j1zou6xtm

March 31, 2024

[1]:
```
!pip install -q -U einops tqdm
##heavily adopted from the annotated Huggingface diffusion model
import math
from inspect import isfunction
from functools import partial
%matplotlib inline
import matplotlib.pyplot as plt
from tqdm.auto import tqdm
from einops import rearrange
import torch
from torch import nn, einsum
import torch.nn.functional as F


#####hyperparameters######
timesteps = 600
device = "cuda" if torch.cuda.is_available() else "cpu"

image_size = 32
channels = 3
epochs = 25
lr = 1e-3

##########

def exists(x):
    return x is not None

def default(val, d):
    if exists(val):
        return val
    return d() if isfunction(d) else d

class Residual(nn.Module):
    def __init__(self, fn):
        super().__init__()
        self.fn = fn
```

```python
    def forward(self, x, *args, **kwargs):
        return self.fn(x, *args, **kwargs) + x

def Upsample(dim):
    return nn.ConvTranspose2d(dim, dim, 4, 2, 1)

def Downsample(dim):
    return nn.Conv2d(dim, dim, 4, 2, 1)

class SinusoidalPositionEmbeddings(nn.Module):
    def __init__(self, dim):
        super().__init__()
        self.dim = dim

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

class Block(nn.Module):
    def __init__(self, dim, dim_out, groups = 8):
        super().__init__()
        self.proj = nn.Conv2d(dim, dim_out, 3, padding = 1)
        self.norm = nn.GroupNorm(groups, dim_out)
        self.act = nn.SiLU()

    def forward(self, x, scale_shift = None):
        x = self.proj(x)
        x = self.norm(x)

        if exists(scale_shift):
            scale, shift = scale_shift
            x = x * (scale + 1) + shift

        x = self.act(x)
        return x

class ResnetBlock(nn.Module):
    """https://arxiv.org/abs/1512.03385"""

    def __init__(self, dim, dim_out, *, time_emb_dim=None, groups=8):
```

```
        super().__init__()
        self.mlp = (
            nn.Sequential(nn.SiLU(), nn.Linear(time_emb_dim, dim_out))
            if exists(time_emb_dim)
            else None
        )

        self.block1 = Block(dim, dim_out, groups=groups)
        self.block2 = Block(dim_out, dim_out, groups=groups)
        self.res_conv = nn.Conv2d(dim, dim_out, 1) if dim != dim_out else nn.
↪Identity()

    def forward(self, x, time_emb=None):
        h = self.block1(x)

        if exists(self.mlp) and exists(time_emb):
            time_emb = self.mlp(time_emb)
            h = rearrange(time_emb, "b c -> b c 1 1") + h

        h = self.block2(h)
        return h + self.res_conv(x)

class ConvNextBlock(nn.Module):
    """https://arxiv.org/abs/2201.03545"""

    def __init__(self, dim, dim_out, *, time_emb_dim=None, mult=2, norm=True):
        super().__init__()
        self.mlp = (
            nn.Sequential(nn.GELU(), nn.Linear(time_emb_dim, dim))
            if exists(time_emb_dim)
            else None
        )

        self.ds_conv = nn.Conv2d(dim, dim, 7, padding=3, groups=dim)

        self.net = nn.Sequential(
            nn.GroupNorm(1, dim) if norm else nn.Identity(),
            nn.Conv2d(dim, dim_out * mult, 3, padding=1),
            nn.GELU(),
            nn.GroupNorm(1, dim_out * mult),
            nn.Conv2d(dim_out * mult, dim_out, 3, padding=1),
        )

        self.res_conv = nn.Conv2d(dim, dim_out, 1) if dim != dim_out else nn.
