Machine Learning for Graphs

Extending GraphCast Into the Past

Martynas Vaznonis

studentnumber: 2701013

Abstract. Forecasting the weather is extremely important to just about everyone. However, traditional numerical weather prediction models have some limitations. Namely, they fail to make use of the preponderance of historical weather data we have and improving them is a tedious and laborious process. Since the release of GraphCast, machine learning based weather prediction has been shown to be competitive with numerical methods. But, despite severe complexity, there are many aspects of GraphCast which might still be improved. This research focuses on efficiently extending how far into the past GraphCast sees by deriving a context state from several older states and using it to improve weather forecasts. While this paper failed to produce satisfactory results, it nonetheless outlines potential paths to extend GraphCast with the context state. The code can be found on GitHub.

Keywords: Weather · Forecast · GNN · Context · GraphCast

1 Introduction

Knowing the weather of next week in advance is beneficial for all. From individuals, who may craft plans around sunshine and rain, to big organizations, which may need to worry about wind conditions, and even to governments, which need to act proactively ere natural disasters strike. For this reason, people have developed ways of forecasting future conditions. Traditionally, this is done by numerical weather prediction (NWP), that is, by utilizing differential equations to model weather dynamics [1, 2]. However, NWP systems have some shortcomings. For one, they are poor at some tasks, including sub-seasonal heat wave [9] and precipitation predictions [14, 13, 3, 15]. Moreover, NWP is slow, and it cannot make use of the preponderance of historical weather data we have accrued.

An alternative to NWP is machine learning based weather prediction methods (MLWP) [10, 18, 8]. They circumvent the aforementioned NWP issues, as well as come with additional benefits, such as capturing patterns not easily represented by equations. One notable MLWP model is GraphCast [8], which uses several complex graph neural networks (GNNs) to autoregressively predict future states. GraphCast was compared against HRES, the best single NWP model [12], and almost ubiquitously surpassed it (though on a lower resolution than HRES is capable of). This showed that MLWP methods can be competitive with traditional ones, and that further research and development may allow us to more accurately predict farther into the future.

GraphCast works by processing the 2 most recent weather states by a series of GNNs, and thus generating the next state. Then the predicted state can be substituted as input, and the subsequent future states unrolled for more distant predictions. The authors claim that any more than the two most recent states provide little performance improvement for the significant computational overhead, and therefore are not worth including [8]. However, improvement, even if little, is observed by including only one additional state. This makes sense considering that the 2 weather states passed into the network do not satisfy the Markov property [4]. Therefore, additional information from the past, can help make better predictions about the future. I endeavor to extend GraphCast by including a special third context state. An overview of the architecture of GraphCast, as well as the extended version can be seen in figure 1.

1.1 Contribution

- Explain why context is useful
- Propose the addition of a context state
- Outline several ways to extract context
- Provide a feasible way to implement context
- Implement an inchoate version of the context extension
- Summarize the difficulties of extending GraphCast
- Discuss potential resolutions and future directions

2 Related Work

The ultimate goal of GraphCast and its predecessors and successors is to accurately predict the weather. As far as I know, no attempts have been made to improve GraphCast by extending its inputs. However, many other directions have been explored successfully. First, GraphCast was improved by realizing that different parts of the world will show different weather patterns. Consequently, this means that fine-tuning on distinct regional data can improve performance within that area [16]. Another improvement was gained by extending GraphCast to model uncertainty [19]. This was suggested as a future direction in the original paper and was considered a major advantage of NWP over GraphCast. Yet even greater advances were made by incorporating diffusion into the GraphCast architecture [11]. Finally, the authors of GraphCast outlined an improvement in the ablation section of the original paper, by creating an ensemble of GraphCast models tuned to predict for different lead times [8].

