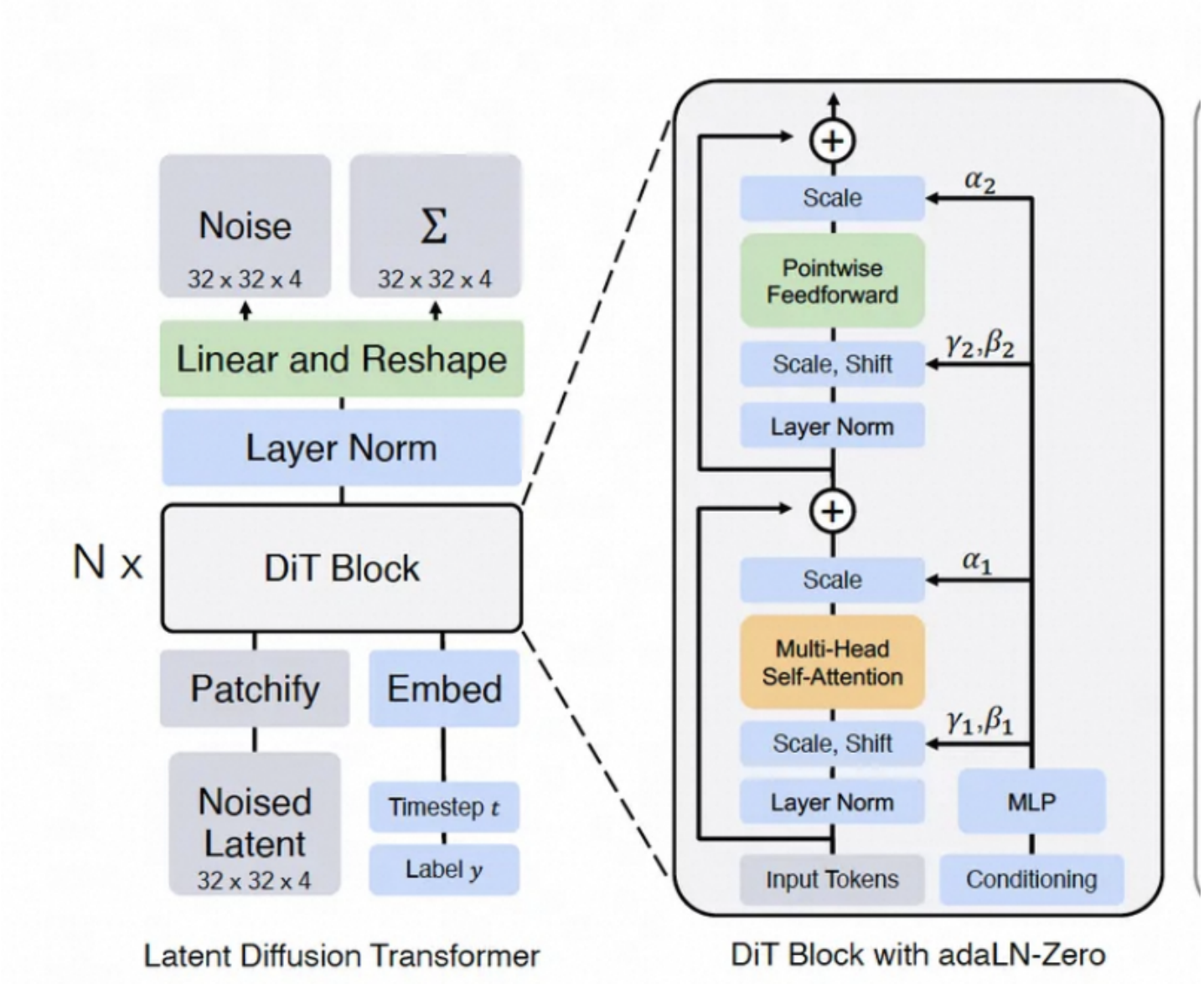
# DiT：从理论到实践，万字长文深入浅出带你学习 Diffusion Transformer



随着近期在视频生成层面的一些研究，对DiT的理解也比去年更加深刻一些，因此这边和大家一起分享下近期的学习进展，并把以前的部分内容做一些补充和优化。

当然纯属一家之言，也欢迎一起讨论交流。

---

# 1.DIT的核心优化思路：

**首先我们看下官方论文中的优势点：**

**1）性能提升**： DiT 在ImageNet基准测试中取得了最先进的FID（Fréchet Inception Distance）结果，特别是在256×256分辨率的基准测试中，实现了2.27的FID，这表明DiT能够生成高质量、高保真度的图像。

**2）可扩展性**： DiT展示了良好的可扩展性，即模型的计算复杂度（以GFLOPs衡量）与生成样本的质量（以FID衡量）之间存在强相关性。通过增加模型的GFLOPs，例如通过增加变换器的深度/宽度或输入tokens的数量，可以显著提高生成图像的质量。

**3）灵活性**： DiT的设计允许研究者通过调整模型的大小、补丁大小和序列长度来探索不同的设计空间，这为未来的研究和应用提供了灵活性。

**4）跨领域研究**： 由于DiT的架构与Vision Transformers（ViTs）相似，它为跨领域的研究提供了可能性，例如将图像生成技术应用于其他视觉任务。

## 下面回到个人的理解层面：

## Q1:如果我们用一句话说完DiT的架构优化，以下两点是必不可少的：

1）通过引入Transformer架构，替换Stable Diffusion及下游变体中最常用的U-Net的架构，而Transformer架构更擅长处理的，往往是时序相关的问题。因此引入Transformer架构带来的最大的好处，其实是解决了Unet模型架构对于时序生成的最大难点。

