# GETRegionDiffusion: Code Implementation Walkthrough

Mathematical Foundations and PyTorch Implementation

Technical Documentation

December 1, 2025

## Contents



## 1 Overview

This document provides a comprehensive walkthrough of the `GETRegionDiffusion` model implementation, explaining the mathematical foundations behind each component and how they map to PyTorch code.

### 1.1 Data Flow

The complete forward pass follows this pipeline:

$$\underbrace{\mathbf{X}}_{\text{Input}} \xrightarrow{\text{Embed}} \underbrace{\mathbf{H}}_{\text{Hidden}} \xrightarrow{\text{Mask}} \underbrace{\tilde{\mathbf{H}}}_{\text{Masked}} \xrightarrow{\text{DiT}} \underbrace{\mathbf{Z}}_{\text{Encoded}} \xrightarrow{\text{Head}} \underbrace{\hat{\mathbf{X}}}_{\text{Output}}$$

Where:

- $\mathbf{X} \in \mathbb{R}^{B \times N \times M}$: Input motif features ($B$=batch, $N$=900 regions, $M$=283 motifs)
- $\mathbf{H} \in \mathbb{R}^{B \times N \times D}$: Embedded features ($D$=768)
- $\tilde{\mathbf{H}}$: Features with masked positions replaced
- $\mathbf{Z}$: Transformer-encoded features
- $\hat{\mathbf{X}}$: Predicted motif features

## 2 Step 1: Region Embedding

**Step 1: RegionEmbed — Linear Projection**

**Purpose:** Project raw motif features from input dimension to model dimension.

**Mathematical Formulation:**

$$\mathbf{H} = \mathbf{X}\mathbf{W}_{\text{embed}} + \mathbf{b}_{\text{embed}}$$

where:

- $\mathbf{X} \in \mathbb{R}^{B \times N \times 283}$ — input motif features
- $\mathbf{W}_{\text{embed}} \in \mathbb{R}^{283 \times 768}$ — learnable weight matrix
- $\mathbf{b}_{\text{embed}} \in \mathbb{R}^{768}$ — learnable bias
- $\mathbf{H} \in \mathbb{R}^{B \times N \times 768}$ — embedded output

**PyTorch Implementation:**

```
class RegionEmbed(BaseModule):
    def __init__(self, cfg: RegionEmbedConfig):
        super().__init__(cfg)
        self.embed = nn.Linear(cfg.num_features, cfg.embed_dim)
        # Linear(283, 768)

    def forward(self, x, **kwargs):
        x = self.embed(x)  # (B, 900, 283) -> (B, 900, 768)
        return x
```

## 3 Step 2: Masking Operation

**Step 2: Masking + Token Replacement**

**Purpose:** Replace masked positions with a learnable mask token while preserving unmasked positions.

**Mathematical Formulation:**
For each position $i$ in the sequence:

$$\tilde{\mathbf{h}}_i = \begin{cases} \mathbf{h}_i & \text{if } m_i = 0 \text{ (unmasked)} \\ \mathbf{e}_{\text{mask}} & \text{if } m_i = 1 \text{ (masked)} \end{cases}$$

Vectorized form using element-wise operations:

$$\tilde{\mathbf{H}} = \mathbf{H} \odot (1 - \mathbf{M}) + \mathbf{E}_{\text{mask}} \odot \mathbf{M}$$

where:

- $\mathbf{M} \in \{0, 1\}^{B \times N \times 1}$ — binary mask (broadcast to hidden dim)
- $\mathbf{E}_{\text{mask}} \in \mathbb{R}^{B \times N \times D}$ — mask token expanded to all positions
- $\odot$ — element-wise (Hadamard) product

**PyTorch Implementation:**

```
# In forward():
# mask_token is a learnable parameter: (1, 1, 768)
mask_token = self.mask_token.expand(B, N, -1)  # -> (B, 900, 768)

# Convert boolean mask to float for multiplication
w = mask.type_as(mask_token)  # True->1.0, False->0.0

# keep original where mask=0, use mask_token where mask=1
x = x * (1 - w) + mask_token * w
```

