



REPRESENTATION LEARNING FOR ATMOSPHERIC DYNAMICS

June 2022

AUTHOR(S):

Pol Garcia Recasens

SUPERVISOR(S):

Ilaria Luise

Christian Lessig





ABSTRACT

.....

The AtmoRep project is a joint effort between climate scientists and CERN physicists. Its goal is to improve our understanding of atmospheric dynamics and how it changes over time, starting from the large amount of observational data accumulated in the last 70 years. This approach is made possible by the newly released ERA5 reanalysis data set, procured by the European Center for Medium Weather Forecasts (ECMWF), that integrates a diverse range of historical measurements into a seamless data set of unprecedented resolution and quality. In this project I have contributed to use state-of-the-art machine learning, specifically transformer architectures, to obtain a data driven description of atmospheric dynamics.



TABLE OF CONTENTS

.....

1	INTRODUCTION	3
2	ARCHITECTURE	4
2.1	Attention mechanism	4
2.2	Multiformer	5
2.3	Training methodologies	6
2.4	Positional encoding	7
2.5	Embedformer	7
3	CONTRIBUTION	7
3.1	Attention maps	7
3.1.1	Prediction plots	8
3.1.2	Attention plots	9
3.2	Data loader	10
3.3	\$CSCRATCH	11
3.4	Embedding external information	12
4	CONCLUSIONS	15
5	REFERENCES	16





1 INTRODUCTION

The AtmoRep project is a joint effort between CERN atmospheric scientist and computer scientists with the goal of applying large-scale representation learning on spatio-temporal atmospheric data. In the Earth sciences, representation learning has not been used yet at large scale, despite having large amounts of unlabeled data available (e.g. satellite observations). AtmoRep aims to use the ERA5 atmospheric reanalysis dataset, that provides 70 years of observations of atmospheric variables, to train a representation network that will help further downstream applications such as hurricane tracking forecasts or super-resolution, also improving the placement of solar and wind farms. To do so, we introduce a novel architecture, based on Transformer models, that is well suited for multiple physical fields with multiple variables. Therefore, it can also be extended to other fields, including oceanic reanalysis or simulation output.

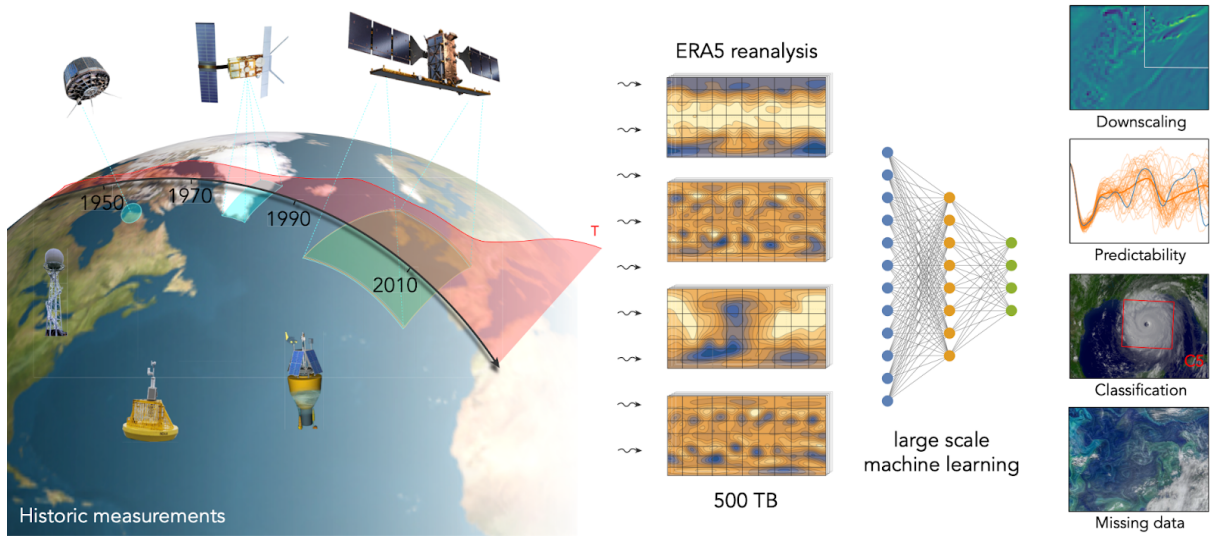


Figure 1: AtmoRep pipeline.

