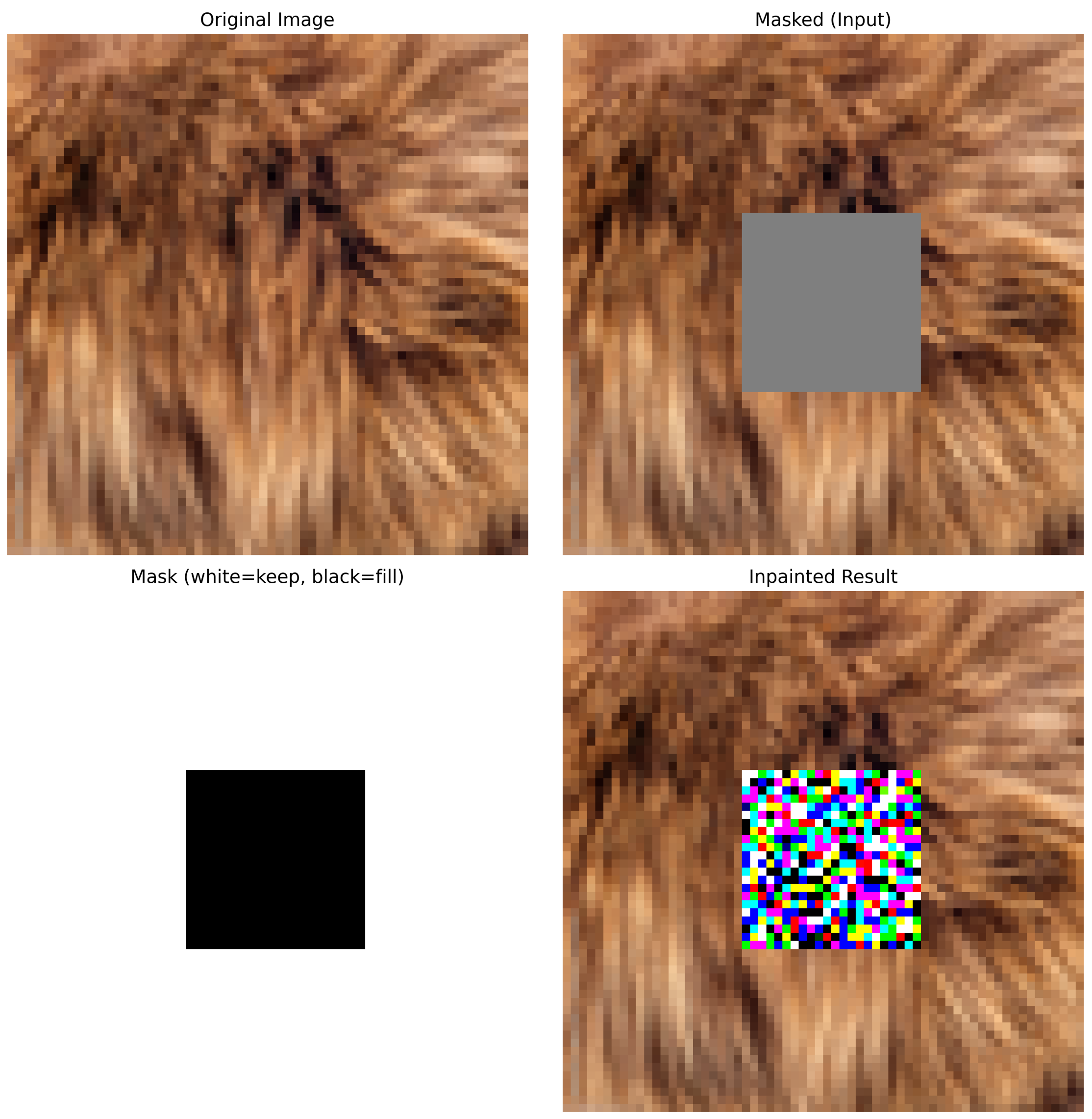
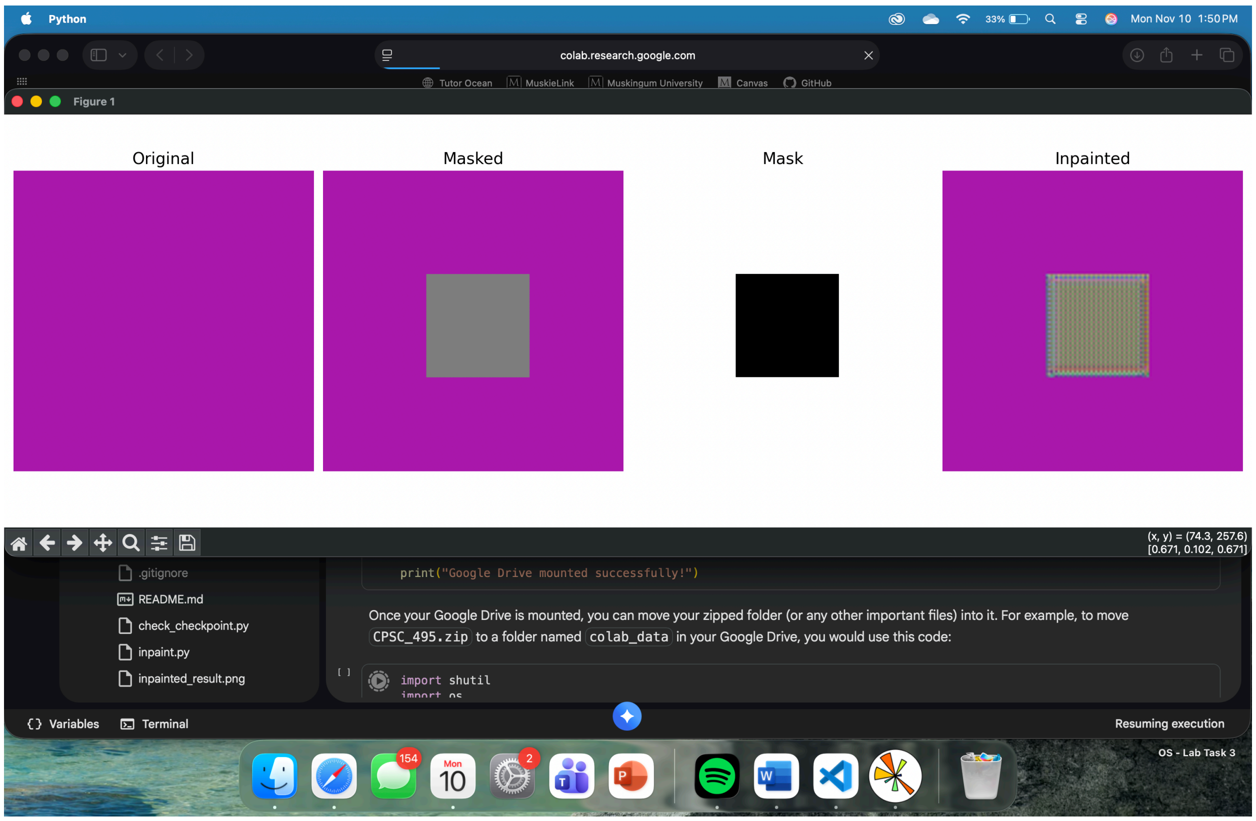
For my senior project I wanted to work with image diffusion and gradient descent because I have always been interested in how modern AI systems actually create or restore images. I thought this would be a good chance to learn something close to what real world generative models do, but without needing massive GPU clusters. At first I planned something more ambitious. I tried to build a gradient descent based face swap system that blended one person's face into another. It sounded cool, but it completely fell apart once I actually tried training it. Diffusion models start from random noise and try to recover an image step by step. That works fine for patterns and textures, but it does not preserve detailed facial identity unless you have a ton of data and compute. My model did not. So the first attempt failed in a pretty dramatic way.