↪Identity()

    def forward(self, x, time_emb=None):
```

```python
        h = self.ds_conv(x)

        if exists(self.mlp) and exists(time_emb):
            assert exists(time_emb), "time embedding must be passed in"
            condition = self.mlp(time_emb)
            h = h + rearrange(condition, "b c -> b c 1 1")

        h = self.net(h)
        return h + self.res_conv(x)

class Block(nn.Module):
    def __init__(self, dim, dim_out, groups = 8):
        super().__init__()
        self.proj = nn.Conv2d(dim, dim_out, 3, padding = 1)
        self.norm = nn.GroupNorm(groups, dim_out)
        self.act = nn.SiLU()

    def forward(self, x, scale_shift = None):
        x = self.proj(x)
        x = self.norm(x)

        if exists(scale_shift):
            scale, shift = scale_shift
            x = x * (scale + 1) + shift

        x = self.act(x)
        return x

class ResnetBlock(nn.Module):
    """https://arxiv.org/abs/1512.03385"""

    def __init__(self, dim, dim_out, *, time_emb_dim=None, groups=8):
        super().__init__()
        self.mlp = (
            nn.Sequential(nn.SiLU(), nn.Linear(time_emb_dim, dim_out))
            if exists(time_emb_dim)
            else None
        )

        self.block1 = Block(dim, dim_out, groups=groups)
        self.block2 = Block(dim_out, dim_out, groups=groups)
        self.res_conv = nn.Conv2d(dim, dim_out, 1) if dim != dim_out else nn.
↪Identity()

    def forward(self, x, time_emb=None):
        h = self.block1(x)
```

```python
        if exists(self.mlp) and exists(time_emb):
            time_emb = self.mlp(time_emb)
            h = rearrange(time_emb, "b c -> b c 1 1") + h

        h = self.block2(h)
        return h + self.res_conv(x)

class ConvNextBlock(nn.Module):
    """https://arxiv.org/abs/2201.03545"""

    def __init__(self, dim, dim_out, *, time_emb_dim=None, mult=2, norm=True):
        super().__init__()
        self.mlp = (
            nn.Sequential(nn.GELU(), nn.Linear(time_emb_dim, dim))
            if exists(time_emb_dim)
            else None
        )

        self.ds_conv = nn.Conv2d(dim, dim, 7, padding=3, groups=dim)

        self.net = nn.Sequential(
            nn.GroupNorm(1, dim) if norm else nn.Identity(),
            nn.Conv2d(dim, dim_out * mult, 3, padding=1),
            nn.GELU(),
            nn.GroupNorm(1, dim_out * mult),
            nn.Conv2d(dim_out * mult, dim_out, 3, padding=1),
        )

        self.res_conv = nn.Conv2d(dim, dim_out, 1) if dim != dim_out else nn.
↪Identity()

    def forward(self, x, time_emb=None):
        h = self.ds_conv(x)

        if exists(self.mlp) and exists(time_emb):
            assert exists(time_emb), "time embedding must be passed in"
            condition = self.mlp(time_emb)
            h = h + rearrange(condition, "b c -> b c 1 1")

        h = self.net(h)
        return h + self.res_conv(x)

class Attention(nn.Module):
    def __init__(self, dim, heads=4, dim_head=32):
        super().__init__()
        self.scale = dim_head**-0.5
        self.heads = heads
```

```
        hidden_dim = dim_head * heads
        self.to_qkv = nn.Conv2d(dim, hidden_dim * 3, 1, bias=False)
        self.to_out = nn.Conv2d(hidden_dim, dim, 1)

    def forward(self, x):
        b, c, h, w = x.shape
