Some fun on learning to forecast with FNOs

1st Kshitij Duraphe
Machine Learning Engineer
Absentia Technologies
Boston, MA
kshitijduraphe5@gmail.com

***Abstract*—I construct a FourierVIT model in order to forecast weather data. The model uses a Vision Transformer with a Fourier Neural Operator and ClimaX-style embeddings to embed lead time and the actual atmospheric data into a unified latent space. The model is created with minimal usage of prebuilt PyTorch libraries. The training is distributed using the FSDP framework and optimization techniques such as mixed-precision training and activation checkpointing to lower memory usage while retaining the same level of performance. I present a somewhat detailed analysis of the rationale behind the model architecture and do some analysis on scaling.**

***Index Terms*—physics-informed learning, distributed training, forecasting**

I. Introduction

Here I develop a custom Vision Transformer (ViT) model tailored for weather simulations using the low-resolution HRES T0 dataset [1]. The specific task is to make the model accept a single input frame from the dataset and condition it on a variable lead time (depending on what forecast you want) and generate a corresponding output frame with the same spatial and channel dimensions.

Deep learning methods for weather forecasting have been shown to achieve state-of-the-art performance compared to traditional methods (see [2], [3], [4], [5]). The problem is often cast as an *image-to-image* problem, where the deep learning model learns the underlying structures in the image and perhaps some of the physics as well.

In the following sections, I detail the methodology behind my approach. I begin with an exploratory analysis that involves loading and visualizing a random frame from the dataset, which helps to understand the intrinsic structure and statistical properties of the data. This also helps to answer objective 5 in the notebook.

I then develop a FourierVIT model using ClimaX-style embeddings [2] along with a Fourier Neural Operator (FNO) [6] for forecasting. I choose forecasting as my task over standard pretraining tasks such as Masked Image Modeling /citemim primarily because of my lack of computational resources. The choice of an FNO is motivated by its ability to efficiently capture global dependencies in spatiotemporal data while remaining computationally scalable. Unlike traditional convolutional or recurrent architectures, which struggle with long-range correlations, an FNO operates directly in the frequency domain, enabling it to learn complex spatial transformations with a reduced number of parameters. This spectral approach is particularly well-suited for weather modeling, where atmospheric dynamics exhibit multi-scale structures that are difficult to capture using purely local operations. Additionally, FNOs are inherently mesh-independent, allowing them to generalize across different spatial resolutions—a crucial property for scaling models from lower-resolution weather simulations to high-resolution global forecasting. ClimaX-style embeddings are used to embed both lead time and image data into a shared latent space. This is done to condition the model on the lead_time variable as asked in the notebook.

The corresponding library contains well-commented code that uses as few libraries as possible in the model design. I use the PyTorch [8] framework for my implementation. The code is ready to deploy more or less out-of-the-box depending on the user environment and only requires the download and creation of the dataset.

An important aspect of my approach is scalability. While the current implementation trains on images of size 512×256×98, scaling to images of size 3600×1800×98 introduces significant challenges. The increased resolution would lead to a drastic growth in memory requirements—particularly within the transformer's self-attention layers, which scale quadratically with the number of patches. Additionally, the computational cost for both the Fourier transforms and the patch embedding layers would increase substantially. I address these challenges by employing activation checkpointing, efficient data parallelism using PyTorch's Fully Sharded Data Parallel (FSDP), and mixed-precision training. These strategies not only optimize memory usage but also ensure that the model remains trainable on multi-GPU setups, as demonstrated in the training output. I discuss the implications of activation checkpointing at higher resolutions and identify the self-attention mechanism as the bottleneck due to its quadratic memory and compute complexity. The report also evaluates the feasibility of employing bf16 (bfloat16) precision across different parts of the computation pipeline, outlining which operations can safely benefit from reduced precision without compromising model accuracy.

Lastly, I provide insights into appropriate normalization strategies for the input data, suggesting a method that leverages statistical properties of the dataset to standardize variable ranges, along with mock code snippets to illustrate the normalization process. The report concludes with a discussion on further memory and performance optimizations that could enhance training, fine-tuning, and inference.

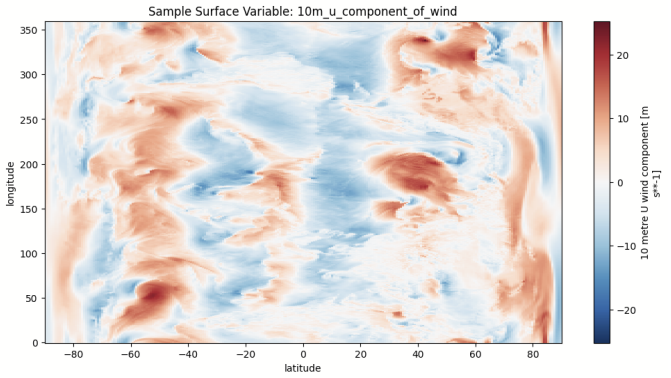

Fig. 1. A sample element om the dataset

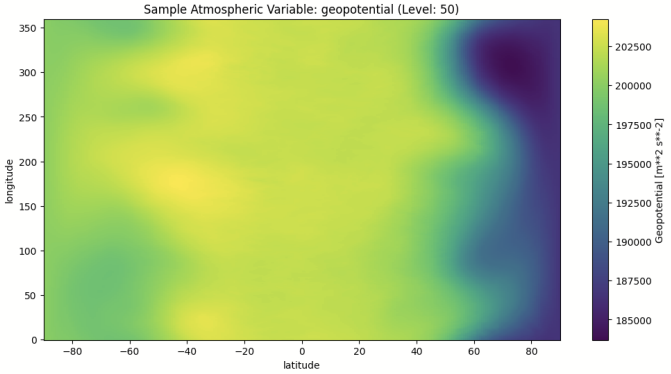

Fig. 2. Another sample element from the dataset

II. Objective 0

Objective 0 asks me to open a random frame in the dataset and visualize it. I have done so in 1 and 2. Since the HRES T0 dataset is 374GB, I used the fsspec library along with xarray and zarr to download only a sample of the dataset and visualize it, as I do not have the storage space to compute on this dataset. Being able to download subsets of the dataset also came in handy when computing the relevant statistics for normalization.

