UNIVERSITY OF NAPLES

"FEDERICO II"

SCHOOL OF POLYTECHNIC AND BASIC SCIENCES

EDUCATIONAL AREA OF MATHEMATICAL, PHYSICAL, AND NATURAL SCIENCES

DEPARTMENT OF PHYSICS "ETTORE PANCINI"

MASTER'S DEGREE IN DATA SCIENCE

Deep learning-based reconstruction of missing data in spatiotemporal sequences

Relators:
Hon. Profs.
Giuseppe Longo
Massimo Brescia
Hon. Dr.
Bruno Buongiorno Nardelli

Candidate:
Francesco Astarita
ID No. P 37/115

Abstract

The analysis of spatiotemporal data is fundamental in many scientific disciplines, yet the presence of missing data poses significant challenges, particularly in high-resolution satellite datasets such as sea surface temperature data from radiometers measuring in the infrared/microwave bands. This work proposes the use of a spatiotemporal neural network based on convolutional and long short - term memory layers (ConvLSTM2D) to reconstruct missing data, leveraging spatial and temporal information to fill gaps caused by cloud cover or instrumental limitations.

The methodology includes:

Data preprocessing, involving the computation of anomalies relative to seasonal climatology and the creation of land-sea masks to separate valid pixels from unusable ones. Sequential preparation, with 15-day temporal windows and spatial tiling optimized for training. Model design, combining ConvLSTM2D layers to capture temporal dynamics and Conv2D layers for spatial refinement.

The model was trained on observed data and validated using metrics such as root mean squared error (RMSE) and qualitative analyses. Preliminary results demonstrate the network capability to reconstruct sea surface temperature anomalies and preserve small scale details. This approach opens new perspectives with respect to traditional methods, such as optimal interpolation, showcasing the effectiveness of deep neural networks in the processing of complex satellite datasets.

Summary

Introduction

The analysis of spatiotemporal data plays a vital role in various scientific disciplines, including oceanography, meteorology, and machine learning. Satellite observations, in particular, are essential tools for monitoring and understanding natural phenomena, climate patterns, and dynamic processes on a global scale. Despite their importance, these datasets present significant challenges, the most notable one being the recovery of missing data due to instrumental and/or physical limitations of the observing systems. The presence of such gaps can severely affect the accuracy of both analyses and predictions. This problem is especially relevant in satellite observations of the sea surface temperature (SST) as those provided by the Copernicus Marine Service, which is the European leading source of satellite-based information for marine environmental monitoring, as shown in figure 1.

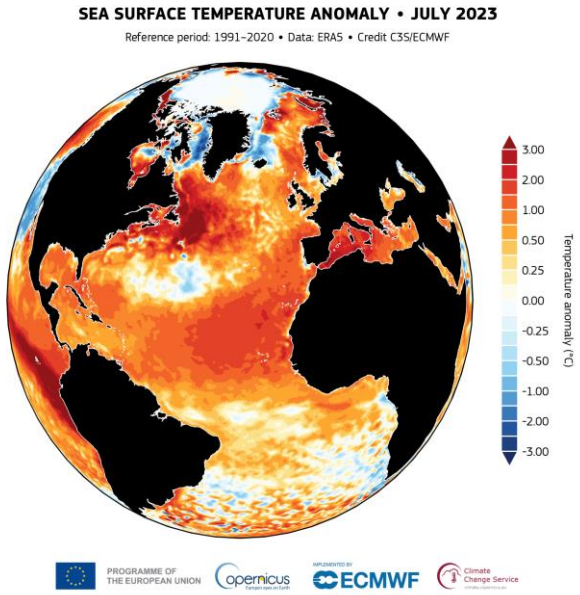


Fig. 1 *- Sea surface temperature anomaly (°C) for July 2023, relative to the 1991-2020 reference period. Data source: ERA5. Credit: Copernicus Climate Change Service/ECMWF.*

(credits: https://climate.copernicus.eu/global-sea-surface-temperature-reaches-record-high?os=roku...&ref=app)

The Copernicus Mediterranean Sea Surface Temperature (hereafter SST_MED) "supercollated" product (Copernicus service product ID: SST_MED_SST_L3S_NRT_OBSERVATIONS_010_012), as shown in fig. 2, delivers high-resolution, daily records covering Mediterranean Sea.

Generated through the integration of multi-sensor observations from satellite instruments like Sea and Land Surface Temperature Radiometer (SLSTR), Visible Infrared Imaging Radiometer Suite (VIIRS), and Advanced Very High Resolution Radiometer (AVHRR), this dataset is classified as a "supercollated" Level 3 (L3S) and is organized in two datasets at different nominal resolutions, one delivered on a 1/16° spatial resolution grid and the other reaching 0.01°. Despite the richness of data provided, one of its major limitations is the high rate of missing values, which often reaches between 60% and 80%.

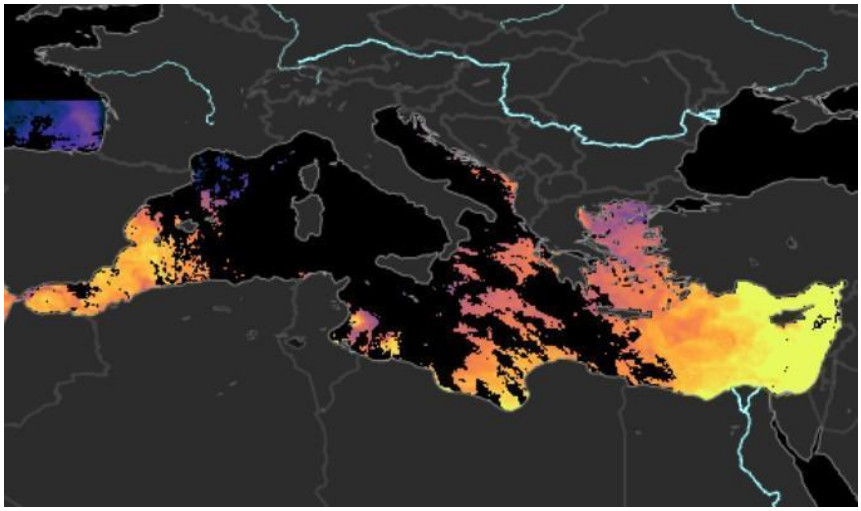


***Fig. 2** Mediterranean Sea - High Resolution and Ultra High Resolution L3S Sea Surface Temperature*

(credits: https://data.marine.copernicus.eu/product/SST_MED_SST_L3S_NRT_OBSERVATIONS_010_012/description)

This is largely due to cloud cover, which obstructs the sea surface view from satellite sensors, especially those operating in the thermal infrared, as well as to other instrumental and satellite mission constraints (e.g. repetitiveness of the orbit, field of view of the antennas, etc.).

Traditional interpolation techniques, commonly employed to address this space-time coverage issues, often fall short when it comes to capturing the intricate spatiotemporal variations inherent in surface ocean dynamics.

These challenges highlight the need to develop and test more advanced methodologies.

Taking advantage of the recent advancements of machine learning/artificial intelligence techniques, this study focuses on the development of a deep learning approach to address the problem of missing data in the SST_MED dataset. Specifically, the approach leverages spatiotemporal models, such as ConvLSTM2D networks, to integrate both spatial and temporal information for reconstructing data gaps. Neural networks offer a significant advantage over conventional methods by their capacity to learn complex patterns and correlations across multiple dimensions. This makes them a particularly promising solution for overcoming the limitations associated with traditional techniques.

The primary objectives of this study can be summarized as follows. Pixels are categorized into three main types: valid or full pixels, which contain meaningful data; empty or flagged pixels, which represent missing information, often due to cloud cover; and land pixels, which correspond to non-oceanic regions. To facilitate subsequent analyses, a mask is built to distinguish between valid sea observations, invalid pixels and land areas.

Another critical objective is the preprocessing of data. This involves calculating anomalies for each pixel by subtracting the seasonal mean value (or climatology) from the corresponding day's measurement. This step aims to learn only the deviations from expected values, reducing the amount of information needed to retrieve at least the background large scale patterns. The dataset is structured with two key channels: anomaly maps and sea/land masks. Additionally, strategies such as normalization are intended to reduce the range of input values so that training becomes more efficient.

Preliminary data analysis is also carried out to better understand the dataset's characteristics. This includes calculating the percentage of cloud cover for each pixel, analyzing the average duration of cloud cover persistence, and determining statistical distributions that provide insights into data variability and patterns.

Once the data have been analyzed and preprocessed, the next phase involves preparing it for model training. This step requires structuring the dataset into temporal sequences over a predefined period—typically fifteen days (selected based on the previous analyses defining characteristic data voids space-time patterns). The input data for these sequences include anomalies and sea/land masks, while the target data consist of the central day in the sequence. This central day is masked to simulate real-world conditions of missing data caused by cloud cover in the predictor sequences. Basically, by training the model on these sequences, it learns how to reconstruct missing information based on the temporal and spatial patterns present in the surrounding data.

The neural network architecture is designed to handle both spatial and temporal information. The model incorporates convolutional networks to learn spatial features and either recurrent models or Transformers to capture temporal dependencies. Once the architecture is established, the model is trained using the prepared sequences, with the goal of optimizing its ability to reconstruct missing data accurately.

Finally, the structure of this document is organized to provide a comprehensive overview of the research process. Section 2 presents a review of the literature, focusing on both traditional and advanced methods for addressing missing data in spatiotemporal datasets. Section 3 describes the dataset in detail, along with the preprocessing techniques employed. Section 4 outlines the model architecture and implementation strategies. The

results of the study are discussed in Section 5, followed by the conclusions and future perspectives in Section 6.

This document is intended to offer a clear and detailed account of the methodology, highlighting the innovations and contributions made through the application of deep learning to climate data reconstruction. By doing so, it provides a foundation for further research and development in this rapidly evolving field.

Theoretical background

Optimal Interpolation: a quick overview

Optimal Interpolation (OI) is a statistical method used to reconstruct missing data by combining observations with background information through a weighted averaging process. It relies on predefined covariance structures to estimate missing values, ensuring smooth and consistent fields. While widely used in oceanography (e.g., Copernicus SST products), OI assumes linear relationships and often oversmooths small-scale features, limiting its ability to capture complex, dynamic ocean patterns.

Applications in Copernicus SST Products

In operational settings such as Copernicus Marine Service, covariance models are carefully calibrated to reflect the physical behaviour of various oceanic regions, including the Mediterranean and Black Seas. The spatial scales of these models are adapted to the specific characteristics of different environments; for example, shorter correlation lengths are used in highly variable coastal zones, while longer ones are applied to more stable open ocean areas (Buongiorno Nardelli et al., 2013). Temporal scales are similarly adjusted, aligning with seasonal and diurnal cycles to ensure that interpolated fields reflect realistic fluctuations.

Applications to Satellite Data

One of the most significant applications of OI is in the generation of Level 4 (L4) products for SST within Copernicus, as shown for example in figure 3. These products provide seamless, high-resolution datasets that fill gaps in raw satellite measurements (Yang et al., 2021). The process begins with Level 2 (L2) data, consisting of geophysical variables retrieved directly from raw satellite images that are often sparse and subject to noise. These are then pre-processed into Level 3 (L3) products, where multiple sensor inputs are combined to a common geographical grid, although gaps may still be present (Buongiorno Nardelli et al., 2013). The final step involves the application of OI, which

Commentato [1]: questa citazione non va bene. cita piuttosto questa (ovunque nel testo al posto di Huang):Yang, C., Leonelli, F. E., Marullo, S., Artale, V., Beggs, H., Nardelli, B. B., Chin, T. M., de Toma, V., Good, S., Huang, B., Merchant, C. J., Sakurai, T., Santoleri, R., Vazquez-Cuervo, J., Zhang, H.-M., & Pisano, A. (2021). Sea Surface Temperature intercomparison in the framework of the Copernicus Climate Change Service (C3S). Journal of Climate, 1–102. https://doi.org/10.1175/jcli-d-20-0793.1

integrates L3 data with background fields and covariance models to produce L4 datasets that are both complete and physically consistent .

