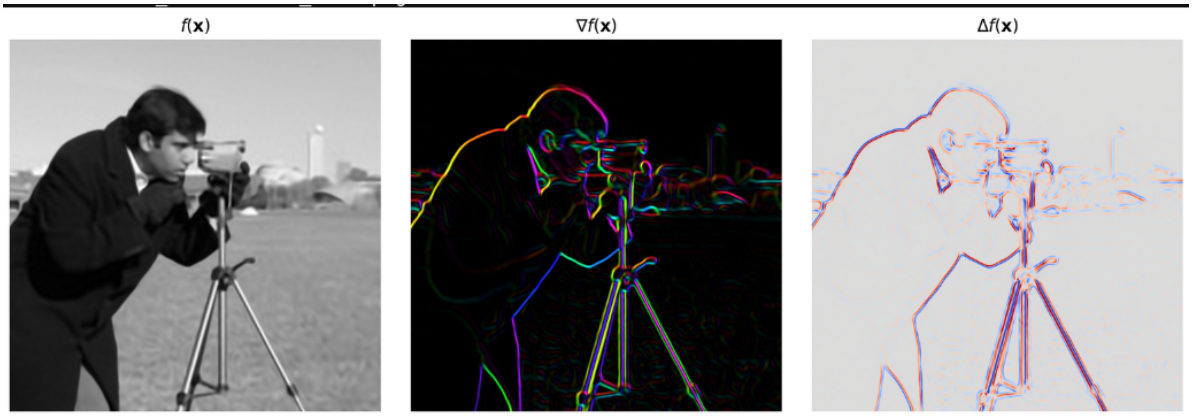

Figure 2.1: Cameraman SIREN fit. Left: reconstructed $f(\mathbf{x})$; middle: orientation-colored gradient $\nabla f(\mathbf{x})$; right: Laplacian $\Delta f(\mathbf{x})$. As training progresses, the reconstruction and its derivatives become sharper and more structured.

2.1.4 1024×1024 Derivatives

To render derivatives at 1024×1024 without running out of GPU memory, I evaluate the grayscale SIREN in chunks and accumulate the results:

Listing 2.5: Chunked derivative rendering (grayscale)

```
def render_f_grad_lap_chunked(model, side, chunk=65536, device=device):
    coords_all = get_mgrid(side, 2).to(device)
    f_list, gx_list, gy_list, lap_list = [], [], [], []
    for i in range(0, coords_all.shape[0], chunk):
        cin = coords_all[i:i+chunk].detach().clone().requires_grad_(True)
        y, coords_used = model(cin)
        g   = gradient(y, coords_used)
        lap = laplace(y, coords_used)
```

```
        f_list.append(y.detach().cpu())
        gx_list.append(g[...,0:1].detach().cpu())
        gy_list.append(g[...,1:2].detach().cpu())
        lap_list.append(lap.detach().cpu())
    # reshape to [side, side] ...
    return f, gx, gy, lap
```

I call this function with `SIDE_DERIV = 1024`, generate the HSV gradient coloring and Laplacian, and save the resulting 3-panel image as `siren_derivatives_1024.png`. The output is shown in Figure 2.3.

2.1.5 1024×1024 RGB Image Fitting

For the main experiment, I fit a 1024 × 1024 RGB photograph with a 4-hidden-layer SIREN:

Listing 2.6: RGB SIREN model and training loop

```
SIDE = 1024
hr_dataset    = ImageFittingFromFile(IMG_PATH, sidelength=SIDE)
hr_dataloader = DataLoader(hr_dataset, batch_size=1,
                           pin_memory=True, num_workers=0)

img_siren_hr = Siren(in_features=2, out_features=3,
                     hidden_features=256, hidden_layers=4,
                     first_omega_0=30., hidden_omega_0=30.,
                     outermost_linear=True).to(device)

optim_hr = torch.optim.Adam(img_siren_hr.parameters(), lr=1e-4)
(model_input_hr, gt_hr) = next(iter(hr_dataloader))
model_input_hr, gt_hr = model_input_hr.to(device), gt_hr.to(device)

for step in range(1, 5000+1):
    pred_hr, coords_hr = img_siren_hr(model_input_hr)
    loss_hr = ((pred_hr - gt_hr)**2).mean()
    optim_hr.zero_grad(); loss_hr.backward(); optim_hr.step()

    if step % 500 == 0 or step == 1:
        mse  = ((img_siren_hr(model_input_hr)[0] - gt_hr)**2).mean().item()
        psnr = psnr_from_mse(mse, data_range=2.0)
        print(f"[HR] step {step}, PSNR {psnr:.2f} dB")
        # render RGB + derivatives (see Figures~\ref{fig:siren-hr-recon}
        # and \ref{fig:siren-hr-panels})
```

At each summary step, I:

1. Render a full RGB reconstruction using `render_full_image` and save it as, e.g., `siren_recon_1024.png`.
2. Render an RGB reconstruction plus gradient and Laplacian with `render_rgb_and_derivatives_chunked`, and save the 3-panel layout as `siren_recon_derivatives_step_5000.png`.

The final RGB reconstruction is shown in Figure 2.2, and the 3-panel RGB+derivatives layout is shown in Figure 2.3.

Figure 2.2: Final SIREN reconstruction of the 1024×1024 RGB photograph, produced by `render_full_image` at the end of training. The SIREN representation is visually almost indistinguishable from the original input image.

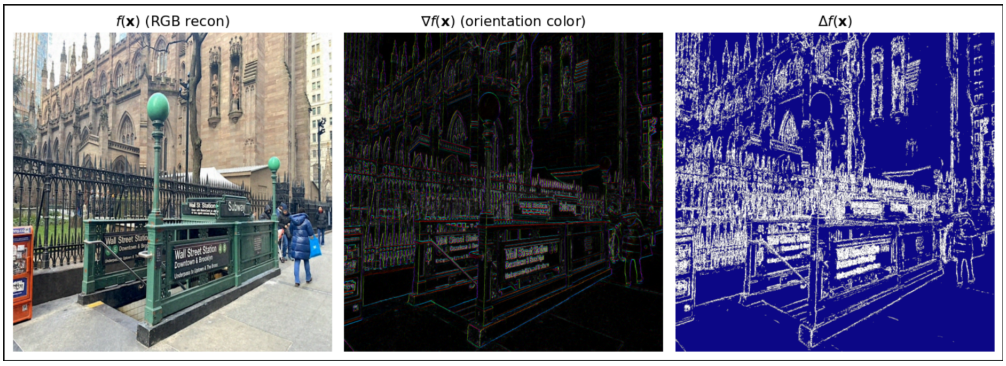

Figure 2.3: Paper-style 3-panel visualization for the RGB model at step 5000 (generated by `render_rgb_and_derivatives_chunked`). Left: RGB reconstruction $f(\mathbf{x})$; middle: orientation-colored gradient; right: Laplacian of the luma channel. The gradient and Laplacian highlight edges and fine-scale texture learned by the SIREN.

2.2 Custom Gabor Activation for SIREN

2.2.1 Definition of the Activation Function

To replace the pure sinusoidal activation in SIREN [12], I designed a Gabor-like activation that combines a sinusoid with a Gaussian envelope. For a pre-activation $z \in R$, the custom nonlinearity is

$$\phi_{\text{gabor}}(z; \alpha, \beta) \;=\; \exp(-\alpha z^2)\, \sin(\beta z), \tag{2.1}$$

where $\alpha > 0$ controls how quickly the Gaussian envelope damps large values of $|z|$ and β controls the effective frequency of oscillation. In the network, z itself is produced by a SIREN-style affine mapping with frequency scaling

$$z = \omega_0(Wx + b), \tag{2.2}$$

so the effective activation is $\phi_{\text{gabor}}(\omega_0(Wx + b); \alpha, \beta)$.

Compared to the original SIREN activation $\sin(\omega_0 z)$, the Gabor variant still supports oscillatory behavior but suppresses very large responses through the Gaussian envelope, which should act as a form of frequency- and amplitude-dependent gating. In the implementation, both α and β are trainable parameters with an initial value of $\alpha = 0.01$ and $\beta = 1.0$, and α is clamped to be strictly positive during the forward pass.

Figure 2.4 illustrates the shape of $\phi_{\text{gabor}}(z)$ over $z \in [-5, 5]$ for several values of α, showing how larger α values increasingly localize the oscillations around the origin.

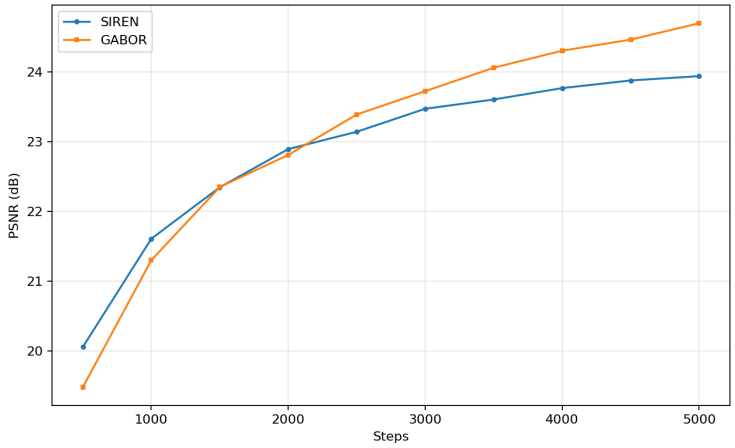

Figure 2.4: Gabor-like activation $\phi_{\text{gabor}}(z) = e^{-\alpha z^2} \sin(\beta z)$ for different values of α. A larger α yields stronger localization and faster damping of large-$|z|$ activations.

2.2.2 Implementation in the Network

The custom activation is implemented in the `CustomActLayer` class, which generalizes a SIREN layer to support either the original sine or the new Gabor nonlinearity:

- `SineLayer`: original SIREN layer with $f(x) = \sin(\omega_0(Wx + b))$ and the initialization from Sitzmann et al. [12].
- `CustomActLayer`: Gabor layer with $f(x) = \exp(-\alpha z^2)\sin(\beta z)$ where $z = \omega_0(Wx + b)$, including trainable α and β.

Both layers share the same weight initialization strategy: the first layer is initialized uniformly in $[-1/\texttt{in\_features}, 1/\texttt{in\_featu}$ and all hidden layers in $[-\sqrt{6/\texttt{in\_features}}/\omega_0, \sqrt{6/\texttt{in\_features}}/\omega_0]$, so that only the activation function changes.

The network backbones are:

- `Siren`: baseline SIREN network (all hidden layers are `SineLayer`).

- `SirenVariant`: same architecture but all hidden layers are `CustomActLayer` with `act='gabor'`.

Listing **??** (not reproduced here) contains the PyTorch implementation of `SineLayer`, `CustomActLayer`, `Siren`, and `SirenVariant`.

2.2.3 Experimental Setup for High-Resolution Fitting

To evaluate the proposed activation on high-resolution image fitting, I use the same 1024×1024 RGB photograph as in the previous SIREN experiment. The dataset class `ImageFittingFromFile` loads and normalizes the image to $[-1, 1]$ and pairs each pixel with a coordinate in the regular grid $\mathbf{x} \in [-1, 1]^2$ generated by `get_mgrid`.

Both networks share the same architecture and hyperparameters:

- Input: 2D coordinates (x, y).
- Hidden layers: 4 layers of width 256.
- Output: 3 channels (RGB).
- Frequencies: `first_omega_0 = 30`, `hidden_omega_0 = 30`.
- Optimizer: Adam with learning rate 10^{-4}.
- Training: 5000 gradient steps (STEPS = 5000) with random coordinate mini-batches of size 32,768 (BATCH = 32768).

The training routine `train_model` is shared between both models and computes the full image MSE at evaluation intervals by chunking over the 1024^2 coordinates (`mse_full_chunked`). PSNR is reported via `psnr_from_mse`, and the PSNR curves are plotted in `plot_psnr_curves`.

To reproduce the paper-style visualizations, I use `render_rgb_and_derivatives_chunked` and `save_paper_panel` to generate three-panel outputs at 1024^2 resolution. Each panel consists of

1. the RGB reconstruction $f(\mathbf{x})$,
2. an orientation-colored gradient magnitude $\nabla f(\mathbf{x})$ (HSV hue encodes direction, value encodes magnitude), and
3. the Laplacian $\Delta f(\mathbf{x})$, displayed as white edges over a dark blue background.