Although ERA5 provides hourly estimates for a large number of atmospheric variables from 1979 onwards, we have focused on temperature, geopotential and the potentials of the wind field. Figure 2 illustrates the vorticity grid for a given hour. The observational data has associated systematic uncertainties concerning, for instance, data sparse locations (e.g pre-satellite era), so the dataset also provides an estimation of the uncertainties via a 10-member ensemble at three-hour intervals. The reanalysis data has been gridded to a regular latitude-longitude grid of 0.25 degrees resolution, sampled at 137 different layers of altitude, and stored in GRIB file format. Although the whole dataset consists of 6 PB of data, we plan to train our network with a subset of 500 TB - 1000 TB. We have made the design choice to assign and store each field per month to a GRIB file of approximately 1.5GB. The latitude of the grid is 720 and the longitude 1440. The number of hours depend on the days of the month, e.g 720 hours for a 30 days month.

To perform representation learning of atmospheric dynamics we introduce the Multiformer, a transformer-based architecture that integrates one transformer per field and couples them using attention. The transformer models provide good quality assessment and interpretability of the predictions due to the attention maps, as we can check which tokens of the input sequence the network is attending to. We have also decided to train



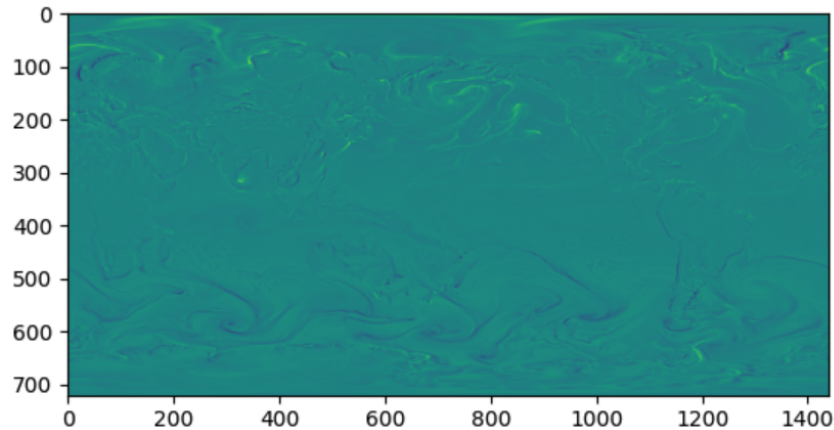


Figure 2: Vorticity grid.

an embedding transformer for each field to improve the quality of the projection.

We followed the “start small, think big” principle. We first trained a transformer model with one field for the standard forecasting task, and iteratively extended the architecture to couple with the different fields. We then switched from the forecasting task, predicting the following token at the timestep $t + 1$ based on the previous t timesteps, to the forecasting BERT training method. Our prototype has evolved from a simple transformer trained on simulated data to a large-scale multiformer trained on ERA5 observational data.

During the 9-weeks summer internship at CERN, I have been able to contribute to AtmoRep by working on tasks:

- Implemented a plotting pipeline that generates the attention maps of each transformer and visualizes them during training.
- Implemented a Data Loader class to abstract the GRIB file representation.
- Benchmarked the performance of CSCATCH, a high performance cache system from Jülich Supercomputing Center.
- Studied different methodologies to include external information to the tokens.

2 ARCHITECTURE

2.1 Attention mechanism

The self-attention mechanism computes a representation of a sequence by relating its different positions via an attention function. The attention function defines a mapping between query-key and value pairs to an output, where the output is computed as a weighted sum of the values, each weight defined as the compatibility of the query with the corresponding key. The queries, keys and values result from the dot product between the input and the learnable query, key and value matrices. Equation 1 describes the “Scaled Dot-Product Attention”, proposed in the “Attention Is All You Need” paper [5]. This attention function improves the standard dot-product attention (simple multiplication of query, keys, values) by scaling the dot product by the number of dimensions





and introducing a non-linear softmax function. The attention map is defined as the dot product between the queries and the values.

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right) V \quad (1)$$

To allow the model to learn to attend to information from different representation subspaces, the Vaswani et.al [5] transformer proposes the Multi-Head Attention. Instead of projecting the query, key, values to d dimensions, each attention head projects its own query, key, values to d/h dimensions, where h is the number of attention heads. The final queries, keys and values result from the concatenation of the attention heads, followed by a projection via a learnable matrix.