2）除了时序相关性的引入之外，第二个最大的好处，就是能并行接受更多的输入信息。这一点可以让生成模型的输入更加灵活。

## Q2:BUT，质量真的提升了吗?

从DiT论文问世之初，包括笔者在内的很多研究同学，都对他的提升是持观望态度的。乃至到目前为止，仍然有很多同学对于PixArt是否超过，或者说达到SDXL的生成质量，也是打问号的。

当然单纯从指标上来看，DiT系列的模型较UNet模型是有提升的。但是从主观生成质量上来说，其实差异有限。

所以个人理解，对于图像任务而言，不管是Unet还是DiT，在目前的训练量下，都能够轻松的达到高质量应用的下线。再往上的差异，对于落地应用其实没有明显优势，更多则是论文中的指标差异了。

## Q3:为什么目前图像生成应用，大多还是在SD-1.5和SDXL，DiT相关的占比很少?

本质上还是围绕SD的一系列算法生态，比如ControlNet，lora，甚至包括Cvitai等开源的基础模型，为下游的应用带来了源源不断的输入，然后下游应用同学又基于这些模型进行二创。

而DiT系列，相关的Lora/Controlnet非常少，乃至于没有。加上训练微调又十分困难，劝退了大量的应用算法同学。

# Q4:那DiT真正的优势是啥?

还是时序。

**之前是怎么做的：** 在DiT前，视频生成算法尝试过大量的架构，核心思路就是一个，引入时间维度T的信息。一种直观的解决方案是，直接在UNet上进行加一维，即变成大家喜闻乐见的3D-Unet。

**加补丁：** 3DUnet确实是有多一维的特征了，但是仍然有个问题没有解决，至少没有被很好的解决时序上的一致性问题。因此还需要加后补丁，比如在Cross Attention 上也加一路输入，包括和IP-Adapter一样，加一路时序的decoupled attention，或者cross attention 中的v用上一帧的v替换当前帧。

**再加补丁：** 但是又有新问题出现，仅仅考虑相邻两帧又不够了，又得考虑多帧的逻辑关联，因此这边又加进一步优化，像是多帧的一致性模块之类的。

而Transformer天然支持时序输入，而且可以并行处理多路输入，也就是说，在Unet中解决不了的问题（时序+多帧输入），在Transformer（DiT）这边是比较容易的问题。所以越来越多的视频生成框架，会选择DiT相关

# Q5:最后代价是什么?

1）**吐槽最多的就是有关于DiT的训练收敛困难问题：**

在推荐系统中，算法同学很自然的能获取到的高质量大规模的训练数据，因此能够让transformer充分发挥自己的潜力。而个体算法研究同学，大概率是没有训练大规模推荐系统算法的需求，因此这个问题天然就不存在。

而图像领域，虽然也能够构建百万级的训练数据，但是却是需要自己去构建的，不能从现有交易系统中直接获取，因此非常考验做大规模数据集的功底。

到视频层面，数据量更少，质量更差，难度就更大。

**2）训练资源和需求：**

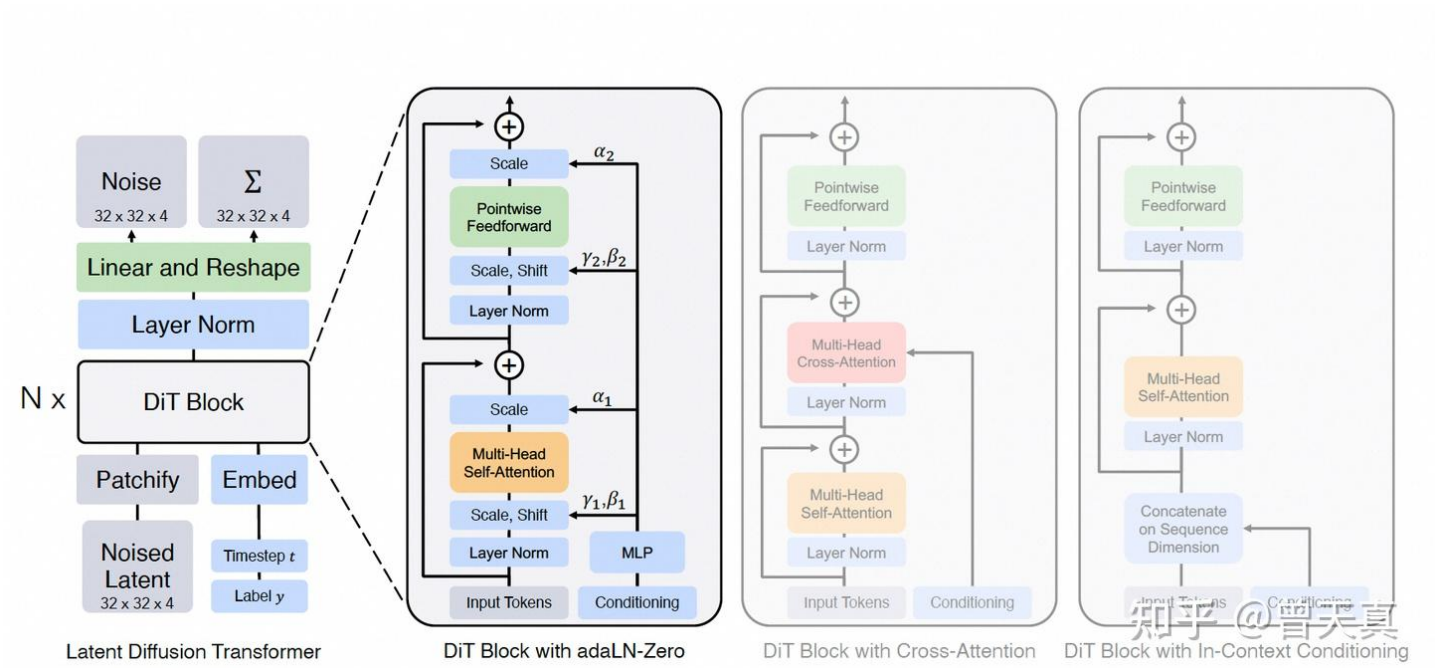
对于数据量，和GPU运算资源都有非常大的要求，因此我们在大多数论文中，会发现很大篇幅都在说明训练资源，时常，以及数据集获取及处理方式。