        qkv = self.to_qkv(x).chunk(3, dim=1)
        q, k, v = map(
            lambda t: rearrange(t, "b (h c) x y -> b h c (x y)", h=self.heads),␣
 ↪qkv
        )
        q = q * self.scale

        sim = einsum("b h d i, b h d j -> b h i j", q, k)
        sim = sim - sim.amax(dim=-1, keepdim=True).detach()
        attn = sim.softmax(dim=-1)

        out = einsum("b h i j, b h d j -> b h i d", attn, v)
        out = rearrange(out, "b h (x y) d -> b (h d) x y", x=h, y=w)
        return self.to_out(out)

class LinearAttention(nn.Module):
    def __init__(self, dim, heads=4, dim_head=32):
        super().__init__()
        self.scale = dim_head**-0.5
        self.heads = heads
        hidden_dim = dim_head * heads
        self.to_qkv = nn.Conv2d(dim, hidden_dim * 3, 1, bias=False)

        self.to_out = nn.Sequential(nn.Conv2d(hidden_dim, dim, 1),
                                    nn.GroupNorm(1, dim))

    def forward(self, x):
        b, c, h, w = x.shape
        qkv = self.to_qkv(x).chunk(3, dim=1)
        q, k, v = map(
            lambda t: rearrange(t, "b (h c) x y -> b h c (x y)", h=self.heads),␣
 ↪qkv
        )

        q = q.softmax(dim=-2)
        k = k.softmax(dim=-1)

        q = q * self.scale
        context = torch.einsum("b h d n, b h e n -> b h d e", k, v)

        out = torch.einsum("b h d e, b h d n -> b h e n", context, q)
```

```
        out = rearrange(out, "b h c (x y) -> b (h c) x y", h=self.heads, x=h,␣
↪y=w)
        return self.to_out(out)

class PreNorm(nn.Module):
    def __init__(self, dim, fn):
        super().__init__()
        self.fn = fn
        self.norm = nn.GroupNorm(1, dim)

    def forward(self, x):
        x = self.norm(x)
        return self.fn(x)

class Unet(nn.Module):
    def __init__(
        self,
        dim,
        init_dim=None,
        out_dim=None,
        dim_mults=(1, 2, 4, 8),
        channels=3,
        with_time_emb=True,
        resnet_block_groups=8,
        use_convnext=True,
        convnext_mult=2,
    ):
        super().__init__()

        # determine dimensions
        self.channels = channels

        init_dim = default(init_dim, dim // 3 * 2)
        self.init_conv = nn.Conv2d(channels, init_dim, 7, padding=3)

        dims = [init_dim, *map(lambda m: dim * m, dim_mults)]
        in_out = list(zip(dims[:-1], dims[1:]))

        if use_convnext:
            block_klass = partial(ConvNextBlock, mult=convnext_mult)
        else:
            block_klass = partial(ResnetBlock, groups=resnet_block_groups)

        # time embeddings
        if with_time_emb:
            time_dim = dim * 4
            self.time_mlp = nn.Sequential(
```

```python
            SinusoidalPositionEmbeddings(dim),
            nn.Linear(dim, time_dim),
            nn.GELU(),
            nn.Linear(time_dim, time_dim),
        )
    else:
        time_dim = None
        self.time_mlp = None

    # layers
    self.downs = nn.ModuleList([])
    self.ups = nn.ModuleList([])
    num_resolutions = len(in_out)

    for ind, (dim_in, dim_out) in enumerate(in_out):
        is_last = ind >= (num_resolutions - 1)

        self.downs.append(
            nn.ModuleList(
                [
                    block_klass(dim_in, dim_out, time_emb_dim=time_dim),
                    block_klass(dim_out, dim_out, time_emb_dim=time_dim),
