def modulate(x, shift, scale):

return x \* (1 + scale.unsqueeze(1)) + shift.unsqueeze(1)

class Attention(nn.Module):

def \_\_init\_\_(self, dim, num\_heads=8, qkv\_bias=False, attn\_drop=0., proj\_drop=0., use\_lora=False, attention\_mode='math'):

super().\_\_init\_\_()

assert dim % num\_heads == 0, 'dim should be divisible by num\_heads'

self.num\_heads = num\_heads

head\_dim = dim // num\_heads

self.scale = head\_dim \*\* -0.5

self.attention\_mode = attention\_mode

self.qkv = nn.Linear(dim, dim \* 3, bias=qkv\_bias)

self.attn\_drop = nn.Dropout(attn\_drop)

self.proj = nn.Linear(dim, dim)

self.proj\_drop = nn.Dropout(proj\_drop)

def forward(self, x):

B, N, C = x.shape

qkv = self.qkv(x).reshape(B, N, 3, self.num\_heads, C // self.num\_heads).permute(2, 0, 3, 1, 4).contiguous()

q, k, v = qkv.unbind(0) # make torchscript happy (cannot use tensor as tuple)

if self.attention\_mode == 'xformers': # cause loss nan while using with amp

x = xformers.ops.memory\_efficient\_attention(q, k, v).reshape(B, N, C)

elif self.attention\_mode == 'flash':

# cause loss nan while using with amp

# Optionally use the context manager to ensure one of the fused kernels is run

with torch.backends.cuda.sdp\_kernel(enable\_math=False):

x = torch.nn.functional.scaled\_dot\_product\_attention(q, k, v).reshape(B, N, C) # require pytorch 2.0

elif self.attention\_mode == 'math':

attn = (q @ k.transpose(-2, -1)) \* self.scale

attn = attn.softmax(dim=-1)

attn = self.attn\_drop(attn)

x = (attn @ v).transpose(1, 2).reshape(B, N, C)

else:

raise NotImplemented

x = self.proj(x)

x = self.proj\_drop(x)

return x

#################################################################################

# Embedding Layers for Timesteps and Class Labels #

#################################################################################

class TimestepEmbedder(nn.Module):

"""

Embeds scalar timesteps into vector representations.

"""

def \_\_init\_\_(self, hidden\_size, frequency\_embedding\_size=256):

super().\_\_init\_\_()

self.mlp = nn.Sequential(

nn.Linear(frequency\_embedding\_size, hidden\_size, bias=True),

nn.SiLU(),

nn.Linear(hidden\_size, hidden\_size, bias=True),

)

self.frequency\_embedding\_size = frequency\_embedding\_size

@staticmethod

def timestep\_embedding(t, dim, max\_period=10000):

"""

Create sinusoidal timestep embeddings.

:param t: a 1-D Tensor of N indices, one per batch element.

These be fractional.

:param dim: the dimension of the output.

:param max\_period: controls the minimum frequency of the embeddings.

:return: an (N, D) Tensor of positional embeddings.

"""

# https://github.com/openai/glide-text2im/blob/main/glide\_text2im/nn.py

half = dim // 2

freqs = torch.exp(

-math.log(max\_period) \* torch.arange(start=0, end=half, dtype=torch.float32) / half

).to(device=t.device)

args = t[:, None].float() \* freqs[None]

embedding = torch.cat([torch.cos(args), torch.sin(args)], dim=-1)

if dim % 2:

embedding = torch.cat([embedding, torch.zeros\_like(embedding[:, :1])], dim=-1)

return embedding

def forward(self, t, use\_fp16=False):

t\_freq = self.timestep\_embedding(t, self.frequency\_embedding\_size)

if use\_fp16:

t\_freq = t\_freq.to(dtype=torch.float16)

t\_emb = self.mlp(t\_freq)

return t\_emb

class LabelEmbedder(nn.Module):

"""

Embeds class labels into vector representations. Also handles label dropout for classifier-free guidance.

"""

def \_\_init\_\_(self, num\_classes, hidden\_size, dropout\_prob):

super().\_\_init\_\_()

use\_cfg\_embedding = dropout\_prob > 0

self.embedding\_table = nn.Embedding(num\_classes + use\_cfg\_embedding, hidden\_size)

self.num\_classes = num\_classes

self.dropout\_prob = dropout\_prob

def token\_drop(self, labels, force\_drop\_ids=None):

"""

Drops labels to enable classifier-free guidance.