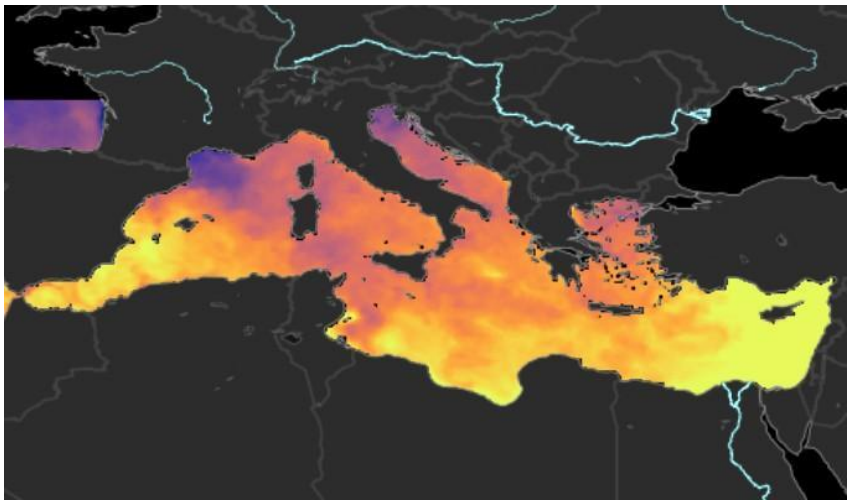


Fig 3 - *Mediterranean Sea ultra high resolution SST observation*

(credits: https://data.marine.copernicus.eu/product/SST_MED_SST_L4_NRT_OBSERVATIONS_010_004/description?view=-&option=-&product_id=-)

Visualization and Comparison

Copernicus provides visual tools to assess the effectiveness of the interpolation process. By comparing raw L3 SST observations with their interpolated L4 counterparts, it becomes evident how OI successfully fills data gaps while maintaining the integrity of underlying patterns. Validation efforts, such as comparisons with in situ buoy measurements, further demonstrate the accuracy of OI-based reconstructions (Yang, C., Leonelli, F. E., Marullo, S., Artale, V., Beggs, H., Nardelli, B. B., Chin, T. M., de Toma, V., Good, S., Huang, B., Merchant, C. J., Sakurai, T., Santoleri, R., Vazquez-Cuervo, J., Zhang, H.-M., & Pisano, A., 2021). Notably, SST estimates in the Mediterranean show bias reductions below 0.1°C, along with improved root mean square differences (RMSD) when compared to ground-truth data (Buongiorno Nardelli et al., 2013). Additionally, OI excels in preserving mesoscale and sub-mesoscale features, which are crucial for understanding complex ocean dynamics .

Theoretical foundations and statistical roots

Deep learning builds upon classical statistical principles while offering significant advantages in modeling capacity and flexibility. Traditional methods, such as regression, kriging, and Optimal Interpolation (OI), rely on predefined models with specific assumptions about the underlying data distribution (Bishop, 1995). In contrast, deep learning thrives on directly extracting patterns from data, allowing for more adaptable and expressive representations (Goodfellow, Bengio, & Courville, 2016).

One of the primary distinctions lies in the nature of modeling assumptions. Statistical methods often impose strict conditions such as linearity or stationarity, making them less effective in handling highly non-linear or dynamic data. Neural networks, however, are largely assumption-free, making them capable of approximating highly complex functions that traditional models struggle to capture (Haykin, 1999). Additionally, while conventional methods tend to be static, deep networks can dynamically adapt to changing data distributions, making them more suitable for modern, large-scale datasets.

At the heart of deep learning models are artificial neurons, inspired by biological neurons. Each artificial neuron receives multiple inputs, applies weights to them, adds a bias term, and processes the result through an activation function.

This can be expressed mathematically as:

$$y = \sigma\left(\sum_{i=1}^{n} w_i x_i + b\right).$$

Here, x_i are the input features, w_i are the corresponding weights, b is the bias term, and σ is the activation function (e.g., ReLU, sigmoid, or tanh).

The role of the activation function is critical, as it introduces non-linearity, enabling the network to model complex relationships in the data.

This capacity is formalized by the **Universal Approximation Theorem**, which states that a neural network with at least one hidden layer and a non-linear activation function can approximate any continuous function on a compact subset of real numbers, given sufficient neurons and appropriate weights (Cybenko, 1989; Hornik, Stinchcombe, & White, 1989). This theoretical result underscores the potential of neural networks to model virtually any function, regardless of complexity. However, in practice, realizing this potential depends on factors such as network architecture, training data, and optimization strategies. Insufficient data, poor design choices, or inadequate training can hinder the network's ability to generalize, despite its theoretical capacity to approximate complex functions (Goodfellow, Bengio, & Courville, 2016).

Historical Overview of Deep Learning

Deep learning has its roots in the mid-20th century, with the emergence of artificial neural networks (ANNs), as in figure 4. The first conceptual model, known as the perceptron, was introduced by Frank Rosenblatt in 1958. Inspired by the functioning of biological neurons, the perceptron demonstrated the ability to learn simple linear patterns through adjustable weights. However, its limitations quickly became apparent, particularly in its inability to handle problems requiring non-linear separations (Minsky & Papert, 1969). Despite these shortcomings, it laid the theoretical foundation for future developments in neural network research.

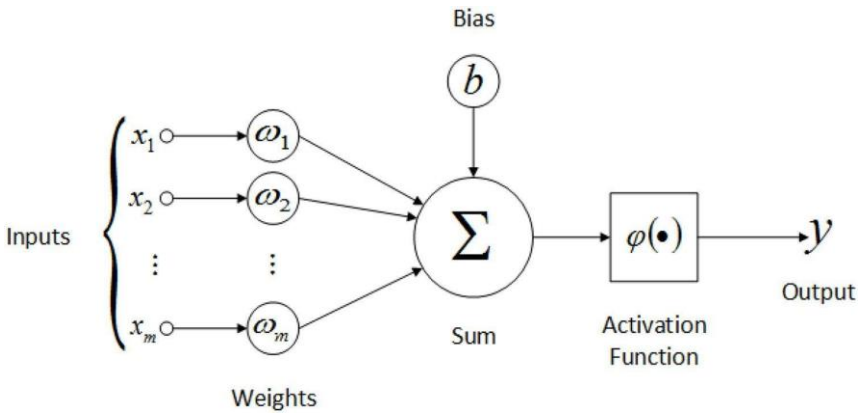


Fig. 4 *– artificial neuron representation*

credits: https://dzone.com/articles/the-artificial-neural-networks-handbook-part-4

A major breakthrough came in the 1980s with the introduction of backpropagation, a method for efficiently computing gradients in multi-layer networks. Proposed by Rumelhart, Hinton, and Williams in 1986, backpropagation (figure 5) enabled multi-layer perceptrons (MLPs) to model complex, non-linear relationships, effectively addressing some of the perceptron's fundamental limitations. This advancement reignited interest in neural networks and helped establish them as powerful tools in computational learning. In the following decades, these methods evolved into the broader field of deep learning, as highlighted in recent overviews by researchers such as LeCun, Bengio, and Hinton (LeCun, Bengio, & Hinton, 2015).

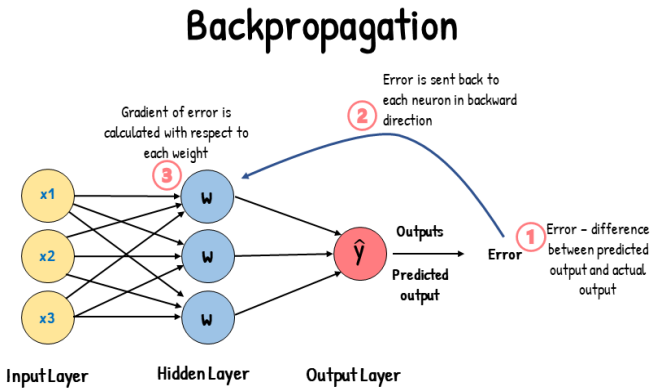


Fig. 5 *- backpropagation diagram*

(credits: https://www.analyticsvidhya.com/blog/2023/01/gradient-descent-vs-backpropagation-whats-the-difference/)

The Transition from Shallow to Deep Architectures

The late 1990s and early 2000s saw a gradual shift toward deeper architectures, driven by advances in computational power and the increasing availability of large datasets. During this period, convolutional neural networks (CNNs), pioneered by Yann LeCun in the 1990s, became a cornerstone of image processing, leveraging spatial hierarchies in data to improve feature extraction (LeCun, Bottou, Bengio, & Haffner, 1998). Meanwhile, recurrent neural networks (RNNs) gained traction for sequential data processing, enabling applications such as speech recognition and time-series forecasting.

The development of Long Short-Term Memory (LSTM) networks by Hochreiter and Schmidhuber in 1997 provided a solution to some of the limitations of earlier RNNs by addressing issues related to long-term dependencies (Hochreiter & Schmidhuber, 1997).

Despite these advancements, deep learning faced significant obstacles. The problem of vanishing gradients hindered the training of deep networks, a challenge extensively studied by Bengio, Simard, and Frasconi in the mid-1990s (Bengio, Simard, & Frasconi,

1994). Limited datasets and hardware constraints further restricted large-scale applications, making deep learning impractical for many real-world scenarios. These challenges slowed the widespread adoption of deep learning, keeping it largely confined to research settings until further breakthroughs in the 2010s (Schmidhuber, 2015).

The Deep Learning Renaissance

The 2010s marked a turning point, ushering in a new era of deep learning fueled by three key factors: the explosion of digital data across various domains, advancements in computational hardware—particularly the widespread adoption of GPUs for accelerating matrix operations—and algorithmic innovations that improved the stability and efficiency of training. Techniques such as dropout (Srivastava et al., 2014), batch normalization (Ioffe & Szegedy, 2015), and more advanced optimizers like Adam (Kingma & Ba, 2015) mitigated overfitting and accelerated convergence, making deep learning models more practical and effective.

This period saw groundbreaking breakthroughs that demonstrated the true potential of deep architectures. One of the most pivotal moments was the introduction of AlexNet in 2012 (figure 6, below), which showcased the power of deep CNNs in the ImageNet competition and dramatically improved the state-of-the-art in image recognition (Krizhevsky, Sutskever, & Hinton, 2012). Meanwhile, the development of long short-term memory (LSTM) networks provided a solution to the vanishing gradient problem in RNNs, enabling the processing of long-sequence data with greater accuracy and reliability (Hochreiter & Schmidhuber, 1997).

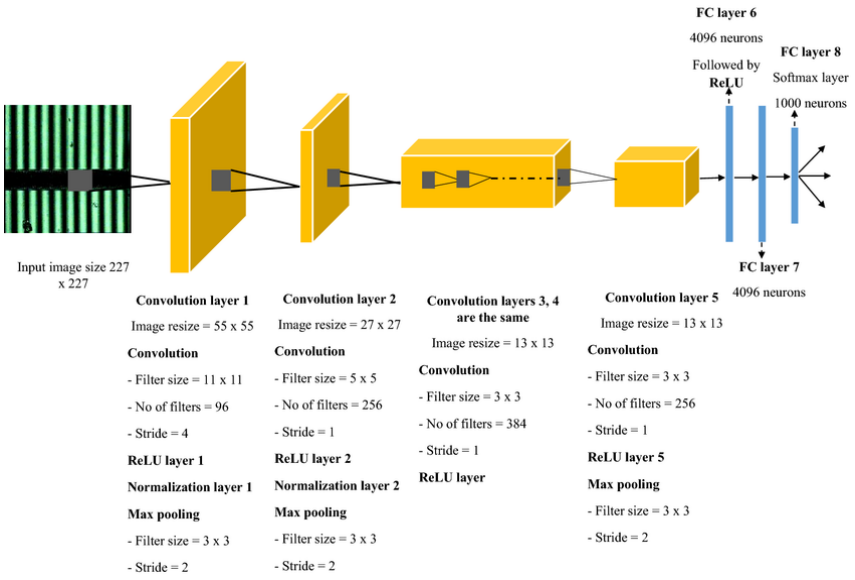


Fig. 6
The architecture of AlexNet CNN (https://www.researchgate.net/figure/The-architecture-of-AlexNet-CNN_fig9_339705064)

Loss Functions in Deep Learning

Loss functions play a crucial role in guiding the training process by quantifying the difference between predicted and actual values (Goodfellow, Bengio, & Courville, 2016). One of the most commonly used loss functions in regression problems is the Mean Squared Error (MSE), which calculates the average squared difference between predicted and true values:

$$L_{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

where y_i represents the true value and $\hat{y}_i$ denotes the predicted value. MSE is particularly useful for applications such as temperature prediction and dynamic topography estimation, although it is highly sensitive to outliers due to the squared term (Murphy, 2012).

Loss functions are fundamental in both traditional machine learning and deep learning, serving as optimization objectives that guide the learning process (Hastie, Tibshirani, & Friedman, 2009). The selection of an appropriate loss function directly impacts model performance and generalization.

Optimization Algorithms in Deep Learning

Optimization methods are essential for adjusting model parameters to minimize loss functions. One of the most fundamental approaches is Stochastic Gradient Descent (SGD, as shown in figure 7), which updates parameters iteratively using the formula:

$$\theta \leftarrow \theta - \eta \nabla_\theta L(\theta)$$

where η represents the learning rate. While simple and effective for small datasets, SGD can converge slowly and is highly sensitive to hyperparameter tuning (Ruder, 2016).

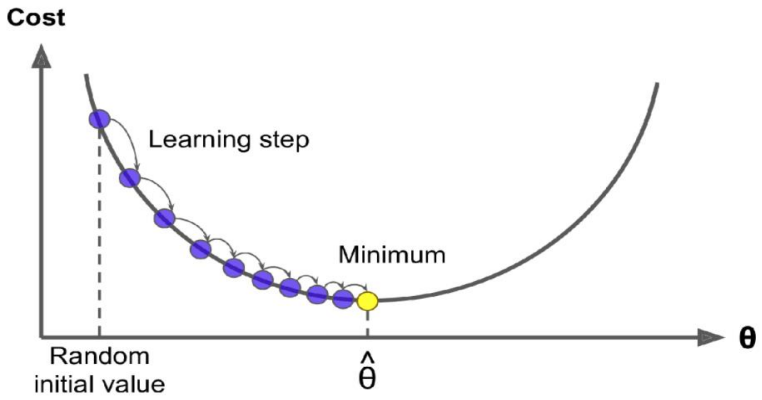


Fig 7 - *Iterations in gradient descent towards the global in this case min*

credits: https://pantelis.github.io/cs634/docs/common/lectures/optimization/sgd/

A more sophisticated approach is Adam (Adaptive Moment Estimation, figure 8), which incorporates momentum and adaptive learning rates to improve convergence:

$$\theta \leftarrow \theta - \eta \left(\frac{m_t}{\sqrt{v_t} + \epsilon} \right)$$

where m_t and v_t represent estimates of the first and second moments of the gradients. Adam is particularly well-suited for training complex models on large datasets with sparse gradients (Kingma & Ba, 2015).