讲完了个人的理解部分，我们还是回到算法本身，结合论文代码来进行详细的解读。

# 2.DIT网络结构概述：

- DiT模型使用Transformer作为其主干网络，替代了传统的U-Net架构。
- 这些模型在Latent Space中训练，通过变换器处理潜在的图像块（patches）。
- **Hidden Dimension (d)**： 每个tokens在序列中都有一个隐藏维度d。这个维度是Transformer内部处理的向量大小。

# 3.DIT模块结构图解：



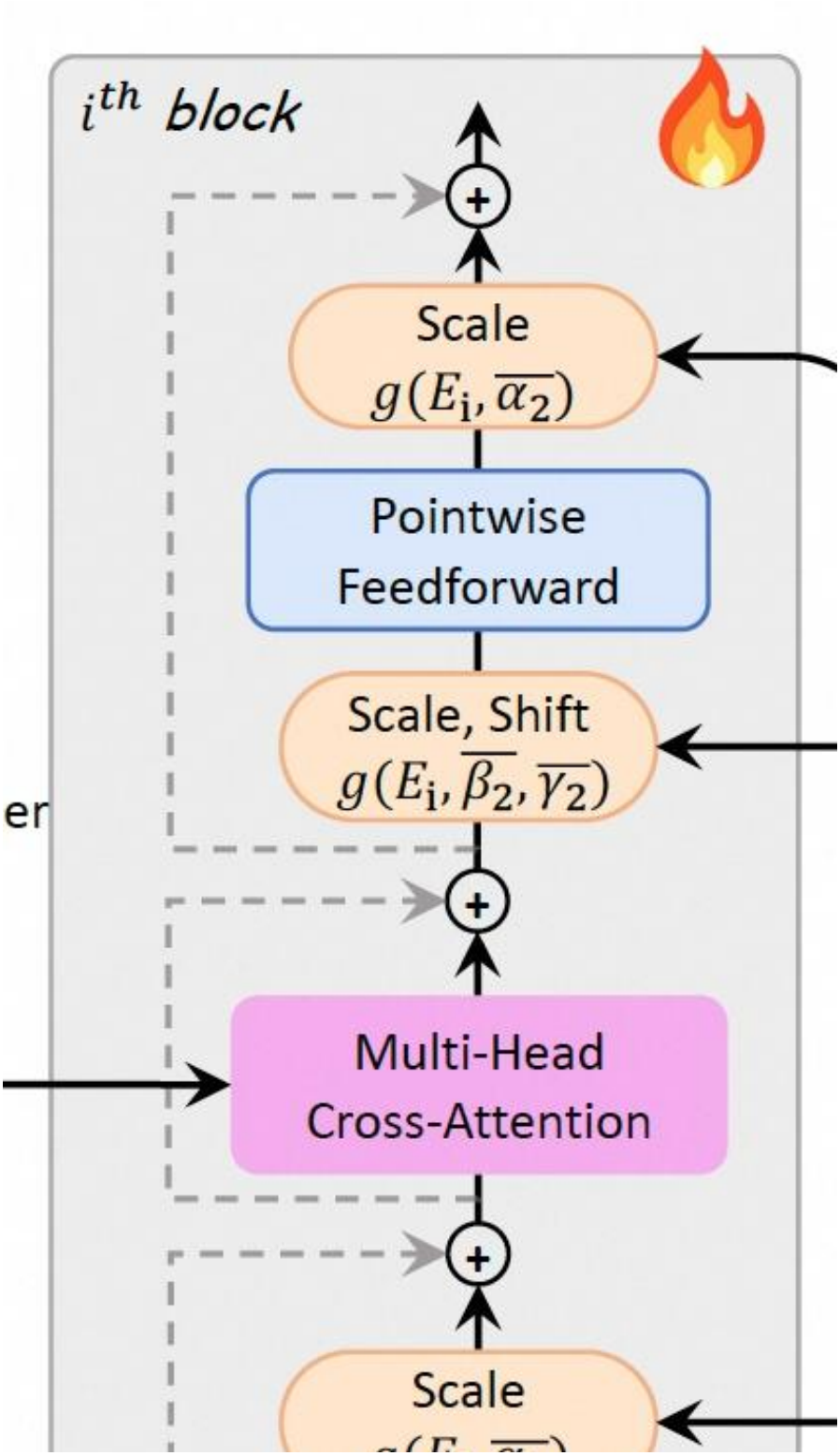
- DiT块是模型的核心，它处理输入的token序列。研究者们探索了四种不同的transformer块设计，以处理条件输入（如噪声时间步t、类别标签c等）。
- 设计包括：
- **上下文条件**： 将t和c的向量嵌入作为额外的令牌添加到输入序列中。
- **交叉注意力块**： 将t和c的嵌入连接成一个长度为2的序列，并在自注意力块之后添加一个额外的多头交叉注意力层。
- **自适应层归一化（adaLN）块**： 在变换器块中用自适应层归一化（adaLN）替换标准的层归一化。
- **adaLN-Zero块**： 在adaLN的基础上，对每个DiT块进行初始化，使其在初始时作为恒等函数。