                    Residual(PreNorm(dim_out, LinearAttention(dim_out))),
                    Downsample(dim_out) if not is_last else nn.Identity(),
                ]
            )
        )

    mid_dim = dims[-1]
    self.mid_block1 = block_klass(mid_dim, mid_dim, time_emb_dim=time_dim)
    self.mid_attn = Residual(PreNorm(mid_dim, Attention(mid_dim)))
    self.mid_block2 = block_klass(mid_dim, mid_dim, time_emb_dim=time_dim)

    for ind, (dim_in, dim_out) in enumerate(reversed(in_out[1:])):
        is_last = ind >= (num_resolutions - 1)

        self.ups.append(
            nn.ModuleList(
                [
                    block_klass(dim_out * 2, dim_in, time_emb_dim=time_dim),
                    block_klass(dim_in, dim_in, time_emb_dim=time_dim),
                    Residual(PreNorm(dim_in, LinearAttention(dim_in))),
                    Upsample(dim_in) if not is_last else nn.Identity(),
                ]
            )
        )
```

```
        out_dim = default(out_dim, channels)
        self.final_conv = nn.Sequential(
            block_klass(dim, dim), nn.Conv2d(dim, out_dim, 1)
        )

    def forward(self, x, time):
        x = self.init_conv(x)

        t = self.time_mlp(time) if exists(self.time_mlp) else None

        h = []

        # downsample
        for block1, block2, attn, downsample in self.downs:
            x = block1(x, t)
            x = block2(x, t)
            x = attn(x)
            h.append(x)
            x = downsample(x)

        # bottleneck
        x = self.mid_block1(x, t)
        x = self.mid_attn(x)
        x = self.mid_block2(x, t)

        # upsample
        for block1, block2, attn, upsample in self.ups:
            x = torch.cat((x, h.pop()), dim=1)
            x = block1(x, t)
            x = block2(x, t)
            x = attn(x)
            x = upsample(x)

        return self.final_conv(x)

def cosine_beta_schedule(timesteps, s=0.008):##cosine beta schedule works best
    """
    cosine schedule as proposed in https://arxiv.org/abs/2102.09672
    """
    steps = timesteps + 1
    x = torch.linspace(0, timesteps, steps)
    alphas_cumprod = torch.cos(((x / timesteps) + s) / (1 + s) * torch.pi * 0.
 ↪5) ** 2
    alphas_cumprod = alphas_cumprod / alphas_cumprod[0]
    betas = 1 - (alphas_cumprod[1:] / alphas_cumprod[:-1])
    return torch.clip(betas, 0.0001, 0.9999)
```

```
# define beta schedule
betas = cosine_beta_schedule(timesteps=timesteps)

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

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

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

def extract(a, t, x_shape):
    batch_size = t.shape[0]
    out = a.gather(-1, t.cpu())
    return out.reshape(batch_size, *((1,) * (len(x_shape) - 1))).to(t.device)

# forward diffusion
def q_sample(x_start, t, noise=None):
    if noise is None:
        noise = torch.randn_like(x_start)

    sqrt_alphas_cumprod_t = extract(sqrt_alphas_cumprod, t, x_start.shape)
    sqrt_one_minus_alphas_cumprod_t = extract(
        sqrt_one_minus_alphas_cumprod, t, x_start.shape
    )

    return sqrt_alphas_cumprod_t * x_start + sqrt_one_minus_alphas_cumprod_t *␣
 ↪noise

def get_noisy_image(x_start, t):
  # add noise
  x_noisy = q_sample(x_start, t=t)