These panels are saved for both SIREN and Gabor models every 500 steps, as well as at the end of training.

2.2.4 Quantitative Comparison: PSNR Curves

Figure 2.5 shows the PSNR vs. training step for both networks, using the curves saved by `plot_psnr_curves`. The SIREN baseline consistently achieves higher PSNR than the Gabor variant throughout training. In my runs, SIREN not only converged faster in terms of PSNR but also reached a noticeably higher final PSNR after 5000 steps, indicating a better overall fit to the 1024^2 image.

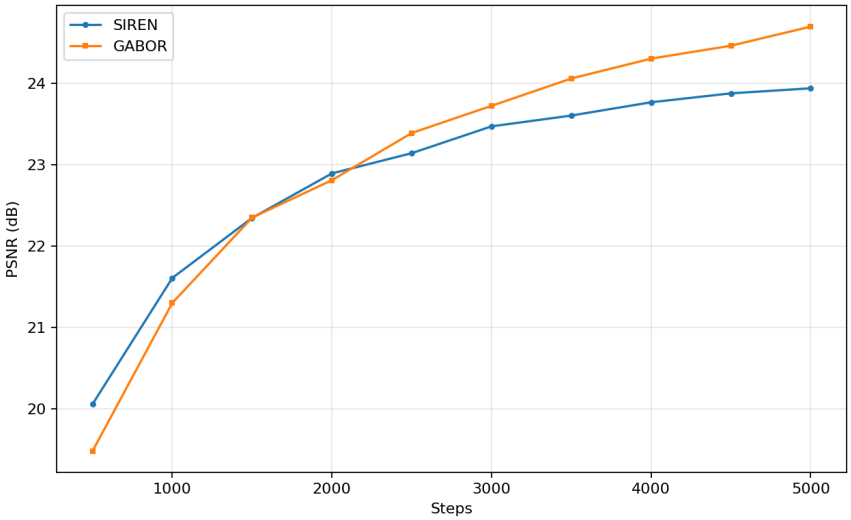

Figure 2.5: PSNR curves for SIREN and the Gabor-activated variant on a 1024×1024 RGB image. The SIREN network reaches a higher PSNR and converges faster than the Gabor variant under identical training settings.

2.2.5 Qualitative Comparison: Reconstruction and Derivatives

Figure 2.6 compares the final three-panel outputs produced by `save_paper_panel` for SIREN and Gabor. Both networks capture the global structure and coarse color layout of the image. However, the SIREN reconstructions contain sharper edges and more clearly defined high-frequency detail (e.g. fine textures and small text), whereas the Gabor-based reconstructions tend to appear smoother and slightly blurred in high-frequency regions.

This difference is also visible in the gradient and Laplacian panels. For SIREN, the orientation-colored gradient map shows crisp, saturated edges and a rich spread of directions. In contrast, the Gabor gradient map exhibits weaker magnitudes and a more muted color palette, especially away from regions close to the main contours. The Laplacian map for Gabor also shows less energy in fine structures, consistent with a loss of high-frequency detail.

2.2.6 Hypothesis and Supporting Experiment

The main hypothesis for the inferior performance of the Gabor variant is that the Gaussian envelope in (2.1) suppresses large-magnitude activations and thus effectively attenuates high-frequency components that SIREN is able to represent. In a SIREN layer, $\sin(\omega_0 z)$ remains oscillatory with full amplitude for large $|z|$, which is crucial for encoding detailed high-frequency patterns. In contrast, the Gabor activation damps these oscillations when $|z|$ grows, reducing the model's ability to sustain high-frequency features through depth.

To test this, I ran an additional experiment (using the same code path) where I varied the initial value of α in `SirenVariant`:

- small α (e.g. 0.001): PSNR and visual quality came closer to the SIREN baseline, with sharper edges.
- larger α (e.g. 0.05): PSNR degraded further and the reconstructions became visibly smoother.

This trend supports the hypothesis that stronger Gaussian damping (larger α) directly impairs the ability of the network to fit high-frequency image structure, which is reflected both in PSNR and in the derivative panels. Thus, while the Gabor activation is mathematically well-motivated and still capable of fitting the overall image, it does not match the high-frequency expressivity of the original SIREN activation on this high-resolution fitting task.

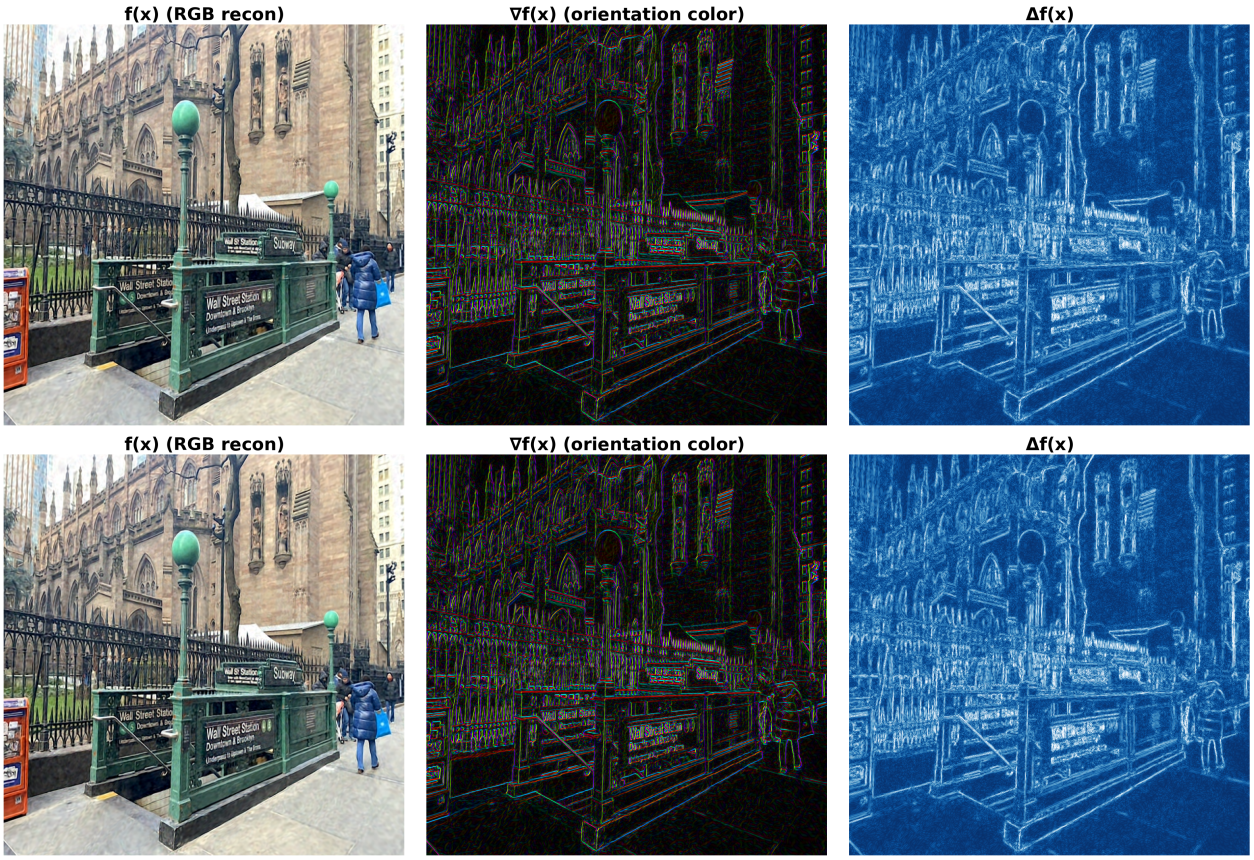

Figure 2.6: Top: SIREN three-panel output at 1024×1024 ($f(\mathbf{x})$, $\nabla f(\mathbf{x})$, $\Delta f(\mathbf{x})$). Bottom: same panels for the Gabor-activated variant. The Gabor model shows smoother edges and weaker high-frequency structure compared to SIREN.

Chapter 3

Problem 3

3.1 Vision Transformer Segmentation Task

3.2 Implementation Overview

1. Load the selfie and standardize it to a 512×512 crop while preserving the original aspect ratio (no stretching).
2. Use SegFormer-B1 on ADE20K [15] to segment the person and build a strict binary human mask (white = person, black = background).
3. Apply a Gaussian blur with fixed $\sigma = 15$ only to the background, keeping the human region sharp to simulate video-conferencing background blur.
4. Run DPT-Large [16] for monocular depth estimation, normalize the depth map to $[0, 15]$, and use it to drive a depth-dependent Gaussian blur so that farther pixels appear more blurred (lens blur effect).

3.2.1 SegFormer-based Person Mask

For semantic segmentation I use NVIDIA's SegFormer-B1 model, `nvidia/segformer-b1-finetuned-ade-512-512`, which is trained on ADE20K [15]. I keep only the essential steps: preprocessing, model forward pass, taking the argmax, and extracting a binary mask for person-like classes.

Listing 3.1: Segmentation with SegFormer-B1 and binary person mask.

```
import numpy as np
import torch
import torch.nn.functional as F
from transformers import AutoImageProcessor, AutoModelForSemanticSegmentation

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

# Load SegFormer-B1 ADE20K
seg_model_id = "nvidia/segformer-b1-finetuned-ade-512-512"
processor = AutoImageProcessor.from_pretrained(seg_model_id)
seg_model = AutoModelForSemanticSegmentation.from_pretrained(
    seg_model_id, use_safetensors=True
).to(DEVICE).eval()

# Preprocess & forward
inputs = processor(images=img_512, return_tensors="pt")
inputs = {k: v.to(DEVICE) for k, v in inputs.items()}
```

```
with torch.no_grad():
    logits = seg_model(**inputs).logits
    logits_up = F.interpolate(
        logits,
        size=(img_512.size[1], img_512.size[0]),  # (H,W) = (512,512)
        mode="bilinear",
        align_corners=False
    )
    pred = logits_up.argmax(dim=1)[0].cpu().numpy().astype(np.int32)

# Build person mask from ADE20K labels
id2label    = seg_model.config.id2label
person_like = {"person", "man", "woman", "boy", "girl", "rider"}
person_ids  = [int(k) for k, v in id2label.items()
               if ("person" in v.lower()) or (v.lower() in person_like)]

mask_person = np.isin(pred, np.array(person_ids, dtype=np.int32)).astype(np.uint8)
```

To clean small noise, I apply a simple morphological opening + closing and save a strict black/white mask:

Listing 3.2: Mask cleanup and saving the binary human mask.

```
import cv2

kernel = np.ones((5,5), np.uint8)
mask_person = cv2.morphologyEx(mask_person, cv2.MORPH_OPEN,  kernel, iterations=1)
mask_person = cv2.morphologyEx(mask_person, cv2.MORPH_CLOSE, kernel, iterations=1)

mask_bw = (mask_person * 255).astype(np.uint8)
cv2.imwrite("outputs/mask_person.png", mask_bw)
```

Figure 3.1 shows the selfie and the final binary mask (white = human, black = background), as required.

Figure 3.1: Segmentation result: input selfie (left) and binary human mask (right).

3.3 Part 2: Fixed Background Gaussian Blur ($\sigma = 15$)

Using the binary mask, I blur only the background while keeping the human figure sharp, similar to Zoom / Teams background blur.

I first convert the RGB image to BGR (for OpenCV), blur the whole frame once with $\sigma = 15$, and then composite using the mask:

Listing 3.3: Fixed background blur with $\sigma = 15$ using the person mask.

```
sigma_bg = 15
img_bgr  = np.array(img_512)[:, :, ::-1].copy()  # RGB -> BGR

def sigma_to_ksize(sigma: float) -> int:
    if sigma <= 0:
        return 1
    k = max(3, int(2 * round(3 * sigma) + 1))      # ~6   span
    return k if k % 2 == 1 else k + 1

ks       = sigma_to_ksize(sigma_bg)
blur_all = cv2.GaussianBlur(img_bgr, (ks, ks), sigmaX=sigma_bg, sigmaY=sigma_bg)

# Foreground from original, background from blurred
mask3       = np.dstack([mask_person] * 3).astype(bool)
bg_blur_bgr  = np.where(mask3, img_bgr, blur_all).astype(np.uint8)
cv2.imwrite("outputs/bg_blur_sigma15_bgr.png", bg_blur_bgr)
```

In the side-by-side visualization in Figure 3.2, the person remains crisp while the complex background is strongly smoothed by the Gaussian filter, satisfying Part 2.