The dataset contains data cubes containing 7 surface variables and 7 atmospheric variables at each timestep. The data are spread across an equiangular lat-lon grid of resolution 256x512. Each atmospheric variable is further divided into 13 layers per variable. This means that at each timestep, we have a 256x512x98 data cube which we have to use for forecasting. Since each variable is the end result of a complex chain of reactions that begin at the Sun and end at the Earth, it is reasonable to assume that many variables will not be directly correlated.

Normalizing and scaling data allows better gradient flow during backpropagation. I download a month's worth of data (specifically, from 2016-01-01 to 2016-02-01) and compute the global mean and standard deviation for each variable across all spatial and temporal dimensions. These statistics provide a basis for standardization, where each variable is transformed using z-score normalization: $X' = \frac{X-\mu}{\sigma}$ where X is the original data, μ is the mean, and σ is the standard deviation. This ensures that the data is centered around zero with a standard deviation of one, improving the numerical stability of the model.

To prevent data leakage, these normalization parameters are computed solely on the training dataset and then applied consistently to validation and test sets. Additionally, I analyze the distribution of each variable, examining its skewness, kurtosis, and normality to determine whether standardization or an alternative transformation—such as log-scaling for heavy-tailed distributions—is more appropriate. For variables with a high percentage of zero values (e.g., precipitation), I apply quantile normalization or min-max scaling to maintain meaningful structure while ensuring numerical stability.

Given the large dataset size, I implement streaming computation using xarray and dask, ensuring that the normalization statistics are derived without loading the entire dataset into memory. This step is crucial for scaling the methodology to higher resolutions (e.g., 3600×1800 spatial grids) without exceeding hardware constraints. Once the normalization parameters are computed, they are stored and used during both training and inference to ensure consistent scaling and improved gradient flow across all training epochs. See I in the appendix for the exact values. Of note is the large scaling in pressure variables and the high range of values across many operators.

In anticipation of using an FNO, I also perform spectral analysis (see II). This is done in order to set the trainable frequency in the SIREN activation function [9]. I then apply log transformations to my wind speed data and normalize all of my data according to the computed statistics.

III. Objective 1

Objective 1 asks about model design. The proposed model integrates Fourier Neural Operators (FNOs), ClimaX-style embeddings, and a Vision Transformer (ViT) to address the complex challenges inherent in weather forecasting. Atmospheric and climate processes are often governed by partial differential equations (PDEs), and FNOs excel at learning global dependencies by operating directly in the frequency domain. This allows the model to capture long-range correlations and multi-scale dynamics with fewer parameters than traditional convolutional approaches.

The Vision Transformer component is employed to leverage self-attention mechanisms that facilitate robust spatial patch embedding and sequence modeling. By decomposing the spatial grid into patches, the Transformer can effectively model interactions among distant regions, a key capability for capturing teleconnections in atmospheric data. This is especially critical in weather forecasting, where remote influences can significantly impact local weather conditions.

ClimaX-style embeddings are incorporated to systematically handle heterogeneous data types, such as multiple atmospheric levels and surface variables. These embeddings preserve the intrinsic physical structure of the data by providing distinct,

learnable representations for each variable. Moreover, the addition of a lead-time conditioning mechanism via a learnable MLP allows the model to dynamically adapt its predictions based on the forecast horizon, ensuring that both short-term and long-term dependencies are adequately captured.

This design is well-suited for high-dimensional, multi-scale weather data, enabling the model to learn complex spatial and temporal relationships while remaining scalable and flexible for various forecasting tasks.

A. Model Design

I designed the FourierViT model from scratch to maintain full control over every aspect of the implementation and to optimize it specifically for the challenges of weather forecasting. I chose not to use high-level PyTorch abstractions for key components such as self-attention and spectral convolutions, because building these components manually allowed me to directly address memory usage, numerical stability, and scalability concerns.

1) Fourier Neural Operators (FNOs): I implemented the FNO blocks using a custom `SpectralConv2d` module that explicitly computes the FFT of the input, manipulates the frequency components, and then applies an inverse FFT. This approach, with explicit FP32 casting during FFT operations, ensures that the spectral representation is computed accurately. FP32 casting during the FFT operation is also required because PyTorch does not natively support setting up computation graphs for complex number operations. I set the number of Fourier modes (e.g., 32 for surface variables and 16 for atmospheric variables) based on empirical evaluation, which strikes a balance between capturing long-range dependencies and maintaining computational efficiency. By directly handling the spectral transformations, I can capture the global correlations in weather data, which are critical when modeling processes governed by partial differential equations.

2) ClimaX-style Embeddings: For the patch embedding, I implemented a custom module (`ViTPatchEmbedding`) inspired by ClimaX that transforms the 98-channel input (resulting from combining atmospheric and surface data) into a learned embedding space. I set the image size to (256, 512) and the patch size to 16, which results in a manageable number of patches while preserving spatial structure. This custom embedding allows me to integrate heterogeneous data types—such as the 7 surface variables and 13 atmospheric levels—in a way that maintains their distinct physical properties. I ensured that the positional embeddings were initialized with a small random value (multiplied by 0.02) to provide a stable starting point for training, which is essential for convergence in high-dimensional spaces.