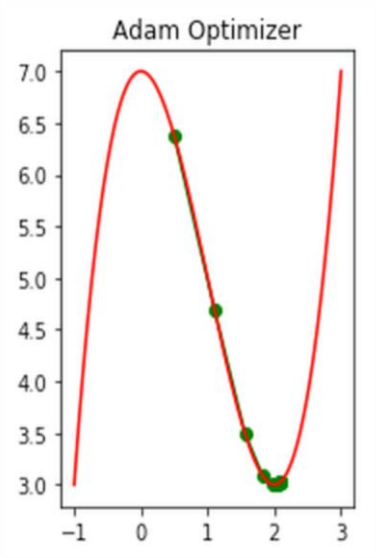


Fig. 8 *- Adam optimizer*

(credits: https://medium.com/ai%C2%B3-theory-practice-business/adam-optimization-algorithm-in-deep-learning-9b775dacbc9f)

Another widely used method is RMSProp, which adapts the learning rate based on the magnitude of recent gradients:

$$\theta \leftarrow \theta - \eta \left(\frac{\nabla_\theta L}{\sqrt{\mathbb{E}\left[\nabla_\theta L\right]^2 + \epsilon}} \right)$$

ensuring more stable convergence in non-stationary environments (Tieleman & Hinton, 2012.

Convolutional Neural Networks (CNNs): a fundamental approach to spatial feature extraction

Convolutional Neural Networks (CNNs) have revolutionized the field of machine learning, particularly in applications dealing with structured spatial data such as images, medical scans, and geospatial information. The origins of CNNs trace back to the 1980s and 1990s, when Fukushima (1980) introduced the Neocognitron, an early model capable of hierarchical feature extraction through convolutional layers. This idea was further refined by LeCun et al. (1989, 1998) with the development of LeNet-5, a pioneering CNN architecture designed for handwritten digit recognition. Their work demonstrated the efficacy of convolutional layers in automatically learning spatial patterns, reducing the need for manual feature engineering (LeCun et al., 1998).

The fundamental operation of a CNN is the convolution, where a small matrix of trainable weights, called a kernel (figure 9), slides over an input image or structured grid. Each convolutional operation produces an output known as a feature map, highlighting specific characteristics such as edges, textures, or more abstract patterns at deeper layers (Krizhevsky, Sutskever, & Hinton, 2012). The strength of CNNs lies in their ability to progressively learn hierarchical representations of data, where shallow layers capture low-level features, and deeper layers extract high-level abstractions (Simonyan & Zisserman, 2015).

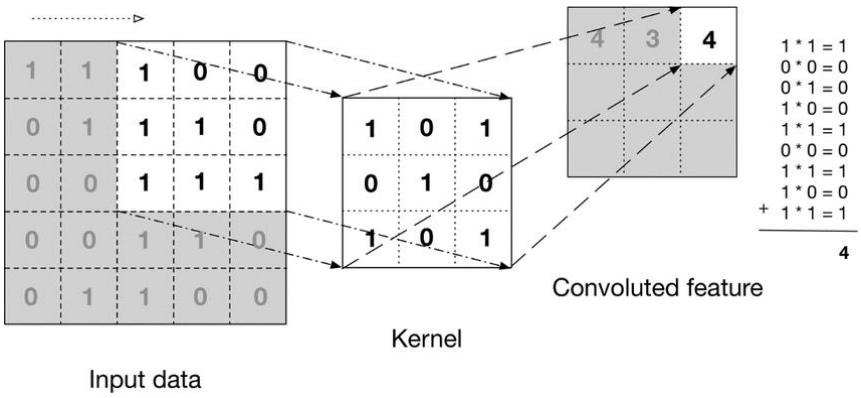


Fig. 9 - *An example of discrete convolution with a 3x3 kernel*

credits: https://www.researchgate.net/figure/Discrete-convolution-with-a-3x3-kernel_fig3_335609766

However, a crucial limitation of standard convolutions is their **fixed receptive field**, meaning that a kernel only considers a small neighborhood of pixels at each step. To address this, **Yu & Koltun (2016)** introduced **dilated convolutions**, which expand the receptive field by inserting gaps between kernel elements. This **dilation rate** determines how far apart the sampled input values are, allowing the model to aggregate multi-scale spatial context without increasing the number of parameters.

Technically, the **dilation rate** is a parameter in convolutional layers that controls the spacing between kernel elements, effectively expanding the **receptive field** of the convolution without increasing the number of parameters or the computational cost significantly. It is particularly useful for capturing patterns at different scales without requiring larger kernels or additional pooling layers.

In a standard convolution (**dilation rate = 1**), the kernel operates on contiguous pixels. However, when the dilation rate is greater than 1, the kernel elements are spaced apart, allowing the model to **skip over intermediate pixels** and effectively "see" a larger region of the input at once.

How does dilation affect feature extraction?

- **Dilation rate = (1,1) (Standard convolution)** → The kernel applies a convolution over **adjacent pixels**, preserving fine details but with a limited receptive field.
- **Dilation rate = (2,2) or higher** → The convolution **skips pixels**, expanding the receptive field and allowing the model to recognize broader patterns without increasing kernel size or adding more layers.

The figure 10, below, provides a good example.

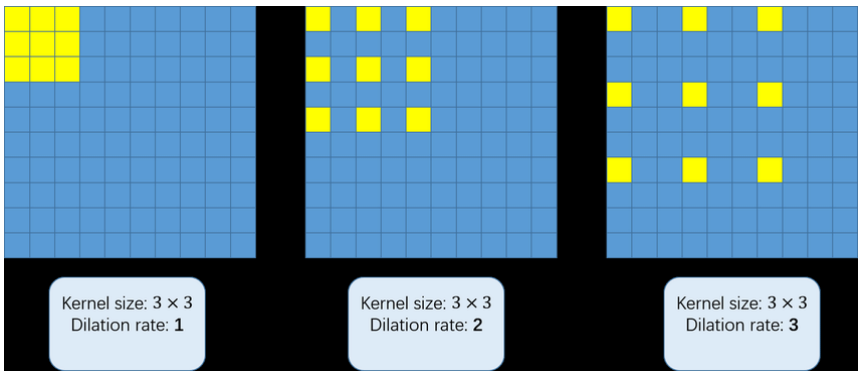


***Fig. 10** - Kernel size 3 x 3 with 3 differents dilation rates*

credits: https://www.researchgate.net/figure/3-convolution-kernels-with-different-dilation-rate-as-1-2-and-3_fig9_323444534

Beyond their application in image recognition, CNNs have proven instrumental in analyzing gridded datasets to build regression models, including those learning from remote sensing images and oceanographic data. Their ability to capture spatial dependencies makes them particularly well-suited for climate modeling starting from observed data, sea surface temperature (SST) analysis, and environmental monitoring in general(Goodfellow, Bengio, & Courville, 2016). Unlike traditional machine learning

approaches that rely on handcrafted features, CNNs learn feature representations directly from raw data, making them highly adaptable to complex, non-linear patterns.

As CNNs became more sophisticated, researchers sought ways to integrate them with sequential models for handling spatiotemporal data, leading to architectures such as ConvLSTM2D, which combines convolutional layers with recurrent structures to model both spatial and temporal dependencies (Shi et al., 2015). These advancements have broadened the scope of CNN applications, making them indispensable tools in modern deep learning.

Long Short-Term Memory (LSTM) networks and their extension to ConvLSTM2D

Understanding Long Short-Term Memory (LSTM) networks

Recurrent Neural Networks (RNNs) have long been the standard architecture for processing sequential data, as they are specifically designed to model temporal dependencies. RNN are designed to deal with sequential data by taking information from previous computational step as input to the successive one. They are based on sequentially connected cells (one per each element in the input sequence).

However, traditional RNNs suffer from a fundamental limitation known as the **vanishing gradient problem**. When processing long sequences, the gradients used for updating the network weights tend to shrink exponentially, making it difficult for the network to retain information from earlier time steps. This drawback severely hampers their ability to model long-range dependencies, which are crucial in many real-world applications, including climate modeling and oceanographic studies.

Long Short-Term Memory (LSTM, introduced by Sepp Hochreiter and Jürgen Schmidhuber, 1997) networks were introduced to overcome this issue by incorporating a memory cell capable of storing and retaining information across long sequences. The LSTM architecture is built around three primary **gates** that regulate the flow of information:

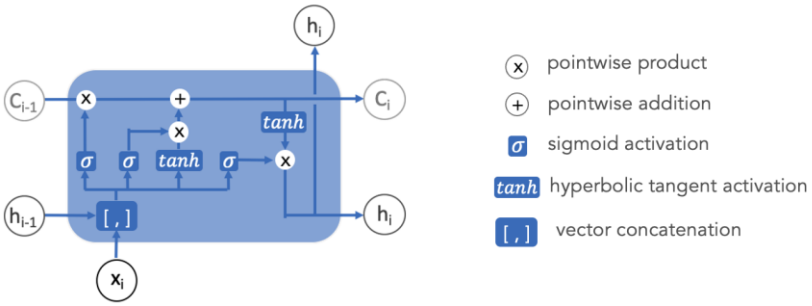


Fig. 11

The Forget Gate decides what portion of the past information should be discarded.

The Input Gate determines what new information should be stored in the memory cell.

The Output Gate controls which information is used to generate the current output.

These mechanisms allow LSTM networks to selectively **retain relevant information while discarding unnecessary details**, making them exceptionally powerful for sequential data modeling. In the context of satellite-derived sea surface temperature (SST) data, this is particularly advantageous, as oceanographic patterns often exhibit long-range dependencies that need to be effectively captured for accurate reconstruction.

From LSTM to ConvLSTM2D: integrating spatial and temporal dependencies

While LSTMs effectively capture temporal dependencies, they inherently **lack spatial awareness**, as they process sequences as one-dimensional data. This is a critical limitation when dealing with geospatial datasets such as SST fields, where both spatial and temporal structures are crucial for meaningful reconstructions.

To address this, Shi et al. (2015) introduced **Convolutional LSTM (ConvLSTM2D)**, an extension of LSTMs that **incorporates convolutional operations within the LSTM structure**. Instead of treating input data as a one-dimensional sequence, ConvLSTM2D processes data as a series of 2D spatial grids over time.

The key difference lies in how information is processed:

Instead of using fully connected layers, ConvLSTM2D **applies convolutional operations** to extract spatial features while preserving temporal dependencies.

The **hidden states and memory cells are now three-dimensional tensors**, allowing the model to process spatially structured data across multiple time steps.

The **gates (forget, input, output)** perform **convolutional transformations** instead of simple matrix multiplications, ensuring that local spatial relationships are preserved.

Here, each operation is performed in a **spatially-aware manner**, meaning that instead of working on isolated features, the model processes structured spatial grids, which is crucial when handling satellite images of SST anomalies.

Why ConvLSTM2D is a good choice for Sea Surface Temperature Reconstruction

The problem of reconstructing missing SST data due to cloud cover and sensor limitations presents two major challenges:

1. **Temporal Continuity:** The missing information must be inferred based on previous and future observations, ensuring that the reconstructed values align with real oceanic processes rather than being mere interpolations.
2. **Spatial Consistency:** Oceanographic structures, such as mesoscale eddies, temperature gradients, and coastal upwellings, exhibit strong spatial correlations. A meaningful reconstruction must preserve these patterns, avoiding excessive smoothing or unrealistic anomalies.

ConvLSTM2D is particularly well-suited to addressing these challenges because:

1. **It captures both spatial and temporal dependencies:** Unlike traditional interpolation methods, which only use static spatial relationships, ConvLSTM2D **learns how temperature patterns evolve over time** while maintaining spatial coherence. This allows the model to make more accurate and physically meaningful predictions.
2. **It adapts to complex non-linear relationships:** Traditional interpolation methods often assume simple statistical relationships between neighboring points. However, oceanic dynamics are highly non-linear, influenced by factors such as currents, winds, and thermohaline circulation. ConvLSTM2D, being a deep learning-based approach, can **learn complex, non-linear dependencies** that classical methods fail to capture.

Technically, for example, in **ConvLSTM2D**, the kernel operates not only in space, as in standard convolutions, but also in time, enabling the network to **store and update information from previous frames**. Each **convolutional filter** applies a kernel to a local region of the input, and the kernel weights are optimized during training to capture patterns useful for prediction tasks.

The **size of the kernel** (e.g., **3×3 or 5×5**) determines **how large the input region is for each convolution operation:**

Larger kernels (e.g., 5×5) → Capture broader patterns **Smaller kernels (e.g., 3×3)** → Improve **precision on fine details** but might struggle to capture larger-scale structures.