3 Background

GraphCast is a complex ML model, constituted of 3 major components (fig 1a), operating over 3 graphs. The 3 graphs are made up of the grid nodes, which define the latitude-longitude partitions over the Earth, and mesh nodes, which define a homogeneous spherical graph over it. Each grid node contains the raw features: 5 surface variables, 6 atmospheric variables repeated 37 times over different altitudes, forcing terms, and constants. The surface and atmospheric variables are also the targets. The forcing variables include features such as total solar radiation, and do not need to be predicted because they are easily calculated. The mesh graph contains both local and long-distance edges and is used by the processor for the bulk of the model's computation. For further mesh graph details, see [8]. The remaining two graphs are bipartite. The grid2mesh graph uses directed edges from grid nodes to mesh nodes. The number of grid nodes connected to a single mesh node is defined by a radius parameter. The mesh2grid graph is the inverse of the grid2mesh graph, but each grid node is connected to exactly the 3 closest mesh nodes. All edges of every graph have 4 features, the length and the relative position between sender and receiver.

The first of the constituent parts is the encoder, which is further subdivided into the embedder and the encoder GNN. The former takes the raw features of the two input states X^t and X^{t-1} and converts them into an embedding. The latter computes a single message-passing step over the grid2mesh graph. Next, the processor uses message-passing over its 16-layer, unshared-weight GNN, defined over the mesh graph. The latent representations of the mesh nodes are then finally transferred to the decoder. The decoder is composed of 2 sub-parts, similar to the encoder. It takes one last message-passing step over the mesh2grid graph, creating the final latent representation of the grid nodes. Furthermore, it translates the latent embeddings back into the feature space. This output is then added to the previous state to finally predict the next state $X^{t+1} = X^t + Y^t$, where Y^t is the model output.

This model is very complex, which forces me to gloss over many details. Some that should be mentioned are that the states are sampled every 6 hours. I.e., state X^t is 6 hours later from X^{t-1}. Moreover, the resolution of the grid nodes is $0.25°$ for the model described in the paper, and $1°$ for the smaller model used in my extension. Also, the number of atmospheric levels is 13 instead of 37 in the smaller version. Last thing of note is that the inputs and outputs of the model are normalized and unnormalized respectively given historical data. For many more details, such as the training regime, loss function, and the target weightings, see [8].

4 Research Reproduction

I managed to download and manipulate the pretrained GraphCast model. I was also able to load preprocessed example batches of various lengths, pass them through the model, make autoregressive predictions, and compare them to the target values. The comparison was done both numerically, allowing me to also compute the loss, and visually, by generating plots of the different variables. I do not include such plots here because of a lack of space, and because they are a direct replication of the original (which can be found at [8], pp. 92-98).

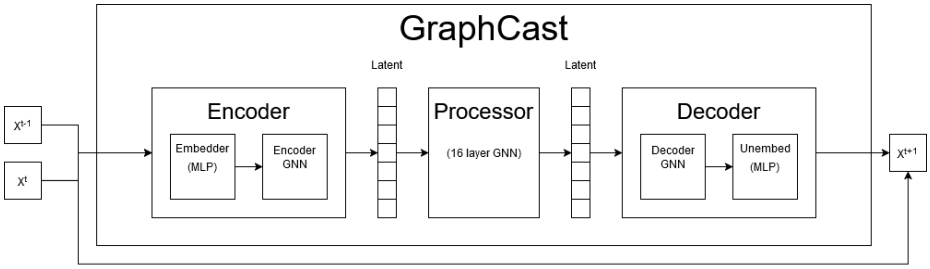
I failed to replicate many other aspects of the original study. I could not perform a single backward pass on my device because of hardware limitations. And I did not pursue this avenue further on Snellius either, because retraining the massive network would have been infeasible in such a short temporal scope.

Subsequently, I managed to stream the ERA5 data, used in the original paper, as well as perform some preprocessing steps. But I ran out of time before I could complete this endeavor and create my own batches akin to the example batches. I successfully computed and appended the solar radiation data. Moreover, I managed to downsample the resolution from the $0.25°$ to the required $1°$. Furthermore, I correctly split the data into inputs and targets. But, most importantly, I failed to convert the `xarray.DataArray` backend from `NumPy` to `JAX`, which prevented me from using the streamed data with the GraphCast model.