"""

if force\_drop\_ids is None:

drop\_ids = torch.rand(labels.shape[0], device=labels.device) < self.dropout\_prob

else:

drop\_ids = force\_drop\_ids == 1

labels = torch.where(drop\_ids, self.num\_classes, labels)

return labels

def forward(self, labels, train, force\_drop\_ids=None):

use\_dropout = self.dropout\_prob > 0

if (train and use\_dropout) or (force\_drop\_ids is not None):

labels = self.token\_drop(labels, force\_drop\_ids)

embeddings = self.embedding\_table(labels)

return embeddings

class TransformerBlock(nn.Module):

"""

A EnDora block with adaptive layer norm zero (adaLN-Zero) conditioning.

"""

def \_\_init\_\_(self, hidden\_size, num\_heads, mlp\_ratio=4.0, \*\*block\_kwargs):

super().\_\_init\_\_()

self.norm1 = nn.LayerNorm(hidden\_size, elementwise\_affine=False, eps=1e-6)

self.attn = Attention(hidden\_size, num\_heads=num\_heads, qkv\_bias=True, \*\*block\_kwargs)

self.norm2 = nn.LayerNorm(hidden\_size, elementwise\_affine=False, eps=1e-6)

mlp\_hidden\_dim = int(hidden\_size \* mlp\_ratio)

approx\_gelu = lambda: nn.GELU(approximate="tanh")

self.mlp = Mlp(in\_features=hidden\_size, hidden\_features=mlp\_hidden\_dim, act\_layer=approx\_gelu, drop=0)

self.adaLN\_modulation = nn.Sequential(

nn.SiLU(),

nn.Linear(hidden\_size, 6 \* hidden\_size, bias=True)

)

def forward(self, x, c):

shift\_msa, scale\_msa, gate\_msa, shift\_mlp, scale\_mlp, gate\_mlp = self.adaLN\_modulation(c).chunk(6, dim=1)

x = x + gate\_msa.unsqueeze(1) \* self.attn(modulate(self.norm1(x), shift\_msa, scale\_msa))

x = x + gate\_mlp.unsqueeze(1) \* self.mlp(modulate(self.norm2(x), shift\_mlp, scale\_mlp))

return x

class FinalLayer(nn.Module):

"""

The final layer of EnDora.

"""

def \_\_init\_\_(self, hidden\_size, patch\_size, out\_channels):

super().\_\_init\_\_()

self.norm\_final = nn.LayerNorm(hidden\_size, elementwise\_affine=False, eps=1e-6)

self.linear = nn.Linear(hidden\_size, patch\_size \* patch\_size \* out\_channels, bias=True)

self.adaLN\_modulation = nn.Sequential(

nn.SiLU(),

nn.Linear(hidden\_size, 2 \* hidden\_size, bias=True)

)

def forward(self, x, c):

shift, scale = self.adaLN\_modulation(c).chunk(2, dim=1)

x = modulate(self.norm\_final(x), shift, scale)

x = self.linear(x)

return x

class EnDora(nn.Module):

"""

Diffusion model with a Transformer backbone.

"""

def \_\_init\_\_(

self,

input\_size=32,

patch\_size=2,

in\_channels=4,

hidden\_size=1152,

depth=28,

num\_heads=16,

mlp\_ratio=4.0,

num\_frames=16,

class\_dropout\_prob=0.1,

num\_classes=1000,

learn\_sigma=True,

extras=2,

attention\_mode='math',

):

super().\_\_init\_\_()

self.learn\_sigma = learn\_sigma

self.in\_channels = in\_channels

self.out\_channels = in\_channels \* 2 if learn\_sigma else in\_channels

self.patch\_size = patch\_size

self.num\_heads = num\_heads

self.extras = extras

self.num\_frames = num\_frames

self.x\_embedder = PatchEmbed(input\_size, patch\_size, in\_channels, hidden\_size, bias=True)

self.t\_embedder = TimestepEmbedder(hidden\_size)

if self.extras == 2:

self.y\_embedder = LabelEmbedder(num\_classes, hidden\_size, class\_dropout\_prob)

if self.extras == 78: # timestep + text\_embedding

self.text\_embedding\_projection = nn.Sequential(

nn.SiLU(),

nn.Linear(1024, hidden\_size, bias=True)

)

num\_patches = self.x\_embedder.num\_patches

# Will use fixed sin-cos embedding:

self.pos\_embed = nn.Parameter(torch.zeros(1, num\_patches, hidden\_size), requires\_grad=False)

self.temp\_embed = nn.Parameter(torch.zeros(1, num\_frames, hidden\_size), requires\_grad=False)

self.pooling = nn.AdaptiveAvgPool1d(64)

self.linear = nn.Linear(in\_features=384, out\_features=1152)

self.linear\_2 = nn.Linear(in\_features=384\*4, out\_features=1152)

self.cov = nn.Conv2d(in\_channels=384, out\_channels=1152, kernel\_size=2, stride=2, bias=False)

self.blocks = nn.ModuleList([

TransformerBlock(hidden\_size, num\_heads, mlp\_ratio=mlp\_ratio, attention\_mode=attention\_mode) for \_ in range(depth)