One example that pushed me to shift directions was when I tried generating an animal image. The fur turned into a blurry mess. It was obvious that the model did not have enough information or training time to understand what real fur looks like. It made me realize that the project needed tighter control. I needed the dataset to be something small, clean, and repeatable. Trying to do facial features with limited hardware was not realistic, and it was stressing the model in ways that were not helpful for learning.

During the process I decided to switch to pattern recognition and inpainting. This is a task that diffusion models can actually handle at small scale. Patterns are simple in structure, and their rules are easy for the model to learn. Instead of dealing with thousands of color combinations or complicated shapes, I created a dataset from scratch. Every image was made of only red and blue pixels. The patterns included checkerboards, stripes, grids, dots, zigzags, waves, concentric shapes, and a few random geometric designs. By limiting the problem to just two colors and clear pixel structures, the model had a much easier time learning the relationships inside each pattern.



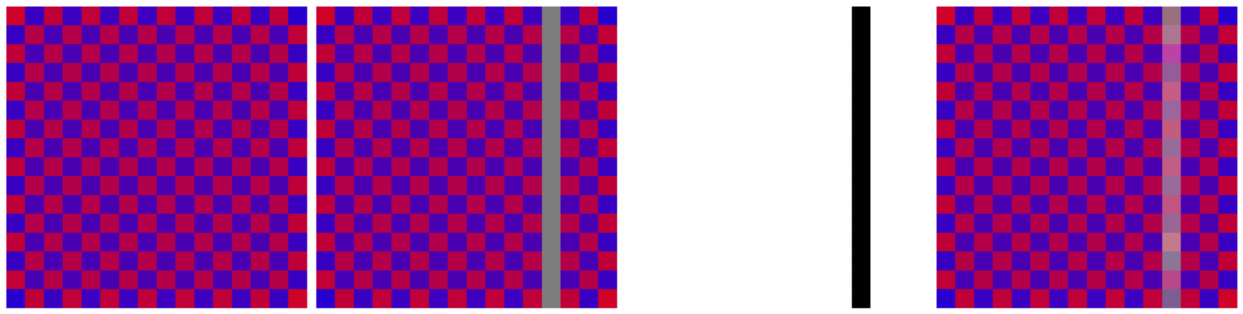
```
print("Google Drive mounted successfully!")
```

Once your Google Drive is mounted, you can move your zipped folder (or any other important files) into it. For example, to move `CPSC_495.zip` to a folder named `colab_data` in your Google Drive, you would use this code:

```
import shutil
import os
```

Another improvement was changing the size. Originally I tried large images, which slowed down everything. Instead, I shrunk all pattern images to 32 by 32 pixels. This is tiny by modern standards, but perfect for the kind of diffusion training I wanted to do. Smaller images mean faster training, more variety in the dataset, and easier visualization of whether the model is learning or not. With this setup, the model could train thousands of samples on my hardware without blowing up the GPU memory.