3) Custom Self-Attention and Transformer Blocks: Instead of using PyTorch's built-in attention modules, I implemented multi-head self-attention from scratch in the `MultiHeadSelfAttention` and `TransformerBlock` classes. This allowed me to define the exact dimensions and operations—for example, splitting the embedding dimension (512) into 8 heads and using a head dimension of 64—and to incorporate my own normalization and activation schemes. I used a simple linear projection for queries, keys, and values, and computed the scaled dot-product attention manually. This low-level approach not only offers greater flexibility for experimentation but also enables detailed profiling of computational cost and memory usage, which is essential when scaling to images of size 3600×1800.

4) Lead-Time Conditioning and Checkpointing: I introduced a lead-time conditioning mechanism using a custom multi-layer perceptron (MLP) with a `SineActivation` function. This component uses an MLP with an input linear layer, a sine activation (with a learnable parameter initialized to 1.0), and a second linear layer to encode the lead time into the same embedding dimension as the patches. I deliberately initialized the weights of the first linear layer with uniform random values within a range determined by the square root of 6 to ensure a balanced starting point. Activation checkpointing is applied both in the FNO blocks and during the Transformer processing to significantly reduce memory footprint during training. This strategy is crucial for enabling experiments on larger resolutions without overwhelming the available GPU memory.

5) Scalability Considerations: Although the current training is conducted on images of size $512\times256\times98$, I designed every component with scalability in mind. The combination of spectral convolution for global context, custom patch embeddings for heterogeneous data, and self-attention for modeling long-range dependencies forms a modular architecture that can be adapted to higher resolutions, such as 3600×1800. I also integrated advanced techniques like mixed precision training (BF16) and distributed training using FSDP to ensure that the model can efficiently handle large-scale data. Every design decision, from the number of Fourier modes to the patch size, was made after careful evaluation of trade-offs between computational efficiency and model accuracy.

IV. Objective 2

This objective asks me to write unit tests for the framework. I implemented a suite of unit tests using Python's `unittest` framework to ensure that every critical component of the project works as intended. The tests were written from scratch to validate the configuration file, dataset handling, model forward pass, and training function execution. Rigorous testing was essential to catch errors early and maintain code reliability as I iterated on the design.

A. Configuration Validation

I created tests to verify that the configuration dictionary contains all required keys (e.g., `'data_path'`, `'batch_size'`, `'patch_size'`, etc.) and that the normalization statistics cover every variable used in the model. This ensures that any missing or malformed configuration parameters are flagged immediately, preventing downstream issues during training or inference.

B. Dataset Handling

Since the weather data is stored in large files, I implemented tests to instantiate the `WeatherSubset` dataset and check its integrity. The tests confirm that the dataset is non-empty and that each sample adheres to the expected tensor dimensionality. This guarantees that data loading and preprocessing work correctly, which is crucial for both training and evaluation.

C. Model Forward Pass

I designed a synthetic test for the `FourierViT` model by feeding it randomized tensors that mimic the structure of real atmospheric and surface data. The test verifies that the model's forward pass produces an output tensor with the expected shape. By enforcing strict shape validation, I ensure that each sub-module (including FNO blocks, patch embeddings, and custom transformer blocks) interacts correctly. This is particularly important given that all components were implemented from scratch rather than relying on high-level abstractions.

D. Training Function Execution

In addition to static tests, I included tests that run through a single epoch of training. These tests check that the training loop executes properly and that loss computation, gradient clipping, and optimizer updates function without error. Integrating these tests into a CI pipeline ensures that future changes do not break the training process.

My unit tests (which can be executed with `python test_project.py`) provide coverage for critical functionalities of the project. They help ensure that the configuration, data loading, model forward pass, and training logic all work together, which is especially useful when deploying in production.

V. Model Training

I trained my model using WSL Ubuntu 22.04 LTC on an NVIDIA RTX 3070 Max-Q GPU to test the model. Below is the terminal output:

```
Using BF16 precision
Loading 4 timestep pairs (~411.0MB)
Epoch 1:   0%|
After input concat: 1.05GB
After FNO blocks: 1.75GB
After patch embedding: 2.07GB
After transformer: 2.08GB
After decoder: 2.19GB
Delta: 1.25GB
Total FWD memory: 1.25GB
Epoch 1/1, Train Loss: 14.9423
Delta: 0.62GB
Total FWD memory: 0.62GB
Validation Loss: 14.8775
```

I conducted some experiments to validate the training procedure of the FourierViT model using a multi-GPU setup. The model was executed in a distributed setting using PyTorch's Fully Sharded Data Parallel (FSDP) and mixed precision, with options to enforce BF16 or FP32 depending on user preference.

A. Training Environment and Configuration

The training script is designed to launch using the command:

```
torchrun --nproc_per_node=<NUM_GPUS>
train_distributed.py --fp32
```

For a single GPU test, I used `--nproc_per_node=1`, which, while not providing parallel speedup, allowed me to verify the end-to-end functionality. The environment was configured to support BF16 precision if available, with fallbacks to FP32. In my experiments, BF16 was supported and enabled, with the following confirmation printed at runtime:

```
Using BF16 precision
```

Additionally, the model was wrapped using FSDP with a mixed precision policy and compiled with `torch.compile` for enhanced performance. The FSDP configuration automatically switched to the `NO_SHARD` strategy for a single GPU, ensuring compatibility.

B. Training Options and Techniques

Several key training options were integrated into the code:

- **Mixed Precision Training:** The model leverages BF16 precision to reduce memory consumption and accelerate computations. Mixed precision is enabled conditionally based on hardware capability.
- **Activation Checkpointing:** Both the FNO blocks and Transformer blocks employ activation checkpointing to manage memory usage, allowing the training of high-dimensional data without exhausting GPU memory.
- **Custom Autocast:** The training loop makes use of the new `torch.amp.autocast` API to ensure operations run in the appropriate precision.
- **Distributed Training:** Using PyTorch's FSDP, the training script supports multi-GPU training. This setup not only helps in parallelizing the workload but also allows the model to be scaled when higher resolutions (e.g., 3600×1800) are used.
- **Gradient Clipping:** To stabilize training, the gradients are clipped at a norm of 1.0 before the optimizer updates, preventing the explosion of gradients.