  # turn back into PIL image
  noisy_image = reverse_transform(x_noisy.squeeze())
  return noisy_image

def p_losses(denoise_model, x_start, t, noise=None, loss_type="huber"):
    if noise is None:
        noise = torch.randn_like(x_start)
    x_noisy = q_sample(x_start=x_start, t=t, noise=noise)
```

```
    predicted_noise = denoise_model(x_noisy, t)
    loss = F.smooth_l1_loss(noise, predicted_noise)
    return loss
```

[2]:
```
# # Import the required libraries
!pip install gdown
import gdown
import gdown
import zipfile
import os
url = 'https://drive.google.com/uc?id=1WO2K-SfU2dntGU4Bb3IYBp9Rh7rtTYEr'
output_path = 'large_file.hdf5'
gdown.download(url, output_path, quiet=False)
import matplotlib.pyplot as plt
import numpy as np
import h5py
with h5py.File('large_file.hdf5', 'r') as file:
    train_imgs = np.array(file['X_jets'][:4096])
    test_imgs = np.array(file['X_jets'][4096:4096+1024])
    train_labels = np.array(file['y'][:4096])
    test_labels = np.array(file['y'][4096:4096+1024])
    print(train_imgs[0].shape)

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import torchvision.transforms.v2 as transforms
class Data(torch.utils.data.Dataset):
    def __init__(self,imgs):
        super().__init__()
        self.transform = transforms.Compose([
            transforms.ToTensor(),
            transforms.Resize((32 ,32)),###small size since diffusion is␣
 ↪expensive
            #transforms.Normalize([0.5,],[0.5,]),
        ])
        self.imgs = imgs
    def __len__(self):
        return len(self.imgs)
    def __getitem__(self,idx):
        img = self.transform(self.imgs[idx])
        return img

train_loader = torch.utils.data.DataLoader(Data(train_imgs), batch_size=128)
val_loader = torch.utils.data.DataLoader(Data(test_imgs), batch_size=128)
```

```python
for imgs in train_loader:
    print(imgs.shape)
    img = imgs[0]
    plt.imshow(img[2])
    break

@torch.no_grad()
def p_sample(model, x, t, t_index):
    betas_t = extract(betas, t, x.shape)
    sqrt_one_minus_alphas_cumprod_t = extract(
        sqrt_one_minus_alphas_cumprod, t, x.shape
    )
    sqrt_recip_alphas_t = extract(sqrt_recip_alphas, t, x.shape)

    # Equation 11 in the paper
    # Use our model (noise predictor) to predict the mean
    model_mean = sqrt_recip_alphas_t * (
        x - betas_t * model(x, t) / sqrt_one_minus_alphas_cumprod_t
    )

    if t_index == 0:
        return model_mean
    else:
        posterior_variance_t = extract(posterior_variance, t, x.shape)
        noise = torch.randn_like(x)
        # Algorithm 2 line 4:
        return model_mean + torch.sqrt(posterior_variance_t) * noise

# Algorithm 2 but save all images:
@torch.no_grad()
def p_sample_loop(model, shape, imginit=None):
    device = next(model.parameters()).device

    b = shape[0]
    # start from pure noise (for each example in the batch)
    img = torch.randn(shape, device=device) if imginit is None else imginit

    imgs = []

    for i in tqdm(reversed(range(0, timesteps)), desc='sampling loop time␣
↪step', total=timesteps):
        img = p_sample(model, img, torch.full((b,), i, device=device,␣
↪dtype=torch.long), i)
        imgs.append(img.cpu().numpy())
    return imgs

@torch.no_grad()
```

```