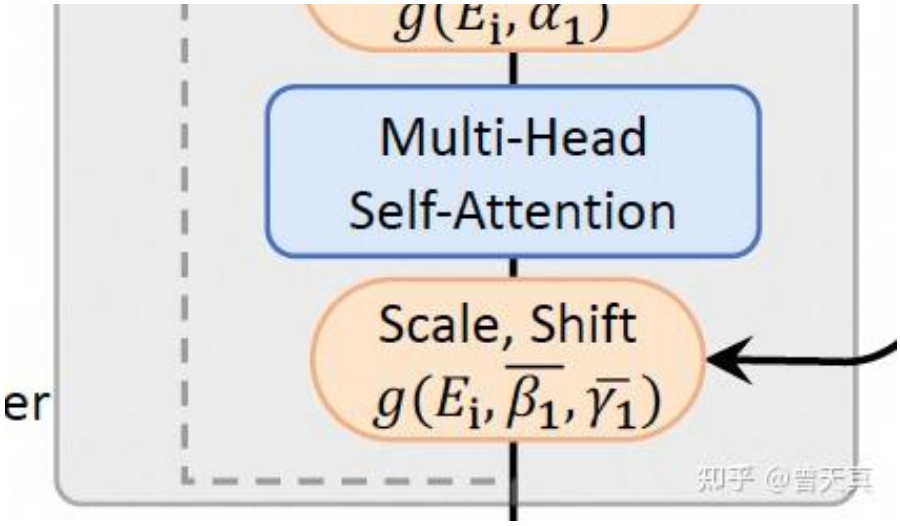
另外关于adaLN-Zero的模块选择上，其实还是有很多不同的适应性和选择问题。

# 一些AdaLN模块的细粒度优化方案：

在PixArt-a中，就有把adaLN-Zero进行优化的先例，一方面adaLN-Zero占据的参数量非常大（**27%**），而在应用中比较有限，因此为了降低参数量，优化成adaLN-single：

只在第一个块中使用时间维度的特征作为输入进行独立控制，并在所有块中共享






# AdaLN-single和adaLN的对比：

## AdaLN-single：

```
class PixArtBlock(nn.Module):
    """
    A PixArt block with adaptive layer norm
(adaLN-single) conditioning.
    """

    def __init__(self, hidden_size, num_heads,
mlp_ratio=4.0, drop_path=0., window_size=0,
input_size=None, use_rel_pos=False,
**block_kwargs):
        super().__init__()
        self.hidden_size = hidden_size
        self.norm1 = nn.LayerNorm(hidden_size,
elementwise_affine=False, eps=1e-6)
```

```
        self.attn = WindowAttention(hidden_size,
num_heads=num_heads, qkv_bias=True,

input_size=input_size if window_size == 0 else
(window_size, window_size),

use_rel_pos=use_rel_pos, **block_kwargs)
        self.cross_attn =
MultiHeadCrossAttention(hidden_size, num_heads,
**block_kwargs)
        self.norm2 = nn.LayerNorm(hidden_size,
elementwise_affine=False, eps=1e-6)
        # to be compatible with lower version
pytorch
        approx_gelu = lambda:
nn.GELU(approximate="tanh")
        self.mlp = Mlp(in_features=hidden_size,
hidden_features=int(hidden_size * mlp_ratio),
act_layer=approx_gelu, drop=0)
        self.drop_path = DropPath(drop_path) if
drop_path > 0. else nn.Identity()
        self.window_size = window_size
        self.scale_shift_table =
nn.Parameter(torch.randn(6, hidden_size) /
hidden_size ** 0.5)
```