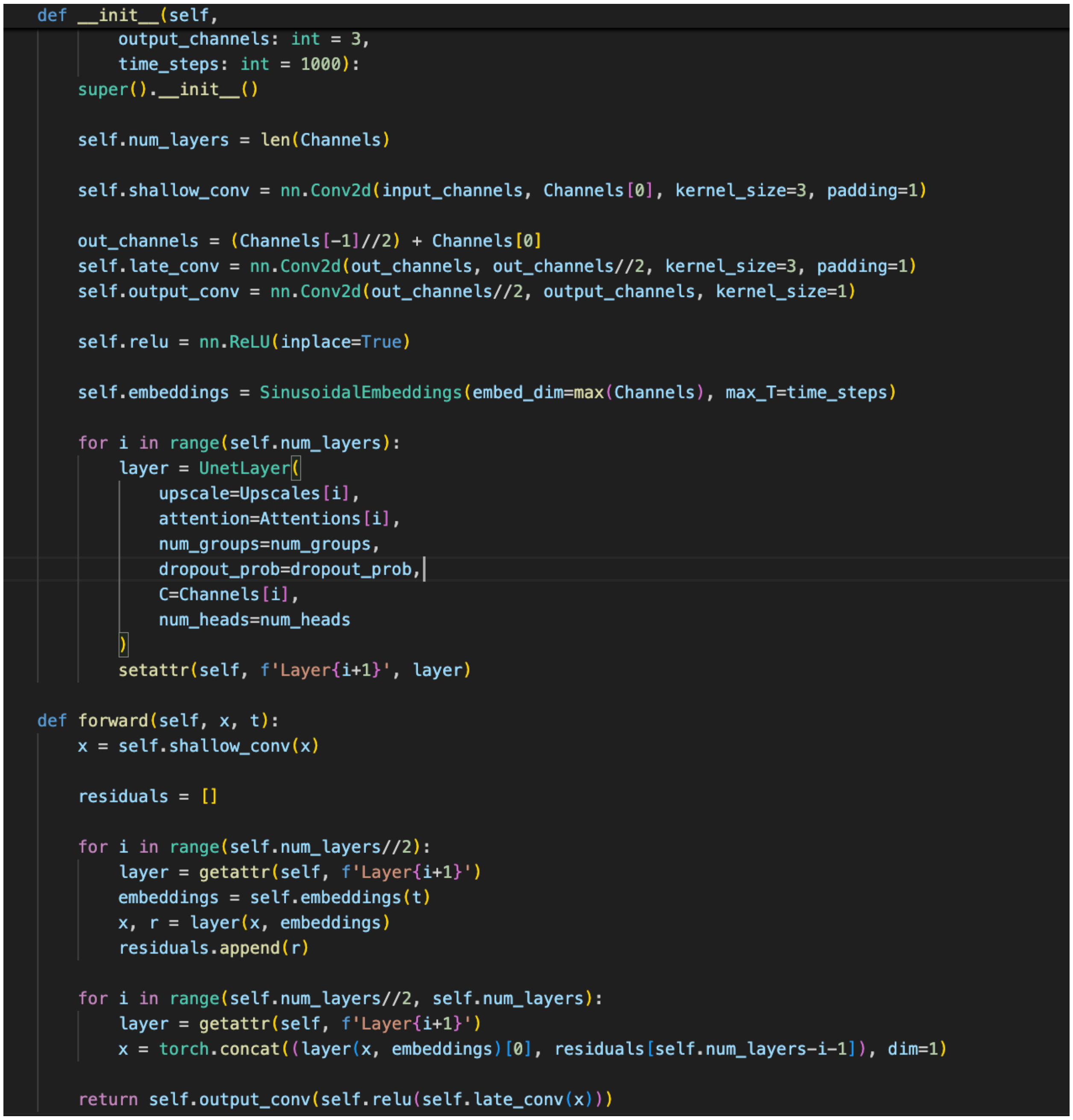
Next came the inpainting task. The idea behind inpainting is straightforward. You take an image, block out part of it with a mask, and then force the model to fill in the missing region. The mask tells the model what pixels it must leave alone and what pixels it needs to recreate. When the diffusion step begins, the model receives a noisy and partially masked image. The goal is to remove noise from the masked section while keeping the unmasked parts stable. Because the mask stays constant throughout the process, the model is guided by the surrounding structure. For example, if a vertical stripe is cut out of the center, the model should extend the stripe straight through the gap. If a dot pattern is missing a dot, the model should place the missing dot in the correct grid position.



This is where diffusion works especially well. The model does not try to recreate the clean image directly in one step. Instead it predicts the noise that was added at each timestep.

Each prediction is compared to the actual noise added earlier, and the model receives penalties based on how far off it is. This gives diffusion models a nice training flow because the loss stays stable and predictable. PyTorch handles backpropagation, and gradient descent updates the weights each time the model makes a wrong noise prediction. After hundreds or thousands of training steps, the model gets better at predicting noise at all levels of corruption. Once it masters noise prediction, it automatically gains the ability to restore images.

