Air-Sea Flux Super-Resolution

Prani Nalluri

January 2025

1 General Problem

We want to super-resolve the inputs to AeroBulk (from low-resolution to high-resolution). Then, use these inputs to calculate the high-resolution heat, momentum, and freshwater fluxes. Below are several approaches that we are exploring to accomplish this.

2 Fourier Transforming Data in Pre-processing

In our initial experiments, we tried to directly super-resolve from low-resolution to high-resolution. This did not work very well, as only large-scale features were picked up on, and not small-scale features. This is likely because our loss function was `MSELoss()`, which computes the point-wise L2 norm between the target and prediction. During training, the CNN tried to minimize this loss. However, low-frequency (large-scale) features contribute more significantly to the loss than high-frequency (small-scale) features. Thus, the CNN returned very low losses even when it was not capturing small=scale behavior well.

For this experiment, we tried Fourier transforming the low-resolution and high-resolution training/testing data. This is to ensure the CNN is able to pick up on the broad range of scales in our data. Fourier transforms are a good choice to address this issue because Fourier transforms separate data based on frequency. This allows us to easily gear the CNN toward picking up on a much broader range of frequencies, or even focus more on a set of frequencies we are interested in (e.g. high frequencies).

Converting to and from Fourier space is very important for this code. Below, we go through some of the basic mathematics required to understand this.

A Fourier transform of some continuous function $f(x)$ is defined as

$$F(k) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i k x} dx$$

The inverse Fourier transform allows us to recover $f(x)$

$$f(x) = \int_{-\infty}^{\infty} F(k) e^{2\pi i k x} dk$$

For numerical data, we use a discrete Fourier transform, as it can work with individual points, rather a continuous function. The expression for a discrete Fourier transform of data x_n is below.

$$X_k = \sum_{n=0}^{N-1} x_n e^{-2\pi i k n/N}, \ k = 0, 1, 2, ..., N-1$$

The inverse discrete Fourier transform is

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{2\pi i k n/N}, \ n = 0, 1, 2, ..., N-1$$

Because we are working with 2D data (images) we use the 2D discrete Fourier transform. Note that the original image is $x_{n,m}$, and has dimensions (N, M).

$$X_{k,l} = \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} x_{n,m} e^{-2\pi i (\frac{kn}{N} + \frac{ml}{M})}$$

The 2D inverse discrete Fourier transform is

$$x_{n,m} = \frac{1}{NM} \sum_{k=0}^{N-1} \sum_{l=0}^{M-1} X_{k,l} e^{2\pi i (\frac{kn}{N} + \frac{ml}{M})}$$

To recover the exact original image, we slightly modify the equation above. This is because we want to remove any numerical errors due to rounding that may have occurred in the inverse Fourier transform calculations. These numerical errors include introducing complex terms to data points. Thus, we use the following equation

$$x_{n,m} = Re\Big(\frac{1}{NM} \sum_{k=0}^{N-1} \sum_{l=0}^{M-1} X_{k,l} e^{2\pi i (\frac{kn}{N} + \frac{ml}{M})}\Big)$$

In code, to convert from real to Fourier space, we do the following using the `fft2` function

$$X_{k,l} = fft2(x_{n,m})$$

To convert from Fourier space to real space we do the following using the `ifft2` function

$$x_{n,m} = real(ifft2(X_{k,l}))$$

Thus, we can say the following two are equivalent.

$$x_{n,m} = real(ifft2(fft2(x_{n,m})))$$

2.1 Our Plan

We intend to train a CNN take the Fourier transform of low-resolution images as input and predict the Fourier transform of the corresponding high-resolution images. The CNN uses the in-built PyTorch function `MSELoss()`. This approach requires us to Fourier transform all training, validation, and test data before working with the CNN. Once the CNN has output its predictions of the Fourier transform of high-resolution data, we can apply an inverse Fourier transform to get our final predictions of the high-resolution fields.

2.2 Current Issues

The issue with this experiment is with recovering the real data after Fourier transforming. Our target data is the Fourier transformed high-resolution images. During CNN testing, the real images from the test data set should be recoverable by taking the inverse Fourier transform of the target images. However, this equivalency is not occurring in our code.

There are several possibilities as to why this may be the case. The `MSELoss()` function in PyTorch is not built for complex values. According to the PyTorch documentation, `MSELoss()` does not throw an error when handling complex values because the loss function can be easily generalized for complex values. We have been searching for an explicit expression for the `MSELoss()` function when it's dealing with complex functions, but have been unable to find one yet.

The next step we will pursue to troubleshoot this issue is to create very simple synthetic data, take its Fourier transform, and attempt to train our CNN on this. This hopefully will allow us to isolate the exact issues in the pipeline for this experiment.