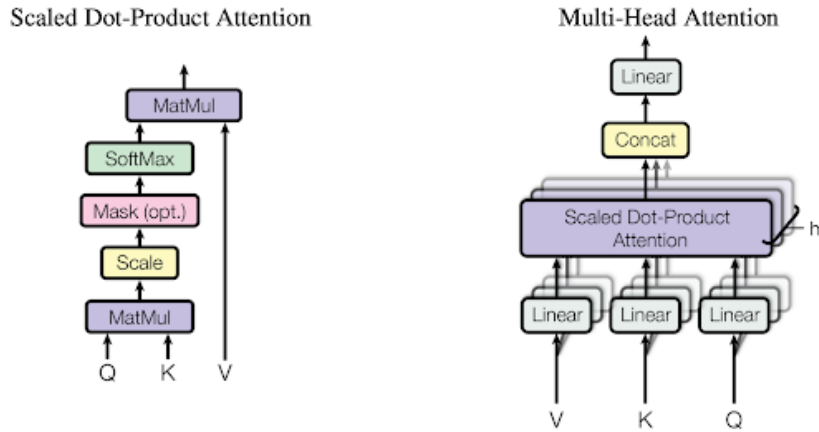


Figure 3: Multi-Head Attention [5].

We can introduce information from one head to another via **cross-attention**, combining the keys and values from one head with the queries of the other. With this mechanism, we are able to merge two different representation subspaces, the queries from one head are used to highlight the values of the other head.

In our architecture we introduce the coupling head. This head applies the cross-attention mechanism by combining the key-values of the head of one transformer, trained with one field (e.g. vorticity), with the queries of another transformer, trained with another field (e.g. divergence). It is important to combine the fields via the cross-attention mechanism to protect the input for each transformer, avoiding mixing them and preserving the interpretability of the attention maps per each transformer/field. We call this a **Interformer**, the transformer architecture that uses coupling heads to combine different fields.

2.2 Multiformer

The **Multiformer** architecture includes an Interformer per dynamic field, using the cross-attention mechanism to combine them on each attention layer. The static fields (e.g. orography) do not change over time, so there is no need to couple them to the other fields. Thus, the attention layer for the static transformer is only based on self-attention. Our complete implementation of the Multiformer is composed of four Interformers, one



per each dynamic field, and one standard transformer for each static field.

Besides including the cross-attention heads, the implementation of the attention layer follows the standard vanilla Transformer. Each head of the layer is composed of an attention block (self or cross attention) followed by an MLP. We also include residual connections, adding the input from the last layer to its own output, to ease the flow of the gradients. We prepend to the source an extra token, named class token, that accumulates the information of the other tokens. The class token is initialized with the mean of the spatio-temporal cube. After the attention layers, we pass the class token through a tail network to make the prediction. Each layer of the tail network is composed by a linear transformation of the input followed by a GeLU non-linear activation function.

To train the model we extended the BERT masked language model, combining the masking with the forecasting task. Moreover, to make the training feasible for large amounts of data we have implemented a data-parallel distributed training with Horovod, using the computational facilities of Jülich Supercomputing Center in a multi-node multi-GPU environment.

2.3 Training methodologies

As mentioned before, we iteratively added complexity to the architecture. We first used the most straightforward training methodology, i.e forecasting. Each element of the batch corresponds to the index of a target token that the network has to predict. Based on this target token, we construct a spatio-temporal cube using the neighborhoods of the token for the previous timesteps. The flattened spatio-temporal cube is going to be the source from which the network will predict the central token for the next timestep. We trained the network using the l_2 loss between the target and predicted tokens.

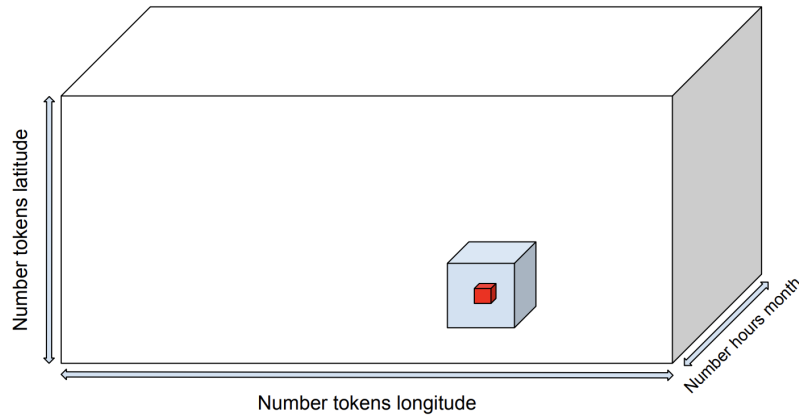


Figure 4: Prediction of the token at time $t+1$ based on the source cube.

Following the implementation of BERT [2], we have improved the training methodology with masked language model (MLM) unsupervised task. We mask a random number of tokens of the spatio-temporal cube with the mean of the cube, and then predict those masked tokens using the class token. Since we are interested in forecasting, and we want to enforce the network to learn to predict future tokens, we also extended the standard masked language modeling task to forecasting masked language modeling. Instead of





masking tokens inside of the spatio-temporal source cube, we mask tokens outside of the cube for future timesteps. This methodology leads to a better convergence than the standard forecast training method.