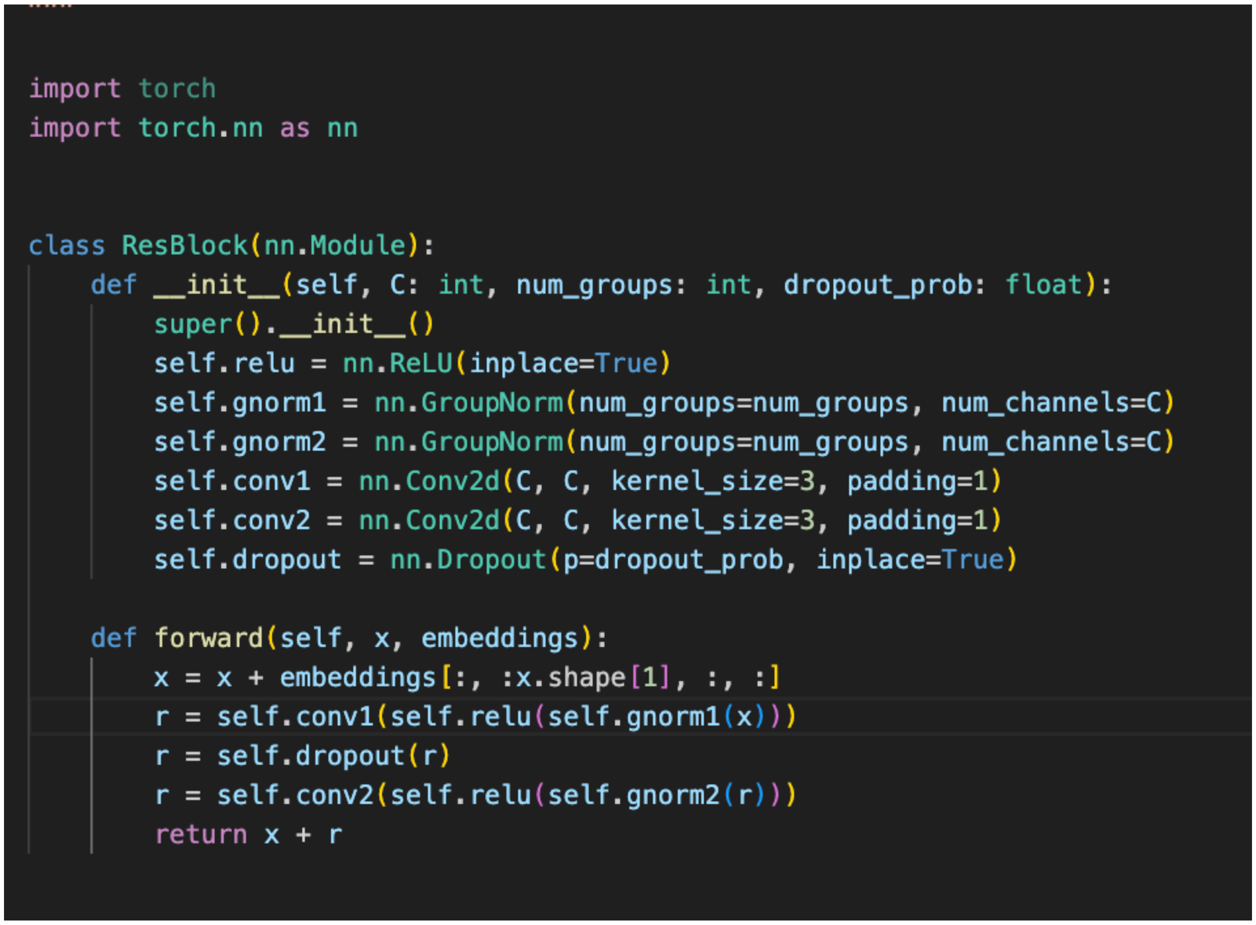
The architecture for the model is a U Net. U Nets are well known for image tasks because they have both downsampling and upsampling paths. The downsampling path pulls out high level information, while the upsampling path rebuilds fine details. The skip connections between the two paths allow detailed features to flow directly into later layers without being lost. This is extremely useful for patterns where alignment matters. If you smooth out or blur small details, you break the structure of the pattern. Skip connections keep those details alive.



```
    def __init__(self,
            output_channels: int = 3,
            time_steps: int = 1000):
        super().__init__()

        self.num_layers = len(Channels)

        self.shallow_conv = nn.Conv2d(input_channels, Channels[0], kernel_size=3, padding=1)

        out_channels = (Channels[-1]//2) + Channels[0]
        self.late_conv = nn.Conv2d(out_channels, out_channels//2, kernel_size=3, padding=1)
        self.output_conv = nn.Conv2d(out_channels//2, output_channels, kernel_size=1)

        self.relu = nn.ReLU(inplace=True)

        self.embeddings = SinusoidalEmbeddings(embed_dim=max(Channels), max_T=time_steps)

        for i in range(self.num_layers):
            layer = UnetLayer(
                upscale=Upscales[i],
                attention=Attentions[i],
                num_groups=num_groups,
                dropout_prob=dropout_prob,
                C=Channels[i],
                num_heads=num_heads
            )
            setattr(self, f'Layer{i+1}', layer)

    def forward(self, x, t):
        x = self.shallow_conv(x)

        residuals = []

        for i in range(self.num_layers//2):
            layer = getattr(self, f'Layer{i+1}')
            embeddings = self.embeddings(t)
            x, r = layer(x, embeddings)
            residuals.append(r)

        for i in range(self.num_layers//2, self.num_layers):
            layer = getattr(self, f'Layer{i+1}')
            x = torch.concat((layer(x, embeddings)[0], residuals[self.num_layers-i-1]), dim=1)

        return self.output_conv(self.relu(self.late_conv(x)))
```

