LSTM - DUGSB

1. NA-ALISTA

在 NA-ALISTA（Neurally Augmented ALISTA） 算法中，LSTM (Long Short-Term Memory) 被用于**自适应地预测每一步迭代的阈值** $\theta^{(k,x^*)}$ **和步长** $\gamma^{(k,x^*)}$，而不是采用固定的或全局优化的参数。具体来说：

[Behrens F, Sauder J, Jung P. Neurally augmented ALISTA[J]. arXiv preprint arXiv:2010.01930, 2020.]

- **输入特征**：

  每步迭代的残差 ℓ_1 范数

  $r^{(k)} = \|\Phi x^{(k)} - y\|_1$

  更新量 ℓ_1 范数

  $u^{(k)} = \|W^T(\Phi x^{(k)} - y)\|_1$

- **输出**：当前迭代的阈值 $\theta^{(k,x^*)}$ 和步长 $\gamma^{(k,x^*)}$，直接用于软阈值函数以更新重建向量：

  $x^{(k+1)} = \eta_{\theta^{(k,x^*)}}\left(x^{(k)} - \gamma^{(k,x^*)} W^T(\Phi x^{(k)} - y)\right)$

- **目标**：使参数根据当前重建状态动态调整，从而获得更紧的误差上界和更好的重建效果。

原本：训练固定 $\theta^{(k)}, \gamma^{(k)}$

Algorithm 1: Neurally Augmented ALISTA

Learnable Parameters: initial cell state $c_0 \in \mathbb{R}^H$, initial hidden state $h_0 \in \mathbb{R}^H$, cell state to output matrix $U \in \mathbb{R}^{2\times H}$ and parameters of LSTM cell.

Input: y

$x \leftarrow 0$; $h \leftarrow h_0$; $c \leftarrow c_0$

for $\{1, \ldots, K\}$ **do**

$r \leftarrow \|\Phi x - y\|_1$

$u \leftarrow \|W^T(\Phi x - y)\|_1$

$c, h \leftarrow \texttt{LSTM}(c, h, [r, u])$

$\theta, \gamma \leftarrow \texttt{Softsign}(Uc)$

$x \leftarrow \eta_\theta\Big(x - \gamma W^T(\Phi x - y)\Big)$

end

Return x;

-> $\theta^{(k)}, \gamma^{(k)}$ 从固定变为 LSTM 动态生成

Algorithm 1: Neurally Augmented ALISTA

Learnable Parameters: initial cell state $c_0 \in \mathbb{R}^H$, initial hidden state $h_0 \in \mathbb{R}^H$, cell state to output matrix $U \in \mathbb{R}^{2\times H}$ and parameters of LSTM cell.

Input: y

$x \leftarrow 0$; $h \leftarrow h_0$; $c \leftarrow c_0$

for $\{1, \ldots, K\}$ **do**

$r \leftarrow \|\Phi x - y\|_1$

$u \leftarrow \|W^T(\Phi x - y)\|_1$

$c, h \leftarrow \texttt{LSTM}(c, h, [r, u])$

$\theta, \gamma \leftarrow \texttt{Softsign}(Uc)$

$x \leftarrow \eta_\theta\Big(x - \gamma W^T(\Phi x - y)\Big)$

end

Return x;

NA-ALISTA 的研究表明，即使使用较小的 LSTM 网络（例如隐藏层维度 $H = 64$），也能显著提升算法的自适应性和重建性能，且计算开销相对于主迭代可以忽略不计。

2. LSTM (Long Short-Term Memory)

LSTM 是一种特殊的循环神经网络（RNN），专门设计来解决传统RNN中的"长期依赖"问题。简单来说，它能够记住很久以前的信息，而不会像人类一样容易忘记。

LSTM 是一种带门控的循环神经网络，能够捕捉时间序列中的长期依赖信息。在每一步迭代中，LSTM 根据当前输入和上一隐藏状态更新内部记忆单元，并生成新的隐藏状态。

（1） 遗忘门（Forget Gate）：决定忘记什么

- 任务：决定从记忆中丢弃哪些旧信息
- 工作原理：查看当前输入和上一步的隐藏状态，给出0到1之间的分数

- 0 = "完全忘记这个信息"
- 1 = "完全保留这个信息"