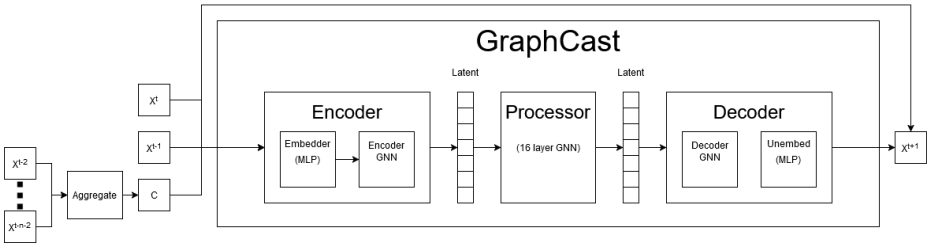
Because I could not use the streamed ERA5 data, replicating any of the claims, skill scores, and statistical tests discussed in the original paper was not possible. That means no comparisons were made with HRES, the efficacy of the model was not validated, extreme weather forecasts were not made, and the effects of training data recency went unexamined in my replication. However, I do not believe I should be judged harshly for this failure as a proper replication in such a short time would have been a nigh impossible Herculean effort for even seasoned AI experts.

5 Research Extension

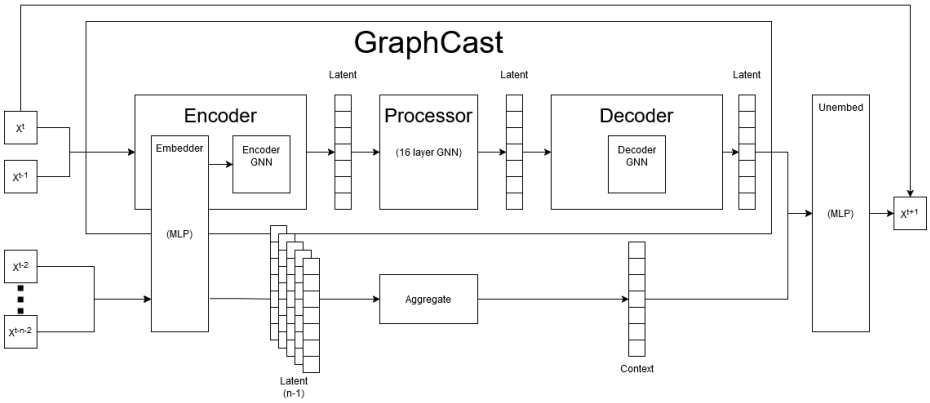
Because the input states do not satisfy the Markov assumption, better performance can be gained by including more information. However, each additional state comes with significant computational overhead and diminishing returns.



(a) *A simplified overview of the GraphCast architecture. The model is constituted of 3 main parts: the encoder, processor, and decoder. The encoder first embeds the two input states, and then computes a single message-passing step over the bipartite grid2mesh graph. Next, the processor computes 16 message-passing steps over the mesh graph. Finally, the decoder decodes with a single message-passing step over the mesh2grid graph and translates the embeddings back into the features. The next state prediction is acquired by adding the model output to the previous state.*



(b) *Ideally, the context state would be added to the input, and the aggregate function could be trained to extract the most important features.*



(c) *Because training GraphCast from scratch is infeasible in this project, the context is added via a parallel pipeline. The context states make use of the pretrained embedder and become translated into the latent space. The aggregation function then extracts a single feature vector. Finally, this vector is concatenated with the latent representation of the grid nodes, and a new MLP is trained to output the residuals of the next state.*

Fig. 1: *The original GraphCast architecture (a), an ideal extension (b), realistic implementation given the brevity of the project (c)*

A more efficient approach is to aggregate only the useful information, and discard what is redundant/useless. This way, for the computational price of one additional state, information from the larger temporal context can be utilized. The general way to derive the context state C is to aggregate over n states from X^{t-n-2} to X^{t-2} (fig 1b). The aggregation can be something as simple as a mean over the states. If older states are considered less informative, this could be a weighted average. An alternative is to train another neural network to extract the features GraphCast can use. A transformer network would be ideal for such a task because it can best summarize the shared information of the n context states [17].

