Emulating Climate Simulations with a Hybrid CNN–LSTM Architecture

https://github.com/123devamthered/cnn-lstm-climate-emulator

Devam Derasary
dderasar@ucsd.edu

Abstract

Accurate climate simulation is vital for planning and environmental policy, but traditional physical models are computationally expensive to run at scale. I propose a hybrid CNN-LSTM (Long Short-Term Memory) architecture that captures both spatial and temporal patterns in historical climate forcing data, enabling efficient and high-fidelity emulation. The CNN backbone extracts spatial features using residual blocks with Group Normalization and SiLU activation, while the LSTM layers model temporal dependencies across 12-month sequences. Enhancements like average pooling, residual projections, and multi-head attention further improve performance and training stability. Through systematic hyperparameter tuning with Optuna, my final model achieves a public Kaggle score of 0.6965 and a private score of 0.7776, placing fifth overall. These results highlight how principled temporal modeling and architecture design can substantially advance climate emulation.

1 Introduction

Climate emulation aims to replicate the outputs of physics-based climate models using faster, data-driven surrogates. Traditional simulators, while essential for forecasting global temperature, precipitation, and extreme weather events, are computationally expensive and slow to run at high resolution. This makes it difficult to generate large ensembles for uncertainty quantification or test policy interventions under multiple future scenarios. Deep learning–based climate emulators offer a powerful alternative by producing accurate outputs at a fraction of the cost and time, enabling real-world applications such as urban planning, disaster response, and environmental risk management [6].

In this project, I tackled the challenge of improving temporal modeling in climate emulation. The starter code provided a fully convolutional architecture (SimpleCNN), which processes each time step independently and focuses solely on spatial relationships. While effective at extracting local features via residual blocks and skip connections, SimpleCNN fails to capture temporal dynamics such as seasonal cycles or long-term trends, which are crucial elements in climate data.

To address this limitation, I developed a hybrid CNN–LSTM (Long Short-Term Memory) architecture [4] (see Sec. 3.1 for details). My model retains the CNN backbone for spatial feature extraction, while integrating multi-layer LSTM blocks to model sequential dependencies across 12-month input sequences. This allows the network to learn both the "what" (spatial patterns) and the "when" (temporal evolution) of the climate system.

I introduced several architectural improvements to boost model performance and training stability:

- **Contribution A:** Replacing ReLU with SiLU (Sigmoid Linear Unit) activation, which helps improve gradient flow and convergence [2] (see Sec. 3.1)
- **Contribution B:** Enlarging convolutional kernel size to 9×9 to expand the spatial receptive field

Preprint. Under review.

- **Contribution C:** Using average pooling instead of max pooling for more stable downsampling
- **Contribution D:** Adding residual projections in the decoder to improve output reconstruction and reduce training noise
- **Contribution E:** Performing extensive hyperparameter optimization with Optuna [1] to fine-tune architecture and training configuration

My final model was trained using PyTorch Lightning and Hydra, and achieved a public Kaggle score of 0.6965 and a private score of 0.7776, placing fifth overall. These results demonstrate how incorporating temporal modeling and principled architectural design can significantly improve the fidelity and usefulness of deep climate emulators.

2 Problem Statement

The objective of this project is to build a data-driven emulator that approximates the output of a computationally expensive climate simulation model. Specifically, I aimed to predict monthly global maps of two key climate variables including surface air temperature (`tas`) and total precipitation (`pr`) based on historical time-series data of five external forcing variables. These include greenhouse gases (CO_2, CH_4), aerosols (SO_2, BC), and top-of-atmosphere incident shortwave radiation (`rsdt`). The dataset is structured under the Shared Socioeconomic Pathways (SSPs), which define plausible global development trajectories with different levels of emissions and climate impacts.

The machine learning task is to train a model on climate simulations from known SSPs (SSP126, SSP370, SSP585), and evaluate its ability to generalize to an unseen future scenario (SSP245). The goal is to accurately emulate climate dynamics while significantly reducing the computational cost of running full physical simulations. Success in this task enables faster exploration of climate futures, with real-world implications in policy-making, risk assessment, disaster preparedness, and environmental planning.