3 Fourier Custom Loss Function

One issue we have encountered in our experiments is ensuring the CNN is able to pick up on all the scales present in the data. It resolves large-scale processes decently, but struggles with smaller scales. We attempt to solve this issue by taking into account the mean-squared error (MSE) of the difference in the prediction and targets plus the MSE between the magnitudes of the normalized Fourier transforms of the prediction and the targets. This custom loss function

can be expressed as such

$$loss = \frac{1}{N}\sum_{k=0}^{N} |f_{target}(x) - f_{output}(x)|^2 + \frac{\alpha}{N}\sum_{k=0}^{N} |F_{target}(k) - F_{output}(k)|^2, \ \alpha \in \Re,$$

where

$$\hat{F}(k) = \frac{F(k)}{max|F(k)|},$$

$$F(k) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i k x} dx$$

Theoretically, the normalized Fourier transformed MSE Loss should account for the smaller scales, and the regular MSE Loss accounts for the rest of the data. We should be able to tune α in order to ensure our predictions are as close to the ground truth as possible. This method has shown a small improvement over the case where just MSE Loss on the data in real space was used (the in-built PyTorch MSELoss() function).

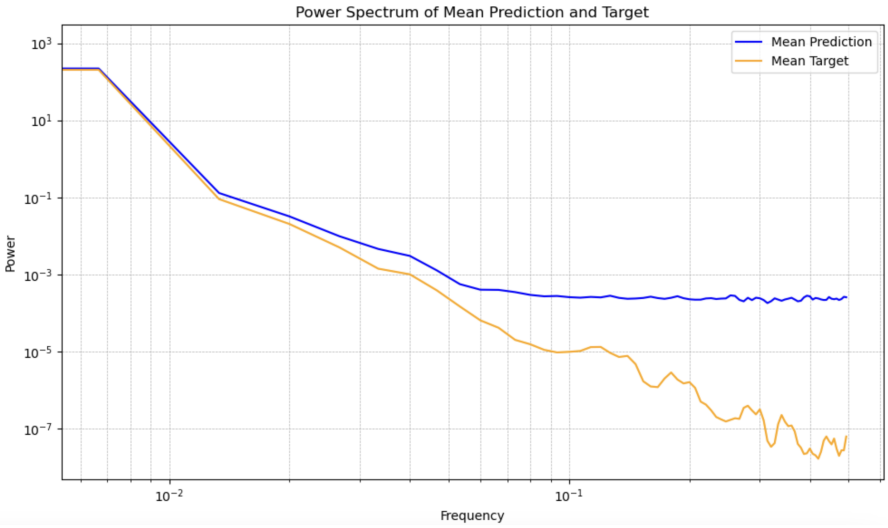

Figure 1: Power spectra for in-built MSELoss function

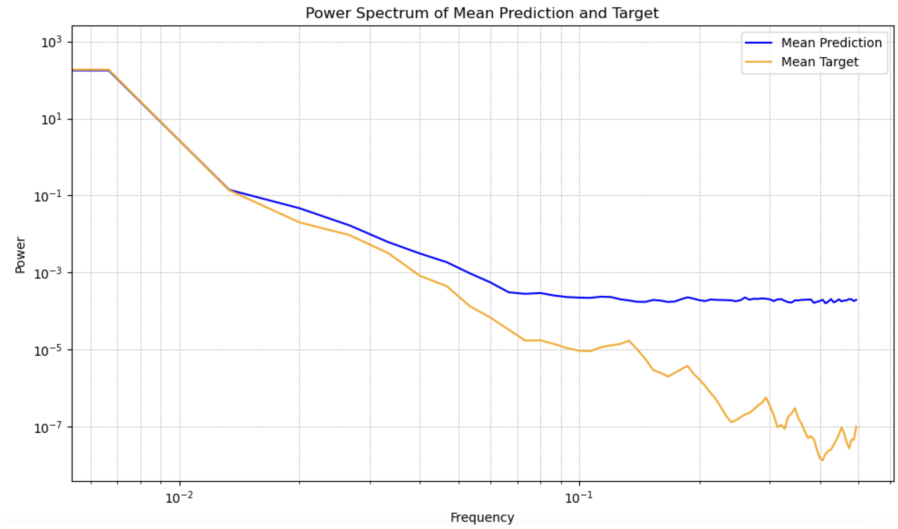

Figure 2: Power spectra for custom loss function

4 Customized Loss Function

One issue we have encountered in our experiments is ensuring the CNN is able to pick up on all the scales present in the data. It resolves large-scale processes decently, but struggles with smaller scales. We attempt to solve this issue by taking into account the mean-squared error (MSE) of the difference in the prediction and targets plus the MSE between the gradients of the prediction and the targets. This custom loss function can be expressed as such

$$loss = ||u_{target} - u_{prediction}||^2 + \alpha||\nabla u_{target} - \nabla u_{prediction}||^2, \ \alpha \in \Re$$

4.1 Current Issues

Theoretically, when $\alpha = 0$, we should be left with the regular in-built PyTorch `MSELoss()`. However, our results when using the in-built `MSELoss()` and when using our custom loss function with $\alpha = 0$ are significantly different. In fact, when our custom loss function at $\alpha = 0$ is used, our CNN performs worse. This indicates that our custom loss function at $\alpha = 0$ is not equivalent to `MSELoss()`. It is unclear why this is the case, since `MSELoss()` is a point-wise measure, as is our custom function. To determine why we have such discrepancies we will use our custom loss function on some simple synthetic data in order to ensure we can troubleshoot the issues with our methodology.

5 U-Net Architecture

Thus far, we have been using a CNN architecture. While this was a good introduction to machine learning generally, we want to try out other architectures for this problem, such as a U-net. This is because U-nets may be able to capture behavior across several scales better than a CNN. We will explore the U-net, along with a few other architectures to test this.