A major issue with such an extension is that it requires both retraining GraphCast, and backpropagating the gradients to the aggregation function. Figure 1c shows an alternative that avoids both of these issues and can be completed within the short time frame of this project. By utilizing the GraphCast embedder, the context states can be processed from raw data into the latent space by a pretrained MLP, which likely extracts useful features. A small issue with this is that the embedder takes 2 input states and produces one latent representation. That means that $n \geq 2$, and that for every latent vector, a sliding window over the context states should be used. Following that, the latent vectors need to be aggregated (averaged; via a transformer; etc.). Lastly, the latent context vector is concatenated to the outputs of the decoder GNN, and the residual Y^t is computed by a new MLP.

Because of time constraints, in my actual implementation the schematic in figure 1c is simplified further still. The number of context states is set to $n = 2$. This means that the embedder produces only 1 latent representation, and the aggregation function becomes the identity.

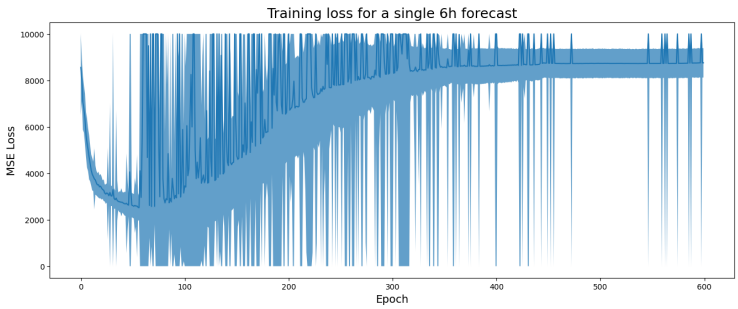
5.1 Experiments & Results

I acquired and concatenated the latent representations of the grid nodes and the contextual state C as described in the previous section for the example batch. I saved this vector, along with X^t, and the targets X^{t+1}. I then created two MLPs (hidden sizes: 2048×4, 1024, 512). One with an input size to accommodate the concatenated vector. The other to only accept the latent representation of the grid nodes (analogous to the original GraphCast implementation). I then trained both of these networks for 600 epochs, 20 times over using mini-batch gradient descent. The batches (batch size of 512) were derived by sampling over the grid points, rather than computing the loss for the entire grid at once. MSE loss was used, and the selected optimizer was Adam (learning rate of 10^{-4}, L2 regularization of 10^{-3}) [7].

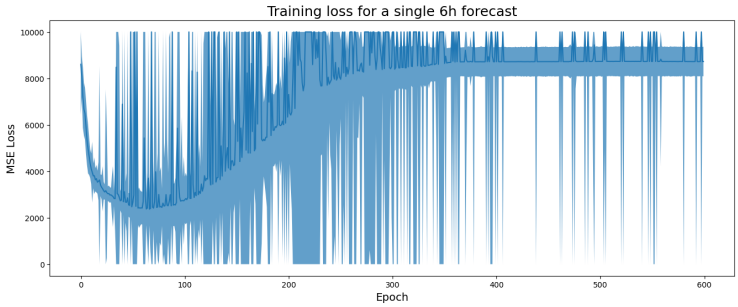
The results can be seen in figure 2. The training losses for the MLP with context and without context are shown by figures 2a and 2b respectively. Both show an unorthodox learning curve, especially given that the MLPs are trained on a single instance. The expectation is that they should overfit, and the loss drop to 0. Instead, the loss drops initially, but then fluctuates and rises to the starting value where it stagnates. The potential reasons and solutions are discussed in

the following section. The similitude of the two curves prevents drawing any conclusions of wether the context state C provides an advantage.

To see if the context features were in any way useful, I examined the model weights. After training, I had 20 context processing MLPs, and I analyzed the first layer weights for the context features. The overall mean was around -0.395, which indicated that the features were being used. Though they could be balanced by the bias term. I then attempted to look at the weights which changed little across runs, hoping that they would represent a proxy for useful features. Only $\sim 0.6\%$ of the weights had a standard deviation $\sigma < 0.1$, very likely due to variance. Moreover, $\sim 90\%$ of them had the highest absolute value. The higher the value, the smaller the variance because of the regularization (fig. 3). Thus, the weights that were randomly high across runs were entailed to have a small standard deviation. So, analysis of the weights also failed to show if the context features were in any way useful.