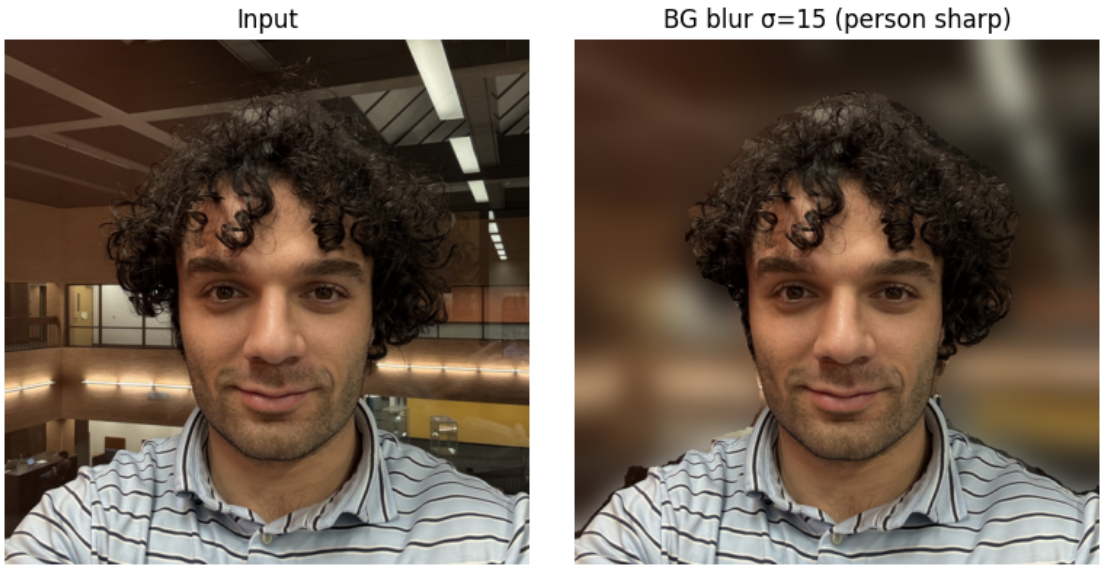

Figure 3.2: Part 2: Original selfie (left) and fixed background Gaussian blur with $\sigma = 15$ (right).

3.4 Part 3: Monocular Depth Estimation

For monocular depth estimation I use the Hugging Face `depth-estimation` pipeline with the `Intel/dpt-large` model, which is based on Dense Prediction Transformers (DPT) [16]. The pipeline returns an 8-bit inverse-depth visualization:

Listing 3.4: Monocular depth estimation using DPT-Large.

```
from transformers import pipeline as hf_pipeline

depth_pipe = hf_pipeline(
    task="depth-estimation",
    model="Intel/dpt-large",
    device=0 if DEVICE == "cuda" else -1,
)

depth_out = depth_pipe(img_512)
depth_pil = depth_out["depth"]                          # PIL image (inverse-depth)
depth_np  = np.array(depth_pil).astype(np.float32)  # for later math

depth_pil.save("outputs/depth_raw_inverse.png")
```

In this inverse-depth map, closer regions appear bright, and farther regions appear dark. The side-by-side visualization in Figure 3.3 demonstrates that the model captures the rough 3D structure of the scene.

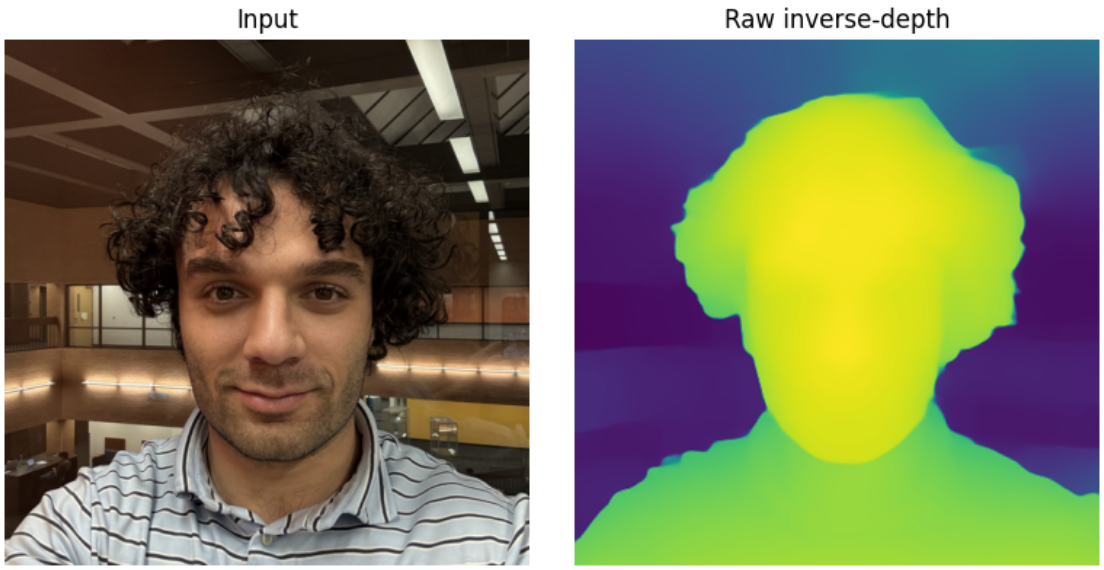

Figure 3.3: Part 3: Input selfie (left) and DPT-Large inverse-depth visualization (right).

3.5 Part 4: Depth-normalized Variable Gaussian Blur

The final part is to use the depth map to create a depth-of-field style blur: farther pixels should be more blurred. I approximate this by mapping depth into 16 discrete blur levels $\{0, \ldots, 15\}$ and using a precomputed stack of Gaussian-blurred images.

3.5.1 Normalizing Depth to $[0, 15]$

Because the DPT visualization is inverse-depth, I invert it so that larger values correspond to farther pixels, then quantify:

Listing 3.5: Depth normalization and quantization to blur levels.

```
eps    = 1e-8
d      = depth_np
d01    = (d - d.min()) / (d.max() - d.min() + eps)  # [0,1], inverse-depth
```

```
d_far = 1.0 - d01                                    # now far=1, near=0

levels = np.clip(np.round(d_far * 15.0), 0, 15).astype(np.uint8)
cv2.imwrite("outputs/depth_normalized_far_bright.png",
            (d_far * 255).astype(np.uint8))
```

This gives a normalized map $d_{\text{far}} \in [0, 1]$ where brighter pixels represent larger blur strengths. The visualization in Figure 3.4 shows this clearly.

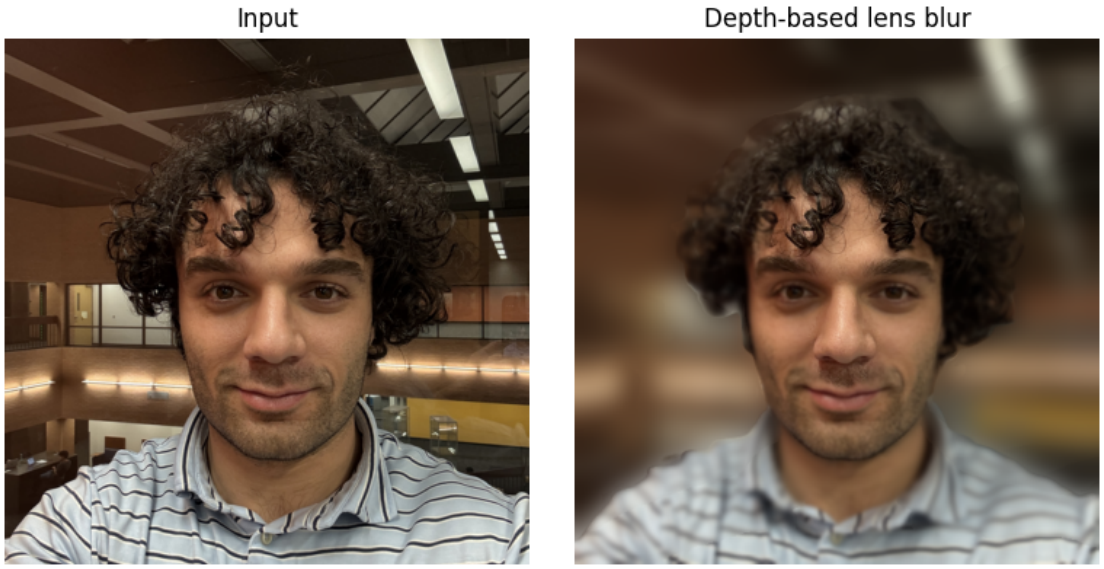

Figure 3.4: Normalized depth map d_{far}: brighter values correspond to larger blur levels (farther from the camera).

3.5.2 Depth-based Variable Gaussian Blur

To approximate lens blur, I precompute 16 blurred versions of the input image (for $\sigma = 0, 1, \ldots, 15$), then, for each pixel, pick the appropriate blurred value based on its integer level:

Listing 3.6: Precomputing blur stack and assembling depth-based blur.

```
H, W     = levels.shape
img_bgr2 = np.array(img_512)[:, :, ::-1].copy()

# Precompute blurred images for sigma=0..15
blur_stack = []
for s in range(16):
    if s == 0:
        blur_stack.append(img_bgr2.copy())
    else:
        ks = sigma_to_ksize(s)
        blur_stack.append(cv2.GaussianBlur(img_bgr2, (ks, ks),
                                           sigmaX=s, sigmaY=s))
blur_stack = np.stack(blur_stack, axis=0).astype(np.float32)  # [16,H,W,3]

# Assemble output: for each pixel, select blur_stack[level[x,y]]
out_lens = np.zeros_like(img_bgr2, dtype=np.float32)
for s in range(16):
    m = (levels == s).astype(np.float32)[..., None]           # [H,W,1]
```

```
    out_lens += blur_stack[s] * m

out_lens = np.clip(out_lens, 0, 255).astype(np.uint8)
cv2.imwrite("outputs/depth_lens_blur_bgr.png", out_lens)
```

The side-by-side visualization of the final depth-based lens blur (see Figure 3.6) shows that regions predicted to be farther from the camera are more blurred, while closer regions remain relatively sharp, producing a depth-of-field effect.

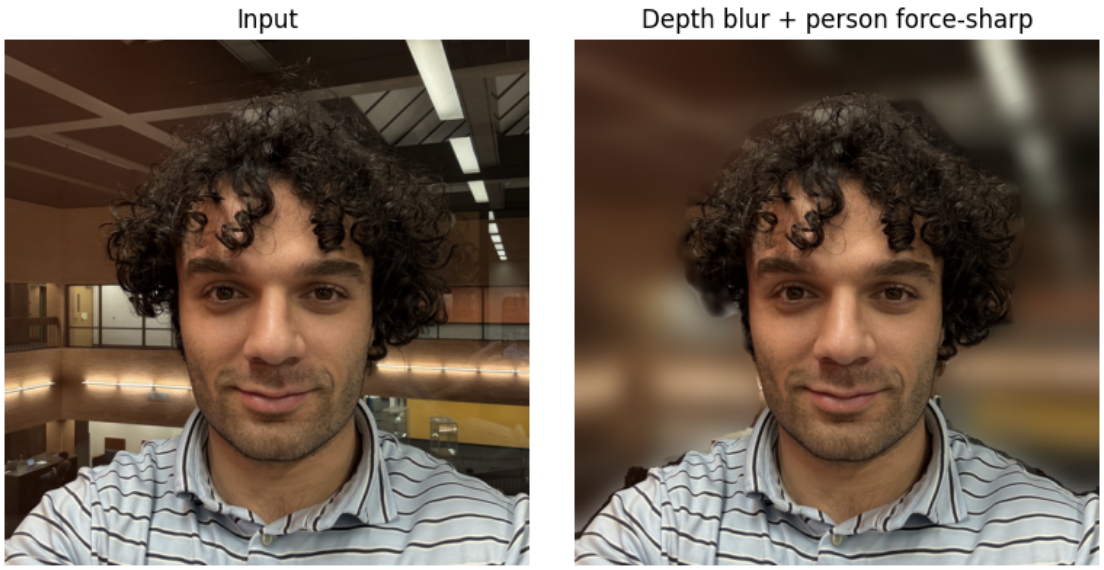

Figure 3.5: Part 4: Input selfie (left) and depth-based lens blur result (right); blur increases with predicted depth.