2.1 Dataset and Preprocessing

The dataset is organized as a sequence of monthly snapshots for each scenario. Each sample consists of gridded climate data over a spatial resolution of 48 latitude × 72 longitude.

Input Tensor (X):

$$X \in \mathbb{R}^{B \times S \times C_{\text{in}} \times H \times W}$$

where B is the batch size, $S = 12$ is the sequence length (representing 12 months), $C_{\text{in}} = 5$ is the number of input variables (CO_2, CH_4, SO_2, BC, and `rsdt`), and $H = 48$, $W = 72$ are the spatial grid dimensions (latitude × longitude).

Output Tensor (Y):

$$Y \in \mathbb{R}^{B \times S \times C_{\text{out}} \times H \times W}$$

where $C_{\text{out}} = 2$ corresponds to the two output climate variables (`tas` and `pr`).

The model generates spatial climate projections for the same 12-month horizon based on the corresponding inputs.

Data Splitting Strategy

The dataset is split by SSP scenario and time as follows:

- **Training set:** 2,943 samples from SSP126, SSP585, and early years of SSP370
- **Validation set:** 120 samples (final 10 years of SSP370)
- **Test set:** 360 samples (30 years from SSP245), although it was mentioned here, but I didn't rely on it since it's corrupted.

Each input sample is shaped as $[12, 5, 48, 72]$ and each output sample as $[12, 2, 48, 72]$, corresponding to a rolling year of climate input/output maps.

Normalization

We maintained the use of Z-score standardization to each variable channel across the training set:

$$X_{\text{norm}} = \frac{X - \mu_{\text{train}}}{\sigma_{\text{train}}}$$

The mean (μ) and standard deviation (σ) were computed channel-wise from all training data and reused for validation and test. Model outputs are denormalized before evaluation to restore their original scale. This normalization pipeline is directly inherited from the starter code without any modifications.

Deviations from Starter Code

We retained the original preprocessing pipeline, including normalization and data loading strategies. No major changes were made to data formatting or splits. Any modifications were limited to model architecture and training configuration, described in later sections. As the test set (SSP245) contains masked target variables and is intended for final evaluation through the competition platform, I excluded it from my development process. All model evaluation and tuning in this report were performed solely on the validation set (final 120 months of SSP370).

3 Methods

My approach is centered on a hybrid CNN-LSTM architecture designed to effectively process spatio-temporal data. The model, f_θ, is trained to minimize the Mean Squared Error (MSE) between the normalized predictions $\hat{Y}_{\text{norm}}$ and the normalized ground truth Y_{norm}.

$$\mathcal{L}_{\text{MSE}} = \frac{1}{B \cdot S \cdot C_{out} \cdot H \cdot W} \sum (Y_{\text{norm}} - \hat{Y}_{\text{norm}})^2$$

Here, B is the batch size, S is the sequence length (12 months), C_{out} is the number of output channels (2 variables: `tas`, `pr`), and H, W are the spatial dimensions (48 and 72, respectively). Predictions are made on normalized values to ensure consistent training dynamics and are denormalized during evaluation.

3.1 Model Architecture

The model (Figure 1) consists of four main components: a CNN backbone for spatial feature extraction, a pooling and projection module, a temporal processing module (LSTM), and a final decoder. I adopted a hybrid architecture that stacks a Long Short-Term Memory (LSTM) network [4] on top of a Convolutional Neural Network (CNN). This combination leverages CNNs' ability to learn localized spatial features with LSTMs' capacity to model temporal dependencies in sequential data. The CNN compresses spatial information at each monthly time step into feature vectors, while the LSTM captures how these spatial features evolve over a 12-month period.