(a)



(b)

Fig. 2: *The training curves for the two MLPs. Training curve for MLP with context features (a). Training curve for MLP without context features (b). The shaded area is the standard deviation. The outputs are clipped between 0 and 10000 for clarity.*

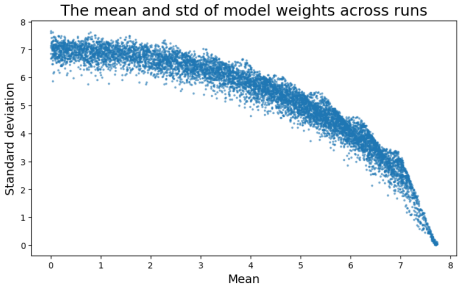


Fig. 3: *The mean and the standard deviation of the model weights are inversely correlated, likely because of regularization. Pearson* $p << 0.0001$.

6 Discussion

It is clear that this implementation of the discussed extension failed to improve over GraphCast. But that does not mean that it cannot possibly work. There are many aspects of this project which contributed to the poor/nonexistent results. To begin with, the training curves in figure 2 indicate that there were fundamental issues with training. Because of the time limitations, I could not do any hyperparameter tuning. Moreover, with the benefit of hindsight, I should have clipped the gradients to prevent those extreme fluctuations (one batch had a loss of $\sim 3.56 * 10^{17}$). Furthermore, GraphCast was trained to produce normalized outputs. Because I could not get the inverse normalization to work with my hacky implementation, I had to hope that my MLPs would simply learn to output the non-standardized residuals.

6.1 Challenges

In this section I will outline the many challenges I ran into during replication and development, to give the reader an idea of the difficulty of this project and the complexity of GraphCast. First, streaming the data came with its own set of issues. The required resolution was not available, preprocessing steps had to be taken, the data format was incorrect, and the pipeline by which the authors of GraphCast processed it was not available. The GraphCast library has 26 Python files that I needed to get acquainted with to find the relevant functions, classes, and instructions. By the end, I made considerable leeway in processing the data, but it was still unsuitable for forward passes through the model. Thus, I had to abandon this pursuit in exchange for more tangible goals.

Next, I endeavored to retrieve the latent representation of the features after they have been embedded by the encoder. This was extremely challenging because the GraphCast object was instantiated in a function, did not contain its own model weights, was wrapped in several other objects, and it was called by a function wrapped in a function, wrapped in another function, and so on. On top of that, this code used the `JAX` library and asynchronous just-in-time (JIT) compilation. So, the call trace was pretty much uninterpretable, and returning the embeddings all the way up this trace seemed unlikely. Thus, I attempted

to circumvent the whole rigmarole, and to simply save the embeddings in the GraphCast object. This was accomplished, and to illustrate the number of object wrappers, this is the line that retrieves the embeddings:

```
predictor.predictor.predictor.predictor.grid2mesh_gnn.embeddings
```

Unfortunately, that would have been too simple. The saved embedding did not return an array of values as expected, but instead gave me an object of type `jax.interpreters.partial_eval.DynamicJaxprTracer`. I unsuccessfully attempted to use the `jax.pure_callback` function to convert these objects to `NumPy` arrays. Further research revealed that these objects are abstract values which define only the shape and type of arrays [5, 6]. The actual values are passed in asynchronously and can only be retrieved by returning them from the JIT-compiled function. Because the embeddings were so many layers deep, this presented a significant obstacle. But, one by one, I began modifying the files of the different classes and wrappers and managed to extricate the embeddings up to the `autoregressive.py` file. Here however, a function which was supposed to return them was wrapped in `haiku.scan`. This made it be called repeatedly and changed the way returns worked. Luckily, I found that I could couple the predictions it was returning with my embeddings into a single tuple. I then continued resolving whatever errors the compiler threw at me, and eventually had the embeddings out in the global environment. The same was repeated for the latent features of the decoder. Ultimately, I had acquired both the embeddings and the latent features.