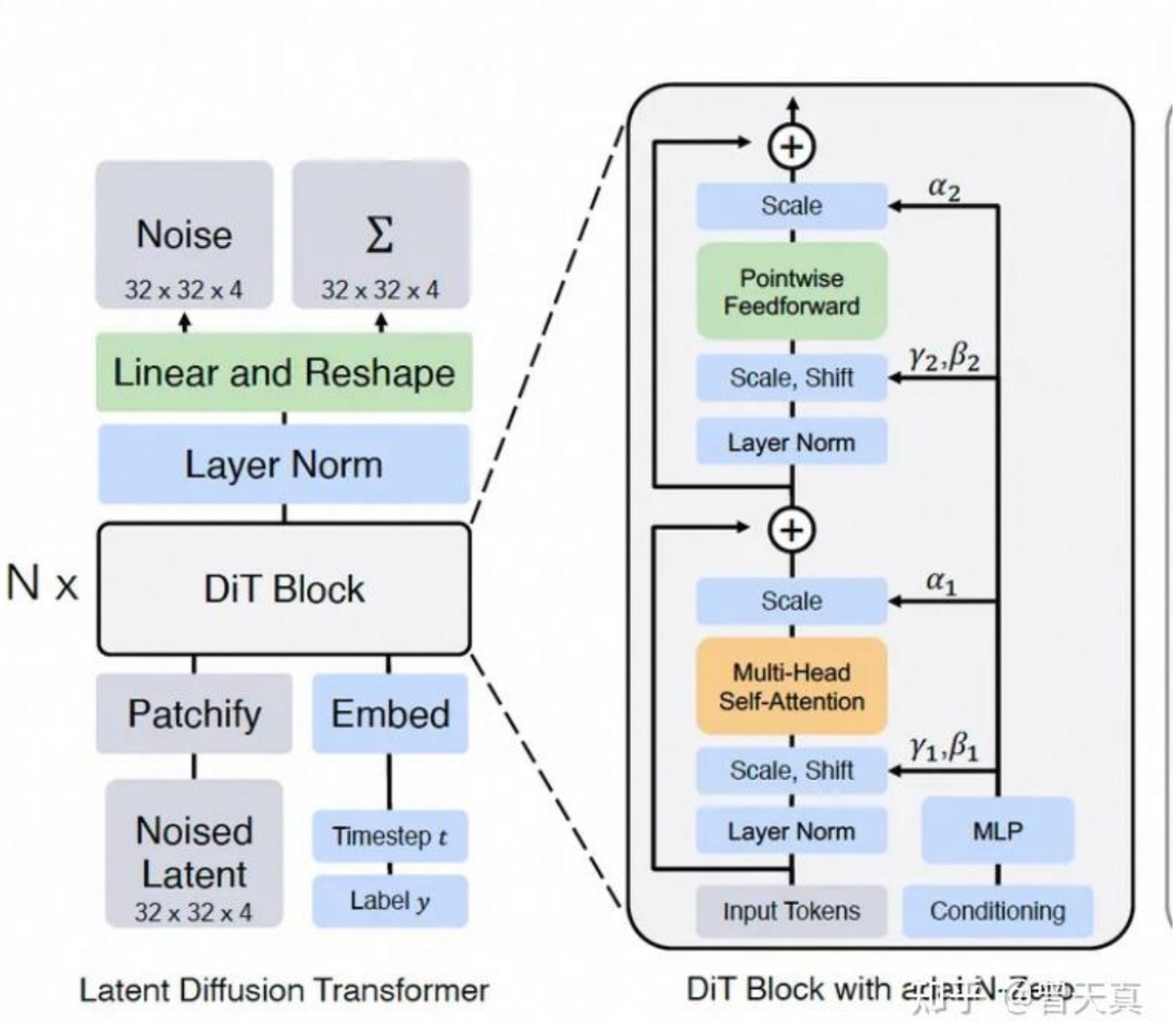
```
    def forward(self, x, y, t, mask=None,
**kwargs):
        B, N, C = x.shape

        shift_msa, scale_msa, gate_msa, shift_mlp,
scale_mlp, gate_mlp =
(self.scale_shift_table[None] + t.reshape(B, 6,
-1)).chunk(6, dim=1)
        x = x + self.drop_path(gate_msa *
self.attn(t2i_modulate(self.norm1(x), shift_msa,
scale_msa)).reshape(B, N, C))
        x = x + self.cross_attn(x, y, mask)
        x = x + self.drop_path(gate_mlp *
self.mlp(t2i_modulate(self.norm2(x), shift_mlp,
scale_mlp)))

        return x
```

**AdaLN-zero：**

```
# adaLN 模块
self.adaLN_modulation = nn.Sequential(
            nn.SiLU(),
            nn.Linear(hidden_size, 6 *
hidden_size, bias=True)
        )
```

# 3.1 DIT模块源码详解：

## 3.1.1 DIT模块：




# 回顾原文+开源代码部分一起学：

```
# DIT的核心子模块,DIT Block
class DiTBlock(nn.Module):
    """
```

```
    A DiT 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)
        # 此处为Multihead-Self-Attention

        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)
        #使用自适应归一化(adaLN)替换标准归一化层

        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
```

## 3.1.2 DIT完整forward：

```
def forward(self, x, t, y):

        x = self.x_embedder(x) + self.pos_embed  #
(N, T, D), where T = H * W / patch_size ** 2
        t = self.t_embedder(t)                   #
(N, D)
        # time step embedding
```

```
        y = self.y_embedder(y, self.training)    #
(N, D)
        c = t + y                                #
(N, D)
        # 送入上述的DIT-Block中
        for block in self.blocks:
            x = block(x, c)                      #
(N, T, D)
        x = self.final_layer(x, c)
# (N, T, patch_size ** 2 * out_channels)

        #编码后仍然需要再进行一次解码，即逆patch化
        x = self.unpatchify(x)                   #
(N, out_channels, H, W)
        return x
```