def sample(model, image_size, batch_size=16, channels=3, imginit=None):
    return p_sample_loop(model, shape=(batch_size, channels, image_size,␣
 ↪image_size),imginit=imginit)


def num_to_groups(num, divisor):
    groups = num // divisor
    remainder = num % divisor
    arr = [divisor] * groups
    if remainder > 0:
        arr.append(remainder)
    return arr


from torch.optim import Adam


model = Unet(
    dim=image_size,
    channels=channels,
    dim_mults=(1, 2, 4,)
)
model.to(device)

optimizer = Adam(model.parameters(), lr=1e-3)

from torchvision.utils import save_image
```

```
Requirement already satisfied: gdown in /opt/conda/lib/python3.10/site-packages
(5.1.0)
Requirement already satisfied: beautifulsoup4 in /opt/conda/lib/python3.10/site-
packages (from gdown) (4.12.2)
Requirement already satisfied: filelock in /opt/conda/lib/python3.10/site-
packages (from gdown) (3.13.1)
Requirement already satisfied: requests[socks] in
/opt/conda/lib/python3.10/site-packages (from gdown) (2.31.0)
Requirement already satisfied: tqdm in /opt/conda/lib/python3.10/site-packages
(from gdown) (4.66.2)
Requirement already satisfied: soupsieve>1.2 in /opt/conda/lib/python3.10/site-
packages (from beautifulsoup4->gdown) (2.5)
Requirement already satisfied: charset-normalizer<4,>=2 in
/opt/conda/lib/python3.10/site-packages (from requests[socks]->gdown) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /opt/conda/lib/python3.10/site-
packages (from requests[socks]->gdown) (3.6)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/opt/conda/lib/python3.10/site-packages (from requests[socks]->gdown) (1.26.18)
Requirement already satisfied: certifi>=2017.4.17 in
```

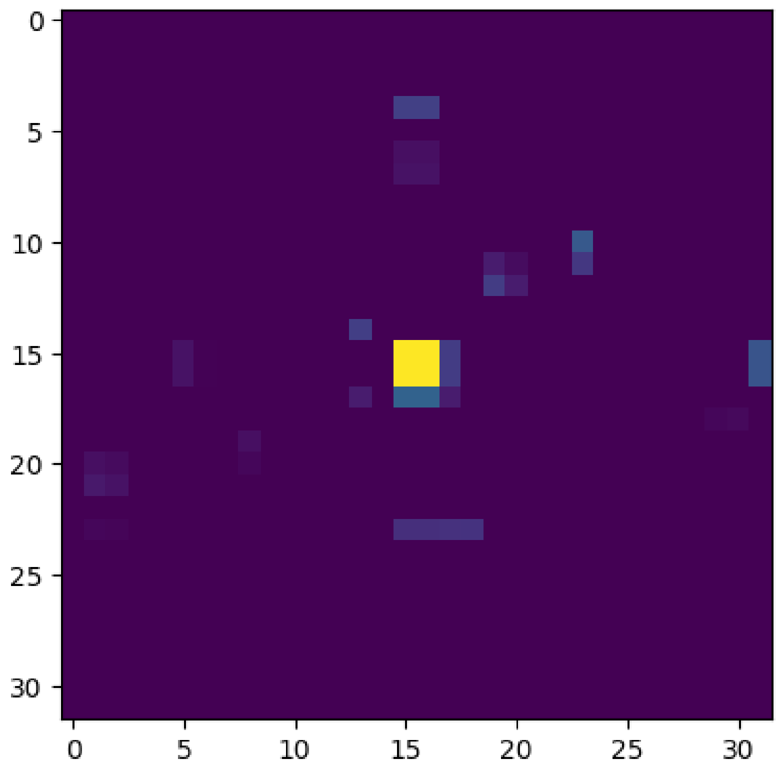
```
/opt/conda/lib/python3.10/site-packages (from requests[socks]->gdown) (2024.2.2)
Requirement already satisfied: PySocks!=1.5.7,>=1.5.6 in
/opt/conda/lib/python3.10/site-packages (from requests[socks]->gdown) (1.7.1)
```

```
Downloading…
From (original):
https://drive.google.com/uc?id=1WO2K-SfU2dntGU4Bb3IYBp9Rh7rtTYEr
From (redirected): https://drive.google.com/uc?id=1WO2K-SfU2dntGU4Bb3IYBp9Rh7rtT
YEr&confirm=t&uuid=15e3e235-b54b-4ff0-963e-8cfbf40621bd
To: /kaggle/working/large_file.hdf5
100%|     | 701M/701M [00:04<00:00, 144MB/s]
```

```
(125, 125, 3)
```

```
/opt/conda/lib/python3.10/site-
packages/torchvision/transforms/v2/_deprecated.py:43: UserWarning: The transform
`ToTensor()` is deprecated and will be removed in a future release. Instead,
please use `v2.Compose([v2.ToImage(), v2.ToDtype(torch.float32, scale=True)])`.
  warnings.warn(
/opt/conda/lib/python3.10/site-
packages/torchvision/transforms/functional.py:1603: UserWarning: The default
value of the antialias parameter of all the resizing transforms (Resize(),
RandomResizedCrop(), etc.) will change from None to True in v0.17, in order to
be consistent across the PIL and Tensor backends. To suppress this warning,
directly pass antialias=True (recommended, future default), antialias=None
(current default, which means False for Tensors and True for PIL), or
antialias=False (only works on Tensors - PIL will still use antialiasing). This
also applies if you are using the inference transforms from the models weights:
update the call to weights.transforms(antialias=True).