Inside each U Net block is a sequence of residual blocks. Residual blocks take an input, apply normalization, ReLU activation, convolution, dropout, and another convolution, then add the input back to the output. This helps stabilize training because the model can learn new transformations without forgetting the original signal. The model also uses sinusoidal embeddings for timesteps. Diffusion models need this because the amount of noise changes every step. The embedding layer turns each timestep value into a high dimensional vector using a mix of sine and cosine waves. This way the network knows exactly how noisy the image should be at that particular point in the process.

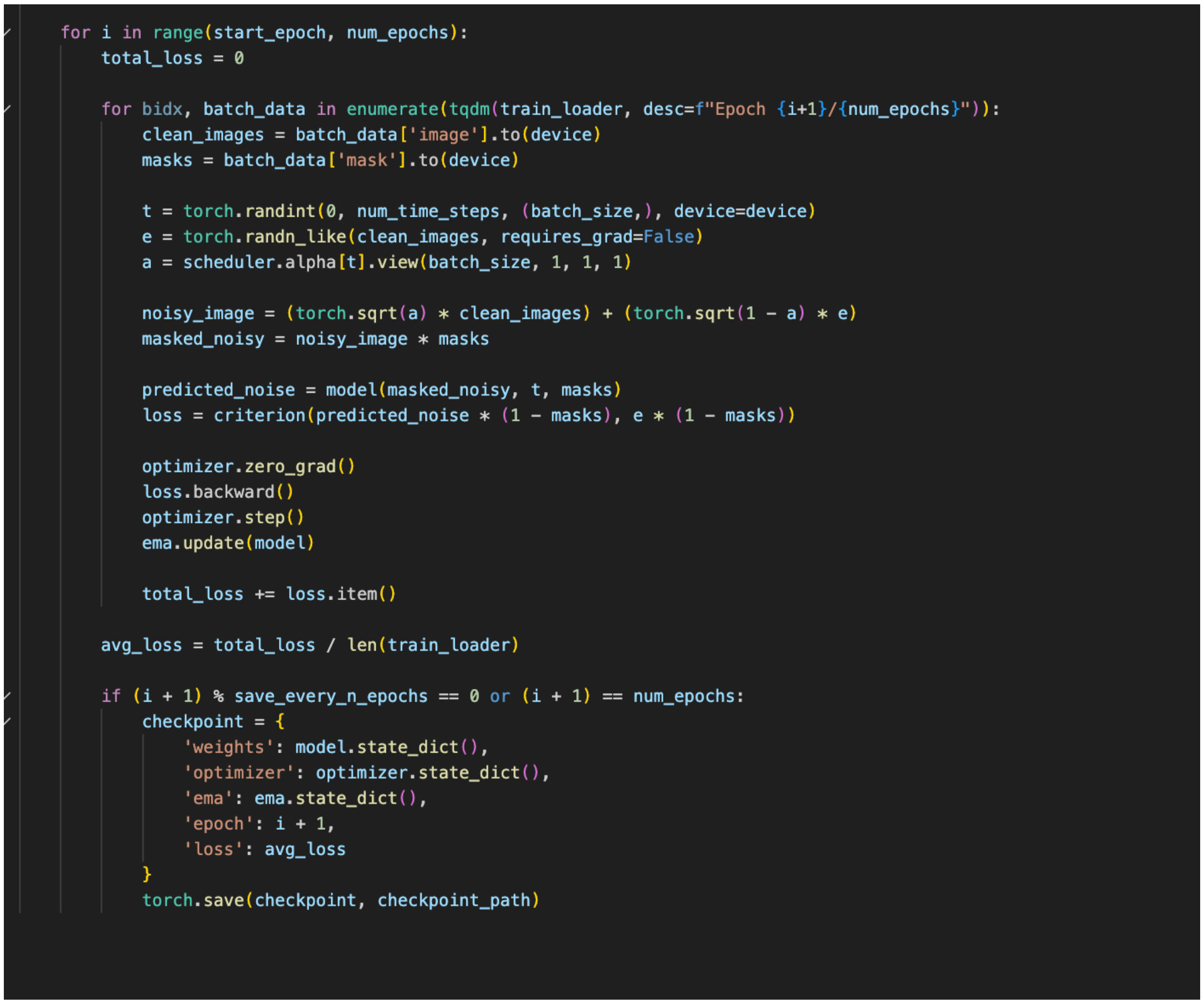
Some parts of the model include attention layers, which allow the network to connect distant pixels. This is especially useful for patterns like stripes, waves, or grids that stretch across the entire image. Without attention, the model might only look at local neighbors, which could cause it to misalign the reconstructed segment. With attention, it can see the picture more globally.



```python
import torch
import torch.nn as nn


class ResBlock(nn.Module):
    def __init__(self, C: int, num_groups: int, dropout_prob: float):
        super().__init__()
        self.relu = nn.ReLU(inplace=True)
        self.gnorm1 = nn.GroupNorm(num_groups=num_groups, num_channels=C)
        self.gnorm2 = nn.GroupNorm(num_groups=num_groups, num_channels=C)
        self.conv1 = nn.Conv2d(C, C, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(C, C, kernel_size=3, padding=1)
        self.dropout = nn.Dropout(p=dropout_prob, inplace=True)

    def forward(self, x, embeddings):
        x = x + embeddings[:, :x.shape[1], :, :]
        r = self.conv1(self.relu(self.gnorm1(x)))
        r = self.dropout(r)
        r = self.conv2(self.relu(self.gnorm2(r)))
        return x + r
```