**Token Initialization:**

```
# Learnable tokens initialized with truncated normal distribution
self.mask_token = nn.Parameter(torch.zeros(1, 1, 768))
self.cls_token = nn.Parameter(torch.zeros(1, 1, 768))
trunc_normal_(self.mask_token, std=0.02)
trunc_normal_(self.cls_token, std=0.02)
```

# 4 Step 3: CLS Token Prepending

## Step 3: Add CLS Token

**Purpose:** Prepend a learnable classification token to the sequence.

**Mathematical Formulation:**

$$\mathbf{H}' = \text{concat}([\mathbf{e}_{\text{cls}}, \tilde{\mathbf{H}}], \text{dim} = 1)$$

where:

- $\mathbf{e}_{\text{cls}} \in \mathbb{R}^{B \times 1 \times D}$ — CLS token (expanded to batch)
- $\tilde{\mathbf{H}} \in \mathbb{R}^{B \times N \times D}$ — masked hidden states
- $\mathbf{H}' \in \mathbb{R}^{B \times (N+1) \times D}$ — output with CLS token

Sequence length changes: $900 \rightarrow 901$

**PyTorch Implementation:**

```
# Expand CLS token to batch size
cls_tokens = self.cls_token.expand(B, -1, -1)  # (1,1,768) -> (B,1,768)

# Concatenate along sequence dimension
x = torch.cat((cls_tokens, x), dim=1)  # (B, 901, 768)
```

## 5 Step 4: Timestep Sampling

### Step 4: Random Timestep Sampling

**Purpose:** Sample a random diffusion timestep for each batch element.

**Mathematical Formulation:**

$$t \sim \text{Uniform}\{0, 1, 2, \ldots, T-1\}$$

where $T = 1000$ (number of diffusion timesteps).
Each sample in the batch gets an independent random timestep:

$$\mathbf{t} = [t_1, t_2, \ldots, t_B] \in \{0, \ldots, 999\}^B$$

**PyTorch Implementation:**

```
# Sample random timesteps for each batch element
t = torch.randint(0, self.num_timesteps, (B,), device=device).long()
# t.shape = (B,), e.g., tensor([342, 891, 127, ...])
```

## 6 Step 5: Timestep Embedding

### Step 5: TimestepEmbedder — Sinusoidal + MLP

**Purpose:** Convert scalar timesteps into rich vector representations for conditioning.

**Mathematical Formulation:**
**Stage 1: Sinusoidal Embedding** (same as Transformer positional encoding)
For timestep $t$ and dimension $d = 256$:

$$\text{PE}(t, 2i) = \sin\left(\frac{t}{\tau^{2i/d}}\right), \quad \text{PE}(t, 2i+1) = \cos\left(\frac{t}{\tau^{2i/d}}\right)$$

where $\tau = 10000$ (max period) and $i \in \{0, 1, \ldots, d/2 - 1\}$.
Equivalently:

$$\omega_i = \exp\left(-\frac{\log(\tau) \cdot i}{d/2}\right), \quad \mathbf{e}_{\text{freq}}(t) = [\cos(t \cdot \boldsymbol{\omega}), \sin(t \cdot \boldsymbol{\omega})]$$

**Stage 2: MLP Projection**

$$\mathbf{c} = \mathbf{W}_2 \cdot \text{SiLU}(\mathbf{W}_1 \cdot \mathbf{e}_{\text{freq}}(t) + \mathbf{b}_1) + \mathbf{b}_2$$

where:

- $\mathbf{W}_1 \in \mathbb{R}^{768 \times 256}$, $\mathbf{b}_1 \in \mathbb{R}^{768}$
- $\mathbf{W}_2 \in \mathbb{R}^{768 \times 768}$, $\mathbf{b}_2 \in \mathbb{R}^{768}$
- $\text{SiLU}(x) = x \cdot \sigma(x)$ where $\sigma$ is sigmoid
- $\mathbf{c} \in \mathbb{R}^{B \times 768}$ — conditioning vector

