STLight: a Fully Convolutional Approach for Efficient Predictive Learning by Spatio-Temporal joint Processing

Andrea Alfarano[1,2*] Alberto Alfarano[3*] Linda Friso[4] Andrea Bacciu[5†]
Irene Amerini[5] Fabrizio Silvestri[5]

[1]DVS, University of Zurich [2]Max Planck Society [3]Meta [4]Google [5]Sapienza, University of Rome

andrea.alfarano@uzh.ch, albealfa@meta.com, lfriso@google.com,
bacciu@diag.uniroma1.it, amerini@diag.uniroma1.it, fsilvestri@diag.uniroma1.it

[*]Equal contribution [†]Work done prior to joining Amazon

The supplementary material provides a comprehensive analysis of the STLight method. Each section contributes unique insights:

In Section A, we extend our evaluation by comparing our model with results from others published STL papers, against longer training duration, and presenting an ablation study on the impact of varying training hyperparameters.

In Section B, we analyze STLight's complexity, breaking down its components and parameter counts.

Section C explores optimal kernel sizes for STLight's convolutional stages.

In Section D, we investigate the impact of weight initialization schemes on training stability.

Section E focuses on optimal training settings.

In Section F, we compare decoding operations' impact on model stability and performance.

Finally, Section G presents the full STLight implementation.

A. Additional Evaluation Results

In Section 4, we used the OpenSTL benchmark to compare our results with public and reproducible outcomes on established benchmark datasets. However, public libraries like OpenSTL do not fully guarantee (1) the correctness of the implementations, (2) the adherence to the original training protocols of each baseline, or (3) the optimality of the default standard training parameters used for learning.

Thus, in this section, we address these limitations.

In Table 1, we compare our model with results published in the literature for relevant STL models on the MMNIST, TaxiBJ, and KTH datasets. While all baseline results in Table 3 are obtained under uniform training settings and protocols, Table 1 lacks this standardization. Given that each model in Table 1 is trained for different (and not always reported) durations, we trained our STLight baseline using the same hyperparameters as in Table 2, but with extended training durations: 2000 epochs for MMNIST and 150 epochs for KTH. Our model still outperforms the other baselines, confirming the OpenSTL benchmark results from Table 3.

Table 1. Comparison of our model results and the results published for each model in literature Across MMNIST, TaxiBJ and KTH datasets. Our model have been trained using hyperparameters of Table 2, except for the training duration that have been extended to 2000 epochs for MMNIST and 150 epochs for KTH.

Model	MMNIST (MSE ↓)	TaxiBJ (MSE ↓)	KTH (SSIM ↑) $10 \rightarrow 20$
ConvLSTM [6]	103.3	48.5	0.712
VPTR-NAR [13]	107.2	-	0.859
VPTR-FAR [13]	63.6	-	0.879
PredRNN [10]	56.8	46.4	0.839
PredRNN++ [8]	46.5	44.8	0.865
MIM [12]	44.2	42.9	-
E3D-LSTM [9]	41.3	43.2	0.879
MAU [1]	29.5	-	-
PhyDNet [3]	24.4	41.9	-
Crevnet [14]	22.3	-	-
PredRNNv2 [11]	48.8	-	-
IAM4VP [5]	15.3	37.2	-
TAU [7]	19.8	34.4	0.911
STLight-L (Ours)	**14.72**	**30.87**	**0.9113**

In Table 2, we extend our model evaluation to the 2000-epoch OpenSTL MMNIST results. Notably, our model corroborates the findings discussed in Section 4.2.1, demonstrating superior performance and efficiency compared to the OpenSTL baselines across both accuracy and computational metrics. Specifically, even with extended training time, STLight-L achieves the best trade-off between accuracy and computational cost, with a significantly lower MSE (14.77) and MAE (47.17), while maintaining a high SSIM (0.9686). As discussed in Section 4.2.1,

our model continues to outperform recurrent architectures like PredRNN++ and MIM in accuracy while requiring fewer FLOPs, and recurrent-free architectures like TAU and SimVP with fewer parameters, establishing STLight-L as both a highly performant and resource-efficient solution.