  warnings.warn(
```

```
torch.Size([128, 3, 32, 32])
```

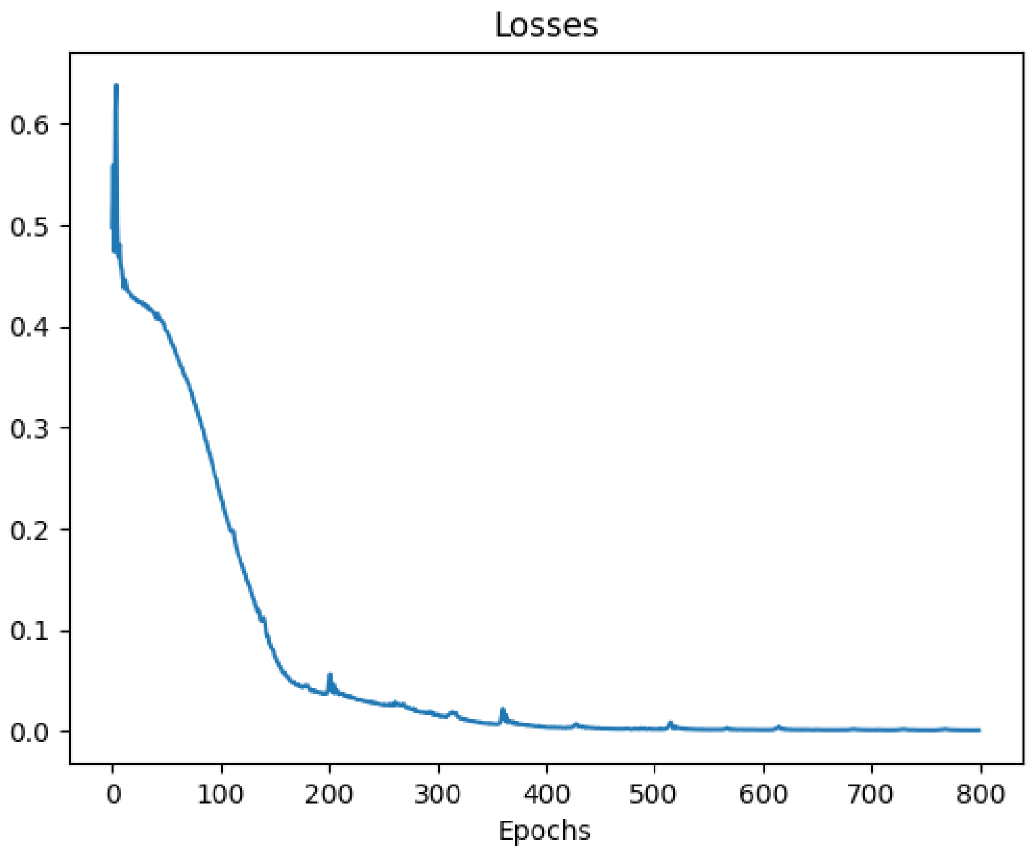


[3]:
```
losses = []
for epoch in range(epochs):
    for step, batch in enumerate(train_loader):
      optimizer.zero_grad()

      batch_size = 128
      batch = batch.to(device)

      # Algorithm 1 line 3: sample t uniformally for every example in the batch
      t = torch.randint(0, timesteps, (batch_size,), device=device).long()

      loss = p_losses(model, batch, t, loss_type="huber")
      losses.append(loss)

      if step % 100 == 0:
        print("Loss:", loss.item())

      loss.backward()
      optimizer.step()
```

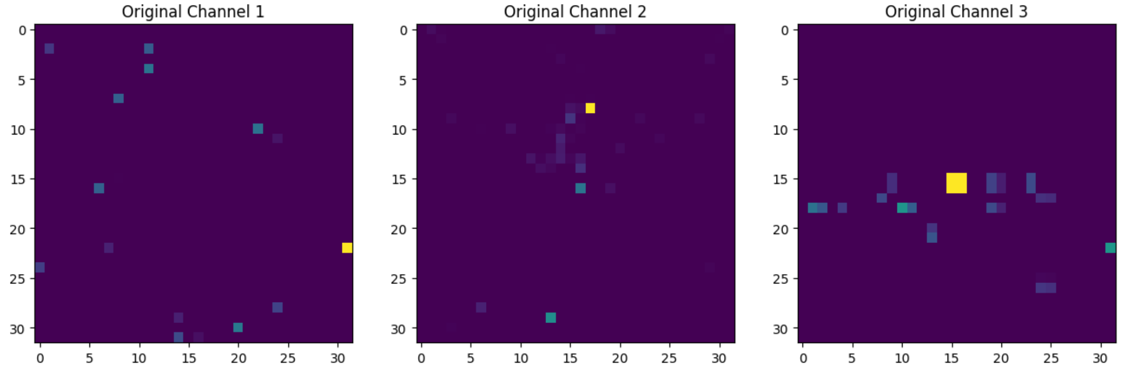
```
Loss: 0.49743813276290894
Loss: 0.4206655025482178
```

```
Loss: 0.3598029315471649
Loss: 0.24995480477809906
Loss: 0.1377469003200531
Loss: 0.05523230880498886
Loss: 0.03757734224200249
Loss: 0.032036300748586655
Loss: 0.026627536863088608
Loss: 0.01830369234085083
Loss: 0.013073218986392021
Loss: 0.006850262638181448
Loss: 0.005789363291114569
Loss: 0.003185780718922615
Loss: 0.002867076313123107
Loss: 0.0018804128048941493
Loss: 0.0029999602120369673
Loss: 0.0013614576309919357
Loss: 0.0011752552818506956
Loss: 0.0012310110032558441
Loss: 0.0011462605325505137
Loss: 0.0007712678052484989