**PyTorch Implementation:**

```
class TimestepEmbedder(nn.Module):
    def __init__(self, hidden_size, freq_embed_size=256):
        super().__init__()
        self.mlp = nn.Sequential(
            nn.Linear(freq_embed_size, hidden_size),  # 256->768
            nn.SiLU(),
            nn.Linear(hidden_size, hidden_size),      # 768->768
        )
        self.frequency_embedding_size = freq_embed_size

    @staticmethod
    def timestep_embedding(t, dim, max_period=10000):
        half = dim // 2  # 128
        # Frequencies: exp(-log(10000) * [0,...,127] / 128)
        freqs = torch.exp(
            -math.log(max_period) *
            torch.arange(0, half, dtype=torch.float32) / half
        ).to(device=t.device)

        # Outer product: t[:, None] * freqs[None]
        args = t[:, None].float() * freqs[None]

        # Concatenate cos and sin -> (B, 256)
        embedding = torch.cat(
            [torch.cos(args), torch.sin(args)], dim=-1)
        return embedding

    def forward(self, t):
        t_freq = self.timestep_embedding(
            t, self.frequency_embedding_size)  # (B, 256)
        t_emb = self.mlp(t_freq)  # (B, 768)
        return t_emb
```

# 7 Step 6: Adaptive Layer Normalization (adaLN)

**Step 6: The `modulate` Function — Core of adaLN**

**Purpose:** Inject timestep conditioning into the transformer by modulating normalized activations.

**Standard LayerNorm:**

$$\text{LN}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \mu}{\sigma} + \beta$$

where $\gamma, \beta$ are **fixed** learnable parameters.

**Adaptive LayerNorm (adaLN):**

$$\text{adaLN}(\mathbf{x}, \mathbf{c}) = (1 + \text{scale}) \odot \frac{\mathbf{x} - \mu}{\sigma} + \text{shift}$$

where scale, shift are **predicted from conditioning c**.

**The modulate function:**

$$\text{modulate}(\mathbf{x}, \text{shift}, \text{scale}) = \mathbf{x} \odot (1 + \text{scale}) + \text{shift}$$

Key insight: $(1 + \text{scale})$ ensures identity behavior when scale $= 0$.

**PyTorch Implementation:**

```
def modulate(x, shift, scale):
    """
    x: (B, L, D) - input tensor
    shift, scale: (B, D) - modulation parameters from timestep
    Returns: (B, L, D) - modulated tensor
    """
    # unsqueeze(1) broadcasts (B, D) -> (B, 1, D) for sequence dimension
    return x * (1 + scale.unsqueeze(1)) + shift.unsqueeze(1)
```

**Why this works:**

- At initialization, adaLN outputs are zero $\Rightarrow$ scale=0, shift=0
- $\text{modulate}(\mathbf{x}, 0, 0) = \mathbf{x} \cdot 1 + 0 = \mathbf{x}$ (identity)
- Model gradually learns conditioning through training

# 8 Step 7: DiTBlock — Transformer Block with adaLN

**Step 7: DiTBlock — The Core Transformer Block**

**Purpose:** Process sequence with self-attention and feed-forward, conditioned on timestep.

**Mathematical Formulation:**

**Stage 1: Compute 6 modulation parameters from conditioning**

$$[\gamma_1, \beta_1, \alpha_1, \gamma_2, \beta_2, \alpha_2] = \text{MLP}_{\text{adaLN}}(\mathbf{c})$$

where $\text{MLP}_{\text{adaLN}}$: $\mathbb{R}^{768} \to \mathbb{R}^{4608}$ $(6 \times 768)$.

**Stage 2: Self-Attention with gated residual**

$$\mathbf{x}' = \mathbf{x} + \alpha_1 \odot \text{Attention}\left(\text{modulate}(\text{LN}(\mathbf{x}), \beta_1, \gamma_1)\right)$$

**Stage 3: Feed-Forward with gated residual**

$$\mathbf{x}'' = \mathbf{x}' + \alpha_2 \odot \text{MLP}\left(\text{modulate}(\text{LN}(\mathbf{x}'), \beta_2, \gamma_2)\right)$$

Where:

- $\gamma_i$ = scale parameters
- $\beta_i$ = shift parameters
- $\alpha_i$ = gate parameters (control residual contribution)
- LN uses `elementwise_affine=False` (no $\gamma, \beta$)