C. Performance Metrics and Resource Usage

During the training run, several key memory usage metrics and loss values were recorded:

- **Memory Usage:**
  - After concatenating inputs, memory usage was approximately 1.05GB.
  - After processing through the FNO blocks, memory usage increased to 1.75GB.
  - After the patch embedding step, it reached 2.07GB.

  - The Transformer blocks maintained a similar footprint (approximately 2.08GB), and the decoder pushed the usage to 2.19GB.
  - The forward pass incurred a total memory usage delta of 1.25GB.
- **Loss Values:**
  - The training loss recorded for Epoch 1 was 14.9423.
  - The validation loss was measured at 14.8775.

These figures confirm that the model operates within reasonable memory limits on a single GPU and that the loss values indicate a stable training process for the given dataset.

D. Discussion and Future Work

The training strategy employed is both scalable and efficient. The use of BF16 precision, combined with activation checkpointing and FSDP, ensures that the model can handle increased input resolutions without incurring prohibitive memory overhead. The integration of `torch.compile` further improves runtime performance, allowing for faster iterations during model development.

VI. Specific questions answered

A. Memory analysis

The profiling output shows that the FNO blocks are the most memory intensive. After concatenating inputs, the memory footprint was about 1.05 GB, which jumped to 1.75 GB after processing through the FNO blocks—an increase of roughly 700 MB. I inserted custom memory profiling calls (using tools like `MemoryProfiler` and `print_memory_usage`) at key stages in the forward pass: after input concatenation, after the FNO blocks, after patch embedding, after the transformer, and after the decoder. This step-by-step analysis revealed that the spectral operations within the FNO blocks (which perform FFT and inverse FFT operations) are the primary contributors to memory usage, with subsequent layers adding only marginal increases.

This is somewhat surprising as the FFT scales as $nlogn$ and transformer architectures are n^2, but this can be explained by the specific choice of patch sizes. I should note that I did face significant checkerboard artifacts in forecasting and this is almost certainly due to the decoder architecture and patch sizes.

B. Scaling to high resolutions

Here are some calculations that provide a first look at scaling to high resolutions

1) Input Resolution and Overall Computation:

- **Spatial Resolution Scaling:** The original resolution contains $512 \times 256 = 131{,}072$ pixels, whereas the new resolution has $3600 \times 1800 = 6{,}480{,}000$ pixels. This is an increase by a factor of roughly 50.
- **Impact:** Every per-pixel operation, including FFTs in the FNO blocks, will need to handle 50× more data, thereby increasing both computation time and memory usage.

2) FNO Blocks (FFT-Based Layers):

- **Computation Increase:** FFT operations scale roughly linearly with the number of pixels. If an FFT on the original image takes time T, then on the new image it will take about $50 \times T$.
- **Memory Footprint:** For example, consider atmospheric data with $C_{atmos} = 7 \times 13 = 91$ channels.
  - *Original:* Total elements $\approx 91 \times 131{,}072 \approx 11.9 \times 10^6$.
  - *New:* Total elements $\approx 91 \times 6{,}480{,}000 \approx 589 \times 10^6$.

  This indicates approximately 50× more memory is needed for these activations. If the FNO blocks originally used about 1.75 GB, scaling by 50 implies a memory requirement in the vicinity of 87.5 GB—keeping in mind that actual usage may vary due to optimizations or reduced precision.

3) ViT Patch Embedding and Transformer Blocks:

- **Patch Embedding Impact:** The current patch size is 16, which yields:
  - *Original Setup:* $\frac{256}{16} \times \frac{512}{16} = 16 \times 32 = 512$ patches.
  - *New Setup:* $\frac{1800}{16} \times \frac{3600}{16} \approx 112.5 \times 225 \approx 25{,}312$ patches (with necessary adjustments or padding for integer division).

  This is roughly a 50× increase in the number of tokens.
- **Self-Attention Complexity:** Self-attention scales quadratically with the number of tokens.
  - *Original:* With 512 tokens, each head computes an attention matrix of size $512 \times 512 \approx 262{,}144$ elements. For 8 heads, this totals roughly 2 097 152 elements (about 8.4 MB using FP32).
  - *New:* With approximately 25 200 tokens, the attention matrix per head has about $25{,}200 \times 25{,}200 \approx 635{,}040{,}000$ elements. For 8 heads, this totals roughly 5,080,320,000 elements—using about 20.32 GB per transformer layer just for the attention scores.

  Backpropagation further doubles this memory requirement (forward plus gradients), potentially exceeding 40 GB per transformer block.

4) Overall Memory and Compute Impact:

- **Memory Footprint (per forward pass, approximate):**
  - After input concatenation: Originally 1.05 GB; new estimate: $1.05 \times 50 \approx 52.5$ GB.
  - After FNO blocks: Originally 1.75 GB; new estimate: $1.75 \times 50 \approx 87.5$ GB.
  - After patch embedding: Likely similar linear scaling, but subsequent transformer layers will dominate due to the dramatic increase in token count.
- **Compute Throughput:**
  - FFT Operations: Approximately 50× slower per image.
  - Transformer Self-Attention: Up to 2,400× more operations due to the quadratic scaling with tokens (i.e., $(25{,}200/512)^2 \approx 2,400$).