Table 2. Quantitative results comparing our model (STLight-L) against OpenSTL baselines, trained for 2000 epochs using the training settings from Table 2. The table reports both accuracy metrics (MSE, MAE, SSIM) and computational metrics (number of parameters, FLOPs) under equivalent training and evaluation conditions on the MMNIST dataset

Model	# Params	FLOPs	MSE	MAE	SSIM
ConvLSTM-S	15.0M	56.8G	22.41	73.07	0.9480
PredNet	12.5M	8.6G	31.85	90.01	0.9273
PhyDNet	3.1M	15.3G	20.35	61.47	0.9559
PredRNN	23.8M	116.0G	26.43	77.52	0.9411
PredRNN++	38.6M	171.7G	14.07	48.91	0.9698
MIM	38.0M	179.2G	14.73	52.31	0.9678
MAU	4.5M	17.8G	22.25	67.96	0.9511
E3D-LSTM	51.0M	298.9G	24.07	77.49	0.9436
PredRNN.V2	23.9M	116.6G	17.26	57.22	0.9624
SimVP+IncepU	58.0M	19.4G	21.15	64.15	0.9536
TAU	44.7M	16.0G	15.69	51.46	0.9721
STLight-L (Ours)	32.9M	32.9M	**14.77**	**47.17**	**0.9686**

B. STLight method complexity

To analyze the complexity of the STLight method, we'll break down its main components: Spatio-Temporal Patches, Patch Shuffle and Reassemble, and the Repeated STLMixer setup. By examining the number of parameters each part uses, we can gain insights into the method's design and its computational demands.

- **Spatio-Temporal Patches** STLight encodes the input frames with a parameter count of $O\left((TC) \cdot d \cdot k_E^2\right)$ because it uses a single convolution, with input channels, output channels and kernel size equal to $T \cdot C$, d, k_E respectively.
- **Patch Shuffle and Reassemble** While the patch shuffle layer doesn't have any learnable parameters, the patch reassemble is based on a single pointwise convolution with input channels, output channels and kernel size respectively equal to d/p^2, $T' \cdot C$, $k_D = 1$. Therefore STLight decodes the processed signals with a parameter count of $O\left(d/p^2 \cdot (T'C) \cdot k_D^2\right) = O\left(d/p^2 \cdot (T'C)\right)$.
- **Repeated STLMixer** The parameter count from our proposed STLMixer architecture is $O\left(\mathtt{de} \cdot (d^2 + d \cdot k_{T_1}^2 + d \cdot k_{T_2}^2)\right)$. In fact, each STLMixer is composed of two depthwise convolutions and one pointwise convolutions. Each convolution has the same input channels and output channel dimensions that are equal to d. Depthwise convolutions perform group convolution with group_size $= d$, hence their parameter counts are $O(d \cdot k_{T_1}^2)$ and $O(d \cdot k_{T_2}^2)$. The pointwise convolution has 1×1 kernel size, so the parameter count for this layer is $O(d^2)$. Summing the three terms and considering that they are $\mathtt{de}$ repeated blocks inside STLight, we obtain the aforementioned complexity.

In order to simplify the parameter count formulas, we consider the following assumptions:

1. $T \cdot C$ and $T' \cdot C$ typically remain below 10, while d often exceeds 1000, hence $T \cdot C \ll d$ and $T' \cdot C \ll d$. Similarly $k_{T_1}^2 \ll d$ and $k_{T_2}^2 \ll d$.
2. $O \leq 2$ and $p \leq 2$, leading to $k_E = p \cdot \max(1, O) \leq 4$ using the formula shown in Section 2.1.

Therefore the parameter counts of our encoder and decoder blocks scale linearly with respect to d, while the parameters count of the repeated STLMixer blocks can be expressed as $O(\mathtt{de} \cdot d^2)$.

C. Optimal k_{T_1} and k_{T_2}

k_{T_1}/k_{T_2}	3	5	7	9	11
3	24.16	22.48	22.33	22.33	22.75
5	24.01	22.67	22.45	22.78	23.03
7	23.75	22.83	22.88	23	24
9	23.45	23.02	23.1	23.38	23.62
11	23.04	23.14	23.24	23.65	24.51

Table 3. STLight MSE comparison for different values of k_{T_1} and k_{T_2}.

This section explores the optimal kernel sizes k_{T_1} and k_{T_2} for the two depthwise convolutional stages within the STLMixer block. While large values for k_{T_1} and k_{T_2} ensure good local context and wider receptive field, they also increase the model's parameter count, possibly leading to worse performances due to overfitting. Our experiments, detailed in Figure 3, show that a small $k_{T_1} = 3$ combined with a larger $k_{T_2} \in \{5, 7\}$ achieves optimal performance while maintaining a lower parameter count. They also indicate that excessively large kernel sizes not only decrease efficiency but also lead to poorer performance. Given the consistency of these findings across various hyperparameters configurations and scenarios, we decided to keep k_{T_1} and k_{T_2} constant during our evaluations.