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$$

（2） 输入门（Input Gate）：决定记住什么

- 任务：决定将哪些新信息存入记忆
- 分为
  - 输入门：决定哪些值需要更新（0到1之间的分数）

    $$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$$
  - 候选记忆：创建新的候选值，可能被添加到记忆中

    $$\tilde{c}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c)$$

（3） 记忆单元更新

- 任务：实际更新记忆单元
- 工作原理：结合遗忘门的建议和输入门的建议来更新记忆

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

（4） 输出门（Output Gate）：决定输出什么

- 任务：决定基于当前记忆要输出什么信息
- 工作原理：查看当前输入和记忆状态，决定输出的内容

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$$

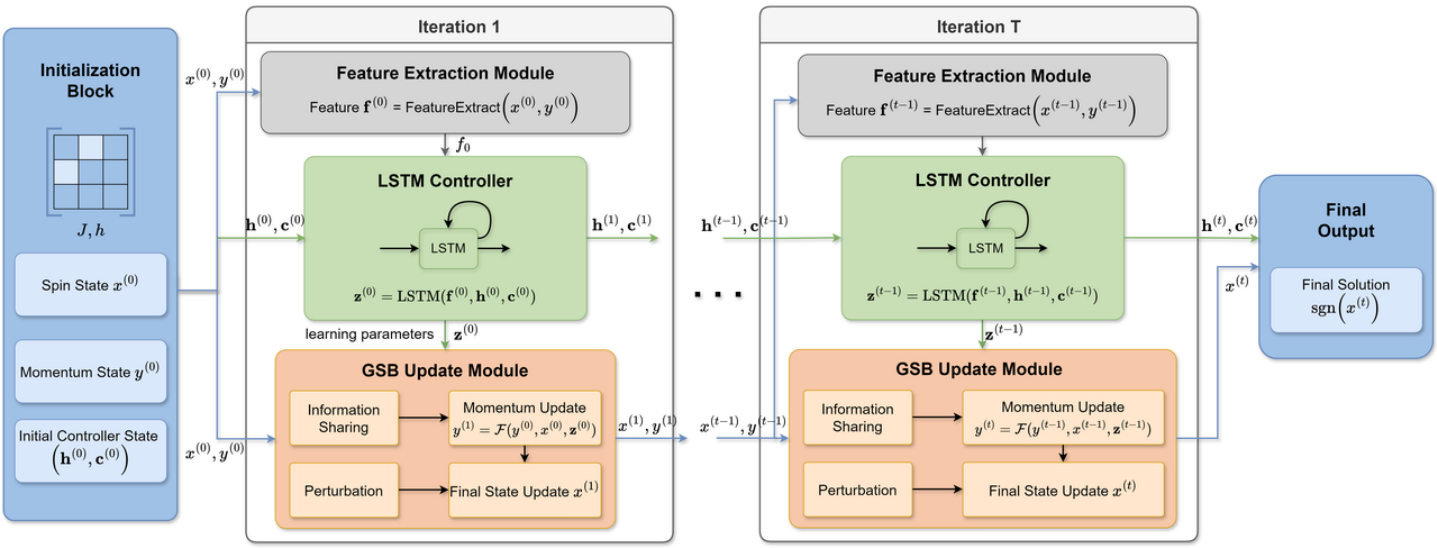
$$h_t = o_t \odot \tanh(c_t)$$

其中：

- σ：Sigmoid 激活函数，输出范围 $[0, 1]$
- $\tanh$：输出范围 $[-1, 1]$
- $\odot$：逐元素乘法
- x_t：当前输入特征向量
- h_{t-1}：上一时刻隐藏状态
- c_{t-1}：上一时刻记忆单元

LSTM 通过门控机制动态选择信息的保留和更新，实现对时间依赖的建模。

3. DUGSB + LSTM



借鉴 NA-ALISTA 中使用 LSTM 动态生成控制参数的思想，在 DUGSB 框架中引入 LSTM 用于动态生成控制参数，指导算法中每个粒子的迭代策略。其核心作用：