CNN Backbone: My spatial feature extractor is a modified version of the `SimpleCNN` baseline, composed of a series of `ResidualBlock` modules. Each ResidualBlock in my CNN backbone follows the skip-connection design introduced in ResNet [3], allowing gradients to flow more easily through deep architectures. Key components include:

- An initial 2D convolution layer to expand the input channels to a higher-dimensional latent space (`cnn_init_dim`).
- A configurable stack of `ResidualBlocks` (`cnn_depth`), each containing two convolutional layers, Group Normalization, and a SiLU activation function. The SiLU (Sigmoid Linear Unit) activation [2] is known to improve gradient flow and learning stability compared to ReLU.
- The CNN processes the input as a sequence of frames reshaped into a batch and outputs features of shape $(B \times S, C_{\text{cnn}}, H, W)$.

Pooling and Feature Projection: To connect spatial features to the temporal model,Iapply:

- **Average Pooling**: A 2D Average Pooling layer downscales each feature map to a fixed spatial size (`average_pool_size`). Compared to max pooling, this method produced more stable and consistent representations.
- **Feature Projection**: Flattened feature maps are passed through a two-layer of multi-layer perceptron (MLP), reducing dimensionality to match the LSTM's hidden state size (`lstm_hidden_dim`) and providing additional learnable capacity.

Temporal Module:

- **LSTM**: A multi-layer unidirectional LSTM learns temporal dependencies across months. LSTMs extend traditional recurrent neural networks (RNNs) by incorporating memory cells with gating mechanisms, enabling them to retain information over longer sequences [4]. This is particularly suitable for climate modeling, where long-term dependencies like seasonal cycles and delayed responses are critical. I used a unidirectional LSTM to preserve causal structure—predictions should depend only on past states.
- **Multi-Head Attention**: To enhance long-term temporal modeling, a multi-head self-attention layer is applied on top of the LSTM outputs. Attention mechanisms allow the model to weigh important time steps differently, improving its ability to focus on relevant moments in the climate sequence [7].

Decoder: The final decoder is a fully connected MLP that maps the LSTM-attention output back to the desired output tensor of shape (C_{out}, H, W) per time step. A residual connection around the decoder block improves gradient flow and reduces reconstruction error.

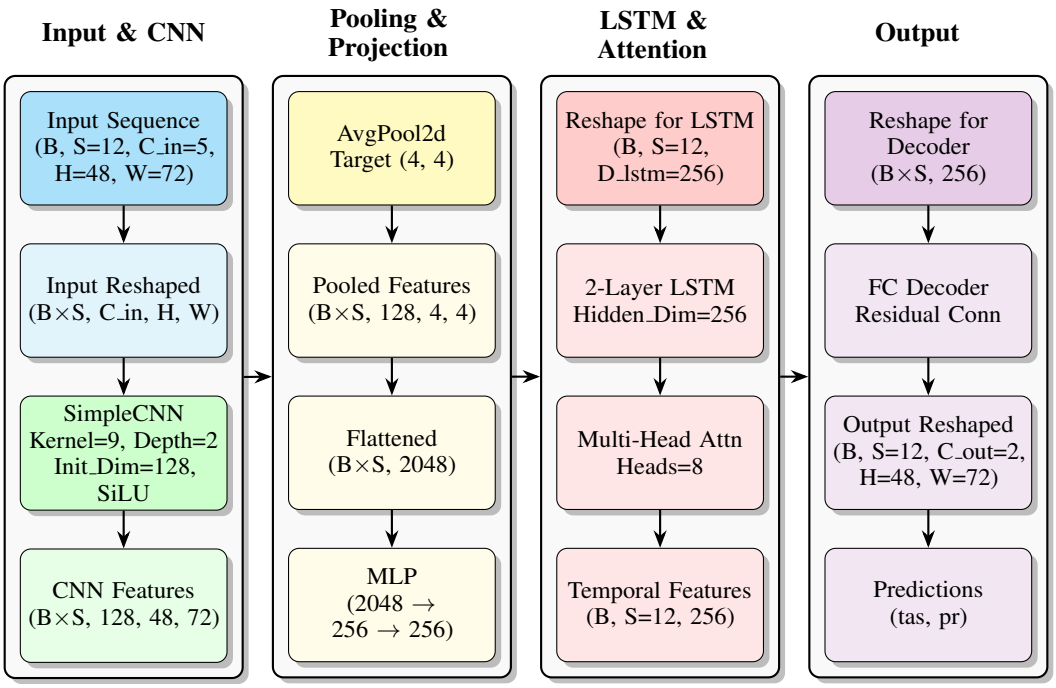

Figure 1: High-level diagram of the final CNN-LSTM architecture.