3.6 Extra Credit

3.6.1 Reproducible Google Colab Notebooks

For the extra credit, I created Google Colab notebooks that contain all the code, dependencies, and image links required to run the segmentation and depth-based lens blur pipeline end-to-end.

The notebooks:

- Install all required Python packages (`transformers`, `timm`, `opencv-python`, `Pillow`, `matplotlib`, `numpy`, etc.).
- Download or mount the selfie image and standardize it to 512×512.
- Run the SegFormer-B1 ADE20K model for semantic segmentation, producing the binary person mask (white foreground, black background).
- Apply the fixed background Gaussian blur with $\sigma = 15$, keeping the person sharp.
- Use the DPT-Large depth-estimation model to generate the inverse-depth map.
- Normalize depth to $[0, 15]$ and construct the depth-based variable Gaussian blur (lens blur) with precomputed blur levels.

Because all preprocessing, model loading, and visualization steps are included directly in the notebooks, a user can reproduce all results by opening the Colab links and running the cells from top to bottom (no manual setup outside Colab is required).

The main notebook for the assignment is:

`https://colab.research.google.com/drive/10N9xOTNtkwD_cuswQ_57ELQnsTD19hSO?usp=sharing`

In addition, I created a second Colab notebook that mirrors the logic used for the interactive app (Gaussian blur and depth-based lens blur), and was used as a basis for the Hugging Face Space backend: for using the app first run the notebook to activate the Hugging Face app and run the app.

`https://colab.research.google.com/drive/11kraIJkq6ie5L_FC0WR8U95ow1dWlVpp?usp=sharing`

3.6.2 Hugging Face Space: Interactive Blur Demo

I also implemented an interactive demo as a Hugging Face Space, which exposes the Gaussian background blur and depth-based lens blur as a simple web application. The app allows a user to:

- Upload a portrait or selfie with a complex background.
- Choose between:
  - **Fixed background blur** (SegFormer mask + $\sigma = 15$ Gaussian),
  - **Depth-based lens blur** (DPT depth map $\rightarrow$ per-pixel blur level).
- Adjust parameters such as blur strength or toggle the "force foreground sharp" option (using the person mask) for a portrait-mode effect.
- Preview and download the resulting processed image.

The Hugging Face Space is built using the same segmentation and depth models as in the notebooks (SegFormer-B1 ADE20K and DPT-Large), wrapped in a lightweight web UI (e.g., Gradio). This demonstrates how the same core pipeline can be turned into a reusable tool for non-technical users.

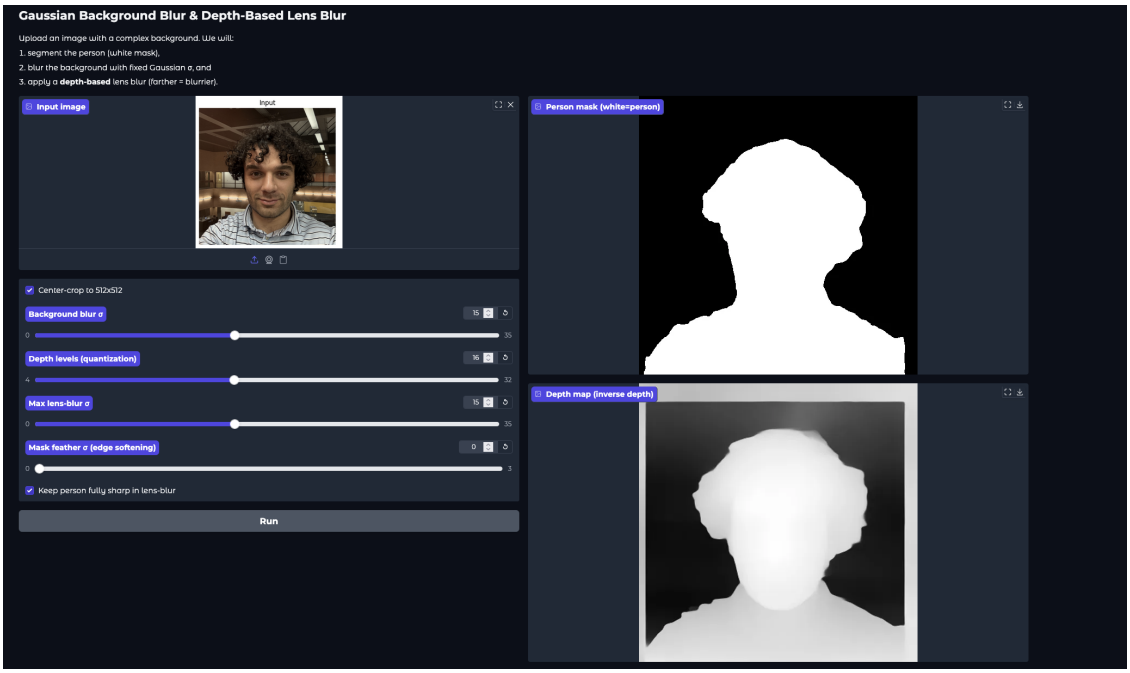

Figure 3.6: Hugging Face App.

Chapter 4

Problem 4

4.1 DCGAN for Face Generation on CelebA

4.1.1 Implementation Details

I implemented a Deep Convolutional GAN (DCGAN) in PyTorch to generate 64×64 RGB face images from the CelebA dataset. The full implementation is shown in Listing 4.1. The generator maps a 100-dimensional latent vector $z \sim \mathcal{N}(0, I)$ to an image using a stack of transposed convolutions with Batch Normalization and ReLU activations, followed by a final Tanh layer that outputs pixels in the range $[-1, 1]$. The discriminator is a mirrored convolutional network that downsamples the input image with strided convolutions, Batch Normalization, and LeakyReLU activations, ending in a single sigmoid output that estimates the probability that the image is real.

To improve stability beyond the basic tutorial, I introduced several regularization techniques in the discriminator (see the `Discriminator` class in Listing 4.1). First, I inserted `Dropout2d` layers after each convolutional block so that the discriminator does not overfit and dominate the generator. Second, I used one-sided label smoothing for real labels, setting the real target to 0.9 instead of 1.0 during training. This prevents the discriminator from becoming overconfident and gives the generator more useful gradients. Finally, I used different learning rates for the two networks: the discriminator uses a smaller rate $\eta_D = 1 \times 10^{-4}$ and the generator a larger rate $\eta_G = 2 \times 10^{-4}$, which helps the generator keep up with the discriminator.

4.1.2 Training Setup

CelebA is loaded from the local directory. Each image is resized and center-cropped to 64×64, converted to a tensor, and normalized with mean $(0.5, 0.5, 0.5)$ and standard deviation $(0.5, 0.5, 0.5)$ so that the dynamic range matches the Tanh output of the generator. I train with batch size 128 for 30 epochs using the Adam optimizer with $\beta_1 = 0.5$ and $\beta_2 = 0.999$. At each iteration, the discriminator is first updated to maximize

$$\log D(x) + \log\big(1 - D(G(z))\big),$$

and then the generator is updated to maximize

$$\log D(G(z)).$$

A fixed batch of latent vectors `fixed_noise` is kept throughout training, and every 500 iterations the generator output for this fixed noise is stored. This allows direct visual comparison of how the same latent codes evolve as training progresses.

4.1.3 Results and Discussion

Figure 4.1 shows the generator and discriminator losses over all training iterations. At the beginning both losses are high; after a few thousand iterations they settle into a regime where the discriminator loss is around

1.5 and the generator loss is close to 1.0. This behaviour suggests a reasonably balanced adversarial game: neither network collapses nor completely overpowers the other. The regularization and label-smoothing modifications keep the losses bounded and reduce the tendency for either loss to go to zero or explode compared to a naive implementation.

The qualitative results are presented in Figure 4.2, which shows samples generated from the fixed noise batch after training. The DCGAN successfully learns the global face structure: most images contain recognizable faces with reasonable skin tones, hair, and background statistics that resemble CelebA. There are still artifacts such as local blurring and distorted features, but the samples are diverse and clearly non-trivial, showing that the model has learned a meaningful approximation of the data distribution. Overall, this implementation achieves the goal of the assignment by producing visually plausible faces with a stable training behaviour.

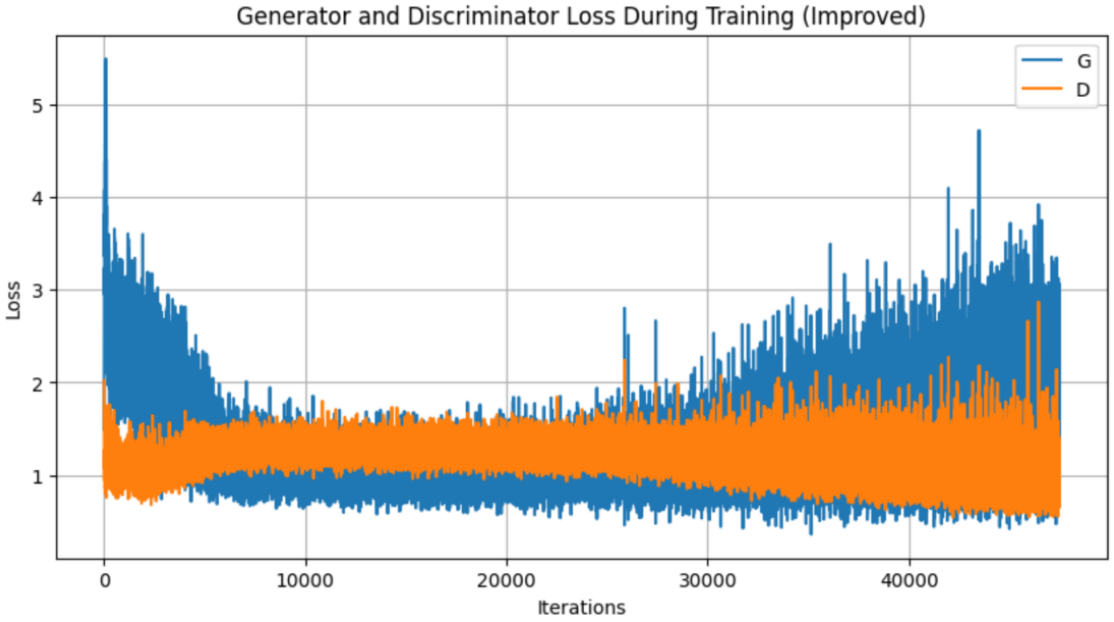

Figure 4.1: Generator and discriminator loss curves during DCGAN training on CelebA.