])

self.final\_layer = FinalLayer(hidden\_size, patch\_size, self.out\_channels)

self.initialize\_weights()

def initialize\_weights(self):

# Initialize transformer layers:

def \_basic\_init(module):

if isinstance(module, nn.Linear):

torch.nn.init.xavier\_uniform\_(module.weight)

if module.bias is not None:

nn.init.constant\_(module.bias, 0)

self.apply(\_basic\_init)

# Initialize (and freeze) pos\_embed by sin-cos embedding:

pos\_embed = get\_2d\_sincos\_pos\_embed(self.pos\_embed.shape[-1], int(self.x\_embedder.num\_patches \*\* 0.5))

self.pos\_embed.data.copy\_(torch.from\_numpy(pos\_embed).float().unsqueeze(0))

temp\_embed = get\_1d\_sincos\_temp\_embed(self.temp\_embed.shape[-1], self.temp\_embed.shape[-2])

self.temp\_embed.data.copy\_(torch.from\_numpy(temp\_embed).float().unsqueeze(0))

# Initialize patch\_embed like nn.Linear (instead of nn.Conv2d):

w = self.x\_embedder.proj.weight.data

nn.init.xavier\_uniform\_(w.view([w.shape[0], -1]))

nn.init.constant\_(self.x\_embedder.proj.bias, 0)

if self.extras == 2:

# Initialize label embedding table:

nn.init.normal\_(self.y\_embedder.embedding\_table.weight, std=0.02)

# Initialize timestep embedding MLP:

nn.init.normal\_(self.t\_embedder.mlp[0].weight, std=0.02)

nn.init.normal\_(self.t\_embedder.mlp[2].weight, std=0.02)

# Zero-out adaLN modulation layers in EnDora blocks:

for block in self.blocks:

nn.init.constant\_(block.adaLN\_modulation[-1].weight, 0)

nn.init.constant\_(block.adaLN\_modulation[-1].bias, 0)

# Zero-out output layers:

nn.init.constant\_(self.final\_layer.adaLN\_modulation[-1].weight, 0)

nn.init.constant\_(self.final\_layer.adaLN\_modulation[-1].bias, 0)

nn.init.constant\_(self.final\_layer.linear.weight, 0)

nn.init.constant\_(self.final\_layer.linear.bias, 0)

def unpatchify(self, x):

"""

x: (N, T, patch\_size\*\*2 \* C)

imgs: (N, H, W, C)

"""

c = self.out\_channels

p = self.x\_embedder.patch\_size[0]

h = w = int(x.shape[1] \*\* 0.5)

assert h \* w == x.shape[1]

x = x.reshape(shape=(x.shape[0], h, w, p, p, c))

x = torch.einsum('nhwpqc->nchpwq', x)

imgs = x.reshape(shape=(x.shape[0], c, h \* p, h \* p))

return imgs

# @torch.cuda.amp.autocast()

# @torch.compile

def forward(

self,

x,

t,

attentions=None,

special\_list=[],

mode="type0",

y=None,

use\_fp16=False,

y\_image=None,

use\_image\_num=0

):

"""

Forward pass of EnDora.

x: (N, F, C, H, W) tensor of video inputs

t: (N,) tensor of diffusion timesteps

y: (N,) tensor of class labels

y\_image: tensor of video frames

use\_image\_num: how many video frames are used

"""

if use\_fp16:

x = x.to(dtype=torch.float16)

elif attentions is not None and mode == "type\_cnn":

attentions = torch.concatenate(attentions, dim=0) # 480,256,384

# attentions.requires\_grad = True

attentions = attentions.permute(0, 2, 1)

attentions = rearrange(attentions, 'b h (m n) -> b h m n', n=16).contiguous() # 480,384,16,16

attentions = self.cov(attentions) # 480,1152, 8,8

attentions = rearrange(attentions, 'b h m n -> b h (m n)').contiguous() # 480,1152, 64

attentions = attentions.permute(0, 2, 1) # 480,64,1152

batches, frames, channels, high, weight = x.shape

x = rearrange(x, 'b f c h w -> (b f) c h w')

x = self.x\_embedder(x) + self.pos\_embed

t = self.t\_embedder(t, use\_fp16=use\_fp16)

timestep\_spatial = repeat(t, 'n d -> (n c) d', c=self.temp\_embed.shape[1] + use\_image\_num)

timestep\_temp = repeat(t, 'n d -> (n c) d', c=self.pos\_embed.shape[1])

if self.extras == 2:

y = self.y\_embedder(y, self.training)

if self.training:

y\_image\_emb = []