3.2 Training and Optimization

The model was trained using the Adam optimizer [5], which adaptively adjusts learning rates based on estimates of first and second moments of gradients. To improve convergence, I employed a `ReduceLROnPlateau` learning rate scheduler that monitors the validation loss and reduces the learning rate when no improvement is observed for a specified number of epochs. To prevent overfitting and reduce training time, I applied early stopping, which terminates training if the validation loss plateaus for several epochs. All experiments were managed using Hydra for configuration control, and Weights & Biases for logging, visualization, and reproducibility.

4 Experiments

I conducted a series of experiments to evaluate my proposed architecture and compare it against a relevant baseline. My primary goal was to improve upon the starter code by explicitly modeling temporal dependencies.

4.1 Baselines

I established two key baselines for comparison: the original `SimpleCNN` starter code and an experimental 3D-CNN.

4.1.1 Baseline A: SimpleCNN from the Starter Code

Baseline A is a fully convolutional neural network that processes each input time step independently, using a deep residual architecture to capture spatial patterns in climate variables. It does not model temporal dependencies and instead treats each of the 12 months in a sequence as a separate input image.

Architecture Overview: The model follows a deep encoder-style structure with four stages. **(1) Initial Layer:** `Conv2d(5, 64, kernel_size=3)` followed by `BatchNorm2d(64)` and `ReLU`, expanding the 5-channel input (CO_2, CH_4, SO_2, BC, rsdt) into 64 feature maps. **(2) Residual Blocks:** Four ResNet-style blocks [3] with skip connections and two conv layers each: `res_blocks.0` (64→128), `res_blocks.1` (128→256), `res_blocks.2` and `res_blocks.3` (256→512). **(3) Dropout:** `Dropout2d(p=0.2)` after the last block for regularization. **(4) Final Projection Head:** `Conv2d(512→256, kernel=3)` → `BatchNorm2d(256)` → `ReLU` → `Conv2d(256→2, kernel=1)`, outputting predictions for `tas` and `pr`.

Training Configuration: The model was trained for a maximum of 20 epochs using the Adam optimizer [5] with a fixed learning rate of 5×10^{-4}. The batch size was 32. Inputs and targets were normalized using channel-wise Z-score statistics.

Performance: While SimpleCNN is effective at capturing spatial features through its deep residual stack, it lacks any temporal modeling and cannot learn seasonality or evolving patterns over time. This makes it a useful baseline for evaluating improvements introduced by temporal models like CNN-LSTM and 3D-CNN. This baseline contains approximately 10.7 million trainable parameters. Trained with a batch size of 32 on an "NVIDIA GeForce RTX 3070 Ti Laptop GPU", it required approximately 2.7 minutes per epoch.

4.1.2 Baseline B: Separable 3D-CNN with Augmented Input Channels

One of my main experiments involved extending the 2D-CNN from the starter code into a 3D convolutional model (developed in the `3d_CNN` branch). This model was designed to capture spatio-temporal correlations across the full input sequence. Key design choices included:

- **Gaussian Noise Augmentation:** Input values were scaled by up to 1% using Gaussian noise to improve generalization through regularization.
- **Causal Temporal Modeling:** Convolutions were restricted to access only past time steps, enforcing causality — a prior grounded in the physical nature of climate prediction.
- **Depthwise-Separable Convolutions:** To reduce parameter count and improve efficiency, I replaced standard 3D convolutions with depthwise separable ones, using 1×1×1 pointwise convolutions to mix channels.
- **Seasonal Encoding:** I augmented the input with two additional channels — `sin_month` and `cos_month` — to help the model learn periodic seasonal patterns.
- **Optimization Strategy:** I used the AdamW optimizer with weight decay for better regularization and convergence.