2.4 Positional encoding

The self-attention layer treats each token independently, ignoring the local position of the token in the spatio-temporal cube. This information can be very relevant for the model, and the most frequent technique to include is the positional encoding. We implement a type of positional encoding, named harmonic positional encoding [5], that adds to the representation of the token its local position inside the spatio-temporal source cube via sinus and cosine functions. A simple indexing of each position can lead to problems as for long sentences the indices can grow large in magnitude, and normalizing sentences with different length is not consistent. As the sinus for each position is different, we have a unique way of encoding each position while being able to deal with sequences of different lengths.

Besides the local index of the token inside the cube, it may be beneficial for the network if we include external information. We append to the projection of each token information about the year, month, day and the latitude and longitude of the predicted token. Therefore, to prepare each token we project them to $d - \text{len}(i)$, where d is the embedding dimension and i the external information array, append i and add the positional encoding.

2.5 Embedformer

Before passing the source tokens to the Multiformer, we first embed them using a linear layer. Adding complexity to this first embedding increases the training cost, and since it can be separated from the Multiformer training, we have decided to implement a second architecture, named **Embedformer**, to learn more complex representations. We pre-train a standard transformer (without cross-attention) per field and, once trained, we can train the Multiformer projecting the tokens using the Embedformer instead of using the linear layer.

3 CONTRIBUTION

3.1 Attention maps

While it is not a fail-safe indicator, attention maps provide a way to assess which tokens of the sequence the network is focusing on. Due to the multi-headed attention, the model learns to attend to information from different representation subspaces, and we can interpret them using the attention maps of each head. Since we are dealing with dynamic fields, we are also able to compare the parts of the sequence that the network is focusing on with the ones that one would theoretically attend to when computing the forecasting.

To increase the interpretability of our model, the first task was to implement a plotting pipeline that illustrates both the test performance and the attention maps. The plots are





generated using the worker with rank 0, so in the case that we are distributing the training to more than one worker (GPU), we set up a barrier to force the workers to wait until the computation of the plots ends. The prediction plots are generated and stored as pdf files after each epoch, and the attention plots are computed once the training has ended.

```
if with_hvd :
    hvd.allreduce(torch.tensor(0), name='barrier')
```

3.1.1 Prediction plots

We validate the training of the model using the Mean Square Error between source and predicted tokens, computing the predictions for a batch of data randomly selected from the year 1979. For each element of the batch, we store the source and predicted tokens in a list. The user just needs to specify via the `iidx` variable which sample of the batch is going to be plotted. We have used the `matplotlib.pyplot` library to compute the plots. In figure 5 and figure 6 we show examples for the target and prediction plots. Our current implementation of the `multiformer` predicts the vorticity field, so both the target and predicted tokens correspond to a vorticity token. One of the future improvements of the training technique would be to predict a token per field, possibly providing stronger gradients for the training.

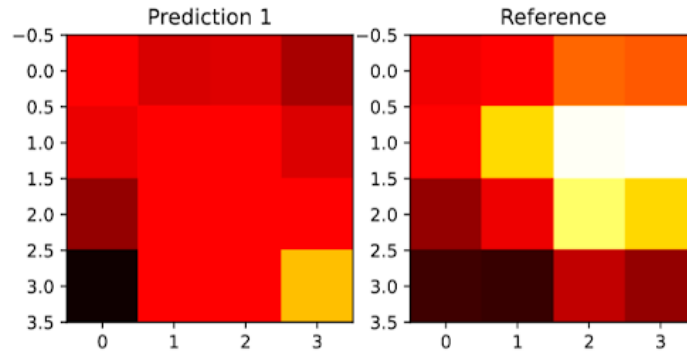


Figure 5: Predicted and reference tokens.

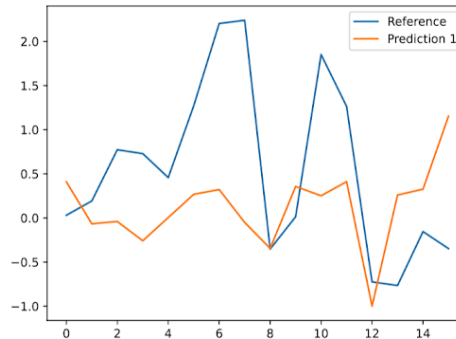


Figure 6: Comparison between predicted and reference values.