Understanding the Role of Convolutional Layers and Filter Selection

In convolutional neural networks (CNNs), convolutional layers play a crucial role in extracting meaningful spatial features from input data. Unlike traditional fully connected layers, where each neuron is connected to every input unit, a **Conv2D layer** applies a set of learnable filters that move across the input, detecting patterns such as edges, textures, and broader spatial structures. These filters act as **feature detectors**, selectively emphasizing important regions of the data while suppressing irrelevant information.

Each filter in a Conv2D layer is optimized during training through backpropagation, meaning that the network learns which spatial features are most relevant for the task at hand. The number of filters, therefore, determines how many distinct spatial features the network can capture. A **larger number of filters allows for a richer, more diverse feature representation**, but this comes at a cost: an excessive number of filters can introduce **redundancy, increase computational complexity, and even lead to overfitting**, where the model memorizes patterns rather than generalizing effectively.

By setting the first Conv2D layer to **64 filters**, we assumed that the model could extract enough distinct spatial features while avoiding unnecessary redundancy. This decision allowed us to strike a **practical balance between accuracy and efficiency**, ensuring that the network was expressive enough to capture key oceanographic structures without becoming overly complex at this stage.

Purpose of the Two-Filter Output Layer

While the first convolutional layer is responsible for learning rich feature representations, the final Conv2D layer in our architecture serves a very different purpose: it must produce a structured, interpretable output that aligns with the task's objectives. Given that our model is designed to reconstruct two distinct outputs—SST anomalies and a land-sea mask—the final convolutional layer has exactly two filters.

However, although the model generates two output channels, the **loss function optimizes only the reconstruction of SST anomalies**, as the land-sea mask is not explicitly considered in the training loss except for the identification of the open ocean areas to be used to estimate the Loss metrics.

Materials and Methods

Data Source

The data utilized for this research was obtained from the Copernicus Marine Service , specifically from the product SST_MED_SST_L3S_NRT_OBSERVATIONS_010_012, which was processed by the Consiglio Nazionale delle Ricerche - Istituto di Scienze Marine - Gruppo di Oceanografia da Satellite (CNR-ISMAR-GOS) institute in Rome. This dataset was selected due to its high spatial resolution and the robust validation methods employed during its creation, ensuring both accuracy and reliability for our analysis.

Technologies for SST Data Collection in the Mediterranean Sea

The sea surface temperature (SST) data used in this study were obtained from the SST_MED_SST_L3S_NRT_OBSERVATIONS_010_012 product, provided by the Copernicus Marine Service (Copernicus). This dataset is classified as super-collated Level 3 (L3S) since it consists of integrated observations from multiple satellite sensors. L3S data are the result of a fusion process involving multi-source satellite measurements that are pre-processed to correct systematic errors, remove outliers, and improve overall data quality. Compared to raw Level 2 (L2) data, L3 data offer greater spatial and temporal consistency, making them better suited for climatological and modeling analyses.

The satellite observations in the dataset are collected from the following instruments and platforms:

SLSTR (Sea and Land Surface Temperature Radiometer) and Sentinel-3A/B Satellites

The SLSTR is an advanced radiometer mounted on the Sentinel-3A and Sentinel-3B satellites (figure 12), which are part of the European Copernicus program. Copernicus is one of the world’s leading environmental monitoring initiatives, managed by the

European Commission in collaboration with the European Space Agency (ESA). The Sentinel-3 satellites are designed to perform multi-spectral observations of both land and ocean surfaces, providing data on various parameters such as temperature, ocean color, altimetry, and ice cover.

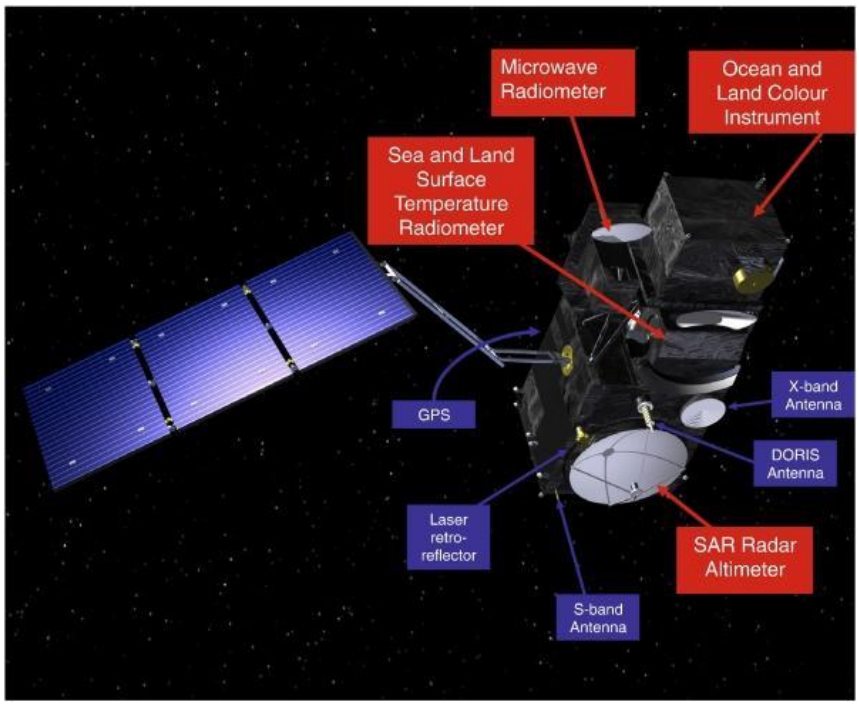


Fig. 12 *- Sentinel 3 - overview*

(credits: https://sentiwiki.copernicus.eu/web/s3-mission)

The SLSTR operates in multiple spectral bands, including:

NIR (Near Infrared): This band is used to detect the reflection of solar radiation from land and sea surfaces and is often combined with other bands to correct for atmospheric influence.

TIR (Thermal Infrared): Thermal infrared bands, centered around wavelengths of 10-12 μm, detect the natural radiation emitted from the sea surface. These bands are crucial for measuring SST, particularly at night, when the absence of solar radiation reduces data distortion.

The SLSTR uses a dual-view scanning system (nadir and inclined) that enhances global coverage and minimizes atmospheric interference. Cross-calibration between the TIR and NIR bands improves the accuracy of temperature measurements.

VIIRS (Visible Infrared Imaging Radiometer Suite) and Suomi-NPP/NOAA-20 Satellites

The VIIRS instrument is installed on the Suomi National Polar-orbiting Partnership (Suomi-NPP) and NOAA-20 satellites, both part of the polar-orbiting weather satellite system managed by the U.S. National Oceanic and Atmospheric Administration (NOAA). Suomi-NPP serves as a bridge between earlier Earth observation systems and the next generation of environmental monitoring satellites.

VIIRS acquires data across a wide range of spectral bands, including both visible and infrared wavelengths. Its TIR bands are essential for measuring SST, while the NIR bands provide information on solar reflection and atmospheric conditions. The spatial resolution of VIIRS ranges from 375 to 750 meters, ensuring high-resolution data suitable for regional-scale studies.

AVHRR (Advanced Very High Resolution Radiometer) and MetOp Satellites

The AVHRR instrument operates on the MetOp satellite series (MetOp-A, MetOp-B, and MetOp-C), which are launched by the European Space Agency (ESA) in collaboration with the European Organization for the Exploitation of Meteorological Satellites (EUMETSAT). AVHRR has been one of the first satellite sensors used operationally for global SST monitoring and atmospheric parameter observation.

AVHRR is a multi-spectral radiometer that includes thermal infrared bands, enabling it to measure radiation emitted by both land and ocean surfaces. Although technologically less advanced than SLSTR and VIIRS, AVHRR remains a valuable data source for long-term environmental monitoring due to its operational continuity and reliability.

Passive Remote Sensing Technology

SST data are acquired through passive remote sensing, a technique that detects the natural radiation emitted or reflected by the Earth's surface. Satellite instruments have the ability to measure this radiation in different spectral bands, including thermal infrared (TIR) and microwaves. Passive remote sensing is particularly advantageous as it does not require an artificial radiation source, thereby reducing both operational costs and environmental impact.

In the TIR bands, the radiation measured is closely related to surface temperature, as described by Planck’s Law. These bands are optimal for SST measurement because they are less influenced by solar reflection, especially during nighttime observations. However, atmospheric conditions can significantly affect the measurements due to absorption and scattering phenomena. To mitigate these effects, raw data undergo radiative transfer corrections that account for local atmospheric variables, such as water vapor concentration.

One of the main challenges of passive remote sensing of the SST is cloud cover, which absorbs and reflects infrared radiation, preventing sensors from directly measuring the sea surface. As a result, SST datasets often contain large percentages of missing data—sometimes between 60% and 80%—in cloud-covered areas. To address this limitation, L3 products are generated through multi-sensor data fusion and algorithms designed to maximize spatial coverage and temporal continuity.

Characteristics of the L3 SST Dataset

The SST_MED_SST_L3S_NRT_OBSERVATIONS_010_012 product is organized into two different datasets providing daily high-resolution files with a spatial grid of 1/16° and 0.01°, respectively. Multi-sensor observations are combined and pre-processed to produce homogeneous maps of sea surface temperature. This data fusion process aims to enhance spatial coverage and data quality while reducing uncertainties related to atmospheric effects and sensor variability.

These L3 SST products are particularly valuable for studying mesoscale ocean phenomena, such as eddies and fronts, and for supporting long-term climatological research. Advanced spatiotemporal reconstruction techniques, including deep learning models, are essential to improve data reliability and fill gaps caused by cloud cover (as shown in the figure 13).

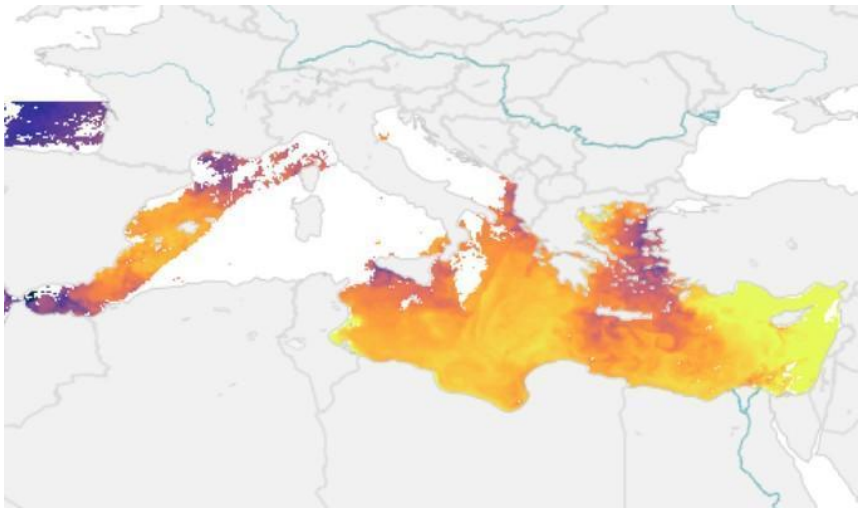


Fig. 13 *- Mediterranean Sea - High Resolution L3S Sea Surface Temperature Reprocessed*
credits: https://data.marine.copernicus.eu/product/SST_MED_PHY_L3S_MY_010_042/notifications

Dataset Characteristics

The dataset comprisesLevel 3S (L3S) satellite observations, which are derived from multi-sensor infrared radiometers. The L3S data, in particular, combines observations from multiple satellite instruments, providing improved coverage and higher spatial resolution compared to single-sensor data. For our study we subsampled the dataset originally provided at 0.01° resolution to a grid regularly spaced at 1/24°. Measurements are focused on nighttime sea surface temperature (SST) to minimize the impact of solar radiation.

Tools and Technologies

1. Data Handling:

- Infrared radiometers to capture SST observations.

- Xarray and NumPy for handling large NetCDF files and performing computational operations efficiently.

2. Processing Algorithms:

- Optimal Interpolation (OI) for creating gap-free SST fields.

3. Hardware:

- High-performance GPUs to train convolutional neural networks (CNNs).

Pre-Processing Steps

The initial step in the workflow involved preparing the L3S data for model training. This included creating masks to identify valid sea pixels, separating them from land and cloud-covered areas. A reference file, specifically L4.nc, was used to generate a static land mask where land pixels were labeled as -1, valid sea pixels were labeled as 1 and invalid values

were labeled 0. This step was essential for ensuring that the dataset accurately represented the oceanic surface.

Temporal Sequence Creation

To reconstruct missing data effectively, the dataset was structured into temporal sequences of 15 days, with the central day (8th day) designated as the target for reconstruction. The choice of a 15-day window was deliberate and based on previous work used to develop OI-based interpolated products (Buongiorno Nardelli et al., 2013), striking a balance between capturing sufficient temporal dynamics and managing computational complexity. Mesoscale and sub-mesoscale oceanographic features (i.e. features characterized by spatial scales of O(1-100 km), which evolve over days to weeks timescales) are well-represented within this window. For each sequence, the central day was masked, simulating the realistic gaps often encountered in satellite observations due to cloud cover or sensor limitations.