- 根据粒子当前状态和历史信息，输出用于调节搜索策略的参数；
- 提供可训练、可适应的控制机制，提高算法稳定性和收敛效率。

3.1 输入特征示例

LSTM 的输入是一个特征向量，可根据问题和粒子状态选择不同特征，例如：

- **当前粒子能量 [1,B]**
- **粒子与全局最优解距离[1,B]**
- **粒子分布的标准差 [1,B]**
- 历史能量变化
- 局部邻域能量差量
- 粒子自身动量信息
- 问题相关约束指标

输入特征向量组合：

$$\text{features}_t = [E(x_t), \|x_t - gbest_t\|, \text{std}(x_t), \Delta E_t, ...] \in \mathbb{R}^{B \times D}$$

其中 B 为 batch size， D 为特征维度。

3.2 输出控制参数

LSTM 的隐藏状态 h_t 经过小型全连接网络（MLP）生成控制参数，例如，这里只选择了两个 GSB 中加入进去的参数：

- a_t ： 用于调节GSB扰动幅度
- b_t ： 用于GSB方向翻转

公式：

$$[a_t, b_t] = \text{Softplus}(\text{MLP}(h_t))$$

使用 Softplus 确保输出非负，便于后续算法使用。

4. LSTM 控制器实现

代码块

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

class LSTM_DUgSB(nn.Module):
    def __init__(self, input_dim=6, hidden_dim=128):
        """
        input_dim: 输入特征维度 D，可根据特征选择调整
        hidden_dim: LSTM 隐藏状态维度
        """
        super().__init__()

        # LSTMCell 用于单步迭代
        # 设计原因:
        # 使用单步 LSTMCell 而非 nn.LSTM 是因为我们希望在每次迭代中动态控制输入
        # 并可以在每步迭代后立即生成控制参数
        self.lstm = nn.LSTMCell(input_dim, hidden_dim)

        # MLP 生成控制参数 a_t 和 b_t
        # 设计原因:
```

```
        # 通过一个两层 MLP 提升非线性表达能力，
        # 输出通过 Softplus 确保为非负，便于表示幅度或权重
        self.param_head = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, 2),
            nn.Softplus()
        )

        # 初始化隐藏状态和记忆单元
        # 设计原因:
        # 可学习的初始隐藏状态和记忆单元，使网络在训练中自适应优化初始化
        self.init_hidden = nn.Parameter(torch.randn(hidden_dim))
        self.init_cell = nn.Parameter(torch.randn(hidden_dim))

    def forward(self, x, hidden=None):
        """
        x: 当前特征向量 (B, input_dim)
        hidden: 上一时刻隐藏状态和记忆单元 tuple (h, c)
        """
        if hidden is None:
            # 初始化 h, c 为可训练参数
            h = self.init_hidden.expand(x.size(0), -1)
            c = self.init_cell.expand(x.size(0), -1)
        else:
            h, c = hidden

        # LSTM 单步更新
        h, c = self.lstm(x, (h, c))

        # 生成控制参数
        ab_params = self.param_head(h)  # 输出 (B, 2)

        ....
```

LSTMController 由三部分组成：

1. **LSTMCell**

代码块

```
self.lstm = nn.LSTMCell(input_dim, hidden_dim)
```

用于单步迭代更新隐藏状态 h_t 和记忆单元 c_t 。

输入是粒子当前状态特征向量 x_t 。

输出隐藏状态 h_t ，包含了历史信息和当前状态信息。

2. **MLP 头部 (param_head)**

```
代码块
1   self.param_head = nn.Sequential(
2       nn.Linear(hidden_dim, hidden_dim),
3       nn.ReLU(),
4       nn.Linear(hidden_dim, 2),
5       nn.Softplus()
6   )
```

将隐藏状态 h_t 映射到具体控制参数 (a_t, b_t) 。

使用 ReLU 增强非线性表达能力。

Softplus 确保输出非负，适合作为幅度或权重参数。

3. **可训练的初始隐藏状态与记忆单元**

```
代码块
1   self.init_hidden = nn.Parameter(torch.randn(hidden_dim))
2   self.init_cell = nn.Parameter(torch.randn(hidden_dim))
```

允许 LSTM 在训练过程中自适应学习最优初始化，而非固定为零。

在 DUGSB 中可提高迭代初期的控制效果和稳定性。