**PyTorch Implementation:**

```
class DiTBlock(nn.Module):
    def __init__(self, hidden_size, num_heads, mlp_ratio=4.0):
        super().__init__()
        # LayerNorm WITHOUT learnable gamma/beta
        self.norm1 = nn.LayerNorm(hidden_size,
                                  elementwise_affine=False, eps=1e-6)
        self.attn = Attention(hidden_size, num_heads=num_heads,
                              qkv_bias=True)
        self.norm2 = nn.LayerNorm(hidden_size,
                                  elementwise_affine=False, eps=1e-6)

        mlp_hidden_dim = int(hidden_size * mlp_ratio)  # 3072
        self.mlp = Mlp(in_features=hidden_size,
                       hidden_features=mlp_hidden_dim)

        # Predict 6 modulation parameters from conditioning
        self.adaLN_modulation = nn.Sequential(
            nn.SiLU(),
            nn.Linear(hidden_size, 6 * hidden_size)  # 768->4608
        )

    def forward(self, x, c):
        # c: conditioning vector, shape (B, 768)

        # Get 6 modulation vectors, each (B, 768)
        modulation = self.adaLN_modulation(c).chunk(6, dim=1)
        shift_msa, scale_msa, gate_msa = modulation[:3]
        shift_mlp, scale_mlp, gate_mlp = modulation[3:]

        # Self-Attention block with gated residual
        normed = modulate(self.norm1(x), shift_msa, scale_msa)
        x = x + gate_msa.unsqueeze(1) * self.attn(normed)

        # Feed-Forward block with gated residual
        normed = modulate(self.norm2(x), shift_mlp, scale_mlp)
        x = x + gate_mlp.unsqueeze(1) * self.mlp(normed)
        return x
```

# 9 Step 8: Self-Attention Mechanism

## Step 8: Multi-Head Self-Attention

**Purpose:** Allow each position to attend to all other positions in the sequence.

**Mathematical Formulation:**
**Stage 1: Linear projections to Q, K, V**

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q, \quad \mathbf{K} = \mathbf{X}\mathbf{W}_K, \quad \mathbf{V} = \mathbf{X}\mathbf{W}_V$$

In practice, computed as single fused projection:

$$[\mathbf{Q}, \mathbf{K}, \mathbf{V}] = \mathbf{X}\mathbf{W}_{QKV}$$

where $\mathbf{W}_{QKV} \in \mathbb{R}^{768\times2304}$.
**Stage 2: Split into heads**

$$\mathbf{Q}_h, \mathbf{K}_h, \mathbf{V}_h \in \mathbb{R}^{B\times h\times N\times d_k}$$

where $h = 12$ heads, $d_k = 768/12 = 64$ per head.
**Stage 3: Scaled dot-product attention**

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

Scale factor: $\sqrt{d_k} = \sqrt{64} = 8$.
**Stage 4: Concatenate heads and project**

$$\text{MultiHead}(\mathbf{X}) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\mathbf{W}_O$$

where $\mathbf{W}_O \in \mathbb{R}^{768\times768}$.

**PyTorch Implementation:**

```
class Attention(nn.Module):
    def __init__(self, dim, num_heads=8, qkv_bias=False):
        super().__init__()
        self.num_heads = num_heads
        head_dim = dim // num_heads  # 768 // 12 = 64
        self.scale = head_dim ** -0.5  # 1/sqrt(64) = 0.125

        self.qkv = nn.Linear(dim, dim * 3, bias=qkv_bias)  # 768 -> 2304
        self.proj = nn.Linear(dim, dim)  # 768 -> 768

    def forward(self, x):
        B, N, C = x.shape  # (B, 901, 768)

        # Compute Q, K, V in one projection
        qkv = self.qkv(x)  # (B, 901, 2304)
        qkv = qkv.reshape(B, N, 3, self.num_heads,
                          C // self.num_heads)
        # Shape: (B, 901, 3, 12, 64)
        qkv = qkv.permute(2, 0, 3, 1, 4)
        # Shape: (3, B, 12, 901, 64)
        q, k, v = qkv[0], qkv[1], qkv[2]

        # Scaled dot-product attention
        attn = (q @ k.transpose(-2, -1)) * self.scale
        attn = attn.softmax(dim=-1)
```

```
        # Apply attention to values
        x = (attn @ v)  # (B, 12, 901, 64)
        x = x.transpose(1, 2).reshape(B, N, C)

        # Output projection
        x = self.proj(x)
        return x
```