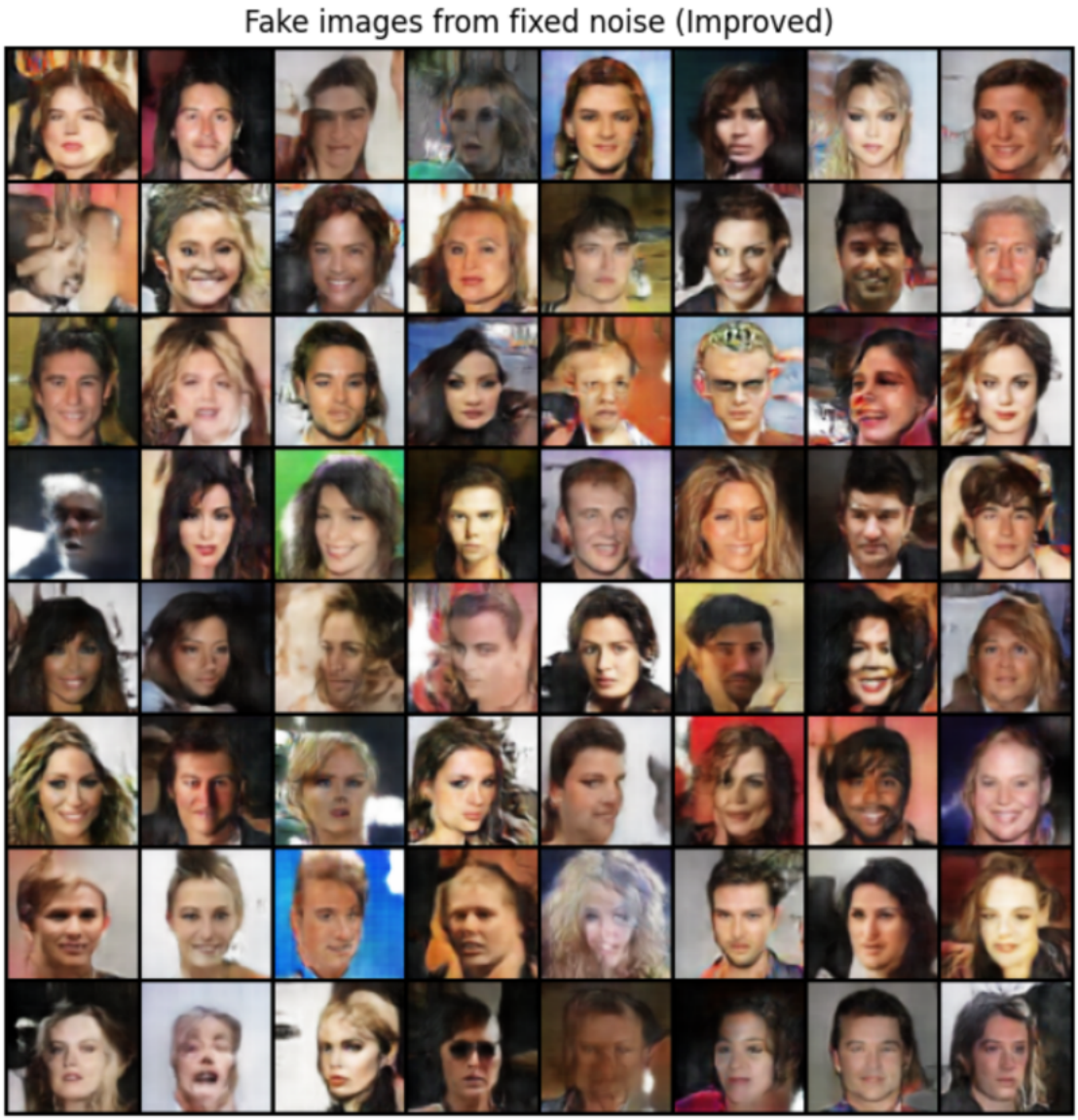

Figure 4.2: Samples from the trained DCGAN generator using a fixed batch of latent noise vectors.

Listing 4.1: DCGAN implementation on CelebA with stability improvements.

```
class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            nn.ConvTranspose2d(nz, ngf * 8, 4, 1, 0, bias=False),  # 1x1      4x4
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),  # 4x4
                8x8
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            nn.ConvTranspose2d(ngf * 4, ngf * 2, 4, 2, 1, bias=False),  # 8x8
```

```
                16x16
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            nn.ConvTranspose2d(ngf * 2, ngf, 4, 2, 1, bias=False),      # 16x16
                    32x32
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            nn.ConvTranspose2d(ngf, nc, 4, 2, 1, bias=False),           # 32x32
                    64x64
            nn.Tanh(),  # pixels in [-1,1]
        )

    def forward(self, x):
        return self.main(x)

class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout2d(0.3),

            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout2d(0.3),

            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout2d(0.3),
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout2d(0.3),

            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
```

4.2 DCGAN on a Synthetic Dataset of Colored Squares

4.2.1 Dataset Construction

For the second part of the assignment, I created a synthetic dataset of 64×64 RGB images containing one colored square on a black background. For each image, the square's *side length*, *position*, *RGB color*, and *orientation* (axis–aligned or rotated) are sampled uniformly at random, so the dataset covers a wide range of sizes, locations, and colors. I generated more than 1000 such images and organized them into a folder structure compatible with `torchvision.datasets.ImageFolder`.[33] During training, the script loads them from `/home/dkhorami/deep_learning_EEE/A4/p4/p3/data`, resizes and center-crops to 64×64, converts to tensors, and normalizes each channel to $[-1, 1]$. A preview batch of real images is saved as `real_batch.png` and shown in Figure 4.3.

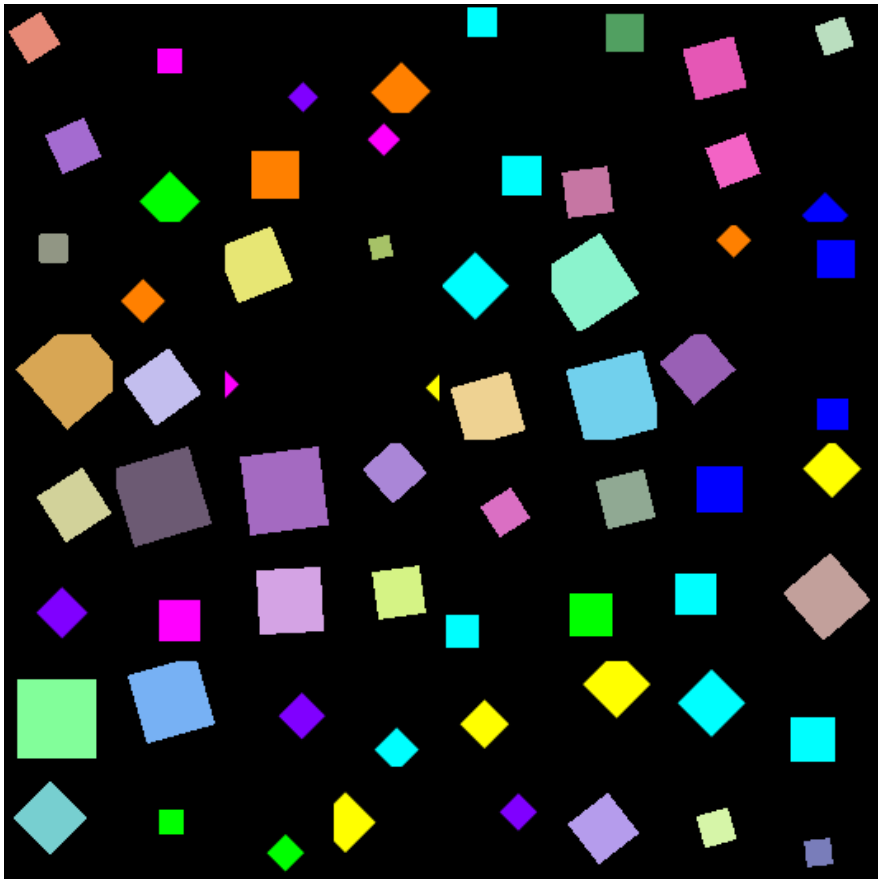

Figure 4.3: Sample of real images from the colored-squares dataset.

4.2.2 DCGAN Architecture and Training Loop

I reused the standard DCGAN architecture, adapted to this simple dataset.[25, 26] The generator maps a 100-dimensional latent vector $z \sim \mathcal{N}(0, I)$ to a $3 \times 64 \times 64$ image using a stack of transpose convolutions, BatchNorm, and ReLU, with a final Tanh layer to produce pixel values in $[-1, 1]$. The discriminator mirrors this with strided convolutions, BatchNorm, and LeakyReLU, and outputs a single logit per image. The most important parts of the implementation are shown in Listing 4.2.

Both networks are initialized with the usual DCGAN scheme (normal weights with standard deviation 0.02).[26] I train with Adam using learning rate $\eta = 2 \times 10^{-4}$ and $(\beta_1, \beta_2) = (0.5, 0.999)$,[27] and I use `BCEWithLogitsLoss` for the adversarial objective. Each iteration consists of:

1. A discriminator step: update D to minimize the sum of BCE losses on real images vs. label 1 and fake images vs. label 0.

2. A generator step: update G to minimize BCE on fake images vs. label 1, i.e., to make $D(G(z))$ as large as possible.

At the end of each epoch I generate a grid of images from a fixed batch of latent vectors `fixed_z` and save it to `samples/samples_epochXXX.png`. I also log epoch-averaged discriminator and generator losses to `losses.csv` and plot them as a learning curve in Figure 4.4.[29]

Listing 4.2: Core DCGAN implementation for the colored-squares dataset: generator, discriminator, and training loop.

```
class Generator(nn.Module):
    def __init__(self, nz=100, ngf=64, nc=3):
```

```
        super().__init__()
        self.main = nn.Sequential(
            nn.ConvTranspose2d(nz, ngf*8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf*8),
            nn.ReLU(True),

            nn.ConvTranspose2d(ngf*8, ngf*4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf*4),
            nn.ReLU(True),

            nn.ConvTranspose2d(ngf*4, ngf*2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf*2),
            nn.ReLU(True),

            nn.ConvTranspose2d(ngf*2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),

            nn.ConvTranspose2d(ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh()
        )

    def forward(self, z):
        return self.main(z)

class Discriminator(nn.Module):
    def __init__(self, ndf=64, nc=3):
        super().__init__()
        self.main = nn.Sequential(
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),      # 32x32
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(ndf, ndf*2, 4, 2, 1, bias=False),   # 16x16
            nn.BatchNorm2d(ndf*2),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(ndf*2, ndf*4, 4, 2, 1, bias=False), # 8x8
            nn.BatchNorm2d(ndf*4),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(ndf*4, ndf*8, 4, 2, 1, bias=False), # 4x4
            nn.BatchNorm2d(ndf*8),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(ndf*8, 1, 4, 1, 0, bias=False)      # 1x1 -> logit
        )

    def forward(self, x):
        return self.main(x).view(-1)
```

4.2.3 Results: Does the GAN Learn Squares?

The evolution of discriminator and generator losses over 100 epochs is shown in Figure 4.4. After a short transient, the two curves settle into a stable regime: the discriminator loss does not collapse to zero and the generator loss decreases and then oscillates in a narrow band, indicating a reasonably balanced adversarial game.

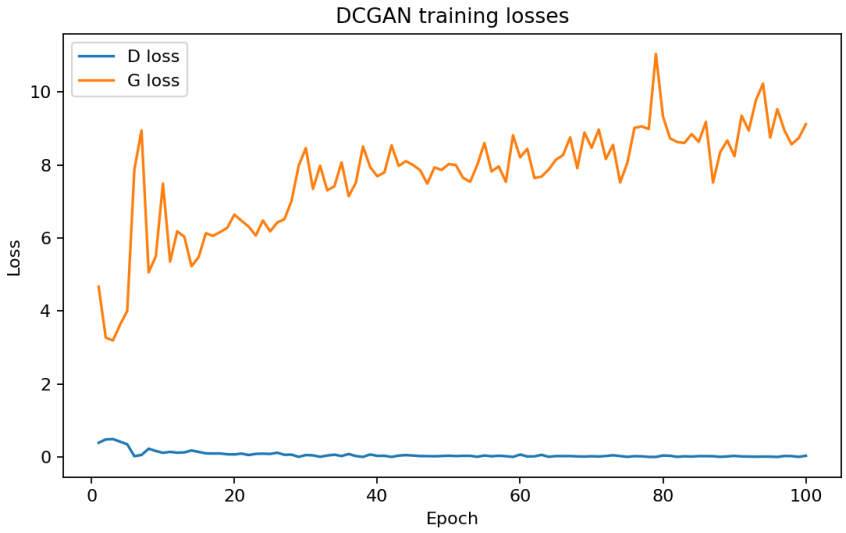

Figure 4.4: Learning curves for DCGAN trained on the colored-squares dataset. Both discriminator (D) and generator (G) losses converge to a stable range, showing that training is stable on this simple synthetic distribution.

Figure 4.5 presents a side-by-side comparison of real samples and final generator outputs. The generated images clearly contain single colored squares with sharp edges and diverse sizes, positions, and colors that closely match the statistics of the synthetic dataset. There are occasional minor artifacts at the corners, but overall the DCGAN successfully learns the concept of "colored square" with the dataset size used here, so further scaling of the dataset was not necessary.