The architecture and hyperparameter details of my best-performing configuration of this 3D-CNN are summarized below:

- **Training Setup:** 50 epochs, batch size of 4, sequence length of 12, and initial learning rate of 1e-3 with a ReduceLROnPlateau scheduler (factor 0.5, patience = 10).

- **Initial Layer:** Input channels (7) are projected to 256 via a 3D convolution, followed by `BatchNorm3d` and ReLU.
- **Backbone:** A stack of 4 residual blocks, each consisting of a dilated 3D convolution, batch normalization, and ReLU activation. Skip connections help maintain gradient flow and expand the temporal receptive field without increasing parameter count.
- **Final Layers:** A dropout layer (rate = 0.4) followed by a 1×1×1 convolution projects the features to the output variables (`tas`, `pr`).
- **Kernel Size:** All convolutions use a kernel size of 3 across all dimensions.

This model had 369k trainable parameters, taking 2.7 minutes per epoch (RTX3070 w/ 8 GB VRAM)

4.2 Evaluation

Models were evaluated using the official Kaggle competition metric, which is a weighted average of three sub-metrics computed for `tas` and `pr`: overall area-weighted RMSE, time-mean area-weighted RMSE, and time-std-dev area-weighted MAE. During development, I used validation loss as my primary indicator for model selection and hyperparameter tuning, as the test set targets were corrupted.

Monthly RMSE:

$$monthly\_rmse(Y) = \sqrt{\frac{1}{T \cdot H \cdot W} \sum_{t=1}^{T} \sum_{h=1}^{H} \sum_{w=1}^{W} (\hat{y}_{t,h,w} - y_{t,h,w})^2}$$

Time-Mean RMSE:

$$time\_mean\_rmse(Y) = \sqrt{\frac{1}{H \cdot W} \sum_{h=1}^{H} \sum_{w=1}^{W} \left(\bar{\hat{y}}_{h,w} - \bar{y}_{h,w}\right)^2}$$

Time-Stddev MAE:

$$time\_stddev\_mae(Y) = \frac{1}{H \cdot W} \sum_{h=1}^{H} \sum_{w=1}^{W} |s_{\hat{y},h,w} - s_{y,h,w}|$$

4.3 Implementation Details

In this section, I focus specifically on the implementation and training details of my final model, the proposed CNN-LSTM architecture (described in sec. 3.1), as this was the core of my design exploration. I intentionally did not perform extensive tuning for the two baselines (`SimpleCNN` and 3D-CNN), reserving my effort for refining the main model.

Computational Setup

I trained and evaluated the final CNN-LSTM model using a single NVIDIA RTX 3070 GPU with 8GB of VRAM. My best run early-stopped after 80 epochs, with each epoch taking approximately 1.75 minutes to complete. The model had around 8.2 million trainable parameters.

Model and Training Configuration

The finalized model used a sequence length of 12 months (after sweeping from 3 to 36), which effectively encodes one full year of input. A batch size of 4 was used across all training runs.

All relevant hyperparameters, including settings for the convolutional layers, LSTM, attention, and training procedure—are summarized in Table 1. These reflect the outcome of iterative tuning and optimization tailored specifically to this architecture.

Hyperparameter Tuning Strategy

I adopted a greedy, sequential tuning strategy due to the large number of interacting hyperparameters. After sweeping the basic hyperparameters, I swept one or two hyperparameters at a time, fixed the best-performing values, and then proceeded with the next sweep. I did this because sweeping many hyperparameters at once came with troubles. For example, I found that the optimal learning rate could shift by more than 100× if bidirectionality was enabled or the projection factor was modified.

I ultimately chose not to use bidirectional LSTMs, as they violate my assumption of causality in climate systems. Gradient clipping was fixed at 5 and not tuned further due to time constraints.