5) Practical Mitigation Strategies:

- **Model Architecture Adjustments:** Increasing the patch size (e.g., to 32 or 64) would reduce the number of tokens, thereby lowering the quadratic cost in self-attention. Alternatively, efficient attention mechanisms (such as Performer or Linformer) or a hierarchical transformer design can help manage the computational complexity.
- **Hardware Considerations:** Higher resolution images will likely require more powerful hardware, such as multiple GPUs with large VRAM and fast interconnects. Reducing the batch size (possibly to 1) or using gradient accumulation techniques may also be necessary to manage memory constraints.
- **Algorithmic Optimizations:** Further improvements in checkpointing strategies (extending beyond FNO blocks to include transformer layers) and selective processing (processing only regions of interest at full resolution) can also help mitigate the increased resource demands.

6) Summary:

- *Pixel Increase:* From 131,072 to 6,480,000 pixels (roughly 50× more).
- *Patch Increase:* From 512 to approximately 25,200 patches (around 50× more tokens).
- *Self-Attention Cost:* Increases from roughly 2 million elements (≈8 MB per block) to approximately 5.08 billion elements (≈20.3 GB per block), leading to an overall cost that may be 2,400× higher.
- *Memory Impact:* The original forward pass required about 1.25 GB peak memory, while scaling naively could push this beyond 50–100 GB before even considering backpropagation.
- *Compute Impact:* FFT operations become 50× more expensive, and transformer self-attention could require up to 2,400× more operations.

Without substantial modifications to both the model architecture and the training strategy, scaling directly to 3600×1800×98 images is likely to be infeasible due to massive increases in computational and memory requirements. Strategic changes—both algorithmic and hardware-related—would be necessary to make training and inference practical at these higher resolutions.

C. Deepening the data cube (changing pressure levels from 13 to 37)

This requires a bit of Fermi estimation.

1) Current Setup: Surface variables: 7 channels Atmospheric variables: $7 \times 13 = 91$ channels Total: $7 + 91 = 98$ channels

2) New Setup: Surface variables: 7 channels Atmospheric variables: $7 \times 37 = 259$ channels Total: $7 + 259 = 266$ channels

3) Scaling Factor:

$$\text{Scaling Factor} = \frac{266}{98} \approx 2.71$$

This means that the number of input channels increases by roughly 2.7×.

4) Impact on the FNO Blocks: **FNO Block Inputs:** The FNO blocks are applied separately to the atmospheric and surface data.

- For the atmospheric data, the number of channels increases from 91 to 259.
- For the surface data, the number of channels remains at 7.

Memory and Compute in FNO Blocks:

- The spectral convolution (FFT) operations and the linear 1×1 convolution inside each FNO block operate on the channel dimension.
- Since these operations scale linearly with the number of channels, the cost (both in terms of memory and computation) for the atmospheric portion will increase by about 2.7×.
- For example, if the FFT-based spectral convolution previously required processing 91 channels, it will now have to handle 259 channels.
- This also increases the size of the weight tensors in these layers (e.g., the parameter tensor in the spectral convolution will increase in size roughly in proportion to the product of the input and output channels).

5) ViT Patch Embedding: **Patch Embedding Input:** The patch embedding layer is implemented as a convolution that takes the entire data cube as input.

- Originally, the convolution input is of shape $(B, 98, H, W)$.
- With more atmospheric levels, the input shape becomes $(B, 266, H, W)$.

Computational Cost:

- The convolution's number of multiply-accumulate operations scales linearly with the number of input channels.
- Thus, the cost in this layer will increase by roughly the same factor of 2.7×.
- However, the output dimension (embedding dimension) remains fixed at 512, so only the cost of processing the input is affected.

6) Decoder and Subsequent Layers: **Decoder Impact:**

- The decoder reconstructs the output based on the aggregated feature maps from the transformer.
- Since the decoder operates on the tokens after the transformer (which uses the fixed embedding dimension), its computational cost is less directly affected by the increased input channel count.
- The main impact in the decoder might be from the larger initial feature maps coming from the FNO and patch embedding stages, but the dominant scaling effect is at the input side.

Transformer Blocks:

- The transformer processes the patch embeddings, which are produced after the initial channel-reducing projection (from 98 or 266 channels to 512).
- Because the transformer operates on a fixed embedding dimension (512) and sequence length determined by

the spatial dimensions and patch size, its computational cost does not directly change with the number of input channels.

7) Overall Memory and Computational Scaling: **Memory Footprint Increase:**

- The parts of the model that directly process the raw data (i.e., the FNO blocks and patch embedding) will see an approximate 2.7× increase in memory usage.
- For example, if the original forward pass consumed 1.75 GB after the FNO blocks for atmospheric data, this could increase to roughly:

$$1.75\,\text{GB} \times 2.7 \approx 4.73\,\text{GB}$$

- Note that the surface data portion is unchanged, but the overall increase is dominated by the atmospheric component.

Computational Cost:

- Similarly, the compute required in the FNO blocks and the patch embedding layer will scale roughly linearly with the number of input channels.
- This means that these components will roughly take 2.7× more compute time (assuming all else is equal).

Bottleneck Considerations:

- While the transformer blocks do not directly scale with the number of input channels, the increased load on the earlier layers (FNO and patch embedding) could lead to increased data movement and overall latency in the network.
- If the architecture is already close to memory or compute limits, this additional factor might necessitate hardware upgrades or further optimizations (such as using mixed precision).