D. Weight initialization

Initial weight settings are crucial for how quickly and effectively a deep learning model learns. In our study, we compare three different initializations:

- **Kaiming-Uniform** Following PyTorch's default weight initialization, we use uniform Kaiming initialization (also known as He initialization) [4] for all the model's convolutional blocks.

- **Kaiming-Normal** As reported in Listing 1, we initialize convolutional layers using gaussian Kaiming initialization.

- **Hybrid** We initialize the patch reassemble layer using the uniform Kaiming initialization and we initialize all the other layers following the approach mentioned in the Kaiming-Normal initialization. We hypothesize that the last layer requires a different initialization because it rearranges the shuffled patches and, unlike other layers, it is not responsible for processing spatial-temporal correlations.

```
def _init_weights(self, m):
    if isinstance(m, nn.Conv2d):
        nn.init.kaiming_normal_(
            m.weight,
            mode='fan_out',
            nonlinearity='relu'
        )
        if m.bias is not None:
            nn.init.constant_(m.bias, 0)
```

Listing 1. Code for the Kaiming-Normal weights initialization.

In Figure 1 we emphasize the importance of carefully selecting initial weight settings to guarantee stable training and reach optimal accuracy, by evaluating the three different initialization schemes mentioned above.

Our experiments show that the Kaiming-Uniform initialization requires several epochs to stabilize before beginning to converge, while the all Kaiming-Normal approach does not lead to stable training.

The optimal strategy is the Hybrid initialization, which leads to stable training while not requiring a number of initial epochs to stabilize.

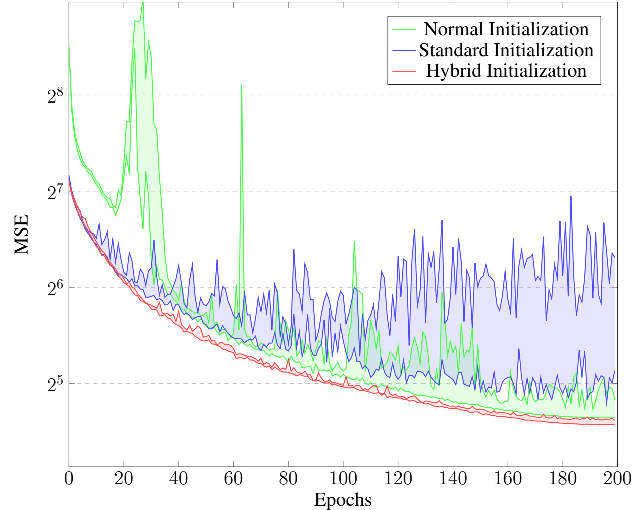

Figure 1. Learning curve comparison for three different weights initializations. For each of them, we report the mimimum and maximum boundaries of the MSE out of 5 runs.

E. Optimal Training Settings

In Tables 4 and **??**, we evaluate the impact of various training hyperparameters on the performance of STLight-L on the MMNIST dataset to identify the most effective training hyperparameter configuration. The learning rate is a critical hyperparameter that influences the model's ability to learn from data while maintaining stability. A low learning rate leads to slow convergence, while a higher learning rate may cause instability, preventing convergence to optimal results [2]. The final div factor determines the minimum learning rate achieved at the end of a training cycle with the OneCycleLR learning rate scheduler, influencing convergence behavior and model performance. We examine the effects of varying the learning rate (LR) and the number of training epochs on key metrics such as MSE, MAE, and SSIM. Tables 4 and **??** show that the default OpenSTL hyperparameters are highly effective for STLight-L, with potential improvements of 0.5 in MSE when using a learning rate of 0.003 instead of 0.001. In fact, in the first section of the table, we observe that reducing the learning rate from 0.003 to 0.0003 results in increased MSE and MAE, indicating that overly small learning rates hinder model performance. Conversely, increasing the learning rate to 0.01 leads to worse results, confirming that the optimal learning rate lies near 0.003. Table 4 also shows that extending the number of training epochs improves all accuracy metrics, confirming that longer training durations significantly enhance model performance.