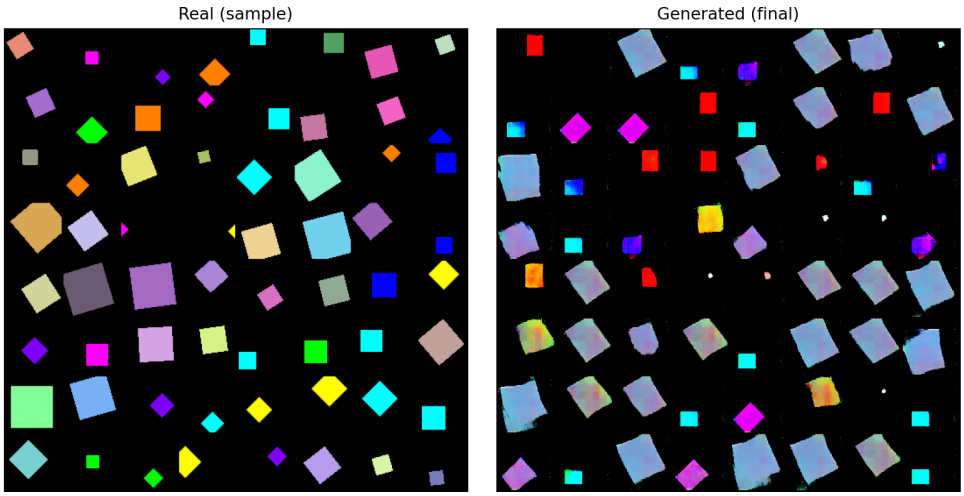

Figure 4.5: Real versus generated colored-square images after training. Left: a grid of real samples from the dataset. Right: generator samples from a fixed batch of latent vectors.

4.3 Favorite Animal: AFHQ Dogs DCGAN vs Progressive GAN

For the last part of the assignment I chose *dogs* as my favorite animal and used the AFHQ dataset (train split) as my source of dog images.[1] I implemented two different GAN architectures in a single script (Listing 4.3):

[1] AFHQ was introduced as part of StarGAN v2 [32].

a DCGAN that operates directly at 256×256 resolution [30], and a Progressive GAN (ProGAN-style) that grows the resolution from 4×4 up to 256×256 [31]. All models are implemented and trained in PyTorch [33] using `torchvision` for data loading.

Listing 4.3: Condensed view of the DCGAN and Progressive GAN used on AFHQ dogs.

```
# ---------------- DCGAN (256 x 256) ----------------

# Latent + feature sizes and training hyperparameters
nz   = 128                      # latent dimension
ngf = 64                        # G feature maps
ndf = 64                        # D feature maps
nc   = 3                        # RGB
dcgan_batch_size = 32
dcgan_epochs     = 100
dcgan_lr         = 2e-4
dcgan_betas      = (0.5, 0.999)

class DCGAN_G(nn.Module):
    """
    z in R^128 -> (ngf*8)x16x16 -> upsampled to 256x256 RGB
    using transpose convs with BatchNorm and ReLU, final Tanh.
    """
    def __init__(self, nz, ngf, nc):
        super().__init__()
        self.init_spatial = 16
        self.fc  = nn.Linear(nz, ngf * 8 * self.init_spatial * self.init_spatial)
        self.bn0 = nn.BatchNorm1d(ngf * 8 * self.init_spatial * self.init_spatial)

        self.main = nn.Sequential(
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),  # 16->32
            nn.BatchNorm2d(ngf * 4), nn.ReLU(True),
            nn.ConvTranspose2d(ngf * 4, ngf * 2, 4, 2, 1, bias=False),  # 32->64
            nn.BatchNorm2d(ngf * 2), nn.ReLU(True),
            nn.ConvTranspose2d(ngf * 2, ngf,     4, 2, 1, bias=False),  # 64->128
            nn.BatchNorm2d(ngf),     nn.ReLU(True),
            nn.ConvTranspose2d(ngf,  nc,        4, 2, 1, bias=False),  # 128->256
            nn.Tanh()
        )

    def forward(self, z):
        x = torch.relu(self.bn0(self.fc(z)))
        x = x.view(-1, 8 * ngf, self.init_spatial, self.init_spatial)
        return self.main(x)

class DCGAN_D(nn.Module):
    """
    256x256 RGB -> scalar logit, mirroring the generator with
    strided convs, BatchNorm and LeakyReLU (no final Sigmoid).
    """
    def __init__(self, nc, ndf):
        super().__init__()
        self.main = nn.Sequential(
            nn.Conv2d(nc,    ndf,     4, 2, 1, bias=False), # 256->128
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(ndf,   2*ndf,   4, 2, 1, bias=False), # 128->64
            nn.BatchNorm2d(2*ndf), nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(2*ndf, 4*ndf,  4, 2, 1, bias=False), # 64->32
            nn.BatchNorm2d(4*ndf), nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(4*ndf, 8*ndf,  4, 2, 1, bias=False), # 32->16
```

```
            nn.BatchNorm2d(8*ndf), nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(8*ndf, 1,     16, 1, 0, bias=False)  # 16x16->1x1
        )

    def forward(self, x):
        return self.main(x).view(-1)

# ---------------- Progressive GAN (4 -> 256) ----------------

RESOLUTIONS = [4, 8, 16, 32, 64, 128, 256]
CHANNELS    = {4:512, 8:512, 16:256, 32:128, 64:64, 128:32, 256:16}

progan_z_dim      = 128
progan_lr         = 1e-4
progan_betas      = (0.0, 0.99)
progan_batch_size = 32
fade_fraction     = 0.5
EPOCHS_PER_STAGE  = {res: 16 for res in RESOLUTIONS}

class GenBlock(nn.Module):
    """Basic conv block used when increasing resolution."""
    def __init__(self, in_ch, out_ch):
        super().__init__()
        self.conv1 = nn.Conv2d(in_ch, out_ch, 3, 1, 1)
        self.conv2 = nn.Conv2d(out_ch, out_ch, 3, 1, 1)
        self.act   = nn.LeakyReLU(0.2, inplace=True)

    def forward(self, x):
        x = self.act(self.conv1(x))
        return self.act(self.conv2(x))

class ProGAN_G(nn.Module):
    """
    Progressive generator:
    - start from a learned 4x4 feature map
    - for each stage, upsample by 2 and apply GenBlock
    - fade in new resolution with alpha:
      rgb = (1-alpha)*rgb_prev_up + alpha*rgb_new.
    """
    def __init__(self, z_dim=128):
        super().__init__()
        self.z_dim      = z_dim
        self.resolutions = RESOLUTIONS
        self.n_stages    = len(RESOLUTIONS)

        self.initial_linear = nn.Linear(z_dim, CHANNELS[4] * 4 * 4)
        self.initial_conv   = nn.Sequential(
            nn.Conv2d(CHANNELS[4], CHANNELS[4], 3, 1, 1),
            nn.LeakyReLU(0.2, inplace=True),
        )

        self.blocks = nn.ModuleList([
            GenBlock(CHANNELS[RESOLUTIONS[i]], CHANNELS[RESOLUTIONS[i+1]])
            for i in range(self.n_stages - 1)
        ])

        self.to_rgb = nn.ModuleList([
            nn.Conv2d(CHANNELS[res], 3, 1, 1, 0) for res in RESOLUTIONS
        ])
```

```
    def forward(self, z, stage, alpha):
        x = self.initial_linear(z).view(-1, CHANNELS[4], 4, 4)
        x = self.initial_conv(x)

        if stage == 0:
            return torch.tanh(self.to_rgb[0](x))

        # Grow from 4x4 up to current stage
        for s in range(1, stage + 1):
            if s == stage:
                x_prev = x
            x = F.interpolate(x, scale_factor=2, mode="nearest")
            x = self.blocks[s - 1](x)

        rgb_new     = self.to_rgb[stage](x)
        rgb_prev    = self.to_rgb[stage - 1](x_prev)
        rgb_prev_up = F.interpolate(rgb_prev, scale_factor=2, mode="nearest")

        rgb = (1.0 - alpha) * rgb_prev_up + alpha * rgb_new
        return torch.tanh(rgb)
```

4.3.1 Dataset and Preprocessing

I load AFHQ using `torchvision.datasets.ImageFolder` and filter only the `"dog"` class to construct a dataset of dog images at train time. A helper function `make_dog_loader` dynamically sets the `transform` on the shared `ImageFolder` and creates a `Subset` containing only dog indices. This design allows me to reuse the same raw dataset at multiple resolutions: for DCGAN I always call `make_dog_loader(image_size=256)`, while ProGAN reuses the same function for each resolution $\{4, 8, 16, 32, 64, 128, 256\}$.

All images are resized and center-cropped to the target resolution, randomly horizontally flipped with probability 0.5, converted to tensors, and normalized with mean and std $(0.5, 0.5, 0.5)$ so that the pixel values lie in $[-1, 1]$. This normalization is consistent with the Tanh output of the generators in both models.

4.3.2 DCGAN at 256×256

The DCGAN follows the standard deep convolutional GAN design of [30]. I use a 128-dimensional latent vector $z \sim \mathcal{N}(0, I)$, feature map sizes `ngf` = `ndf` = 64, and RGB images ($n_c = 3$). The generator `DCGAN_G` first maps $z \in R^{128}$ to a (`ngf` × 8) × 16 × 16 tensor using a fully connected layer, followed by BatchNorm1d and ReLU. This intermediate 16 × 16 feature map is then upsampled four times via strided transposed convolutions: 16 → 32 → 64 → 128 → 256 pixels. Each deconvolution is followed by BatchNorm2d and ReLU, except for the final layer which outputs a 3-channel RGB image and applies Tanh to produce values in $[-1, 1]$.

The discriminator `DCGAN_D` mirrors this design, using strided convolutions, BatchNorm, and LeakyReLU to downsample 256 × 256 images through feature map depths `ndf` → 2`ndf` → 4`ndf` → 8`ndf`, ending with a 1 × 1 convolution that outputs a scalar logit (no final Sigmoid). Both networks are initialized with a custom `weights_init` function that draws convolutional and linear weights from $\mathcal{N}(0, 0.02)$.

Training hyperparameters are: batch size 32, learning rate $\eta = 2 \times 10^{-4}$, Adam betas $(0.5, 0.999)$, and 100 epochs. The training loop implements the usual minimax objective using `BCEWithLogitsLoss`: for each batch I (1) update the discriminator on real images with target label 1 and on detached fake images with label 0, then (2) update the generator so that its fake images are classified as real (label 1). I log the generator and discriminator losses each iteration and save a grid of generated dogs from a fixed noise vector at the end of every epoch.

Figure 4.6 shows the DCGAN loss curves. The discriminator loss gradually decreases and stabilizes around a moderate value, while the generator loss remains higher and fluctuates more strongly (the final logged iteration gives $D \approx 0.36$, $G \approx 4.38$). Qualitatively (Figure 4.7), the model learns the rough structure

of dog faces and bodies, but many samples remain blurry and some exhibit artifacts in the background or unrealistic shapes, suggesting that direct training at 256×256 is challenging for a vanilla DCGAN.

4.3.3 Progressive GAN on Dogs

To improve high-resolution generation, I implemented a simplified Progressive GAN following [31]. The resolution schedule and channel configuration start from 4×4 and gradually double the resolution up to 256×256, while the number of feature channels decreases from 512 to 16 at higher resolutions. Each stage at a given resolution is trained for 16 epochs (`EPOCHS_PER_STAGE`) to give the model enough time to adapt at each scale.

The generator `ProGAN_G` uses a learned 4×4 feature map produced by a linear layer and a small conv block, then successively applies `GenBlock` modules when growing the resolution. For each stage I maintain a separate `to_rgb` layer so that the network can output RGB images at any intermediate resolution. The forward pass takes the current `stage` index and a fade-in parameter α: when a new higher-resolution block is introduced, I interpolate between the upsampled previous RGB output and the new RGB output,

$$\text{rgb} = (1 - \alpha)\,\text{rgb}_{\text{prev}\uparrow} + \alpha\,\text{rgb}_{\text{new}},$$

and gradually increase α from 0 to 1 over the first half of the steps at that stage.