Table 1: Final hyperparameter configuration for the CNN-LSTM model.

Parameter Group	**Hyperparameter**	**Value**
Data	Sequence Length	12
	Batch Size	4
CNN	Initial Channels (`cnn_init_dim`)	128
	Depth (`cnn_depth`)	2
	Kernel Size	9
	Dropout (`cnn_dropout_rate`)	0.4
	Activation	SiLU
Pooling & Projection	Pool Size (`average_pool_size`)	4
	Projection Factor	1.0
LSTM & Attention	Hidden Dim (`lstm_hidden_dim`)	256
	Layers (`n_lstm_layers`)	2
	Dropout (`lstm_dropout`)	0.1
	Attention Heads	8
Training	Optimizer	Adam
	Learning Rate	5e-5
	Weight Decay	2.4e-5
	LR Scheduler Patience	5
	Precision	16-mixed

4.4 Results

My final CNN-LSTM model (described in section 3) significantly outperformed the baselines. It achieved a public Kaggle score of **0.6965** and a private score of **0.7776**, placing 5th. Table 2 summarizes the validation metrics for my final model and my Baseline A & B. The CNN-LSTM demonstrates substantially lower (better) errors across nearly all metrics, particularly the time-mean RMSE for `tas`, indicating superior ability to capture long-term climate averages.

Table 2: Comparison of final validation metrics between the 2D-CNN (Baseline A), 3D-CNN (Baseline B), and my proposed CNN-LSTM model. Lower is better for all metrics.

Variable	**Metric**	**Baseline A (2D-CNN)**	**Baseline B (3D-CNN)**	**Final Model (CNN-LSTM)**
`tas`	Time-Stddev MAE	0.6080	0.4465	**0.2027**
	Time-Mean RMSE	1.1524	2.2704	**0.4314**
	Monthly RMSE	2.3448	2.7353	**1.2035**
`pr`	Time-Stddev MAE	0.8543	**0.6873**	0.7589
	Time-Mean RMSE	0.3867	0.5792	**0.2429**
	Monthly RMSE	2.0905	2.1358	**1.9956**
Validation Loss		0.1817	0.1987	**0.1594**
Public Kaggle Score		2.0058	2.4030	**0.6965**

4.5 Ablations

The majority of the features I decided to implement were based on comparing otherwise identical models with and without that feature. A couple example cases of ablation done after the final submission are the kernel size and the use of attention. Table 3 presents a quantitative comparison of my final model against two ablated versions—one without the multi-head attention layer and another using a smaller 3×3 convolution kernel instead of 9×9. These comparisons isolate the impact of each component on model performance.

Table 3: Comparison of final validation metrics between my final model unaltered, with a 3x3 kernel, and without the use of attention. Lower is better for all metrics.

Variable	Metric	Final	No Attention	3x3 Kernel
`tas`	Time-Stddev MAE	0.2027	**0.1751**	0.2282
	Time-Mean RMSE	0.4314	**0.3933**	0.4182
	Monthly RMSE	1.2035	**1.1990**	1.2012
`pr`	Time-Stddev MAE	0.7589	**0.7389**	0.7528
	Time-Mean RMSE	0.2429	0.2466	**0.2357**
	Monthly RMSE	1.9956	1.9973	**1.9942**
Training Loss		0.1298	0.1345	**0.1281**
Validation Loss		**0.1594**	0.1597	**0.1594**

In this case, I can see that disabling attention is considerably beneficial for predicting tas, while swapping back to a 3x3 kernel is marginally better for predicting pr. This highlights a drawback of my general methodology of ”finalizing” a small range of optimal values for hyperparameters one at a time, and a difficulty that comes along with increasing quantity of hyperparameters. The result also strongly indicates that it would be beneficial to separate the prediction of tas and pr. Other ablations include comparing activation functions (ReLU, SwiGLU, SiLU, and GeGLU) and precision types.