Table 4. Ablation Study on STLight-L with Fixed Hyperparameters (dim=1400, depth=16, kernel_size_1=3, kernel_size_2=7, patch_size=2) and Varying Training Learning Rate, and the Number of Epochs

LR	Final Div	Epoch	MSE	MAE	SSIM
↑ Varying Learning Rate (LR)					
0.0003	10000	200	27.42	76.86	0.937
0.001	10000	200	22.28	66.11	0.950
0.003	10000	200	21.80	64.90	0.951
0.01	10000	200	38.47	90.42	0.939
↑ Varying Number of Epochs (Epoch)					
0.001	10000	500	18.88	58.59	0.957
0.001	10000	1000	17.89	54.25	0.962
0.001	10000	2000	14.77	47.17	0.969

FinalDivFactor/lr	0.0003	0.001	0.003	0.01
1000	26.04	22.41	22.01	35.99
3000	27.14	22.49	21.85	38.35
10000	27.42	22.33	21.8	38.47

Table 5. STLight-33M MSE comparison for different values of Learning Rate (lr) and FinalDivFactor.

F. Order of the decoding operations

We investigate the optimal procedure for decoding the tensor $Z''_T \in \mathbb{R}^{B\times d\times H/p\times W/p}$ into the desired tensor of the predicted frames $\mathcal{B}'_{T'}$. We evaluated two choices:

- **Shuffle-Reassemble** We first perform patch shuffle, obtaining a tensor of shape $B \times d/p^2 \times H \times W$. Subsequently, the tensor is reassembled using a 1×1 convolutional layer, with d/p^2 input channels and $T' \cdot C$ output channels.
- **Reassemble-Shuffle** We first perform patch rearrange, using a 1×1 convolutional layer, with d input channels and $T' \cdot C \cdot p^2$ output channels, obtaining an intermediate tensor of shape $B \times (T' \cdot C \cdot p^2) \times H/p \times W/p$. Subsequently, we perform patch shuffle on the intermediate tensor, obtaining $B \times (T' \cdot C) \times H \times W$.

Both choices are able to effectively decode the output stage. In Figure 2, we train STLight for different configuration settings and we report the standard deviation of the differences in loss values between consecutive epochs ("dispersion") to get a measure of how much variation we get in the loss reduction. The figure clearly illustrates that Shuffle-Reassemble notably curtails the feature dispersion. This adjustment leads to an uptick in model performance and promotes stability.

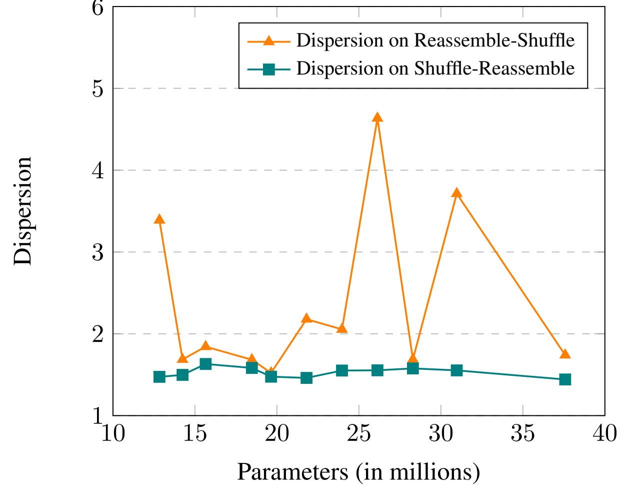

Figure 2. Dispersion vs Number of parameters on different decoding operations order

G. STLight Code

We report the full implementation of the STLight model. We encourage readers to use STLight through our OpenSTL implementation which will be publicly available.

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

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

    def forward(self, x):
        return self.fn(x) + x

def STLMixer(dim, K_1, k_2):
    return nn.Sequential(
        Residual(  # depthwise convolution block
            nn.Sequential(
                nn.Conv2d(dim, dim, K_1, groups=dim, padding="same"),
                nn.Conv2d(dim, dim, K_2, groups=dim, padding="same",
                          dilation=3),
                nn.GELU(),
                nn.BatchNorm2d(dim),
            )
        ),
        nn.Sequential(  # pointwise convolution block
            nn.Conv2d(dim, dim, kernel_size=1),
            nn.GELU(), nn.BatchNorm2d(dim)
        ),
    )

def PatchEncoder(in_layers, dim, patch_size, overlapping):
    return nn.Sequential(
        nn.Conv2d(in_layers, dim,
                  kernel_size=patch_size * max(1, overlapping),
                  stride=patch_size,
                  padding=max(0, overlapping - 1) * patch_size // 2
                  ),
        nn.BatchNorm2d(dim),
        nn.GELU(),
    )