D. Bottlenecks if activation checkpointing is applied

At higher resolutions, the transformer's self-attention layers become the major bottleneck even when activation checkpointing is used. Here's a detailed analysis with numbers:

- In a lower-resolution setting, suppose the patch embedding produces around 512 tokens. In that case, each self-attention layer must compute an attention matrix with roughly 512^2, or 262,000 elements per head. With, say, 8 heads, that's around 2.1 million elements.
- If you increase the resolution significantly—resulting in, for example, 25,200 tokens—the attention matrix per head would have roughly $25{,}200^2$, or about 635 million elements. For 8 heads, that scales to roughly 5.08 billion elements. This is a roughly 2,400× increase in the number of comparisons and intermediate values that need to be stored and processed.
- Activation checkpointing helps by not storing all intermediate activations during the forward pass; it recomputes them during the backward pass. However, the cost of self-attention remains because the large attention matrices are required during the forward computation and need to be computed—even if not all are kept in memory simultaneously. This quadratic scaling in the number of tokens means that, despite checkpointing, the memory required to compute these attention matrices (and the time to compute them) grows dramatically.
- As a result, even though checkpointing reduces the memory footprint by trading off extra computation, the self-attention layers still dominate the overall memory usage and computational cost at higher resolutions. This is why, for very high-resolution inputs, the transformer self-attention becomes the critical bottleneck in the architecture.

E. What computations can be performed using bf16?

As discussed before: Yes, my model can perform all but the Fourier transform computations using bf16, because of PyTorch lacking native support for bf16 computations for complex numbers.

F. Data normalization

I have done basic normalizing of all data and log-transformed wind. I explained how this is done in detail above. This is also available in the code itself.

A cursory literature search reveals the following types of normalization done to specific types of data. This is done to ensure that the data are in a suitable format for the model to learn from. Below is a summary of the normalization techniques used for different types of weather data:

- **Wind data:** Wind data is often log transformed to reduce skewness and improve the distribution of the data. This is because wind data tends to have a heavy-tailed and positively skewed distribution [11]. Log transformation helps to bring the mean of the data closer to zero and reduces the impact of extreme values [10].
- **Rain data:** Rain data is normalized using a specific transformation function to bring the data into a range of [0, 1] [13]. This transformation function is designed to spread out the values closest to 0, making it easier for the model to learn from the data.
- **Temperature, pressure, and humidity data:** These types of data are often normalized using max-min normalization or Z-score normalization to prevent overfitting and ensure that the model can learn the characteristics of each physical quantity [14].
- **Precipitation data:** Precipitation data is often normalized using max-min normalization due to its wide range of 0 values and the presence of many samples with no or sporadic precipitation [14].
- **Atmospheric variables:** Atmospheric variables such as wind speed and relative humidity are often normalized using techniques such as KL divergence to ensure that the model's results adhere to physical laws [15].

The choice of normalization technique depends on the specific characteristics of the data and the type of model being used. Normalization is just one aspect of data preprocessing, and other techniques such as data transformation and feature

scaling may also be used to prepare the data for training, as done in [12].

G. Further memory and performance optimizations to improve training/inference

This is the section where I hypothesize about possible improvements, but each of these will require significant investigation.

1) Mixed Precision (BF16/FP16): Using BF16 or FP16 arithmetic reduces the memory footprint and increases throughput by halving the memory needed per floating-point parameter (e.g., 4 bytes become 2 bytes for FP16). For example, if the model uses 1.25 GB of memory in FP32 per forward pass, switching to FP16/BF16 could reduce this to approximately 0.65 GB—about a 50% reduction. GPUs like the NVIDIA A100 can see a 2× or higher speedup in compute-bound kernels with mixed precision. There is a slight tradeoff in numerical precision; during finetuning, accuracy might drop by less than 1%, and for inference, many models tolerate these minor deviations.

2) Post-Training Quantization: Converting weights and activations from FP32 to INT8 (or even lower) further reduces memory requirements and accelerates inference on specialized hardware. For instance, converting to INT8 can reduce the storage size by a factor of 4 compared to FP32. On hardware optimized for INT8 operations, such as many modern GPUs and NPUs, inference throughput can improve by 2–4× over FP32. Quantization-aware training might be necessary if accuracy drops exceed 1–2%, so a common approach is to finetune in FP32/mixed precision and then apply quantization with calibration on a validation set.

3) Efficient Attention Mechanisms: Standard multi-head self-attention scales quadratically with sequence length. For a 512-token sequence, the attention matrix per head has roughly $512^2 \approx 262\,144$ elements, leading to about 2.1 million elements for 8 heads. In high-resolution cases with around 25,200 tokens, this cost increases dramatically. Methods such as Linformer or Performer approximate attention with linear or near-linear complexity. By reducing the quadratic cost to linear, the attention computation cost can drop from $O(N^2)$ to $O(N)$; for a 25,200-token sequence, this could mean a reduction in computation by a factor of roughly 2421 in the worst case. The tradeoff is that approximation errors may reduce accuracy by 1–3% if not carefully tuned.

4) Patch Size Adjustments: Increasing the patch size reduces the number of tokens passed to the transformer. For example, increasing the patch size from 16 to 32 on 3600×1800 images changes the number of patches from approximately 25,200 tokens to about

$$\frac{3600}{32} \times \frac{1800}{32} \approx 112.5 \times 56.25 \approx 6\,300 \text{ tokens.}$$

This is a reduction by a factor of 4 in sequence length, which lowers the self-attention computational cost by roughly $4^2 = 16$×. However, a larger patch size may reduce the spatial resolution in the transformer's input and could degrade the model's ability to capture fine-grained features by a few percentage points in accuracy.

5) FFT Optimizations: FFT operations in FNO blocks are highly optimized in libraries such as cuFFT, but further tuning can help. For example, using lower precision (BF16/FP16) for FFTs can reduce memory bandwidth usage and potentially yield similar speed gains—often around a 50% reduction in compute cost. Additionally, batching FFT calls can maximize throughput. If an FFT on a 512×256 image takes time T in FP32, switching to BF16 might reduce it to roughly $0.5T$, and for 3600×1800 images (with 50× more data), the time could scale to about $50T$ without optimizations, or roughly $25T$ with BF16 and batching. The tradeoff is a potential introduction of noise, though in practice, the effect is often minimal when handled properly.