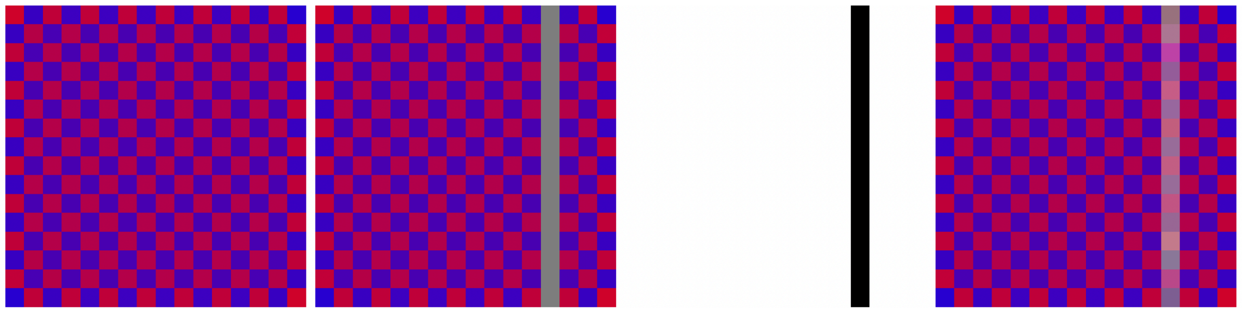
Once everything was set up, I ran training on the dataset. Early in training the model produced complete nonsense. Diffusion models usually do because they have not learned any structure yet. After a few epochs, the model started recognizing simpler patterns like stripes. A bit later it began picking up checkerboards. Eventually it learned to reconstruct more complicated shapes. The more patterns I gave it, the better it became at predicting the missing regions. I used mean squared error as the loss function because the goal was to predict the correct noise values. During training the model would run:

1. Pick a random timestep
2. Add noise to a clean image
3. Apply the mask
4. Predict the noise in the masked region
5. Compare the prediction to the real noise
6. Update the weights using gradient descent



```
for i in range(start_epoch, num_epochs):
    total_loss = 0

    for bidx, batch_data in enumerate(tqdm(train_loader, desc=f"Epoch {i+1}/{num_epochs}")):
        clean_images = batch_data['image'].to(device)
        masks = batch_data['mask'].to(device)

        t = torch.randint(0, num_time_steps, (batch_size,), device=device)
        e = torch.randn_like(clean_images, requires_grad=False)
        a = scheduler.alpha[t].view(batch_size, 1, 1, 1)

        noisy_image = (torch.sqrt(a) * clean_images) + (torch.sqrt(1 - a) * e)
        masked_noisy = noisy_image * masks

        predicted_noise = model(masked_noisy, t, masks)
        loss = criterion(predicted_noise * (1 - masks), e * (1 - masks))

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        ema.update(model)

        total_loss += loss.item()

    avg_loss = total_loss / len(train_loader)

    if (i + 1) % save_every_n_epochs == 0 or (i + 1) == num_epochs:
        checkpoint = {
            'weights': model.state_dict(),
            'optimizer': optimizer.state_dict(),
            'ema': ema.state_dict(),
            'epoch': i + 1,
            'loss': avg_loss
        }
        torch.save(checkpoint, checkpoint_path)
```

After enough training, I finally started getting results that made sense. When I masked out certain spots in a checkerboard, the model filled in the missing sections with the correct alternating colors. When I removed part of a zigzag, the model continued the line at the correct angle. When I tested it on dot grids, the missing dots appeared in the right positions. All of this confirmed that the model had actually learned the structural rules behind each pattern.



Switching from face swapping to pattern inpainting was the best decision for this project. It made everything more attainable with my hardware and allowed me to understand diffusion at a level that actually felt practical. I learned how diffusion models use noise prediction as the core target, how gradient descent optimizes the parameters, and how controlling the dataset can completely change the model's performance. The project taught me that simpler problems can be much better for learning than jumping straight into complicated ones like faces. In the end, my model is able to recreate missing regions in geometric patterns with strong accuracy, and the results make sense visually. It showed me how the pieces of diffusion, gradient descent, architecture design, and dataset control all fit together into a working machine learning system.

https://pytorch.org/docs/stable/index.html