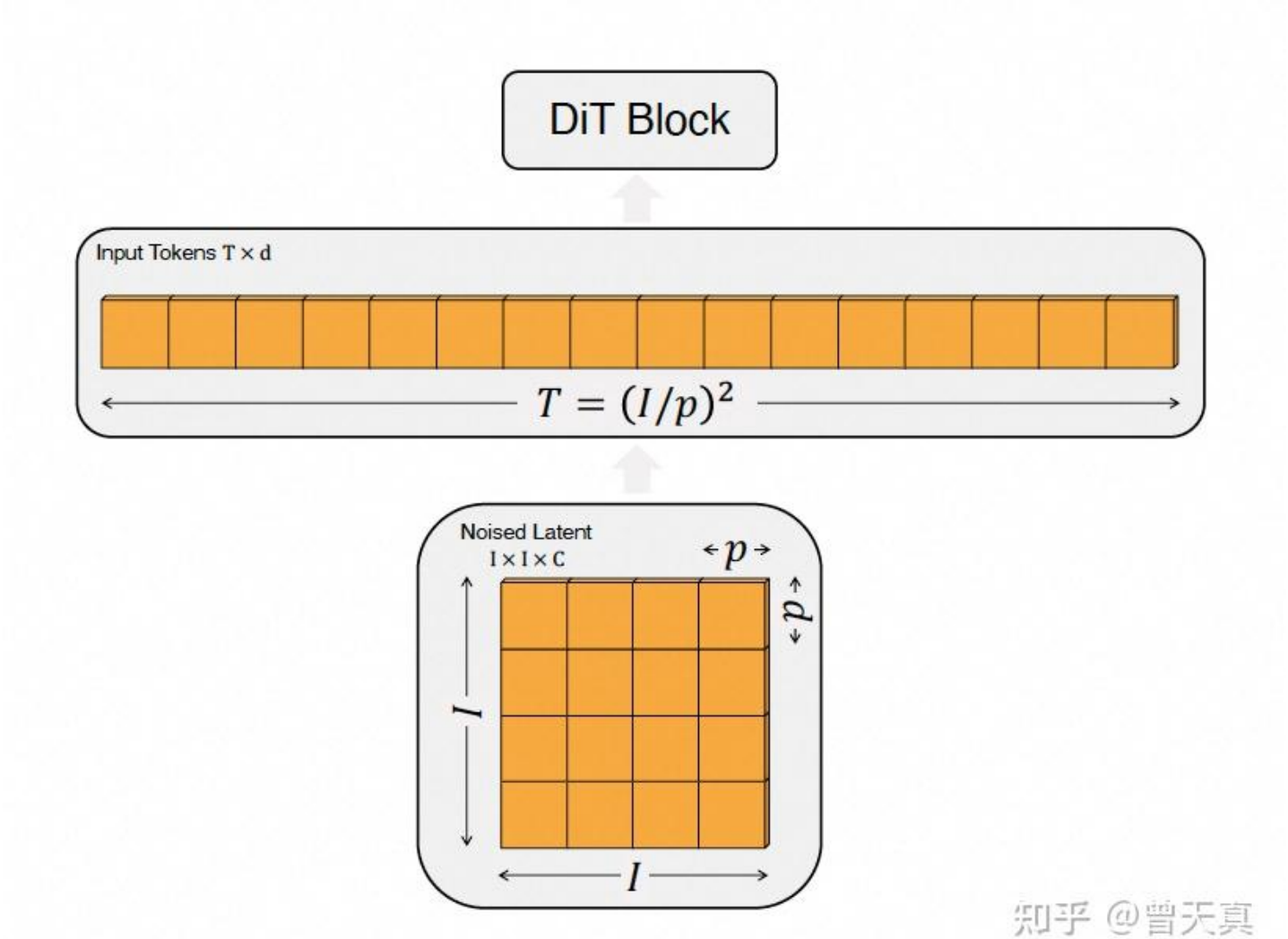
## 3.1.3 带CFG的forward：

```
def forward_with_cfg(self, x, t, y, cfg_scale):

        half = x[: len(x) // 2]
        combined = torch.cat([half, half], dim=0)
        model_out = self.forward(combined, t, y)
        #确保精确的可重复性，默认只在三个通道上应用无分类器
引导
        eps, rest = model_out[:, :3], model_out[:,
3:]
        cond_eps, uncond_eps = torch.split(eps,
len(eps) // 2, dim=0)
        #条件控制权重参数
        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=1)
```

# 4. DiT的输入参数规格：

针对DIT的网络输入层而言，DIT也设计了很多提升质量小细节，总结来说：

- DiT模型的输入规格涉及到将图像的噪声潜在表示分割成小块，并将这些小块转换成一个长序列，以便Transformer可以处理。
- 这个过程的计算复杂度随着patch大小的减小而增加，因为需要处理更多的tokens。
- 这种设计允许模型在潜在空间中有效地处理图像数据，并在生成过程中利用Transformer的强大能力。



# 4.1 时序embedding 模块构建

整体编码方式参考了openai的glide编码模式，和StableDiffusion是有一定的差异。

Timestep_embedding 后接一层MLP。

```
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):

        # 整体参考OpenAI 的glide实现
        # 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):
        t_freq = self.timestep_embedding(t,
self.frequency_embedding_size)
        # 注意timestep后接上mlp层
        t_emb = self.mlp(t_freq)
        return t_emb
```