However, I needed the context embeddings, but I only had the embeddings from the input states. Next, I figured out how to split off 4 input states and pass them to GraphCast. Furthermore, I modified GraphCast to process the older two states into the embeddings. Finally, I returned these embeddings via the newly paved exit. These were no easy feats either, but I shall not expand on them with as much detail to preserve space.

Regardless, after an enormous amount of work, I had both the context embeddings and the latent features of an example batch. I quickly set up some MLPs in `PyTorch` and uploaded everything to Snellius. I then spent a few hours on analyzing the data but could no longer add or try anything else as the time had run out. To sum up, I hope this section suffices to show that I really did all I could in the time that was available to me.

6.2 Conclusion

Whether the addition of a context state in GraphCast helps or not remains unclear. I attempted to implement it in this project but given the time constraints and the complexity of the problem, I could not get any definitive results. Still, the theoretical argument remains, and the extensions I outline in figure 1b, if the computational budget is sufficient, or figure 1c, if the resources are limited, should be tested properly in the future. The issues with training the MLPs seen in figure 2 might be circumvented by hyperparameter tuning and gradient clipping. Loading in more data would also permit more extensive testing. In conclusion, in this paper I outline a way of implementing the context state, the many issues I have encountered, as well as potential solutions.

References

1. Bauer, P., Thorpe, A., Brunet, G.: The quiet revolution of numerical weather prediction. Nature **525**(7567), 47–55 (2015)
2. Benjamin, S.G. *et al.*: 100 years of progress in forecasting and NWP applications. Meteorological Monographs **59**, 13–1 (2019)
3. Espeholt, L. *et al.*: Deep learning for twelve hour precipitation forecasts. Nature communications **13**(1), 1–10 (2022)
4. Frydenberg, M.: The chain graph Markov property. Scandinavian journal of statistics (1990)
5. JAX docs: FAQ, https://jax.readthedocs.io/en/latest/faq.html#how-can-i-convert-a-jax-tracer-to-a-numpy-array.
6. JAX docs: Thinking in JAX, https://jax.readthedocs.io/en/latest/notebooks/thinking_in_jax.html#jit-mechanics-tracing-and-static-variables.
7. Kingma, D.P.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
8. Lam, R. *et al.*: GraphCast: Learning skillful medium-range global weather forecasting. arXiv preprint arXiv:2212.12794 (2022)
9. Lopez-Gomez, I. *et al.*: Global extreme heat forecasting using neural weather models. Artificial Intelligence for the Earth Systems **2**(1) (2023)
10. Nguyen, T. *et al.*: ClimaX: A foundation model for weather and climate. arXiv preprint arXiv:2301.10343 (2023)
11. Price, I. *et al.*: GenCast: Diffusion-based ensemble forecasting for medium-range weather. arXiv preprint arXiv:2312.15796 (2023)
12. Rasp, S. *et al.*: WeatherBench: a benchmark data set for data-driven weather forecasting. Journal of Advances in Modeling Earth Systems **12**(11), e2020MS002203 (2020)
13. Ravuri, S. *et al.*: Skilful precipitation nowcasting using deep generative models of radar. Nature **597**(7878), 672–677 (2021)
14. Shi, X. *et al.*: Deep learning for precipitation nowcasting: A benchmark and a new model. Advances in neural information processing systems **30** (2017)
15. Sønderby, C.K. *et al.*: Metnet: A neural weather model for precipitation forecasting. arXiv preprint arXiv:2003.12140 (2020)
16. Subich, C.: Efficient fine-tuning of 37-level GraphCast with the Canadian global deterministic analysis. arXiv preprint arXiv:2408.14587 (2024)
17. Vaswani, A.: Attention is all you need. Advances in Neural Information Processing Systems (2017)
18. Weyn, J.A., Durran, D.R., Caruana, R.: Can machines learn to predict weather? Using deep learning to predict gridded 500-hPa geopotential height from historical weather data. Journal of Advances in Modeling Earth Systems **11**(8), 2680–2693 (2019)
19. Zheng, L., Xing, R., Wang, Y.: GraphCast-Qtf: An improved weather prediction model based on uncertainty quantification methods. In: 2024 12th International Conference on Agro-Geoinformatics (Agro-Geoinformatics), pp. 1–5 (2024)