3.1.2 Attention plots

Each head of each block of the transformer has its own attention matrices. To get the attention maps and values given a source token, we prepare the token and evaluate its forward propagation through the network. We slightly changed the forward function to return the attention map and the values instead of returning the prediction. Intuitively, the query and keys determine which values to focus on. The following function retrieves the attention and values from both the self-attention and cross-attention heads.

```
def get_attention( self, fields) :
    '''Get attention map (for analysis)'''
    # layer norm for each field
    fields_lnormed = []
    for ifield, field in enumerate( fields) :
        fields_lnormed.append( self.lnorms[ifield](field) )

    atts = []
    vsh = []

    for head in self.heads_self :
        att, v = head.get_attention( fields_lnormed[0], fields_lnormed[0] )
        atts.append( att)
        vsh.append( v)

    for ifield, head in enumerate( self.heads_coupling) :
        att, v = head.get_attention( fields_lnormed[0], fields_lnormed[ifield+1])
        atts.append( att)
        vsh.append( v)

    return torch.cat( atts, 0), torch.cat( vsh, 0)
```

Similarly to the test function, we compute the predictions and get the attentions from a batch randomly selected from 1979. We control the number of attention maps that are generated with a variable that indicates the number of patches that are randomly selected from each file, corresponding to one month of data. Once the attention values are stored on disk, the plotter reads the files and computes the plots. We plot the attention maps, the values, the source and the target-predicted tokens for a specific field.

With the attention maps we are able to assess which tokens of the sequence the network attends to and how the attention evolves during time. We plot the attention maps per layer, head and timestep, so we can easily visualize which parts of the sequence each head focuses on. Deeper layers are more typically semantic, but each token also accumulates additional context each time self-attention is applied [1] which also adds more noise. The following plots are computed with the vorticity multiformer coupled to the orography field. The last head corresponds to the cross-attention head i.e the head that combines the queries from the orography transformer and the key-values from the vorticity multiformer. In figure 7 we can easily observe that its attention map has a different shape 7. Moreover, in figure 8 we show that, by applying the sigmoid function to the previous attention maps, we are able to highlight the most relevant parts 8.

Figure 9 shows the evolution of the vorticity source token for the different timesteps. Given the index of the target token, the source token is composed by a window of neighbors that surround the target index.



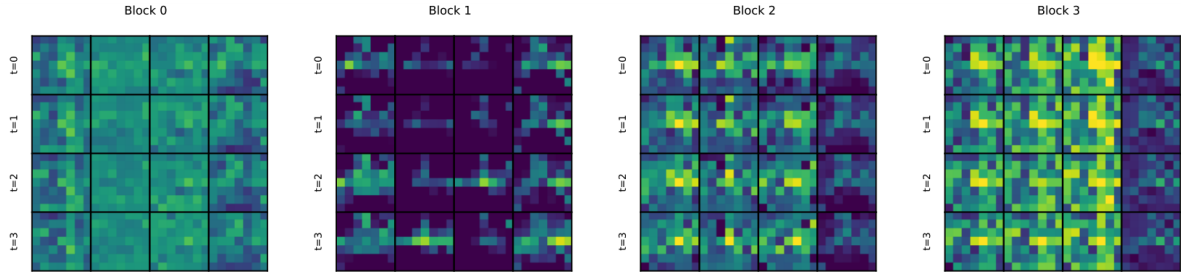


Figure 7: Time evolution of the attention maps for each layer and head.

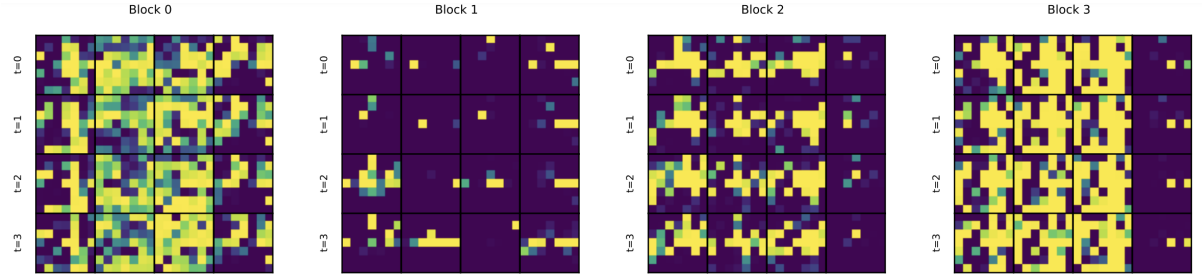


Figure 8: Time evolution of the sigmoid-attention maps for each layer and head.