# 4.2 Label Embedding 构建

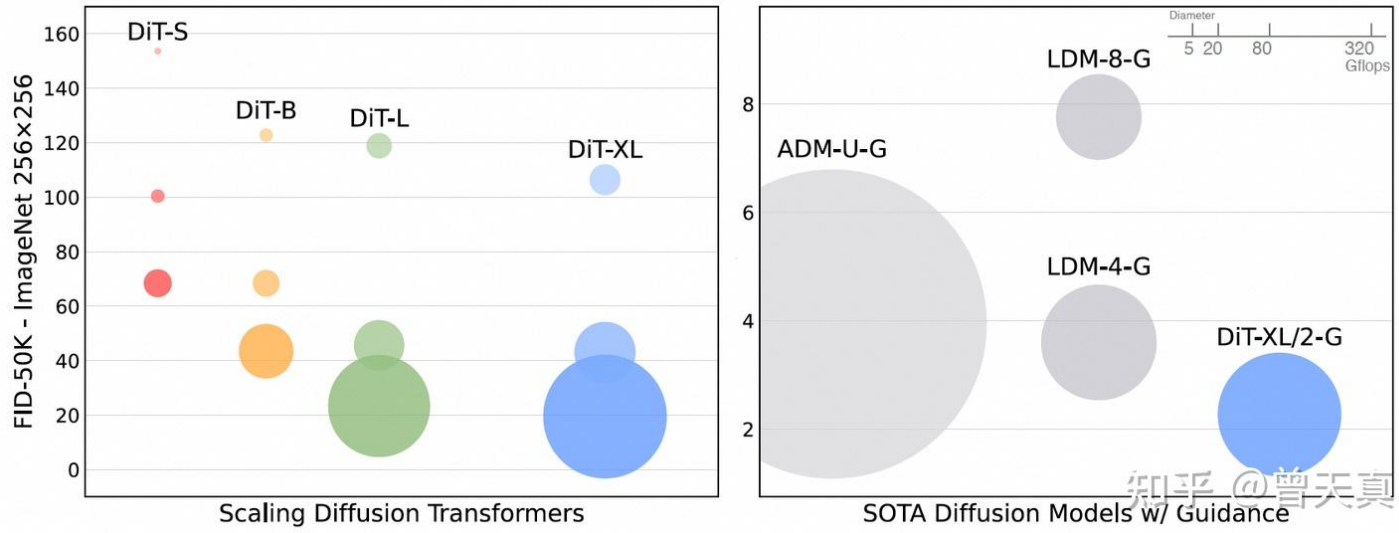
LabelEmbedder 在原文中提到为了能够高效使用 Classifier-Free Guidance而引入的Dropout层，具体的实现就在这个模块中。

```
class LabelEmbedder(nn.Module):
    # Label 编码
    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):
        # 加入的Dropout层用以实现Classifier-Free
Guidance
        if force_drop_ids is None:
            drop_ids = torch.rand(labels.shape[0],
device=labels.device) < self.dropout_prob
```

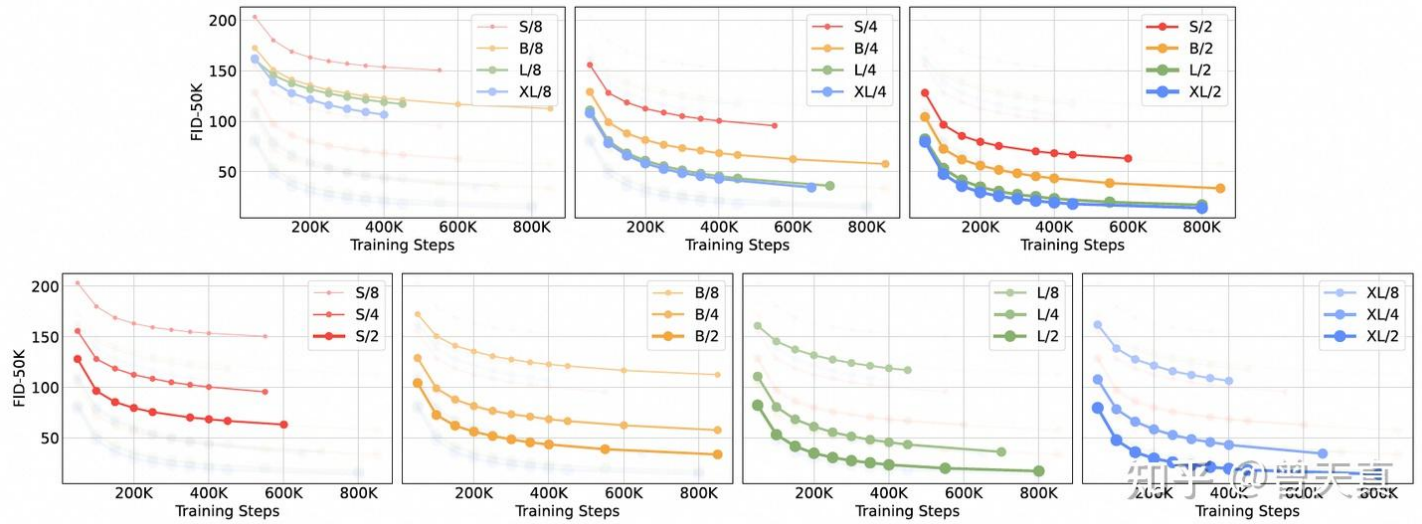
```
        else:
            drop_ids = force_drop_ids == 1
        labels = torch.where(drop_ids,
self.num_classes, labels)
        return labels

    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
```

# 5.DIT与UNet效果对比：

- 传统的U-Net架构在扩散模型中被广泛使用，但DiT-XL/2在性能上超越了这些模型，展示了DiT架构的优势。




# 6.DIT模型全阶段优化：



- 扩展DiT模型在训练的所有阶段都能改善FID。我们展示了12个DiT模型在训练迭代过程中的FID-50K。
- 顶部行：我们在保持补丁大小不变的情况下比较FID。
- 底部行：在保持模型大小不变的情况下比较FID。
  Transformer Backbone在所有模型大小和补丁大小下都能产生更好的生成模型。