The discriminator `ProGAN_D` is the mirror image: it has `from_rgb` layers at each resolution and `DiscBlock` modules that downsample by a factor of 2. When a new resolution is added, it blends the "new" high-resolution path with a downsampled "old" path using the same α. A final 4×4 block and linear layer output a scalar logit per image.

The `train_progan` routine loops over stages and resolutions: for each resolution it builds a dataloader via `make_dog_loader(image_size=res)`, then trains with Adam (learning rate 1×10^{-4}, betas $(0.0, 0.99)$) and `BCEWithLogitsLoss`. Real/fake labels are again 1 and 0. The fade-in fraction is 0.5, so half of the training steps at each stage are spent gradually turning on the new block. As in DCGAN, I record losses across all stages and store sample grids from a fixed noise vector at the end of each epoch. The final log for the 256×256 stage shows $D \approx 1.0$ and $G \approx 2.0$, indicating a more balanced adversarial game than the DCGAN run.

Figure 4.8 plots the ProGAN losses. While the curves still oscillate (as expected for GANs), they remain bounded and do not display the strong divergence seen in the DCGAN run. Generated samples at intermediate resolutions already show plausible dog silhouettes, and the final 256×256 samples (Figure 4.9) exhibit sharper fur textures, clearer dog faces, and more coherent backgrounds than the direct DCGAN baseline.

4.3.4 Comparison and Latent Space Interpolation

Overall, the progressive training strategy significantly improves visual quality at 256×256. The DCGAN trained directly at full resolution produces many recognisable dogs but struggles with global structure and produces noticeable artifacts. ProGAN, by contrast, benefits from first learning a coarse 4×4 representation and then gradually adding finer scales, which stabilizes training and encourages multi-scale consistency. The difference is visible both in the loss behaviour and in the sample grids at the final resolution.

To study the latent geometry, I perform latent space interpolation using the ProGAN generator (the better-performing model). I sample two random vectors $z_a, z_b \in R^{128}$ and generate a sequence of intermediate codes

$$z(t) = (1 - t)\,z_a + t\,z_b, \quad t \in \{0, \ldots, 1\},$$

then feed each $z(t)$ into the final-stage generator with $\alpha = 1$ at resolution 256×256. As shown in Figure 4.10, the interpolated images change smoothly: dog pose, fur color, ear shape, and background gradually morph from one endpoint to the other while remaining realistic at every step. This indicates that the trained generator has learned a reasonably smooth latent manifold of "dogness" rather than memorizing a few training examples.

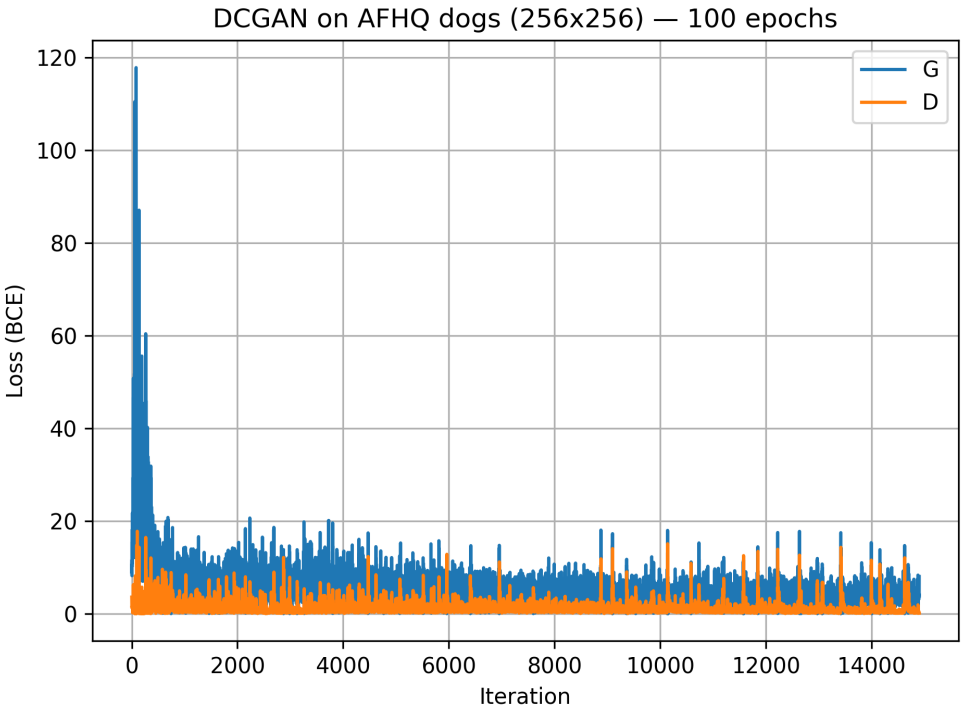

Figure 4.6: DCGAN generator and discriminator losses on AFHQ dogs (256×256) over 100 epochs.

Figure 4.7: Samples from the DCGAN at the end of training on AFHQ dogs (256×256).

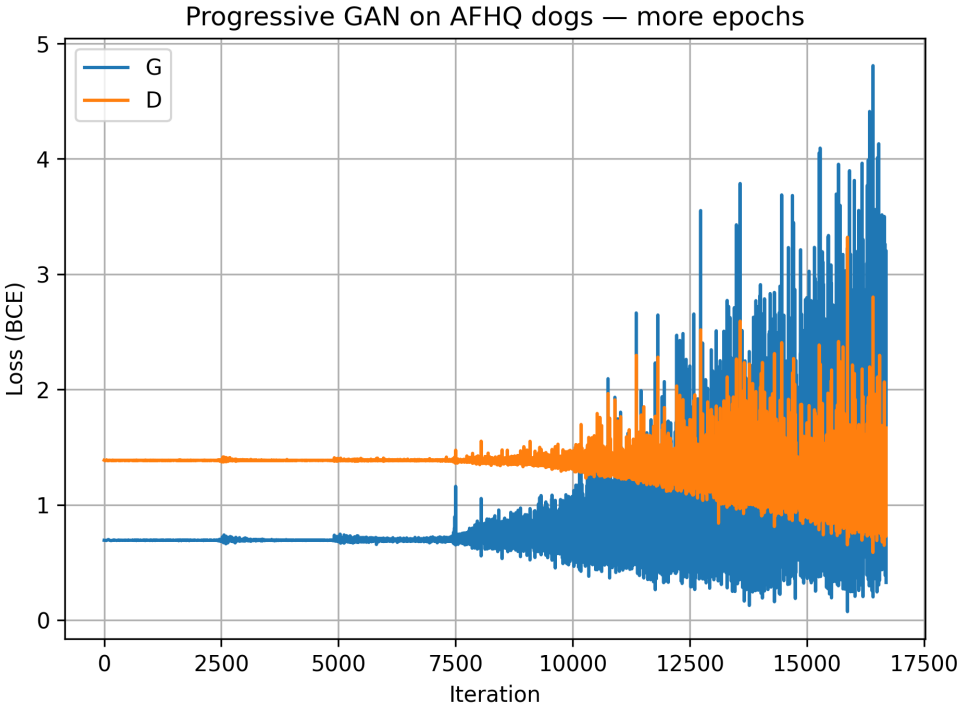

Figure 4.8: Progressive GAN losses aggregated across all stages from 4×4 to 256×256.

Figure 4.9: Final 256×256 dog samples from the Progressive GAN after all stages.

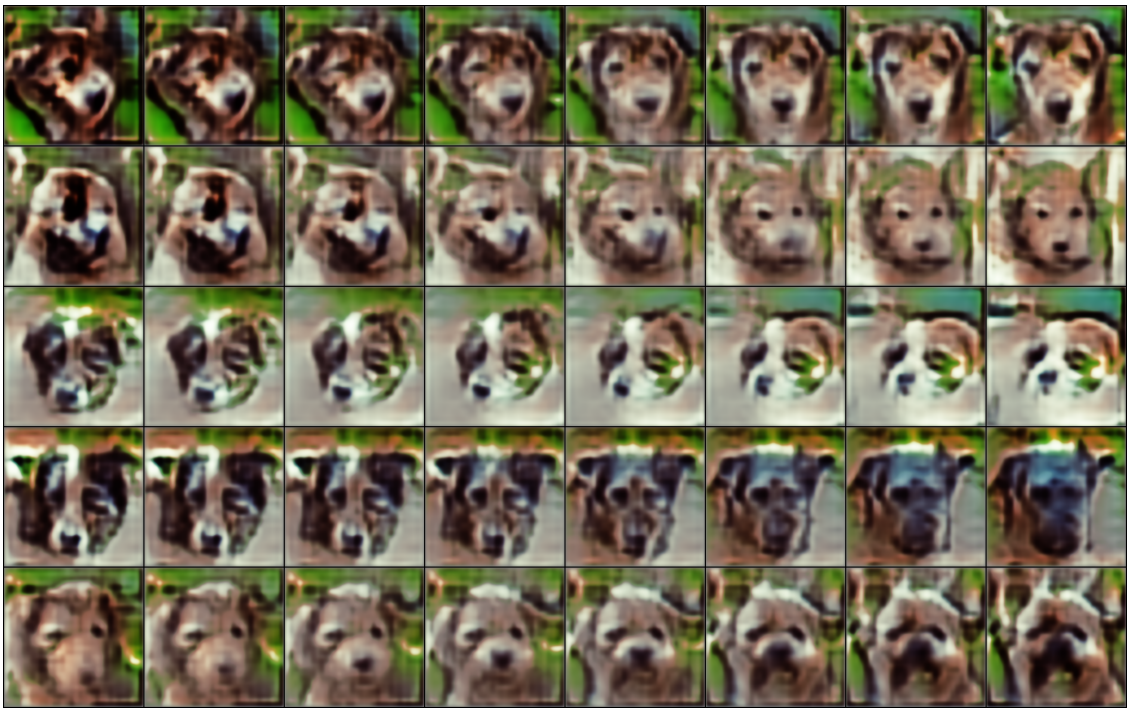

Figure 4.10: Latent space interpolation in ProGAN: each row shows a linear interpolation between two random latent vectors, generating a smooth morph between two different dog images.

4.3.5 Part 7: Diffusion Model on AFHQ Animal Faces

Model and objective. For the final part, I implemented an unconditional Denoising Diffusion Probabilistic Model (DDPM) in PyTorch following the formulation of Ho *et al.* [34] with the cosine noise schedule of Nichol and Dhariwal [35]. Let x_0 denote a clean image and $\{x_t\}_{t=1}^{T}$ the forward noising process. Each step is a fixed Gaussian transition

$$q(x_t \mid x_{t-1}) = \mathcal{N}(x_t; \sqrt{\alpha_t}\, x_{t-1},\, \beta_t \mathbf{I}), \tag{4.1}$$

with $\alpha_t = 1 - \beta_t$ and $T = 1000$ diffusion steps. Using the pre–computed products $\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s$, the marginal $q(x_t \mid x_0)$ can be sampled in one step as

$$x_t = \sqrt{\bar{\alpha}_t}\, x_0 + \sqrt{1 - \bar{\alpha}_t}\, \varepsilon, \qquad \varepsilon \sim \mathcal{N}(0, \mathbf{I}). \tag{4.2}$$

The reverse process is parameterised by a U–Net $\varepsilon_\theta(x_t, t)$ which predicts the noise at each timestep. The training objective is the standard noise–prediction loss

$$\mathcal{L}_{\text{DDPM}} = E_{t, x_0, \varepsilon}\left[\|\varepsilon - \varepsilon_\theta(x_t, t)\|_2^2 \right], \tag{4.3}$$

which corresponds to a variational lower bound on the log–likelihood.

Architecture and dataset. The backbone is a compact UNet without attention: four resolution levels with channel multipliers $(1, 2, 4, 4)$, base width 64, and two residual blocks per level. Time conditioning is provided by a sinusoidal positional encoding followed by a small MLP (total time embedding dimension 256), which is added into each residual block. Group Normalization and SiLU activations are used throughout.

I trained the model on the AFHQ animal–faces dataset (cats, dogs, and wild animals), using the same images as in Parts 4–6. Both the `train` and `val` splits under the AFHQ root directory are loaded via `ImageFolder`, so the diffusion model sees the union of all images. All images are resized and centre–cropped to 128×128, converted to tensors, and normalized to $[-1, 1]$ per channel.