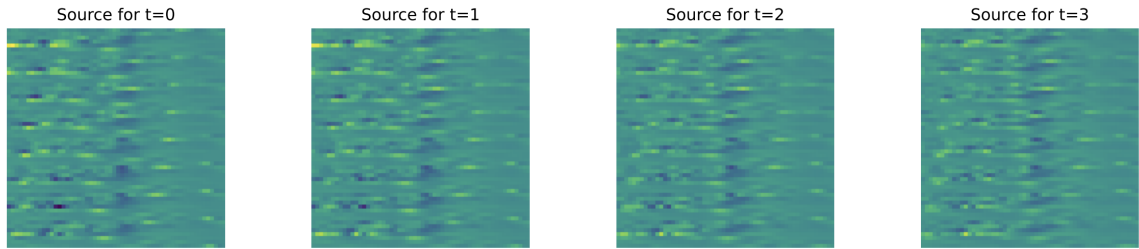


Figure 9: Time evolution of the source tokens.

We can also plot the values of a specific head [10](#), corresponding to the linear projection of the token using the learned value matrix as a sanity check. The following scatterplot shows the values from the five tokens.

3.2 Data loader

The data loader class abstracts the file representation and improves the reusability of the code. Each file corresponds to one month of ERA5 data for a given field. Given a set of year/months and fields, the data loader returns a list with shape [year/month, [field, x]], where x is a torch tensor with shape (hours/month, latitude, longitude). Therefore, for each randomly selected month of a year, we load the data from the file for each different field.

In order to generate the datasets for train and test, we shuffle the selected files and randomly pick a set of indices, representing the target tokens, from which we construct the spatio-temporal cube for the sources. We have to ensure that each target index has



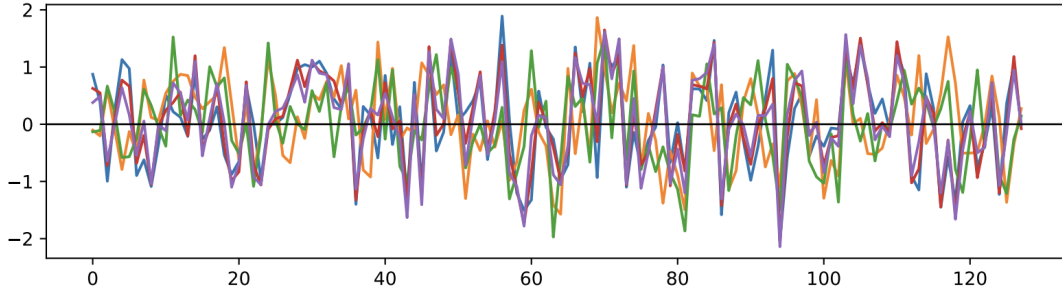


Figure 10: Values of the head for five selected tokens.

access to the previous *num_steps_past* hours of data so, if specified, we pad the file to prepend the selected number of hours from the past month of data, corresponding to the previous file. We prepend at least *num_steps_past* to properly construct the source for the first indices of the month.

To load the GRIB files, we use the xarray library that supports the GRIB engine using the cfgrib interface. However, we have experimentally found that xarray adds a memory overhead to the files so we decided to extend the data loader to support other file formats. After the file is loaded into a tensor, we tokenize the data so each grid point corresponds to a shape (tokensize, tokensize). Therefore, we extend the tensor with shape (hours_month, lat, long) to a tensor with shape (hours_month, lat/toksize, long/toksize, toksize, toksize). The following code gives an example of the grib file loader.

```
def grib_file_loader(fname, field, time_padding = [0,0], static=False) :
    ds = xr.open_dataset(fname, engine='cfgrib')[field]
    if not static:
        ds = ds[time_padding[0]:(ds.time.shape[0] - time_padding[0])]
        x = torch.from_numpy(np.array(ds, dtype=np.float32))
    ds.close()

    return x
```

To ease the convergence of the model and make the training feasible for small machines, the user can also smooth the data of the file via the blur functionality of the opencv library.

```
if self.smoothing > 0 :
    sm = self.smoothing
    data_ym = [torch.from_numpy( cv.blur( data_ym[k].numpy(), (sm,sm))).unsqueeze(0)
               for k in range(data_ym.shape[0])]
    data_ym = torch.cat( data_ym, 0)
```

3.3 \$CSCRATCH

We have been awarded computing power at the Juelich Supercomputer Center, so we will train our model with the Booster module of the JUWELS machine. To store the ERA5 data, we use \$CSCRATCH, a GPFS temporary storage location for applications with large size and I/O demands. JUWELS uses Slurm [3] as the resource manager and batch system, controlling the jobs are submitted to different modules.