class STLight(nn.Module):
    def __init__(
        self, in_layers, out_layers, dim, depth, patch_size,
        overlapping, K_1, K_2
    ):
        super().__init__()
        self.out_layers = out_layers
        self.patch_encoder = PatchEncoder(in_layers, dim, patch_size,
                                          overlapping)
        self.net = nn.ModuleList([STLMixer(dim, K_1, K_2) for _ in range(depth)])
        self.patch_reassemble = nn.Conv2d(dim // patch_size**2,
                                          out_layers, kernel_size= 1)
        self.up = nn.PixelShuffle(patch_size)
        self.patch_encoder.apply(self._init_weights)
        self.net.apply(self._init_weights)

    def forward(self, x):
        B, T, C, H, W = x.shape
        x = x.reshape(B, T * C, H, W)
        x = self.patch_encoder(x)

        for i, block in enumerate(self.net):
            if i == len(self.net) // 3:
                x1 = x
            if i == 2 * len(self.net) // 3:
                x = x + x1
            x = block(x)
        x = self.up(x)
        x = self.patch_reassemble(x)
        return x.reshape(B, self.out_layers // C, C, H, W)
```

Figure 3. Full STLight implementation

References

[1] Zheng Chang, Xinfeng Zhang, Shanshe Wang, Siwei Ma, Yan Ye, Xiang Xinguang, and Wen Gao. Mau: A motion-aware unit for video prediction and beyond. *Advances in Neural Information Processing Systems*, 34:26950–26962, 2021. 1

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org. 3

[3] Vincent Le Guen and Nicolas Thome. Disentangling physical dynamics from unknown factors for unsupervised video prediction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11474–11484, 2020. 1

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015. 3

[5] Minseok Seo, Hakjin Lee, Doyi Kim, and Junghoon Seo. Implicit stacked autoregressive model for video prediction. *arXiv preprint arXiv:2303.07849*, 2023. 1

[6] Xingjian Shi, Zhourong Chen, Hao Wang, Dit-Yan Yeung, Wai-Kin Wong, and Wang-chun Woo. Convolutional lstm network: A machine learning approach for precipitation nowcasting. *Advances in neural information processing systems*, 28, 2015. 1

[7] Cheng Tan, Zhangyang Gao, Lirong Wu, Yongjie Xu, Jun Xia, Siyuan Li, and Stan Z Li. Temporal attention unit: Towards efficient spatiotemporal predictive learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 18770–18782, 2023. 1

[8] Yunbo Wang, Zhifeng Gao, Mingsheng Long, Jianmin Wang, and S Yu Philip. Predrnn++: Towards a resolution of the deep-in-time dilemma in spatiotemporal predictive learning. In *International Conference on Machine Learning*, pages 5123–5132. PMLR, 2018. 1

[9] Yunbo Wang, Lu Jiang, Ming-Hsuan Yang, Li-Jia Li, Mingsheng Long, and Li Fei-Fei. Eidetic 3d lstm: A model for video prediction and beyond. In *International conference on learning representations*, 2018. 1

[10] Yunbo Wang, Mingsheng Long, Jianmin Wang, Zhifeng Gao, and Philip S Yu. Predrnn: Recurrent neural networks for predictive learning using spatiotemporal lstms. *Advances in neural information processing systems*, 30, 2017. 1

[11] Yunbo Wang, Haixu Wu, Jianjin Zhang, Zhifeng Gao, Jianmin Wang, S Yu Philip, and Mingsheng Long. Predrnn: A recurrent neural network for spatiotemporal predictive learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(2):2208–2225, 2022. 1

[12] Yunbo Wang, Jianjin Zhang, Hongyu Zhu, Mingsheng Long, Jianmin Wang, and Philip S Yu. Memory in memory: A predictive neural network for learning higher-order non-stationarity from spatiotemporal dynamics. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 9154–9162, 2019. 1

[13] Xi Ye and Guillaume-Alexandre Bilodeau. Vptr: Efficient transformers for video prediction. In *2022 26th International Conference on Pattern Recognition (ICPR)*, pages 3492–3499. IEEE, 2022. 1

[14] Wei Yu, Yichao Lu, Steve Easterbrook, and Sanja Fidler. Crevnet: Conditionally reversible video prediction, 2019. 1