6) Fuse Operations and Operator-Level Optimizations: Fusing linear operations—such as the two linear layers in the transformer's MLP—can reduce kernel launch overhead and improve cache efficiency. This operator fusion can sometimes lower the operational overhead by 10–20% per layer, especially for small matrix multiplications. The tradeoff is that it requires careful engineering or reliance on libraries (like NVIDIA's TensorRT) that support such fusions.

7) Distributed Inference: For extremely high-resolution images, partitioning the image into spatial regions and processing them on multiple GPUs can be beneficial. For example, if a single high-resolution image requires 100 GB of GPU memory for one pass, splitting the image across 4 GPUs could reduce the per-device requirement to approximately 25 GB (ignoring overhead). However, this introduces communication overhead for synchronization and recombination of outputs, potentially adding 10–30% latency depending on the interconnect (e.g., NVLink versus PCIe).

8) Asynchronous Data Loading and Preprocessing: Utilizing multi-threaded or asynchronous data loading pipelines ensures that GPUs are not idle while waiting for data. If data loading was causing 20–30% idle time, asynchronous loading could nearly eliminate this overhead, thereby improving throughput by a similar margin. The tradeoff is increased pipeline management complexity and potential I/O bottlenecks if the storage system does not keep up.

9) TorchScript / ONNX and TorchCompile: Converting the model to a compiled format (using TorchScript, ONNX, or torch.compile) can significantly speed up inference by optimizing the computation graph, fusing operations, and eliminating Python overhead. Reports indicate that such optimizations can yield 1.5–3× faster inference. For example, if inference time is 100 ms per image in eager mode, it might be reduced to 50–70 ms. The tradeoff is that debugging and modifying the model after compilation becomes more challenging, and not all operations may be fully optimized.

10) Advanced Checkpointing: Beyond the current activation checkpointing, more granular checkpointing strategies can be applied to further reduce memory usage. For instance, for a transformer block that uses 40 GB during the backward pass, aggressive checkpointing might cut peak memory usage by

30–40% at the expense of a 20–30% increase in forward pass time. This extra computation may be acceptable during finetuning if it allows training on hardware with limited memory.

11) Gradient Accumulation: When forced to use very small batch sizes (e.g., 1 or 2), gradient accumulation simulates a larger batch size by accumulating gradients over several forward passes before performing an update. For example, accumulating gradients over 8 iterations can simulate an 8× larger batch size without increasing peak memory usage. The tradeoff is a slight increase in overall training time and additional complexity in adjusting the learning rate schedule.

12) Summary of Numbers and Tradeoffs:

- *Mixed Precision:* Can reduce memory usage by 50% (e.g., from 1.25 GB to 0.65 GB per pass) and potentially double the speed on compatible hardware.
- *Efficient Attention:* For 25,200 tokens, quadratic attention can require up to 20–40 GB per block; linear attention methods might reduce this by a factor of 5–10.
- *Patch Size Increase:* Increasing from 16×16 to 32×32 patches can reduce tokens by 4× (e.g., 25,200 → 6,300 tokens), lowering attention cost by 16×, with a potential accuracy tradeoff of 1–3%.
- *FFT and Kernel Optimizations:* Lower precision FFT can yield 50% savings in compute and memory, and operator fusions can reduce overhead by 10–20%.
- *Distributed Inference:* Splitting a high-resolution image over 4 GPUs can reduce per-GPU memory requirements from 100 GB to 25 GB, though with an additional 10–30% communication overhead.
- *Compilation:* TorchScript/TorchCompile may improve inference speed by 1.5–3×.
- *Checkpointing & Gradient Accumulation:* Aggressive checkpointing can reduce peak memory by 30–40%, while gradient accumulation allows simulating larger batch sizes without increasing memory usage.

13) Conclusion: For both finetuning and inference on the FourierViT model, applying mixed precision, efficient attention mechanisms, patch size adjustments, FFT and kernel optimizations, as well as distributed and asynchronous processing, are key to reducing memory footprint and computation time. For example, mixed precision can halve memory usage while efficient attention methods can reduce the potential 2,400× scaling cost in self-attention to a more manageable level. Distributed strategies and graph optimizations can further improve inference speeds by 2–3×. Although each optimization has tradeoffs—such as slight accuracy loss or increased computational overhead—when combined judiciously, these strategies enable the processing of extremely high-resolution images on modern high-end GPUs.

VII. Conclusion

I know that this paper isn't very pleasing to the eyes to read, but I blame that largely on the lack of figures and loss curve graphs that I would have been able to provide if I had access to a stronger GPU. I hope that you read this thoroughly and look at my code as well. It is available at https://github.com/ksd3/fouriervitforecasting. This is a private repository, so please request access!

References

[1] S. Rasp, S. Hoyer, A. Merose, I. Langmore, P. Battaglia, T. Russell, et al., "WeatherBench 2: A benchmark for the next generation of data-driven global weather models," *J. Adv. Model. Earth Syst.*, vol. 16, p. e2023MS004019, 2024. [Online]. Available: https://doi.org/10.1029/2023MS004019

[2] T. Nguyen, J. Brandstetter, A. Kapoor, J. K. Gupta, and A. Grover, "ClimaX: A foundation model for weather and climate," *arXiv preprint arXiv:2301.10343*, 2023. [Online]. Available: https://arxiv.org/abs/2301.10343