The high-performance storage tier (HPST) is a cache layer on top of the \$SCRATCH file system providing high bandwidth and low latency access, with the goal of speeding up I/O in compute jobs. Due to our high demanding data loading process, where we have to load terabytes of data, we wanted to benchmark the Jülich cache system to check if it could reduce the cost of the data loading. As it is an inconsistent cache, the user has to take care of the state of each file. This is done via the command line tool ime-ctl.

The first step was to move the grib files to the cache via the ime-ctl –prestige command. With the flag –globres=fs:cscratch@just, we are able to prevent Slurm from allocating resources when the cache system is unavailable. Although one would expect that caching our data files improves the loading process, we proved that CSCRATCH has a deficient performance. Further investigation led us to conclude that it has to lookup for the file location in disk to ensure the data consistency between the cached and the original version of the file. This ends up adding an overhead in the loading process due to the relatively small monthly file size of 1.5GB. Although we have proved that with bigger files, e.g yearly files of 18GB, the cache system becomes more competitive, we have continued to use the monthly files because it is important for the machine learning.

3.4 Embedding external information

The attention model ignores the positional information of the input sequence and doesn't introduce any spatial inductive bias, returning the same output regardless of the order of the inputs [4]. However, for image processing the spatial structures are important and the network must know how to exploit spatial relationships in the input data, so one must explicitly add this information. Via the harmonic positional encoding, we add to the representation of the sequence the **local position** of each token inside the spatio-temporal source cube.

There is extra information, such as the year, the day in the year, the hour in the day and the latitude and longitude of the target token that may be useful for the network. Our first approach was to append a tensor with the external information to the representation of the token. However, we experimentally show that the network is not using the global information. If we assign a hardcoded value to each of the five dimensions, the performance of the model doesn't degrade. For a 12-epochs trained model, we get the following average test loss:

- Baseline: 0.9719
- Year hardcoded to 1980: 0.9855
- Hour hardcoded to 0: 0.9719
- Hour hardcoded to 12: 0.9719
- Lat and long hardcoded to 0: 0.9719

Although the network performance is slightly worse when we harcode the year, and one would conclude that it is using the year information, the performance omitting the other fields is exactly the same as the performance of the baseline model. Figure 11 shows





Figure 11: Comparison between trainings hardcoding each information.

the test loss results of the different hardcoded configurations.

The baseline configuration appends the external information to the projection of the token. Since the network appears to ignore most of it, we have tried different alternatives. Figure 12 shows the results of each configuration, none of them appears to improve the performance of the baseline model. The configurations that we have tried are the following ones:

- Append the information to every token
 - Project the token information tensor before appending.
 - Append the token information to the token and then project the result to `dim_embed` dimensions. This configuration leads to the *append info pre embed* loss in figure 12.
- Initialization of the class token
 - Learnable embedding of the information into class token. This configuration leads to the *class token init with info* loss in figure 12.
- Add the external information after the attention layer
 - We have appended the information after the last attention layer, and before the predictive tail network. This configuration leads to the *info after transformer* loss in figure 12.

One would think that the global latitude and longitude of the token is relevant for the network to compute the predictions, as it is meaningful for theoretical weather forecasting. Since appending directly the external information to each token doesn't improve the performance of the model, we propose a variation of the positional encoding that includes the global position. The latitude and longitude define the global position of the target token in the grid, so in order to compute the global position of each token in the cube,