# 10 Step 9: Feed-Forward Network (MLP)

## Step 9: Two-Layer MLP with GELU Activation

**Purpose:** Add non-linear transformations to each position independently.

**Mathematical Formulation:**

$$\text{MLP}(\mathbf{x}) = \mathbf{W}_2 \cdot \text{GELU}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$$

where:

- $\mathbf{W}_1 \in \mathbb{R}^{3072\times768}$ — expansion layer
- $\mathbf{W}_2 \in \mathbb{R}^{768\times3072}$ — compression layer
- Expansion ratio $= 4$ $(768 \to 3072 \to 768)$

**GELU activation:**

$$\text{GELU}(x) = x \cdot \Phi(x) \approx 0.5x\left(1 + \tanh\left[\sqrt{\frac{2}{\pi}}(x + 0.044715x^3)\right]\right)$$

where $\Phi(x)$ is the standard Gaussian CDF.

**PyTorch Implementation:**

```
class Mlp(nn.Module):
    def __init__(self, in_features, hidden_features=None,
                 out_features=None, act_layer=nn.GELU, drop=0.0):
        super().__init__()
        out_features = out_features or in_features
        hidden_features = hidden_features or in_features

        self.fc1 = nn.Linear(in_features, hidden_features)
        self.act = act_layer()  # GELU
        self.fc2 = nn.Linear(hidden_features, out_features)
        self.drop = nn.Dropout(drop)

    def forward(self, x):
        x = self.fc1(x)   # 768 -> 3072
        x = self.act(x)   # GELU activation
        x = self.fc2(x)   # 3072 -> 768
        x = self.drop(x)
        return x
```

## 11 Step 10: DiT Encoder Assembly

## Step 10: GETRegionDiTEncoder — Stacking 12 Blocks

**Purpose:** Combine TimestepEmbedder and 12 DiTBlocks into a complete encoder.

**Mathematical Formulation:**
**Complete forward pass:**

$$\mathbf{c} = \text{TimestepEmbedder}(t) \tag{1}$$

$$\mathbf{h}^{(0)} = \mathbf{H}' \quad \text{(input with CLS token)} \tag{2}$$

$$\mathbf{h}^{(\ell)} = \text{DiTBlock}^{(\ell)}(\mathbf{h}^{(\ell-1)}, \mathbf{c}) \quad \text{for } \ell = 1, \ldots, 12 \tag{3}$$

$$\mathbf{Z} = \text{LayerNorm}(\mathbf{h}^{(12)}) \tag{4}$$

Note: The same conditioning vector $\mathbf{c}$ is used for all 12 blocks.

### PyTorch Implementation:

```
class GETRegionDiTEncoder(nn.Module):
    def __init__(self, embed_dim=768, depth=12, num_heads=12, mlp_ratio=4.0):
        super().__init__()

        # Timestep embedding
        self.t_embedder = TimestepEmbedder(embed_dim)  # 768

        # Stack of 12 DiT blocks
        self.blocks = nn.ModuleList([
            DiTBlock(embed_dim, num_heads, mlp_ratio=mlp_ratio)
            for _ in range(depth)  # 12 blocks
        ])

        # Final LayerNorm (with affine params for output)
        self.norm = nn.LayerNorm(embed_dim, eps=1e-6)

    def forward(self, x, t):
        """
        x: (B, 901, 768) - embedded regions with CLS
        t: (B,) - timestep indices
        """
        # Convert timestep to conditioning vector
        c = self.t_embedder(t)  # (B, 768)

        # Pass through all DiT blocks
        for blk in self.blocks:
            x = blk(x, c)  # Same c for all blocks

        # Final normalization
        x = self.norm(x)  # (B, 901, 768)
        return x
```

### Critical: Zero Initialization for Stability

```
def initialize_weights(self):
    # ... standard init ...

    # Zero-out adaLN layers for identity initialization
    for block in self.blocks:
        nn.init.constant_(block.adaLN_modulation[-1].weight, 0)
        nn.init.constant_(block.adaLN_modulation[-1].bias, 0)
```

This ensures scale = shift = gate = 0 at init, so:

- $\text{modulate}(\mathbf{x}, 0, 0) = \mathbf{x}$ (identity)
- $\text{gate} \cdot \text{output} = 0$ (no residual contribution)