The files were preliminary sorted chronologically to ensure consistency in the temporal dimension. For each sequence, 15 consecutive days of data were extracted, with SST and corresponding masks generated for each day. Cloud-covered pixels were identified using the mask, allowing the model to focus on valid sea observations. This approach not only preserved the integrity of the data but also provided a robust input for the model to learn spatiotemporal patterns.

Computation of anomalies with respect to daily climatological fields

Sea surface temperature (SST) climatological anomalies provide fundamental information for oceanographic and climatological studies, as they allow researchers to quantify deviations from expected conditions. Rather than analyzing absolute SST values, which are heavily influenced by seasonal and latitudinal variations, the computation of anomalies enables a clearer identification of patterns linked to specific

oceanographic/climatic events, such as marine heatwaves, transient eddy features, warm/cold anomalies, and also to some extent their broader response associated with climate change. This approach ensures that the focus remains on significant deviations rather than on predictable seasonal cycles, which could otherwise obscure the detection of anomalous behaviors in ocean temperatures. From the perspective of gappy data interpolation, working with anomalies also represents an advantage, as mean patterns and typical temporal signals associated with annual cycles do not need to be learned from the data, reducing the problem complexity to more "local" regressions.

The computation of SST anomalies relies on **removing the seasonal climatological component** from the observed SST values. This is typically achieved by comparing each daily SST measurement against a **corresponding daily climatology**, which represents the long-term average for that specific day of the year. Mathematically, the SST anomaly at a given location and time is computed as:

SST Anomaly = `Observed SST − Climatological SST`

The **climatological SST** is derived from historical records spanning multiple decades, typically over a reference period such as **1985-2015**, as the one used here. These reference periods are chosen to provide a sufficiently stable and representative baseline, ensuring that natural variability is well accounted for while still allowing the detection of longer-term trends and extreme events.

One of the major advantages of using SST anomalies rather than absolute temperatures is that it eliminates the **dominant seasonal cycle** from the dataset. This is particularly beneficial in machine learning applications, as it reduces unnecessary complexity in the input data. Without anomaly computation, the model would need to learn to distinguish between expected seasonal variations and actual anomalies, increasing the risk of misattributing patterns and leading to a less efficient learning process. By focusing directly on deviations from climatological patterns, the model can prioritize detecting meaningful features, such as **the emergence of unexpected warming or cooling events, shifts in oceanic circulation, or the influence of external factors like atmospheric conditions and ocean-atmosphere interactions**.

From a computational perspective, this transformation also plays a crucial role in **data preprocessing and normalization**. In deep learning applications, input features with vastly different ranges can slow down convergence or lead to suboptimal weight updates during training. Since SST values can vary significantly depending on geographical location and seasonal cycles, their direct use in a neural network could introduce unwanted biases. Converting SST into anomalies ensures that the input distribution is more uniform, making it easier for the model to learn meaningful representations without being dominated by location-specific absolute temperature differences.

Furthermore, in the context of **gap-filling models**, such as the one employed in this study, working with anomalies rather than raw temperatures ensures that the reconstruction task focuses on deviations rather than background climatology. When a deep learning model is tasked with predicting missing SST values, it benefits from learning patterns of variability rather than absolute temperature levels, which are largely dictated by well-known climatological processes. This is particularly relevant for applications where cloud-covered satellite observations create missing data gaps, requiring a reliable interpolation based on surrounding spatial and temporal information.

Ultimately, the computation of SST anomalies relative to daily climatological fields serves as a **crucial preprocessing step** in the development of robust and generalizable machine learning models for oceanographic applications. It allows for a more **effective learning process, reduces the impact of seasonal biases, and enhances the interpretability of model predictions** by focusing on meaningful deviations rather than absolute values. Future research may further refine this approach by incorporating **multi-variable anomaly detection**, such as combining SST anomalies with wind speed, salinity, or atmospheric pressure anomalies, to create a more comprehensive framework for understanding oceanic variability.

Tile Creation and Spatial Dimensions

The extracted sequences were further divided into smaller spatial tiles to optimize the training process. Each tile was 250x190 pixels in size. This size has been chosen to guarantee that enough observations are included in input also in presence of large atmospheric perturbations, while still keeping the memory requirements manageable for

GPU-based training. The choice of dimensions was guided by the need to balance spatial coverage with computational efficiency.

To maximize data usage, the tiles were extracted with 50% overlap, ensuring that all valid regions were covered multiple times. This overlap also helped in reducing edge effects during the training phase.

A key criterion for determining the validity of a tile was the proportion of valid sea pixels it contained. Tiles with fewer than 10% valid sea pixels were excluded from the dataset to avoid training the model on unreliable data. The percentage threshold was calculated by identifying pixels classified as sea (non-land and non-cloud) and comparing them to the total number of pixels in the tile. This step ensured that the tiles retained a meaningful amount of oceanic data for reconstruction tasks.

Normalization and Output

Normalization of the SST data was performed to standardize the input values for the neural network, enhancing its ability to learn effectively. For each tile sequence, the mean SST value was subtracted. Missing values, represented as NaN, were replaced with zeros during this process to allow numerical computations.

Beyond centering the data around the mean, we calculated the standard deviation across all sequences **(max_std_seq),** which was saved separately. This value serves to normalize the input data while keeping the information on the upper ocean temperature gradients consistent across the timeseries .

The processed tiles were saved in .npz format, a compact and efficient file format for numerical data. Each file contained the following components:

- Normalized sequence: The input data for training, with anomalies standardized.

- Normalized target: The corresponding target data for reconstruction, standardized using the same statistics.

- Mean and standard deviation: Values used for normalization, enabling future denormalization or analysis.

This file organization allows for streamlined access and handling of data during the training phase. The `npz` format is particularly advantageous in managing large-scale datasets, as it provides high compression and rapid loading times, crucial for iterative training processes.

This detailed process of data preparation and tile generation forms the foundation of the model training phase, ensuring that the input data is both accurate and computationally efficient. Further sections will detail the model architecture and training strategies used in this study.

Training Neural Networks on Satellite Data

This section describes the methodology employed to train our neural network aimed at reconstructing spatiotemporal climate data. The overarching goal is to address gaps in temperature records caused by phenomena such as data corruption or cloud cover. By leveraging sequential satellite observations and an advanced neural network architecture, this project integrates efficient data handling with robust predictive modeling to achieve high-quality reconstructions.

Model Design

The starting model architecture was designed to exploit both the temporal and spatial structure of the data, and slightly different configurations have been tested successively to try to improve model performance.

At its core we have a ConvLSTM2D layer, which processes the sequence of 15 time steps. This layer adapts LSTM architecture to perform convolutional operations over spatial dimensions, processing sequences of 2D spatial data while preserving temporal memory. The predictor data include both temperature anomalies and land-valid-invalid pixel mask as separate channels. Unlike 3D convolutions, which apply kernels across both spatial and temporal dimensions, Conv2D LSTM focuses on modelling temporal evolution without explicitly convolving over the time axis. By setting `return_sequencesFalse`, the ConvLSTM2D layer reduces the temporal sequence to a single temporal output.

Following this, the architecture incorporates two Conv2D layers to refine the spatial representation:

- The first Conv2D layer applies 64 filters with a kernel size of (3, 3) and ReLU activation. This step extracts complex spatial features, enhancing the models ability to capture localized variations within the data.

- The second Conv2D layer reduces the output to two channels using a linear activation function. These two channels correspond to the final predicted anomaly map and the land mask, which are crucial for accurately reconstructing the central days missing data.

The figure 14 serves as a useful example.

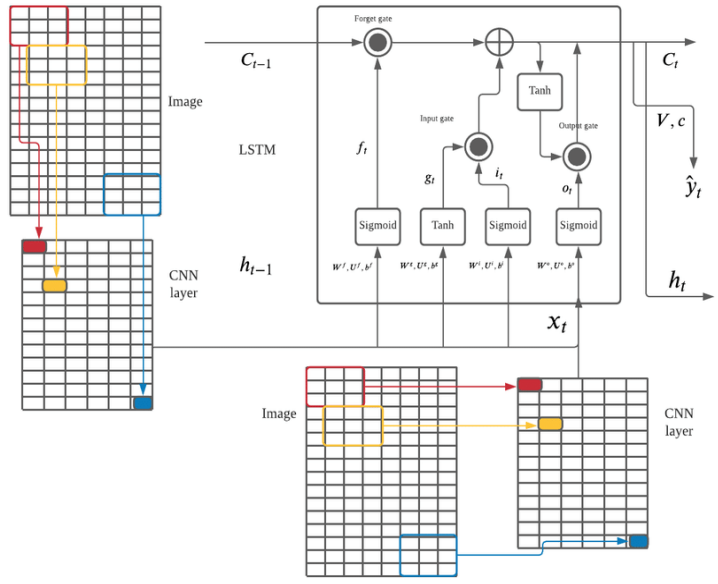


Fig. 14 *- Architecture of ConvLSTM2D in one layer*

credits: https://www.researchgate.net/figure/Architecture-of-ConvLSTM2D-in-one-layer_fig2_356851239

The decision to use two channels reflects the dual requirements of the task: reconstructing the data anomalies while preserving the structural integrity of the land-ocean distinction and only learning from valid SST anomaly data. By integrating temporal processing (via

ConvLSTM2D) with spatial refinement (via Conv2D), the model achieves a balance between capturing high-level patterns and preserving fine-grained details.

Training and validation setup

The dataset was divided into two subsets: 4000 samples for training and 638 samples for independent testing (as shown in fig. 14). This test set is used exclusively after training to evaluate the model's generalization to unseen scenarios, playing a crucial role in verifying model robustness and ensuring it is not overfitted to the training data. It is important to distinguish this from the validation process, which is conducted during training.

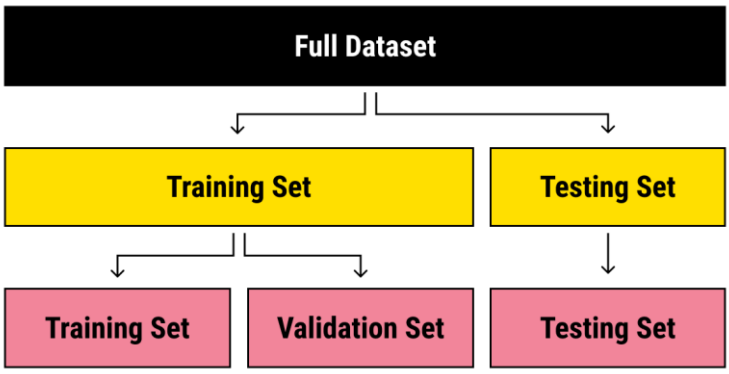


Fig. 15 - *an idea of the splitting full dataset in training and testing set*
credits: https://labelyourdata.com/articles/machine-learning-and-training-data

During training, a portion of the training data is withheld at each epoch for validation purposes. This validation set is not used to update model weights but instead provides feedback on model performance. It helps fine-tune hyperparameters and prevents overfitting by identifying when the model begins to degrade in generalization performance. In this project, validation is performed by sampling sequences from the

training set to evaluate how well the model reconstructs missing values on the central day of a 15-day sequence.

TensorFlow was selected as the development framework due to its powerful ecosystem and efficient handling of large datasets. Its seamless integration with GPU acceleration significantly speeds up the training process, particularly for computationally intensive layers like ConvLSTM2D.

To further enhance the training process, two callbacks were employed:

- EarlyStopping: This mechanism monitors the validation loss and halts training when no improvement is observed for 10 consecutive epochs. By preventing unnecessary iterations, EarlyStopping mitigates the risk of overfitting and optimizes computational resources.

- ModelCheckpoint: This callback saves the model's weights whenever a new minimum validation loss is achieved. This ensures that the best-performing version of the model is preserved, even if subsequent epochs lead to degradation in performance.

Together, these strategies contribute to a stable and efficient training process, reducing the likelihood of overtraining and ensuring the retention of optimal parameters.

The training and validation sequences consist of 15 days, with the model aiming to reconstruct the missing information on the central day. This reconstruction task is vital for handling data gaps caused by cloud cover, leveraging both spatial and temporal patterns. Cloud masks are used to indicate the areas requiring reconstruction, while a land-sea distinction is implemented by setting land pixels to zero, ensuring the model focuses exclusively on relevant sea surface temperature data.

Performance during both validation and testing is assessed using error metrics like Mean Squared Error (MSE) and Mean Absolute Error (MAE). These metrics provide insight into the model's ability to accurately fill missing values. Improvements in these metrics across epochs signal progress in the learning process, guiding the refinement of hyperparameters and architecture where necessary.

Fine-tuning and Evaluation of the ConvLSTM-Based Models for SST Anomaly Reconstruction

The process of optimizing a deep learning model for reconstructing missing SST anomalies required a systematic approach to fine-tuning the model architecture. The initial design centered around a ConvLSTM2D layer, followed by convolutional layers to refine spatial details. However, early experimental results indicated that key architectural components such as kernel size, dilation rate, and feature extraction depth played a significant role in model performance. By adjusting these parameters, we aimed to achieve the best trade-off between bias minimization, reconstruction accuracy, and computational efficiency.

Architectural adjustments and their rationale

The first test concerned the tuning of the **dilation rate** in the first Conv2D layer. The dilation rate controls the receptive field without increasing the number of parameters, allowing the model to capture larger-scale spatial structures. We experimented with **dilation rates of (2,2) and (1,1)**, observing that a higher dilation rate led to enhanced recognition of broad spatial features (in Model 2). The final model (Model 3) also adopted **(2,2) dilation**.