# print(y\_image)

for y\_image\_single in y\_image:

# print(y\_image\_single)

y\_image\_single = y\_image\_single.reshape(1, -1)

y\_image\_emb.append(self.y\_embedder(y\_image\_single, self.training))

y\_image\_emb = torch.cat(y\_image\_emb, dim=0)

y\_spatial = repeat(y, 'n d -> n c d', c=self.temp\_embed.shape[1])

y\_spatial = torch.cat([y\_spatial, y\_image\_emb], dim=1)

y\_spatial = rearrange(y\_spatial, 'n c d -> (n c) d')

else:

y\_spatial = repeat(y, 'n d -> (n c) d', c=self.temp\_embed.shape[1])

y\_temp = repeat(y, 'n d -> (n c) d', c=self.pos\_embed.shape[1])

elif self.extras == 78:

text\_embedding = self.text\_embedding\_projection(text\_embedding)

text\_embedding\_video = text\_embedding[:, :1, :]

text\_embedding\_image = text\_embedding[:, 1:, :]

text\_embedding\_video = repeat(text\_embedding, 'n t d -> n (t c) d', c=self.temp\_embed.shape[1])

text\_embedding\_spatial = torch.cat([text\_embedding\_video, text\_embedding\_image], dim=1)

text\_embedding\_spatial = rearrange(text\_embedding\_spatial, 'n t d -> (n t) d')

text\_embedding\_temp = repeat(text\_embedding\_video, 'n t d -> n (t c) d', c=self.pos\_embed.shape[1])

text\_embedding\_temp = rearrange(text\_embedding\_temp, 'n t d -> (n t) d')

output = []

for i in range(0, len(self.blocks), 2):

spatial\_block, temp\_block = self.blocks[i:i+2]

if self.extras == 2:

c = timestep\_spatial + y\_spatial

elif self.extras == 78:

c = timestep\_spatial + text\_embedding\_spatial

else:

c = timestep\_spatial

x = spatial\_block(x, c)

if i // 2 in special\_list:

output.append(x)

x = rearrange(x, '(b f) t d -> (b t) f d', b=batches)

x\_video = x[:, :(frames-use\_image\_num), :]

x\_image = x[:, (frames-use\_image\_num):, :]

# Add Time Embedding

if i == 0:

x\_video = x\_video + self.temp\_embed

if self.extras == 2:

c = timestep\_temp + y\_temp

elif self.extras == 78:

c = timestep\_temp + text\_embedding\_temp

else:

c = timestep\_temp

x\_video = temp\_block(x\_video, c)

x = torch.cat([x\_video, x\_image], dim=1)

x = rearrange(x, '(b t) f d -> (b f) t d', b=batches)

if self.extras == 2:

c = timestep\_spatial + y\_spatial

else:

c = timestep\_spatial

x = self.final\_layer(x, c)

x = self.unpatchify(x)

x = rearrange(x, '(b f) c h w -> b f c h w', b=batches)

# print(x.shape)

if attentions is not None:

features = torch.concatenate(output, dim=0) #480,64,1152

else:

features = attentions

return x, attentions, features

def forward\_with\_cfg(self, x, t, y, cfg\_scale, use\_fp16=False):

"""

Forward pass of EnDora, but also batches the unconditional forward pass for classifier-free guidance.

"""

# https://github.com/openai/glide-text2im/blob/main/notebooks/text2im.ipynb

half = x[: len(x) // 2]

combined = torch.cat([half, half], dim=0)

if use\_fp16:

combined = combined.to(dtype=torch.float16)

model\_out = self.forward(combined, t, y, use\_fp16=use\_fp16)

# For exact reproducibility reasons, we apply classifier-free guidance on only

# three channels by default. The standard approach to cfg applies it to all channels.

# This can be done by uncommenting the following line and commenting-out the line following that.

# eps, rest = model\_out[:, :self.in\_channels], model\_out[:, self.in\_channels:]

# eps, rest = model\_out[:, :3], model\_out[:, 3:]

eps, rest = model\_out[:, :, :4, ...], model\_out[:, :, 4:, ...] # 2 16 4 32 32

cond\_eps, uncond\_eps = torch.split(eps, len(eps) // 2, dim=0)

half\_eps = uncond\_eps + cfg\_scale \* (cond\_eps - uncond\_eps)

eps = torch.cat([half\_eps, half\_eps], dim=0)

return torch.cat([eps, rest], dim=2)