# 12 Step 11: CLS Token Removal & Prediction Head

## Step 11: Output Processing

**Purpose:** Remove CLS token and project back to motif space.

**Mathematical Formulation:**
**Stage 1: Remove CLS token**

$$\mathbf{Z}_{\text{regions}} = \mathbf{Z}[:, 1:, :] \in \mathbb{R}^{B \times N \times D}$$

(Remove first token, keep positions 1 to 900)
**Stage 2: Linear projection to output**

$$\hat{\mathbf{X}} = \mathbf{Z}_{\text{regions}} \mathbf{W}_{\text{head}} + \mathbf{b}_{\text{head}}$$

where:

- $\mathbf{W}_{\text{head}} \in \mathbb{R}^{768 \times 283}$
- $\hat{\mathbf{X}} \in \mathbb{R}^{B \times 900 \times 283}$ — predicted motif features

**PyTorch Implementation:**

```
# In forward():
# After encoder
x = self.encoder(x, t)  # (B, 901, 768)

# Remove CLS token (index 0)
x = x[:, 1:]  # (B, 900, 768)

# Project to output dimension
x_masked = self.head_mask(x)  # (B, 900, 283)

# head_mask is defined as:
self.head_mask = nn.Linear(768, 283)  # in_features, out_features from config
```

# 13 Step 12: Loss Computation

## Step 12: Masked MSE Loss

**Purpose:** Compute reconstruction loss only on masked positions.

**Mathematical Formulation:**

$$\mathcal{L} = \frac{1}{|\mathcal{M}|} \sum_{i \in \mathcal{M}} \|\hat{\mathbf{x}}_i - \mathbf{x}_i\|_2^2$$

where:

- $\mathcal{M} = \{i : m_i = 1\}$ — set of masked indices
- $\hat{\mathbf{x}}_i \in \mathbb{R}^{283}$ — predicted motif vector at position $i$
- $\mathbf{x}_i \in \mathbb{R}^{283}$ — ground truth motif vector

**Implementation trick:** Zero out unmasked positions before loss:

$$\text{pred}_{\text{masked}} = \hat{\mathbf{X}} \odot \mathbf{M}, \quad \text{target}_{\text{masked}} = \mathbf{X} \odot \mathbf{M}$$

Then MSE naturally ignores zeros (they contribute 0 to loss).

**PyTorch Implementation:**

```python
def before_loss(self, output, batch):
    """Prepare predictions and targets for loss computation."""
    x_masked, x_original, mask = output

    # Apply mask to both predictions and targets
    # Masked positions: compute loss
    # Unmasked positions: multiply by 0 (ignore)
    pred = {'masked': x_masked * mask}    # (B, 900, 283) * (B, 900, 1)
    obs = {'masked': x_original * mask}

    return pred, obs

# Loss is then computed as:
# loss = MSELoss(pred['masked'], obs['masked'])
```

**From config (dit.yaml):**

```yaml
loss:
  components:
    masked:
      _target_: torch.nn.MSELoss
      reduction: mean
  weights:
    masked: 1.0
```

# 14 Step 13: Diffusion Schedule

**Step 13: Noise Schedule Setup**

**Purpose:** Precompute diffusion noise schedule parameters.

**Mathematical Formulation:**
**Linear beta schedule:**

$$\beta_t = \beta_{\text{start}} + \frac{t}{T-1}(\beta_{\text{end}} - \beta_{\text{start}})$$

with $\beta_{\text{start}} = 0.0001$, $\beta_{\text{end}} = 0.02$, $T = 1000$.
**Derived quantities:**

$$\alpha_t = 1 - \beta_t \tag{5}$$

$$\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s \quad \text{(cumulative product)} \tag{6}$$