[3] C. Bodnar, W. P. Bruinsma, A. Lucic, M. Stanley, A. Vaughan, J. Brandstetter, P. Garvan, M. Riechert, J. A. Weyn, H. Dong, J. K. Gupta, K. Thambiratnam, A. T. Archibald, C.-C. Wu, E. Heider, M. Welling, R. E. Turner, and P. Perdikaris, "A foundation model for the Earth system," *arXiv preprint arXiv:2405.13063*, 2024. [Online]. Available: https://arxiv.org/abs/2405.13063

[4] I. Price, A. Sanchez-Gonzalez, F. Alet, et al., "Probabilistic weather forecasting with machine learning," *Nature*, vol. 637, pp. 84–90, 2025. [Online]. Available: https://doi.org/10.1038/s41586-024-08252-9

[5] J. Pathak, S. Subramanian, P. Harrington, S. Raja, A. Chattopadhyay, M. Mardani, T. Kurth, D. Hall, Z. Li, K. Azizzadenesheli, P. Hassanzadeh, K. Kashinath, and A. Anandkumar, "FourCastNet: A global data-driven high-resolution weather model using adaptive Fourier neural operators," *arXiv preprint arXiv:2202.11214*, 2022. [Online]. Available: https://arxiv.org/abs/2202.11214

[6] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, and A. Anandkumar, "Fourier Neural Operator for Parametric Partial Differential Equations," *arXiv preprint arXiv:2010.08895*, 2021. [Online]. Available: https://arxiv.org/abs/2010.08895

[7] V. Hondru, F. A. Croitoru, S. Minaee, R. T. Ionescu, and N. Sebe, "Masked Image Modeling: A Survey," *arXiv preprint arXiv:2408.06687*, 2025. [Online]. Available: https://arxiv.org/abs/2408.06687

[8] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," *arXiv preprint arXiv:1912.01703*, 2019. [Online]. Available: https://arxiv.org/abs/1912.01703

[9] V. Sitzmann, J. N. P. Martel, A. W. Bergman, D. B. Lindell, and G. Wetzstein, "Implicit Neural Representations with Periodic Activation Functions," *arXiv preprint arXiv:2006.09661*, 2020. [Online]. Available: https://arxiv.org/abs/2006.09661

[10] T. Weber, A. Corotan, B. Hutchinson, B. Kravitz, and R. Link, "Technical note: Deep learning for creating surrogate models of precipitation in Earth system models," *Atmos. Chem. Phys.*, 2019.

[11] J. Manero, J. Béjar, and U. Cortés, "Dust in the Wind...: Deep Learning Application to Wind Energy Time Series Forecasting," *Energies*, 2019.

[12] J.-P. Lai, Y.-M. Chang, C.-H. Chen, and P.-F. Pai, "A Survey of Machine Learning Models in Renewable Energy Predictions," *Appl. Sci.*, 2020.

[13] V. Bouget, D. B'er'eziat, J. Brajard, A. Charantonis, and A. Filoche, "Fusion of rain radar images and wind forecasts in a deep learning model applied to rain nowcasting," *Remote Sens.*, 2020.

[14] L. Xiang, J. Xiang, J. Guan, F. Zhang, Y. Zhao, and L. Zhang, "A Novel Reference-Based and Gradient-Guided Deep Learning Model for Daily Precipitation Downscaling," *Atmosphere*, 2022.

[15] W. Li, Z. Liu, K. Chen, H. Chen, S. Liang, Z. Zou, *et al.*, "DeepPhysiNet: Bridging Deep Learning and Atmospheric Physics for Accurate and Continuous Weather Modeling,"

VIII. Appendix

TABLE I
STATISTICAL ANALYSIS OF WEATHER VARIABLES

Variable	Storage (MB)	Min	Max	Mean	Std Dev	Skewness	Kurtosis
10m U Wind	67.100	−32.140	33.470	0.160	5.534	0.383	0.193
10m V Wind	67.100	−30.140	26.720	−0.245	4.619	0.039	0.527
10m Wind Speed	67.100	0.078	36.550	6.270	3.644	0.802	0.512
2m Temperature	67.100	222.310	318.440	277.600	19.876	−0.466	−1.026
Geopotential	872.400	−6266.420	204,281.440	77,741.200	59,489.740	0.480	−0.783
Mean Sea Level Pressure	67.100	92,338.160	107,380.690	100,918.000	1488.290	−0.588	0.734
Specific Humidity	872.400	0.000	0.024	0.002	0.004	2.763	7.338
Surface Pressure	67.100	50,028.260	105,408.200	96,669.810	9505.800	−2.428	5.032
Temperature	872.400	182.110	319.680	243.120	28.721	0.208	−1.038
Total Precipitation (6hr)	67.100	0.000	0.000	0.000	0.000	–	–
U Component of Wind	872.400	−65.160	122.260	8.244	14.905	1.193	2.286
V Component of Wind	872.400	−86.930	90.750	0.044	9.761	−0.008	3.288
Vertical Velocity	872.400	−11.990	12.540	0.005	0.156	−2.431	74.845
Wind Speed	872.400	0.051	124.420	15.021	12.660	1.735	3.811

TABLE II
SPECTRAL AND STATISTICAL PROPERTIES OF WEATHER VARIABLES

Variable	Level	Entropy	Dom. Freq	Mean	Variance	Avg. Grad	PSD Bands
10m U Wind	None	3.010	1	0.490	31.630	1.060	287.000
10m V Wind	None	3.660	1	−0.120	21.580	0.960	287.000
10m Wind Speed	None	3.120	1	6.380	13.370	0.880	287.000
2m Temperature	None	0.450	1	277.710	373.270	1.000	287.000
Mean Sea Level Pressure	None	1.390	1	100,747.630	30,059,764.000	90.110	287.000
Surface Pressure	None	2.440	1	96,509.890	90,201,904.000	750.560	287.000
Total Precipitation (6hr)	None	N/A	0	0.000	0.000	0.000	287.000