Why does this matter in our model?

In our experiments, we tested **dilation rates of (2,2) and (1,1)** in the first **Conv2D** layer.

- A **higher dilation rate (2,2)** improved the ability to recognize **large-scale spatial patterns**.
- A **lower dilation rate (1,1)** showed lower performance.

The second refinement involved increasing the **kernel size** of the ConvLSTM2D layer from **(3×3) to (5×5)** . This adjustment allowed the model to focus on an even larger receptive field, allowing to capture broader spatial dependencies

Model 1 architecture

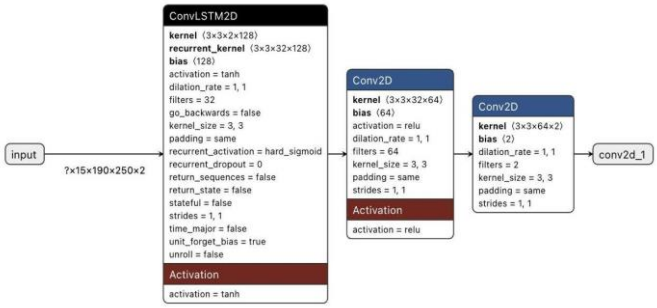


Fig. 16

Model 2 architecture

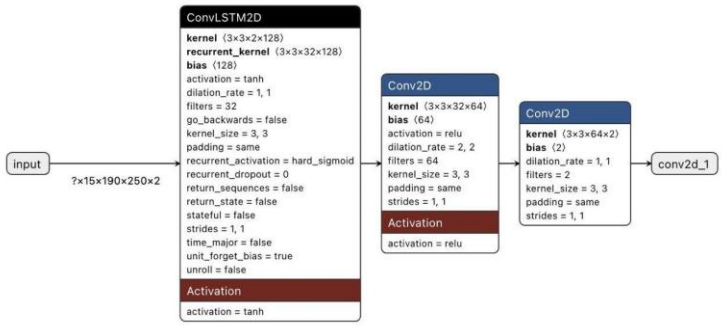


Fig. 17

Model 3 architecture

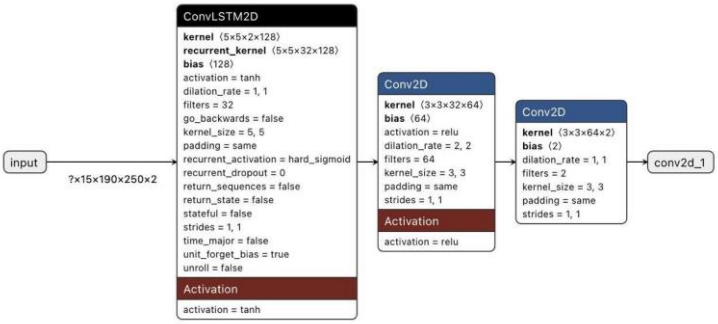


Fig. 18

Ultimately, **Model 3 adopted a dilation rate of (1,1)**, ensuring precise spatial reconstruction while relying on the **ConvLSTM structure to capture broader temporal dependencies**.

Lastly, we optimized the **number of filters in Conv2D layers**. A high number of filters in the first Conv2D layer allows the model to capture diverse spatial patterns, but excessive feature maps can introduce redundancy. Our final model used **64 filters in the first convolutional layer**, followed by a **2-filter output layer**, corresponding to the predicted SST anomalies and the land-sea mask. This design ensured that both spatial and temporal patterns were well captured while maintaining computational efficiency.

Results and discussion

Beyond static performance metrics, it was essential to examine the **training and validation loss curves**, as shown in the figures below. The training behavior of each model provided insight into its generalization ability:

- **Model 1** exhibited fluctuating validation loss, suggesting **higher variance and potential overfitting** in later epochs.
- **Model 2** showed a more stable loss convergence, indicating improved generalization.

- **Model 3** displayed the most **consistent training dynamics**.

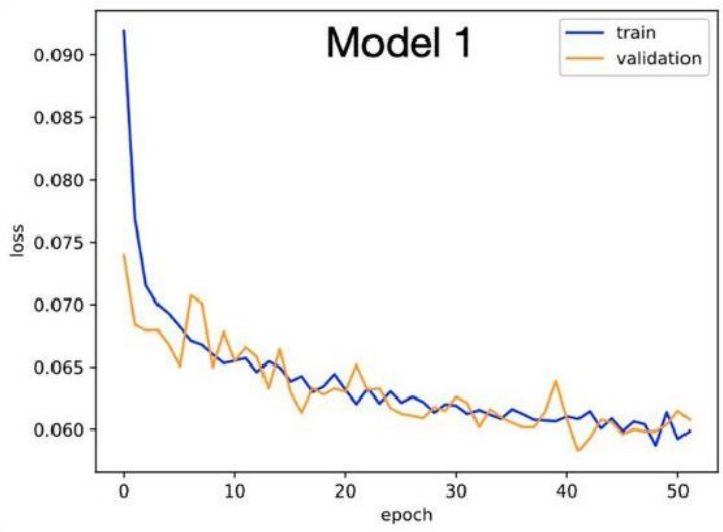


Fig. 19 *- model 1 training and validation loss curves*

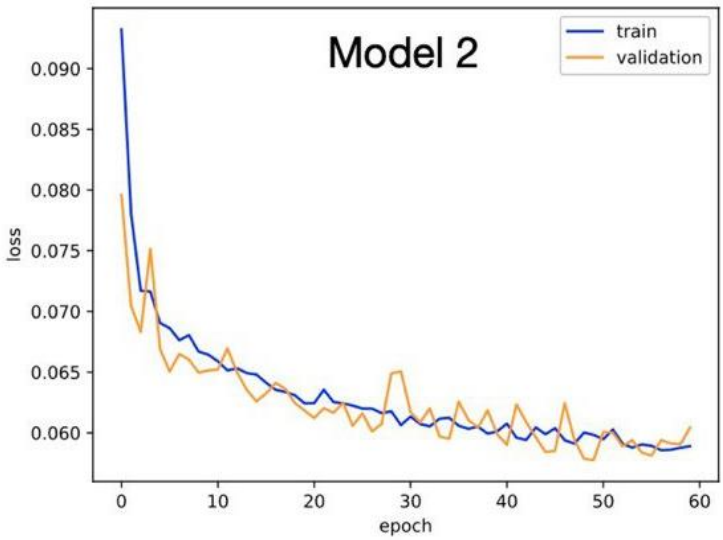


Fig. 20 *- model 2 training and validation loss curves*

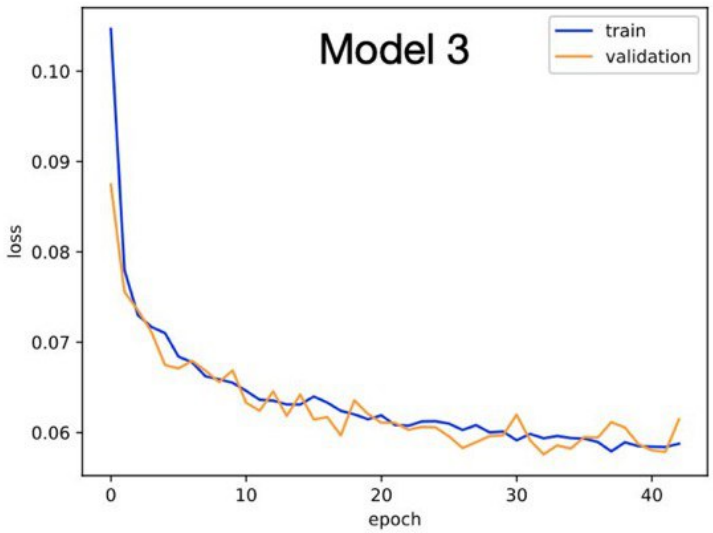


Fig. 21 - *model 3 training and validation loss curves*

Comparative Performance Analysis

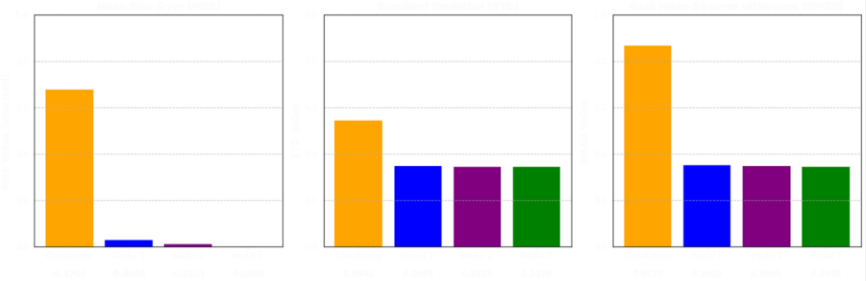
To evaluate the effectiveness of these modifications, we assessed the three models (**Model 1, Model 2, and Model 3**) against a **climatology baseline**.

The benchmark climatology, representing a simple daily mean climatological reconstruction, exhibited a strong negative bias (**MBE = -0.6761**) and high error (**RMSD = 0.8678**), highlighting the need for an advanced learning-based approach.

The deep learning models progressively improved upon this baseline. **Model 1** significantly reduced bias (**MBE = -0.0300**) and error (**RMSD = 0.3502**), demonstrating the advantage of spatiotemporal learning. **Model 2** further refined the predictions (**MBE = -0.0131**, **RMSD = 0.3465**), suggesting improved balance between bias correction and error minimization. **Model 3**, incorporating the final set of architectural optimizations,

achieved a nearly zero bias (**MBE = -0.00096**) and the lowest RMSD (**0.3459**), confirming its superiority in accuracy and reliability.

Model	Mean Bias Error (MBE)	Standard Deviation (STD)	Root Mean Squared Difference (RMSD)
Climatology	-0.6761	0.5441	0.8678
Model 1	-0.0300	0.3489	0.3502
Model 2	-0.0131	0.3462	0.3465
Model 3	**-0.00096**	**0.3459**	**0.3459**



Evaluation and Visualization

After training, the model was evaluated on the test set to assess its predictive accuracy. Predictions for the central time step were compared against the ground truth values, with the Root Mean Squared Error (RMSE) used as the primary metric. RMSE quantifies the average deviation of the predictions from the actual values, providing an intuitive measure of model performance. In the present exercise, this metric is compared to the RMSE associated with the climatological background, which thus serves as a first order benchmark.

In addition to numerical evaluation, qualitative insights were gained through visualization. Predictions and ground truth data were converted back to sull SST values and displayed side by side. These visual comparisons revealed how well the model reconstructed missing data and highlighted areas where its predictions diverged from the true values. By examining these discrepancies, we can identify potential limitations of the model and inform future improvements.

Considerations

The configuration of our convolutional LSTM and pure convolutional layers—starting with 64 filters for feature extraction and concluding with a two-filter output layer—was primarily determined by practical constraints rather than extensive hyperparameter optimization. Ideally, the number of filters and layers would require dedicated tuning to achieve an optimal balance between expressiveness and efficiency. However, due to computational resource limitations, we were unable to systematically explore alternative configurations.

While the selected architecture performed sufficiently well for our task, it is likely that a more thorough exploration of filter counts and layer depth could further refine the model's ability to capture fine-scale SST anomalies. Future work should investigate the impact of these architectural choices to determine whether alternative configurations could improve predictive accuracy while maintaining computational feasibility.

Fine-tuning and Evaluation of the ConvLSTM-Based Models for SST Anomaly Reconstruction

The process of fine-tuning our ConvLSTM-based models aimed at achieving an optimal balance between accuracy and computational efficiency, ensuring robust reconstruction of missing SST anomaly values. The modifications introduced in Model 3, including the refined kernel size, optimized dilation rate, and improved filter selection, contributed to its superior performance in reconstructing spatial and temporal SST anomalies.

To evaluate the effectiveness of these refinements, we compared the model's output against both the observed target values and the climatological baseline, as shown in the **figure 22, below**. This visualization provides a qualitative assessment of the model's predictive capabilities by displaying:

1. **Target SST values (left panel)**: The actual SST data, which serves as the ground truth. These values were excluded from the input data sequence and have never been seen before by the model
2. **Predicted SST values by Model 3 (center panel)**: The reconstructed SST field as generated by the model, based on the spatiotemporal dependencies learned from the available data.
3. **Climatological SST baseline (right panel)**: The corresponding daily climatological mean, representing an average expected SST for that specific time of year.

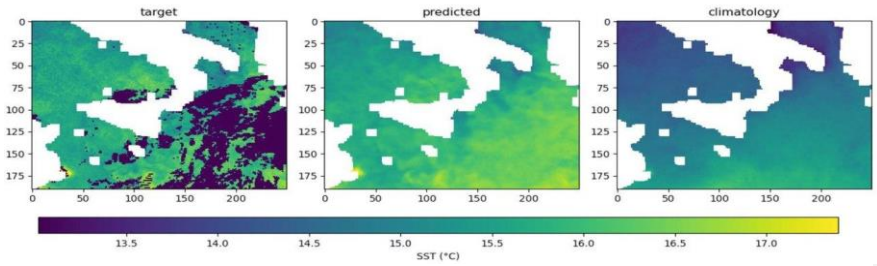


Fig. 22 target vs predicted vs climatology values comparison.