$$\sqrt{\bar{\alpha}_t} \quad \text{(for signal scaling)} \tag{7}$$

$$\sqrt{1-\bar{\alpha}_t} \quad \text{(for noise scaling)} \tag{8}$$

**Forward diffusion process:**

$$\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$$

**PyTorch Implementation:**

```
def _setup_diffusion(self, diff_cfg):
    """Setup noise schedule for diffusion."""
    num_timesteps = diff_cfg.get('num_timesteps', 1000)
    beta_start = diff_cfg.get('beta_start', 0.0001)
    beta_end = diff_cfg.get('beta_end', 0.02)

    # Linear schedule: [0.0001, ..., 0.02]
    betas = torch.linspace(beta_start, beta_end, num_timesteps)
    alphas = 1.0 - betas  # alpha_t = 1 - beta_t

    # Cumulative product: alpha_bar_t
    alphas_cumprod = torch.cumprod(alphas, dim=0)

    # Register as buffers (non-trainable)
    self.register_buffer('betas', betas)
    self.register_buffer('alphas', alphas)
    self.register_buffer('alphas_cumprod', alphas_cumprod)
    self.register_buffer('sqrt_alphas_cumprod',
                         torch.sqrt(alphas_cumprod))
    self.register_buffer('sqrt_one_minus_alphas_cumprod',
                         torch.sqrt(1.0 - alphas_cumprod))
    self.num_timesteps = num_timesteps
```

# 15 Complete Forward Pass Summary

# 16 Parameter Count

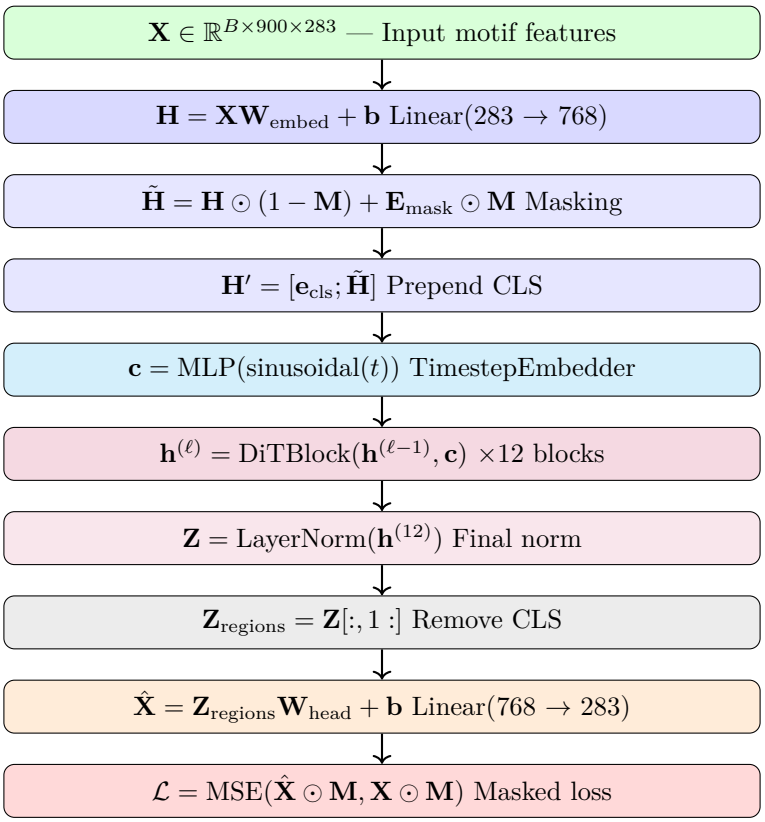


Figure 1: Complete forward pass with mathematical operations at each step.

Table 1: Approximate parameter count for GETRegionDiffusion

| **Component** | **Computation** | **Parameters** |
|---|---|---|
| RegionEmbed | $283 \times 768 + 768$ | 217,856 |
| TimestepEmbedder | $256 \times 768 + 768 + 768 \times 768 + 768$ | 787,968 |
| DiTBlock ($\times 12$) | | |
| - qkv | $768 \times 2304 + 2304$ | 1,771,776 |
| - proj | $768 \times 768 + 768$ | 590,592 |
| - MLP fc1 | $768 \times 3072 + 3072$ | 2,362,368 |
| - MLP fc2 | $3072 \times 768 + 768$ | 2,360,064 |
| - adaLN | $768 \times 4608 + 4608$ | 3,543,552 |
| Subtotal per block | | 10,628,352 |
| All 12 blocks | | 127,540,224 |
| Final LayerNorm | $768 \times 2$ | 1,536 |
| head_mask | $768 \times 283 + 283$ | 217,627 |
| Tokens (mask + cls) | $768 \times 2$ | 1,536 |
| **Total** | | $\approx$ **129M** |