Loss: 0.0008282518829219043
Loss: 0.0009146773954853415
Loss: 0.0017155862879008055
```

[5]:
```
plt.plot([l.detach().cpu() for l in losses])
plt.title("Losses")
plt.xlabel("Epochs")
plt.show()
```

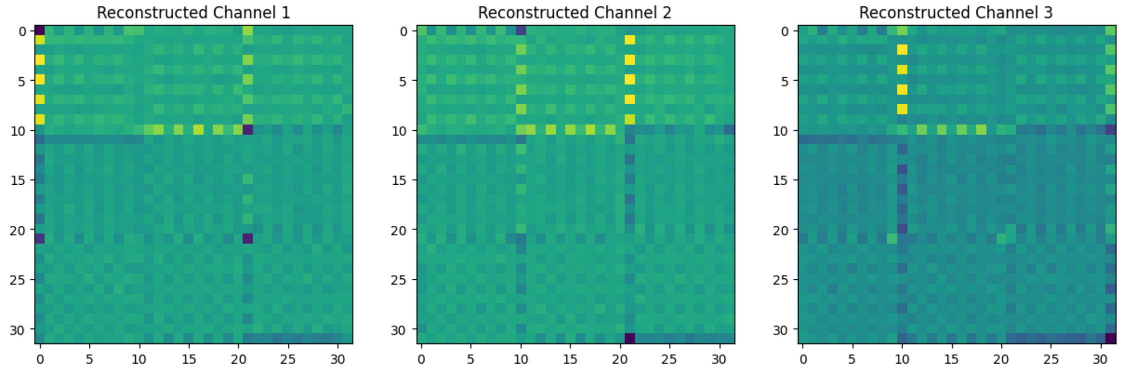


[6]:
```
import matplotlib.pyplot as plt

# Get the image from the validation loader
for imgs in val_loader:
    img = imgs[10].unsqueeze(0)
    break

# Plot all three channels using subplots
fig, axs = plt.subplots(1, 3, figsize=(15, 5))

# Plot each channel separately
for i in range(3):
    axs[i].imshow(img.squeeze(0).permute(1, 2, 0).cpu()[:, :, i])
    axs[i].set_title(f'Original Channel {i+1}')

plt.show()
```




```
[7]: # sample 64 images
samples = sample(model, image_size=image_size, batch_size=64,␣
 ↪channels=channels, imginit=img.to(device))

# Show a random one
random_index = 63  # Index of the image to display
image_to_display = samples[-1][random_index].reshape(image_size, image_size,␣
 ↪channels)

# Plot all three channels using subplots
fig, axs = plt.subplots(1, 3, figsize=(15, 5))

# Plot each channel separately
for i in range(3):
    axs[i].imshow(image_to_display[:,:,i])
    axs[i].set_title(f'Reconstructed Channel {i+1}')

plt.show()
```

```
sampling loop time step:   0%|          | 0/600 [00:00<?, ?it/s]
```



```
[ ]:
```