The comparison reveals key insights into the model's reconstruction ability. Unlike the climatology-based estimation, which provides a generalized representation based on long-term averages, Model 3 successfully **captures finer-scale variations in SST patterns**, including localized warming and cooling trends. The reconstructed SST field maintains **spatial coherence**, filling in missing regions with a level of detail that closely resembles the true target values. Additionally, the model effectively preserves **coastal**

and gradient structures, which are typically more challenging to reconstruct due to their higher spatial variability.

Despite these improvements, some discrepancies remain, particularly in regions with high variability, where the predicted values exhibit minor deviations from the ground truth. This is expected, as **the model reconstructs missing values based on learned spatiotemporal dependencies rather than direct observations**. However, compared to the climatological baseline, which lacks the ability to adapt to real-time conditions, Model 3 demonstrates significantly enhanced predictive accuracy.

These results confirm that **fine-tuning key architectural parameters can significantly improve the reconstruction of missing SST values**, enabling the model to better capture short-term fluctuations that would otherwise be smoothed out by climatological estimates. Future enhancements will be needed to further refine these reconstructions by r incorporating more sophisticated model architectures.

Conclusion and future directions

The progressive improvements across the three models demonstrate the impact of architectural design on deep learning-based **SST anomaly reconstruction**. Model 3 emerged as the best configuration, achieving **minimal bias, high accuracy, and strong generalization**. These results highlight the advantages of combining **ConvLSTM for temporal awareness with Conv2D layers for spatial refinement**.

Looking ahead, potential enhancements might include **testing different architectures such as mixed convLSTM2D and U-net and implementing attention mechanisms for selective feature focus**. Ultimately, this study confirms that **deep learning offers a powerful approach for reconstructing missing SST data**, demonstrating strong potential for operational oceanographic applications.

References

Beauchamp, M., Thompson, J., Georgenthum, H., Febvre, Q., & Fablet, R. (2022). Learning neural optimal interpolation models and solvers. *arXiv preprint*, arXiv:2211.07209. https://doi.org/10.48550/arXiv.2211.07209

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, *5*(2), 157–166.

Bishop, C. M. (1995). *Neural Networks for Pattern Recognition*. Oxford University Press.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.

Bouttier, F., & Courtier, P. (1999). Data assimilation concepts and methods. *Meteorological Training Course Lecture Series*. ECMWF. Recuperato da https://www.ecmwf.int/

Buongiorno Nardelli, B., Marullo, S., Santoleri, R., & Tronconi, C. (2022). A new operational Mediterranean diurnal optimally interpolated sea surface temperature product. *Earth System Science Data*, *14*(9), 4111–4130.

Copernicus Marine Service. (n.d.). Global Ocean Gridded L4 Sea Surface Heights and Derived Variables.

Cressie, N. (1990). The origins of kriging. *Mathematical Geology*, *22*(3), 239–252. https://doi.org/10.1007/BF00889887

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, *2*(4), 303–314.

Daley, R. (1991). *Atmospheric data analysis*. Cambridge University Press.

Gaspari, G., & Cohn, S. E. (1999). Construction of correlation functions in two and three dimensions. *Quarterly Journal of the Royal Meteorological Society*, *125*(554), 723–757. https://doi.org/10.1002/qj.49712555417

Ghil, M., & Malanotte-Rizzoli, P. (1991). Data assimilation in meteorology and oceanography. *Advances in Geophysics*, *33*, 141–266. https://doi.org/10.1016/S0065-2687(08)60442-2

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.

Haykin, S. (1999). *Neural Networks: A Comprehensive Foundation*. Prentice Hall.

Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, *9*(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*.

Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *Proceedings of the 32nd International Conference on Machine Learning*, 448–456.

Ito, T. (2019). Optimal interpolation of global dissolved oxygen: 1965–2015. *Geoscience Data Journal*, *6*(2), 105–119. https://doi.org/10.1002/gdj3.130

Johnson, J. E., Lguensat, R., Fablet, R., Cosme, E., & Le Sommer, J. (2022). Neural fields for fast and scalable interpolation of geophysical ocean variables. *arXiv preprint*, arXiv:2211.10444. https://doi.org/10.48550/arXiv.2211.10444

Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. *International Conference on Learning Representations (ICLR)*.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems, 25*, 1097–1105.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444.

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278–2324.

Lorenc, A. C. (1981). A global three-dimensional multivariate statistical interpolation scheme. *Monthly Weather Review*, *109*(4), 701–721. https://doi.org/10.1175/1520-0493(1981)109<0701:AGTMSI>2.0.CO;2

Minsky, M., & Papert, S. (1969). *Perceptrons: An Introduction to Computational Geometry*. MIT Press.

Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective*. MIT Press.

Reynolds, R. W., Rayner, N. A., Smith, T. M., Stokes, D. C., & Wang, W. (2002). An improved in situ and satellite SST analysis for climate. *Journal of Climate*, *15*(13), 1609–1625. https://doi.org/10.1175/1520-0442(2002)015<1609:AIISAS>2.0.CO;2

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, *65*(6), 386–408.

Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint*, arXiv:1609.04747.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, *61*, 85–117.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, *15*(1), 1929–1958.

Tieleman, T., & Hinton, G. (2012). Lecture 6.5 - RMSProp: Divide the gradient by a running average of its recent magnitude. *Coursera: Neural Networks for Machine Learning*.

Yang, C., Leonelli, F. E., Marullo, S., Artale, V., Beggs, H., Nardelli, B. B., Chin, T. M., de Toma, V., Good, S., Huang, B., Merchant, C. J., Sakurai, T., Santoleri, R., Vazquez-Cuervo, J., Zhang, H.-M., & Pisano, A. (2021). Sea Surface Temperature intercomparison in the framework of the Copernicus Climate Change Service (C3S). Journal of Climate, 1–102. https://doi.org/10.1175/jcli-d-20-0793.1

Code Walkthrough and Explanation

When developing machine learning models, training the network is just the beginning of the process. The goal isn't merely to have the model perform well on the data it has seen during training but to ensure that it can handle new, unseen data effectively. This ability to generalize to new data is essential in real-world applications. Without a separate test phase, we would have no reliable way to measure how well the model can generalize. Let's explore why this distinction between training and testing is crucial.

First, there's the problem of overfitting, which is a common challenge in machine learning. During training, the model learns patterns in the data, but if it's not carefully managed, it can also learn irrelevant details or noise specific to the training set. As a result, the model may perform exceptionally well on the training data but fail when confronted with new data because it hasn't learned the underlying general patterns—only the peculiarities of the training set. This is why we test the model on a separate dataset that it has never seen before. By evaluating the model's performance on this test set, we can gauge whether it's truly capturing meaningful patterns or merely memorizing the training data.

Another reason for this separate test phase is to ensure generalization. In machine learning, a model is only considered successful if it can apply what it has learned to new data effectively. The test dataset simulates a real-world scenario by presenting the model with data outside of its training experience. This gives us a clearer picture of how it might

perform once deployed in a real environment. In a production setting, data often contains variations or complexities that the model hasn't encountered during training. The test phase helps reveal potential weaknesses in the model's ability to adapt to such situations.

Furthermore, this testing step is essential for model comparison and selection. During development, it's common to train multiple models or experiment with different architectures and hyperparameters. Without a test phase, we wouldn't have a reliable way to compare these models and decide which one is the best candidate for deployment. Metrics such as RMSE (Root Mean Squared Error) and the R^2 score play a crucial role here. For example, a model might achieve a low RMSE, indicating it can predict values with minimal error, but if the R^2 score is low, it may mean the model is not capturing the variability in the data effectively. Testing multiple models under the same conditions allows us to make informed decisions based on these metrics.

An equally important concern is the avoidance of data leakage. Data leakage occurs when information from the test set unintentionally influences the training process, leading to overly optimistic performance estimates. This can happen in subtle ways, such as reusing the same data for both training and evaluation, causing the model to already "know" the test data. By ensuring that the test dataset is completely separate and unseen during training, we prevent this issue and obtain more accurate performance estimates.

It's also worth noting the distinction between validation and testing. During training, we often use a validation set to fine-tune hyperparameters and evaluate intermediate performance. However, this validation set is still indirectly involved in training decisions and can't provide a fully unbiased evaluation. The test set, on the other hand, is only used after training is complete. It acts as the final benchmark, giving us an objective measure of the model's effectiveness in a production-like scenario.

In this particular code, the test phase serves all these purposes. First, it ensures data independence by evaluating the model on test data that it has never seen before, stored in **list_IDs_test**. The **data_generator** function feeds this test data into the TensorFlow dataset, which is processed efficiently in batches. This setup replicates a production environment, where data is handled in streams rather than being loaded all at once. Once the model makes predictions on this test set, we compare the predictions with the actual ground truth values using key metrics like RMSE and the R^2 score. These metrics are critical in regression tasks, where predicting continuous values accurately is the goal. A low RMSE indicates that the model's predictions are close to the actual values on average, while a high R^2 score demonstrates that the model explains a large portion of the variance in the data.

Together, these steps help ensure that the model is reliable, robust, and ready for real-world deployment. By systematically separating training, validation, and testing phases, we can trust that the performance metrics reflect the model's ability to handle new data. Without this structured approach, we would risk deploying a model that appears to perform well in controlled conditions but ultimately fails in practice. The test phase is our safeguard against such failures, giving us confidence that the model can meet real-world expectations.

This code implements a neural network to process temporal-spatial data using a combination of ConvLSTM and Conv2D layers. The goal is to effectively train and evaluate a model designed to handle complex sequential data with both temporal and spatial dimensions. Below, I provide a thorough, conversational breakdown of each code section to explain not only what each part does but also why it's important and how it contributes to the overall objective.

Importing Libraries

```
import numpy as np
import tensorflow as tf
```

```
from tensorflow.keras import layers, models
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
from tensorflow.keras import backend as K
from sklearn.metrics import mean_squared_error, r2_score
import matplotlib.pyplot as plt
import math
```

First, the script imports several essential libraries. These libraries are fundamental tools in the fields of machine learning, data manipulation, and visualization.

numpy: This is a core library for numerical computing. It offers powerful tools for handling large multi-dimensional arrays and performing mathematical operations.

tensorflow/keras: TensorFlow is a popular deep learning framework, and Keras (built into TensorFlow) simplifies the process of defining and training neural networks.

sklearn: Scikit-learn provides utility functions for evaluating model performance, including the Mean Squared Error (MSE) and R^2 (coefficient of determination) metrics.

matplotlib: This library is used to create visualizations, such as plots of training and validation losses.

math: Provides access to basic mathematical operations, though its use here is minimal.

These libraries form the backbone of the code, enabling everything from data preprocessing to model evaluation.

Utility Function: Finding the Closest Power of 2

```
def closest_power_of_2(n):
    return 2 ** np.round(np.log2(n))
```

This utility function serves a specific purpose: it calculates the nearest power of 2 for any given number `n`. Why is this important? Many machine learning frameworks and algorithms perform more efficiently when working with data sizes that are powers of 2.

This optimization can lead to faster computation and better memory alignment on hardware like GPUs.

For example, if the validation dataset size isn't a power of 2, the function helps round it up or down to the nearest compatible value.

Defining Hyperparameters

```
val_split = .15
n_epochs = 100
batch_size = 4
pat = 10
n_file = 4096
val_size = closest_power_of_2(int(n_file * val_split))
data_size = n_file - val_size
n_batch = data_size / batch_size
```

Here, several key parameters for model training are defined:

val_split: Specifies that 15% of the data will be used for validation. This split allows the model to be evaluated on unseen data during training, helping to monitor overfitting.

n_epochs: The maximum number of times the model will iterate over the entire training dataset.

batch_size: Defines how many samples the model processes at once during training. A smaller batch size like 4 can help when working with large models or limited memory, though it may lead to noisier gradient updates.

pat: This parameter sets the patience level for early stopping. If the validation loss doesn't improve after 10 epochs, training will halt to avoid overfitting.

n_file: The total number of data files available for training.

val_size: The number of files allocated to the validation set, rounded to the nearest power of 2.

data_size: The remaining files used for training.

n_batch: The number of batches per epoch, calculated by dividing the training data size by the batch size.

These hyperparameters are crucial because they define the structure and flow of the training process.

Defining File Paths

```
LOAD_PATH = 'frast/SST_NET/training'
SAVE_FILE                                                              =
'frast/SST_NET/checkpoint/model_checkpoint_b'+str(batch_size)+'_dilate
d_deep_long_5.keras'
model_name                                                             =
'frast/SST_NET/SST_ConvLSTM2D_1layer_b'+str(batch_size)+'_dilated_deep
_long_5.h5'
file_loss_eps                                                          =
'frast/SST_NET/loss_SST_ConvLSTM2D_1layer_b'+str(batch_size)+'_dilated
_deep_long_5.eps'
```

These file paths determine where the data is loaded from and where the model and training results are saved.

LOAD_PATH: Directory containing the training data.

SAVE_FILE: Location where the model checkpoint (best weights) will be saved.

model_name: Final model file path.

file_loss_eps: File path to save a plot of the training and validation loss curves.