# 7.Patch化操作：

- 通过“patchify”过程，空间表示被转换成一个token序列。
- 序列的长度T由输入图像的空间尺寸除以补丁（Patch）大小的平方决定，即T = (I/p)^2：
- 例如，如果输入图像是256×256像素，补丁（Patch）大小是4×4像素，那么序列长度T将是(256/4)^2 = 64^2 = 4096。

```
from timm.models.vision_transformer import
PatchEmbed, Attention, Mlp
        ...
        # patch化操作使用了
timm.models.vision_transformer 中自带的编码

        self.x_embedder = PatchEmbed(input_size,
patch_size, in_channels, hidden_size, bias=True)
        self.t_embedder =
TimestepEmbedder(hidden_size)
        self.y_embedder =
LabelEmbedder(num_classes, hidden_size,
class_dropout_prob)
        num_patches = self.x_embedder.num_patches
        # pose embedding 使用了sin-cos编码方式
        self.pos_embed =
nn.Parameter(torch.zeros(1, num_patches,
hidden_size), requires_grad=False)
```

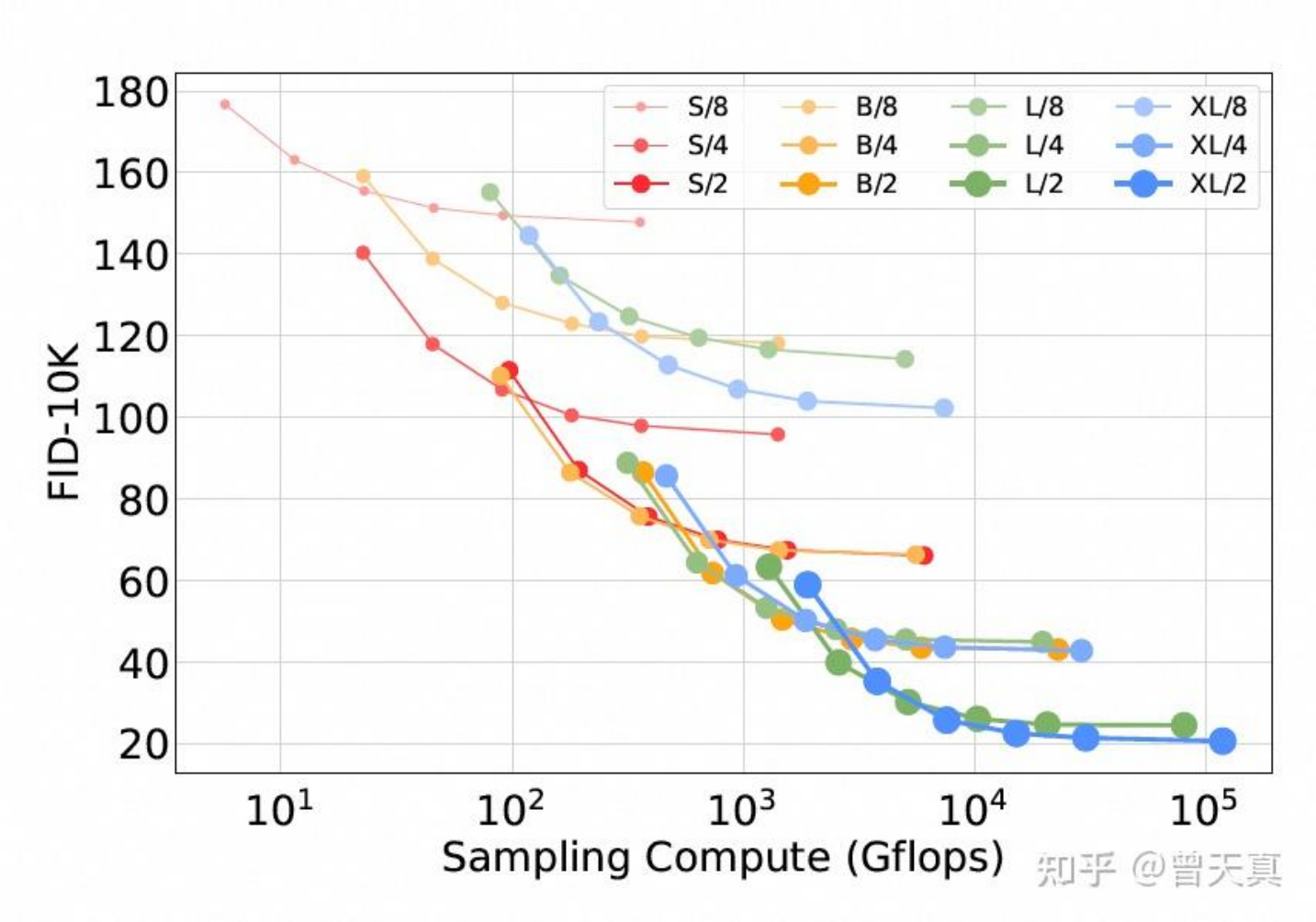
# 7.1 逆Patch化操作：

```
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
```

# 7.2 可扩展性分析：

研究者们分析了DiT模型的可扩展性，即模型复杂度（以GFLOPs衡量）与样本质量（以FID衡量）之间的关系。他们发现，具有更高GFLOPs的DiT模型（通过增加Transformer的深度/宽度或输入Tokens的数量）通常具有更低的FID，表现出更好的性能。




# 【-END-】

对AIGC相关应用，算法前沿以及创业/工作感兴趣的同学，可以加微信：**Zeng_AIGC** ， ***备注：研究方向+学校/公司 + 知乎*** 即可加入交流群。

欢迎大家与创业团队，大厂leader以及顶尖名校的算法研究同学共同交流。

**另外想入AIGC算法方向，但是缺少AIGC质量项目，或者有疑惑的同学，也欢迎咨询：Zeng_AIGC，*备注：AIGC+咨询***

*对于AIGC，扩散模型相关应用，算法，实践感兴趣的欢迎关注我～*