5 Discussion

My work demonstrates that a carefully designed hybrid CNN-LSTM architecture can serve as a high-fidelity climate emulator. The explicit separation of spatial and temporal feature extraction proved highly effective. The most impactful techniques were the introduction of LSTM layers to model sequential data and the expansion of the CNN's receptive field via larger kernel sizes. One surprising finding was how sensitive hyperparameter tuning was; the optimal learning rate, for instance, could change by orders of magnitude when a single architectural feature like bidirectionality was toggled. This underscores the need for isolated, systematic tuning.

Limitations: My primary bottleneck was the computational cost of hyperparameter sweeping. The greedy, sequential approach to tuning is very unlikely to find the global optimum. Furthermore, Optuna's database schema is tied to specific versions and search spaces, which created friction when I wanted to add new features or change tuning ranges, often forcing me to restart the study.

Future work: One of my largest blunders was not honing in more on the loss function for training, especially since the metrics used to determine the Kaggle score were given to us right in the project description. If I were to redo the project, I would delegate a significant portion of time to finding or crafting a better loss function (Perhaps for each variable). Furthermore, I only had one normalization method in my data pre-processing, which I could have added to given more time.

I also had a separate branch (also on a CNN-LSTM) that added several forms of data augmentation, but I did not have sufficient time to explore that hyperparameter space. Given more resources and time, I would experiment with incorporating some of those into my main model.

In addition to that, I could have tried using different optimizers like RMSProp or SGD.

For Future Work, I was also thinking of implementing a U-Net + CNN-LSTM model, which is a SOTA model for weather and climate spatiotemporal modeling.

6 Contributions

I independently completed all core components of this project, including the model architecture, training pipeline, experiments, and final report:

- **Devam Derasary:** Led the design and implementation of the CNN–LSTM architecture for climate emulation, including preprocessing pipeline modifications for temporal input handling, decoder strategy experiments, and adaptive learning rate scheduling. Built and tested multiple model variants, including a 3D-CNN with data augmentation and depthwise-separable convolutions. Conducted extensive hyperparameter optimization using tools like Optuna and managed all experimentation on GPU hardware. Additionally, explored a Transformer-based architecture for spatiotemporal modeling. Authored the final report and presentation, and managed model evaluation and logging via Weights Biases.

7 Acknowledgements

In addition to all of the paper being cited and course materials, this paper has also been written with the help of ChatGPT (4.0) and Gemini (2.5). ChatGPT 4.0 has been used to help polishing the paper by fixing the grammar, rephrasing some vague sentences and brainstorming the concept. However, I was the one who made the decisions and interpretations on this paper.

7.1 Reference Statement for ChatGPT (4.0) Use

ChatGPT was used for general coding and debugging assistance, and referenced for suggestions to better the model.

7.2 Reference Statement for Grok (3.0) Use

Grok 3.0 was used for general coding and debugging assistance, and in particular helped with feature implementation in the 3D-CNN model.

7.3 Reference Statement for Gemini (2.5) Use

Gemini 2.5 was used for general coding and debugging assistance, and in particular helped with the setup of the Gemini-CNNLSTM-Test branch, alongside LR scheduling and Early stopping.

7.4 Reference Statement for Claude (3.7-sonnet and 4.0-sonnet) Use

Claude-3.7-sonnet and Claude 4.0-sonnet helped with general debugging and the implementation of the Hydra Optuna hyper-parameter sweeper in the Gemini-CNNLSTM-Test branch. It also helped with refactoring code, implementing different activation functions, and hyperparameterizing features across multiple branches.

References

[1] Takuya Akiba, Shotaro Sano, Takeru Yanase, Toshihiko Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2623–2631, 2019.

[2] Stefan Elfwing, Eiji Uchibe, and Kenji Doya. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning. *Neural Networks*, 107:3–11, 2018.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, pages 770–778, 2016.

[4] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

[5] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[6] Stephan Rasp, Michael S Pritchard, and Pierre Gentine. Deep learning to represent subgrid processes in climate models. *Proceedings of the National Academy of Sciences*, 115(39):9684–9689, 2018.

[7] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.