Proper file management ensures that training outputs are organized and can be easily retrieved for later use.

Custom Loss Function

```
def custom_mse_loss(y_true, y_pred):
    channel_0_true = y_true[..., 0]
    channel_0_pred = y_pred[..., 0]
    condition = tf.greater(tf.abs(channel_0_true), 0.0)
    mask = tf.cast(condition, tf.float32)
    squared_errors = tf.square(channel_0_true - channel_0_pred)
    masked_squared_errors = squared_errors * mask
    loss = tf.reduce_sum(masked_squared_errors) / (tf.reduce_sum(mask)
+ K.epsilon())
    return loss
```

This function implements a custom version of the Mean Squared Error (MSE) loss. However, there is an important distinction: the loss calculation is masked to only consider relevant data points.

The model focuses solely on the first channel of the target and prediction tensors (`channel_0_true` and `channel_0_pred`).

A condition is applied to create a mask that filters out irrelevant data points where the target is zero.

The masked errors are averaged to compute the final loss value.

This approach ensures that the model does not waste effort on irrelevant parts of the data, which can improve learning efficiency and accuracy.

Data Generator

```
def data_generator(list_IDs, dimX, dimY, shuffle=True):
    max_std_seq                                                    =
np.load(f'{LOAD_PATH}/max_std_seq.npz')['max_std_seq']
    indexes = np.arange(len(list_IDs))
    while True:
        if shuffle:
            np.random.shuffle(indexes)
```

```
        for idx in indexes:
            ID = list_IDs[idx]
            data = np.load(f'{LOAD_PATH}/{ID}.npz')
            X = data['sequence'] / max_std_seq
            Y = data['target'] / max_std_seq
            yield X, Y
```

This function dynamically loads and yields training data in batches. It performs normalization to scale the input data.

Shuffling the data at the start of each epoch helps prevent the model from overfitting to a specific data order.

The generator allows for efficient data handling, especially when dealing with large datasets that cannot fit entirely into memory.

Defining Dataset Parameters

```
params = {
    'dimX': (15, 190, 250, 2),
    'dimY': (190, 250, 2),
    'shuffle': True
}
list_IDs_train = [f"training_data_{i}" for i in range(1, n_file+1)]
train_gen = data_generator(list_IDs_train, **params)
```

The `params` dictionary specifies the input and output dimensions for the data.

A list of training data IDs is generated based on file names.

These parameters guide how the generator prepares and structures each batch of data for training.

Creating the Dataset

```
tf_dataset_train = tf.data.Dataset.from_generator(
```

```
    lambda: train_gen,
    output_types=(tf.float32, tf.float32),
    output_shapes=(
        tf.TensorShape([*params['dimX']]),
        tf.TensorShape([*params['dimY']])
    )
).batch(batch_size).prefetch(1)
```

TensorFlow's `tf.data.Dataset` API converts the generator into a dataset object.

The dataset is batched and pre-fetched to optimize training performance by overlapping data loading and model execution.

Model Definition

```
input_shape = (15, 190, 250, 2)
model = models.Sequential([
    layers.Input(shape=input_shape),
    layers.ConvLSTM2D(filters=32, kernel_size=(5, 5), padding="same",
return_sequences=True),
    layers.ConvLSTM2D(filters=32, kernel_size=(5, 5), padding="same",
return_sequences=False),
    layers.Conv2D(filters=64, kernel_size=(3, 3), dilation_rate=2,
activation='relu', padding='same'),
    layers.Conv2D(filters=2, kernel_size=(3, 3), activation='linear',
padding='same')
])
model.compile(optimizer='adam', loss=custom_mse_loss)
```

The model architecture consists of ConvLSTM2D layers to capture both temporal and spatial features, followed by Conv2D layers to further refine spatial patterns.

The model is compiled with the Adam optimizer and the custom loss function.

Early Stopping and Checkpointing

```
early_stopping = EarlyStopping(patience=pat, restore_best_weights=True,
monitor='val_loss', mode='min')
checkpoint = ModelCheckpoint(filepath=SAVE_FILE, save_best_only=True)
```

Early stopping prevents overfitting by terminating training if the validation loss does not improve after a set number of epochs.

Model checkpoints save the best-performing model during training.

Training and Plotting

```
history = model.fit(train_dataset, epochs=n_epochs,
validation_data=val_dataset, steps_per_epoch=n_batch,
callbacks=[early_stopping, checkpoint])
model.save(model_name)
train = history.history['loss']
val = history.history['val_loss']
plt.plot(train, color='blue', label='train')
plt.plot(val, color='orange', label='validation')
plt.title('ConvLSTM2D - 1 layer')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend()
plt.savefig(file_loss_eps, dpi=150)
```

The model is trained using the specified parameters. Loss curves are plotted to visualize how well the model performs on training and validation data.

Model Evaluation Code with Explanation

Library Imports and Path Setup

This section imports all necessary libraries for model evaluation. TensorFlow manages the neural network's structure, data handling, and evaluation, while numpy and sklearn assist in numerical computations and metric evaluation respectively. The paths for loading

training data and saving the model are defined as global variables for flexibility. The code assumes the presence of a trained model and sets up output paths for results.

```
LOAD_PATH = 'frast/SST_NET/training'
batch_size = 32  # Placeholder for the batch size variable, assumed here
for the example.
SAVE_FILE = 'frast/SST_NET/test/model_checkpoint_TEST_b' +
str(batch_size) + '.keras'
model_name = 'frast/SST_NET/test/SST_ConvLSTM2D_TEST_b' +
str(batch_size) + '.h5'
file_loss_eps = 'frast/SST_NET/test/loss_SST_ConvLSTM2D_TEST_b' +
str(batch_size) + '.eps'
```

Test Dataset Preparation

Here, we are preparing the test data that the model will evaluate. The test data identifiers are generated from a known index range. The custom function 'data_generator' is assumed to provide batches of data based on these identifiers. TensorFlow's 'from_generator' method wraps the generator to create a dataset object, which supports efficient data streaming with batching and prefetching, minimizing I/O overhead during model evaluation.

```
list_IDs_test = [f"training_data_{i}" for i in range(4097, 4638)]

params = {'dimX': (128, 128, 3), 'dimY': (1,)}
test_gen = data_generator(list_IDs_test, **params)

tf_dataset_test = tf.data.Dataset.from_generator(
    lambda: test_gen,
    output_types=(tf.float32, tf.float32),
    output_shapes=(tf.TensorShape(params['dimX']),
tf.TensorShape(params['dimY']))
).batch(batch_size).prefetch(1)
```

Model Predictions

In this step, the pre-trained model is used to predict output values based on the input data from the test set. The 'predict' method processes each batch and returns the predictions as a numpy array. These predictions will later be compared with the actual values to compute evaluation metrics.

Perform model predictions on the test dataset

```
print("Evaluating the model on the test dataset...")
predictions = model.predict(tf_dataset_test)
```

Extracting Actual Values

This section iterates through the 'tf_dataset_test' to extract and collect the actual target values ('y') from each batch. We use the 'as_numpy_iterator' method to convert TensorFlow tensors into numpy arrays for easier manipulation. After collecting all targets, they are concatenated into a single array, allowing us to compare them directly with the model's predictions.

```
actuals = []
for _, y in tf_dataset_test.as_numpy_iterator():
    actuals.append(y)
actuals = np.concatenate(actuals
```

Metric Calculation

This code computes two key evaluation metrics:
1. RMSE (Root Mean Square Error): RMSE is the square root of the average squared differences between predicted and actual values. It measures how far off the model's predictions are on average. Lower values indicate better accuracy.
2. R^2 Score: The R^2 score indicates how well the model explains the variability of the target data. A score of 1 means perfect predictions, while a score of 0 indicates no explanatory power beyond random guessing.

```
rmse = np.sqrt(mean_squared_error(actuals.flatten(),
```

```
predictions.flatten()))
print(f"RMSE on the test set: {rmse}")

r2 = r2_score(actuals.flatten(), predictions.flatten())
print(f"R² Score on the test set: {r2}")
```

The script is written in Bash and is designed for use in a high-performance computing (HPC) environment managed by **SLURM** (Simple Linux Utility for Resource Management). It automates the execution of a Python-based machine learning model training process on a computing cluster. Let's break down the code, explaining each component while also discussing related concepts such as Linux, SLURM, and clusters.

SLURM Directives

The script begins with a set of SLURM directives, denoted by lines starting with `#SBATCH`. These lines instruct SLURM on how to allocate resources and manage the job.

```
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --partition=gpus
#SBATCH --gres=gpu:4  # Request 1 GPU
```

This directive requests **1 node** (i.e., one physical or virtual machine) to run the job. A node is a unit of computational power consisting of CPUs, memory, and potentially GPUs.

`#SBATCH --ntasks=1:`

It specifies that the script requires **1 task** to run. In parallel computing, a task typically corresponds to a process or job that runs independently. Since this script doesn't involve parallel tasks, only one task is requested.

`SBATCH --partition=gpus:`

A **partition** in SLURM is a grouping of nodes configured for specific purposes, such as nodes optimized for GPU computations. This script is requesting access to the GPU partition.

`#SBATCH --gres=gpu:4:`

This line requests **4 GPUs**. The script can potentially run highly parallelized code that leverages GPU acceleration for faster training.

These directives guide SLURM in efficiently managing shared resources in the cluster. Without SLURM, managing resource allocation manually in such environments would be extremely complex.

Setting Up the Environment

The script continues by setting up the necessary software environment.

```
source /nfsexports/SOFTWARE/anaconda3.OK/setupconda.sh
```

This line **sources** a script (`setupconda.sh`), which prepares the system to use **Anaconda**, a popular Python distribution. Sourcing runs a script within the current shell, enabling environment variables and other settings to take effect without spawning a new shell. In HPC workflows, Python environments are often managed with tools like **Conda** to ensure compatibility and reproducibility of software dependencies.

Verifying Python Installation

```
echo "python"
which python
echo
```

Here, the script prints the word `python` and then executes `which python`. The `which` command shows the path to the Python interpreter that will be used. This step is a sanity check to ensure that the correct Python installation is active before proceeding. In cluster

environments, where multiple Python versions and environments may coexist, it's crucial to confirm that the correct interpreter is being used.

Activating a Conda Environment

```
echo "(activate env)"
conda activate tensorflowgpu_xarray
echo
```

The script activates a Conda environment named `tensorflowgpu_xarray`. This environment likely contains the required packages for training the model, such as **TensorFlow** and **xarray**. Activating a Conda environment modifies the shell's environment variables to prioritize the environment's Python installation and libraries.

Launching the Training Script

```
echo "launch SST model training"
python       -u              frast/SST_NET/training_SST_model.py       >
frast/SST_NET/training_SST_model_u4_dilated_deep_long_5.log 2>&1
echo
```

The script launches the Python training script (`training_SST_model.py`) using the following command:

```
python -u frast/SST_NET/training_SST_model.py
```

The `-u` flag tells Python to run in **unbuffered mode**, meaning that output is written directly to the terminal or log file without being delayed in buffers. This ensures that log messages appear in real-time, which is essential for monitoring long-running jobs on a cluster.

The script redirects both **standard output** and **standard error** to a log file:

```
>     frast/SST_NET/training_SST_model_u4_dilated_deep_long_5.log
2>&1
```

The `>` symbol indicates output redirection, and `2>&1` combines standard error (file descriptor 2) with standard output (file descriptor 1). This allows all output (including errors) to be captured in the specified log file for later analysis.

What is SLURM, and why is it used in clusters?

SLURM is an open-source job scheduler designed to manage large-scale clusters efficiently. In HPC environments, hundreds or thousands of users may need access to limited resources (CPUs, GPUs, memory). SLURM allocates these resources fairly and optimizes job scheduling to maximize cluster utilization. It allows users to define job requirements, such as the number of nodes and tasks, through directives like the ones in this script.

By using SLURM, users can submit batch jobs without worrying about resource conflicts or manual resource management. SLURM handles queuing, prioritization, and resource cleanup, making it an indispensable tool in scientific and enterprise computing environments.

Why Linux and why it is used for clusters

Linux is a widely used open-source operating system known for its stability, security, and flexibility. It is the operating system of choice for most HPC clusters for several reasons:

Scalability: Linux can handle thousands of simultaneous users and processes, making it ideal for cluster environments.

Customizability: Linux allows administrators to configure the system to meet specific performance and security requirements.

Open Source: Organizations can modify and optimize Linux for their needs without licensing costs.

Strong Networking and Resource Management: Linux supports advanced networking protocols and tools like SLURM, making it well-suited for distributed computing.

Why use a cluster for machine learning?

Training complex machine learning models, especially deep learning models, requires significant computational resources. Clusters provide access to high-performance hardware, including multiple GPUs and large amounts of memory. By distributing workloads across multiple nodes, clusters can dramatically reduce training time. Additionally, clusters offer robust job scheduling and resource management, ensuring efficient utilization of shared resources.

Conclusions

This script automates the process of launching a deep learning training job on a cluster managed by SLURM. It requests 4 GPUs, sets up the necessary Python environment, verifies the Python installation, and runs a training script while capturing output in a log file. The use of Linux, SLURM, and clusters is essential in this context to efficiently manage resources, handle parallel workloads, and accelerate machine learning research.