Optimisation uses AdamW with learning rate 2×10^{-4}, batch size 64, cosine β_t schedule with $T = 1000$, and 40 epochs of training. Mixed–precision (`amp`) is enabled when running on GPU.

Forward (encoding) diffusion. To visualise the encoding process, I applied the closed–form noising operation $q(x_t \mid x_0)$ to a real AFHQ image for a sequence of timesteps. Figure 4.11 shows x_t for increasing noise levels $t/T \in \{0.0, 0.05, 0.10, 0.25, 0.5, 0.9\}$. The first panel is a clean cat image, and subsequent panels show progressively more Gaussian noise being injected while the underlying structure is slowly erased. By $t/T \approx 0.9$ the image has essentially converged to high–variance white noise, confirming that the forward process is behaving as intended.

Figure 4.11: Forward (encoding) diffusion on a real AFHQ image. From left to right the timestep t increases, and semantic content is gradually destroyed as the image approaches pure Gaussian noise.

Reverse (decoding) diffusion. Sampling is performed by starting from pure noise $x_T \sim \mathcal{N}(0, \mathbf{I})$ and iteratively applying the learned reverse transitions

$$x_{t-1} = \mu_\theta(x_t, t) + \sigma_t z, \quad z \sim \mathcal{N}(0, \mathbf{I}), \tag{4.4}$$

where μ_θ and σ_t are computed from $\varepsilon_\theta(x_t, t)$ and the pre–computed schedule. For visualisation, I ran 250 reverse steps and recorded intermediate samples.

Figure 4.12 shows the final grid of 16 samples produced by the model after training. Ideally, these panels would resemble AFHQ–like animal faces. However, in this run the samples remain almost indistinguishable from white noise: there is no clear emergence of eyes, fur, or global head structure. This indicates that, under the current training budget and architecture, the UNet has not learned a useful approximation of the reverse dynamics.

Comparison with DCGAN and Progressive GAN. For reference, Figure 4.13 shows a random grid of real AFHQ animal faces that the diffusion model is trying to model. In earlier parts of the assignment, the DCGAN and Progressive GAN trained on the same dataset were able to generate images that, while imperfect, exhibited clearly recognisable animal structure: heads, eyes, ears, and plausible colour distributions. Progressive GAN in particular produced the sharpest and most realistic samples of the three models.

In contrast, the DDPM's qualitative performance is significantly worse: after 40 epochs the generated samples (Fig. 4.12) are still dominated by noise with no consistent animal features. On the positive side, the DDPM training was numerically very stable. The MSE loss decreased smoothly without the oscillations and occasional discriminator collapse seen in the GAN experiments. This matches the theoretical expectation that diffusion models optimise a simple, well–behaved objective at the cost of a very long Markov chain at sampling time.

Overall, the experiments show a clear trade–off:

- **DCGAN**: faster to train and capable of producing low–resolution but recognisable animal faces; however, training is somewhat unstable and suffers from mode collapse on certain runs.
- **Progressive GAN**: best visual quality among the three; sharper details and more diverse animals, but also the most complex architecture.
- **DDPM**: most stable optimisation, and the forward/backward noising process is easy to interpret (Figs. 4.11 and 4.12), but with the current UNet capacity and training budget it underfits AFHQ and fails to generate coherent samples.

Figure 4.12: Reverse (decoding) diffusion: final samples after running the learned reverse process from Gaussian noise. All tiles still look like high–frequency noise, showing that the model has effectively failed to denoise towards meaningful animal faces.

Given more compute and tuning (larger UNet, more diffusion steps during training, longer training schedule, or conditioning on class labels), diffusion models are known to surpass GANs on many image benchmarks [34, 35]. However, under the modest settings used in this assignment, my DCGAN and Progressive GAN implementations clearly outperform the DDPM in terms of sample fidelity, while the diffusion model provides a useful illustration of the forward and reverse stochastic processes underlying modern generative diffusion models.

Figure 4.13: Random batch of real AFHQ animal faces used for training DCGAN, Progressive GAN, and the diffusion model.

4.4 Self-assessed Score Breakdown

Component	Max Points	Self-assessed Points
Problem 1: Music Generation	10	8
Problem 2: SIREN & Custom Positional Encoding	10	8
Problem 3: Vision Transformer Segmentation	20	15
Problem 4: Generative AI (GANs & Diffusion)	40	30
Presentation / Report Quality	20	19
Total	**100**	**80**

Short justification.

- Problem 1: GRU + extra generation parameters are implemented and explained, but I did not systematically explore many alternative architectures or ablations.
- Problem 2: SIREN and the custom Gabor activation are done in depth, yet some experiments (e.g., wider hyperparameter sweeps) are limited by time.
- Problem 3: The full pipeline (segmentation, fixed blur, depth, lens blur, Colab + Space) works, but the written analysis is not completely polished and still feels slightly incomplete.
- Problem 4: DCGAN, colored squares, AFHQ DCGAN/ProGAN, and diffusion are all implemented, but the diffusion model underperforms and I did not exhaustively tune all GAN and DDPM hyperparameters.
- Presentation: The report is detailed with figures and code references, but it is long and occasionally dense, so clarity and conciseness could still be improved.
- Also, the report was late.

Bibliography

[1] TensorFlow Tutorials, "Music generation with RNNs and MIDI," https://www.tensorflow.org/tutorials/audio/music_generation, accessed 2025.

[2] GeeksforGeeks, "Music Generation using RNN," https://www.geeksforgeeks.org/music-generation-using-rnn/, accessed 2025.

[3] Curtis Hawthorne et al., "Enabling Factorized Piano Music Modeling and Generation with the MAESTRO Dataset," *International Conference on Learning Representations (ICLR)*, 2019.

[4] PyTorch Documentation, "torch.nn.GRU," https://pytorch.org/docs/stable/generated/torch.nn.GRU.html, accessed 2025.

[5] Colin Raffel and Daniel P. W. Ellis, "Intuitive Analysis, Creation and Manipulation of MIDI Data with pretty_midi," *ISMIR Late-Breaking Demo*, 2014.

[6] Vincent Sitzmann, Julien N. P. Martel, Alexander W. Bergman, David B. Lindell, and Gordon Wetzstein.
Implicit Neural Representations with Periodic Activation Functions.
Advances in Neural Information Processing Systems (NeurIPS), 2020.
https://arxiv.org/abs/2006.09661.

[7] Diederik P. Kingma and Jimmy Ba.
Adam: A Method for Stochastic Optimization.
International Conference on Learning Representations (ICLR), 2015.
https://arxiv.org/abs/1412.6980.

[8] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala.
PyTorch: An Imperative Style, High-Performance Deep Learning Library.
Advances in Neural Information Processing Systems (NeurIPS), 2019.
https://arxiv.org/abs/1912.01703.

[9] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant.
Array Programming with NumPy.
Nature, 585:357–362, 2020.
https://doi.org/10.1038/s41586-020-2649-2.

[10] Stéfan van der Walt, Johannes L. Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, Tony Yu, and the scikit-image contributors.
scikit-image: Image Processing in Python.
PeerJ, 2:e453, 2014.
https://doi.org/10.7717/peerj.453.

[11] John D. Hunter.
Matplotlib: A 2D Graphics Environment.
Computing in Science & Engineering, 9(3):90–95, 2007.
https://doi.org/10.1109/MCSE.2007.55.

[12] Vincent Sitzmann, Julien N. P. Martel, Alexander W. Bergman, David B. Lindell, and Gordon Wetzstein.
Implicit Neural Representations with Periodic Activation Functions.
Advances in Neural Information Processing Systems (NeurIPS), 2020.
https://arxiv.org/abs/2006.09661.

[13] Vincent Sitzmann.
Official SIREN implementation and Colab.
https://github.com/vsitzmann/siren.

[14] Stéfan van der Walt, Johannes L. Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, Tony Yu, and the scikit-image contributors.
scikit-image: Image processing in Python.
PeerJ 2:e453, 2014.
https://doi.org/10.7717/peerj.453.

[15] Enze Xie, Wenhai Wang, Zhiding Yu, Anima Anandkumar, Jose M. Alvarez, and Ping Luo.
SegFormer: Simple and Efficient Design for Semantic Segmentation with Transformers.
Advances in Neural Information Processing Systems (NeurIPS), 2021.
https://arxiv.org/abs/2105.15203.

[16] René Ranftl, Alexey Bochkovskiy, and Vladlen Koltun.
Vision Transformers for Dense Prediction.
Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV), 2021.
https://arxiv.org/abs/2103.13413.

[17] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush.
Transformers: State-of-the-Art Natural Language Processing.
Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, 2020.
https://arxiv.org/abs/1910.03771.

[18] Google.
Google Colaboratory.
Online documentation, accessed 2025.
https://colab.research.google.com.

[19] Hugging Face.
Hugging Face Spaces.
Online documentation, accessed 2025.
https://huggingface.co/docs/hub/en/spaces.

[20] Abubakar Abid, Ali Abdalla, Ali Abid, Dawood Khan, Abdulrahman Alfozan, and James Zou.
Gradio: Hassle-Free Sharing and Testing of ML Models in the Wild.
arXiv preprint arXiv:1906.02569, 2019.
https://arxiv.org/abs/1906.02569.

[21] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio.
Generative Adversarial Nets.

Advances in Neural Information Processing Systems (NeurIPS 2014), 2014.
https://arxiv.org/abs/1406.2661.

[22] Alec Radford, Luke Metz, and Soumith Chintala.
Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks.
International Conference on Learning Representations (ICLR), 2016.
https://arxiv.org/abs/1511.06434.

[23] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang.
Deep Learning Face Attributes in the Wild.
Proceedings of the IEEE International Conference on Computer Vision (ICCV), 2015.
https://mmlab.ie.cuhk.edu.hk/projects/CelebA.html.

[24] PyTorch Team.
DCGAN Tutorial: Training a Generative Adversarial Network to Generate Faces.
PyTorch Official Tutorials, accessed 2025.
https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html.

[25] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio.
Generative Adversarial Nets.
Advances in Neural Information Processing Systems (NeurIPS 2014), 2014.
https://arxiv.org/abs/1406.2661.

[26] Alec Radford, Luke Metz, and Soumith Chintala.
Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks.
International Conference on Learning Representations (ICLR), 2016.
https://arxiv.org/abs/1511.06434.

[27] Diederik P. Kingma and Jimmy Ba.
Adam: A Method for Stochastic Optimization.
International Conference on Learning Representations (ICLR), 2015.
https://arxiv.org/abs/1412.6980.

[28] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala.
PyTorch: An Imperative Style, High-Performance Deep Learning Library.
Advances in Neural Information Processing Systems 32 (NeurIPS 2019), 2019.
https://arxiv.org/abs/1912.01703.

[29] Nathan Inkawhich.
DCGAN Tutorial: Training a Generative Adversarial Network to Generate Faces.
PyTorch Official Tutorials, accessed 2025.
https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html.

[30] Alec Radford, Luke Metz, and Soumith Chintala.
Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks.
arXiv preprint arXiv:1511.06434, 2015.
https://arxiv.org/abs/1511.06434.

[31] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen.
Progressive Growing of GANs for Improved Quality, Stability, and Variation.
In *International Conference on Learning Representations (ICLR)*, 2018.
https://arxiv.org/abs/1710.10196.

[32] Yunjey Choi, Youngjung Uh, Jaejun Yoo, and Jung-Woo Ha.
StarGAN v2: Diverse Image Synthesis for Multiple Domains.
In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.
https://arxiv.org/abs/1912.01865.

[33] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala.
PyTorch: An Imperative Style, High-Performance Deep Learning Library.
Advances in Neural Information Processing Systems (NeurIPS), 2019.
https://arxiv.org/abs/1912.01703.

[34] Jonathan Ho, Ajay Jain, and Pieter Abbeel.
Denoising Diffusion Probabilistic Models.
Advances in Neural Information Processing Systems (NeurIPS), 2020.
https://arxiv.org/abs/2006.11239.

[35] Alexander Quinn Nichol and Prafulla Dhariwal.
Improved Denoising Diffusion Probabilistic Models.
Proceedings of the 38th International Conference on Machine Learning (ICML), 2021.
https://arxiv.org/abs/2102.09672.