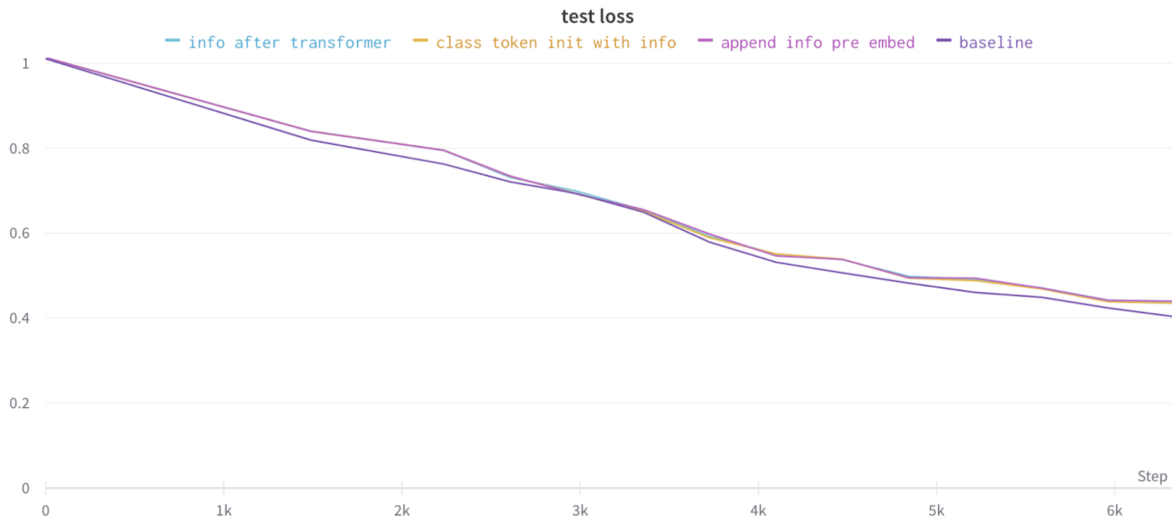


Figure 12: Comparison between trainings for each configuration.

we have to add or subtracting the local indices to the global position of the target.

The following code illustrates our positional encoding, we apply the sinus function to the latitude and the cosine function to the longitude. The *div* variable is still defined as in the standard positional encoding 2.4. The *target_t_idx* variable introduces the information about the timestep of the token inside the cube.

```
pe[:, :, 0::2] = torch.sin((target_t_idx * div)) + 0.5 * torch.sin((target_lat_idx * div))
pe[:, :, 1::2] = torch.cos((target_t_idx * div)) + 0.5 * torch.sin((target_lon_idx * div))

x = x + pe[:, :x.size(1)]

return x
```

Figure 13 shows the results with the new positional encoding. We blurred the data to ease the convergence of the model. One can observe that the new positional encoding with the global information of the token keeps converging to the old positional encoding with the local position of the token. If we train the model with the global position of the token, ignoring the local positions inside the cube, we get the same performance as if we train without positional encoding. *latlong encoding* corresponds to the new positional encoding, *pos encoding + latlong* adds the latitude and longitude the old positional encoding, and *pos encoding latlong* sets the position of all tokens with the global position of the target.

The data used to train the model has been corrected via token normalization, normalizing the values of each token using its values for all the timesteps of the month. One hypothesis is that the token normalization removes the differences between tokens, so the global position of the token doesn't add useful information to the model. We computed the average of each mean and standard deviation per token of the month, and normalized every token uniformly. Figure 14 shows that the baseline positional encoding with the new normalization converges better than the new positional encoding, therefore rejecting the hypothesis.

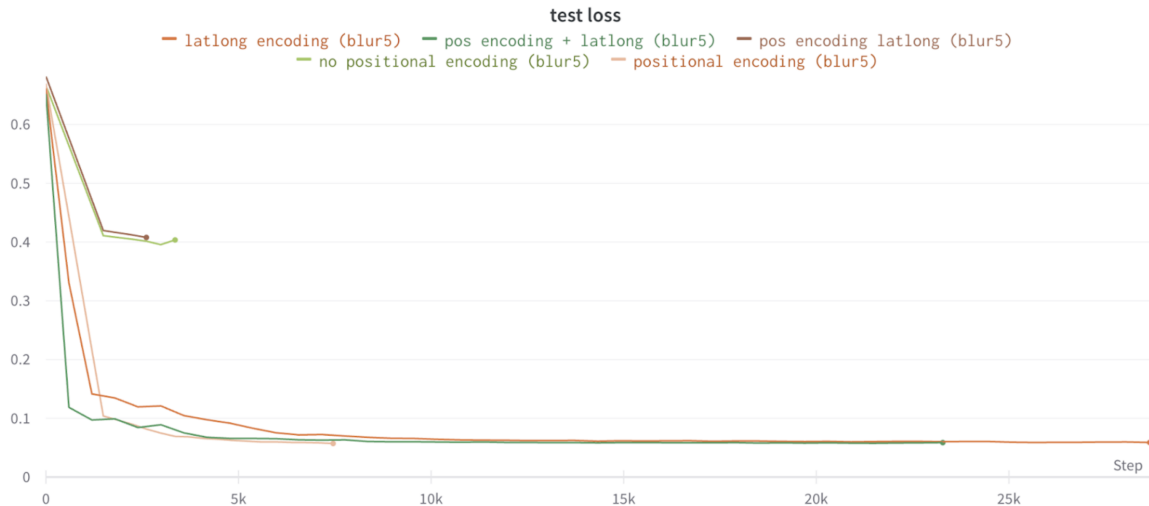


Figure 13: Comparison between the different positional encodings.

4 CONCLUSIONS

In this work, I presented my contribution to the AtmoRep project in the 9-weeks summer internship at CERN. I applied Deep Learning with the goal of improving our understanding of atmospheric dynamics, starting from the large amount of observational data accumulated in the last 70 years. I first implemented a plotting pipeline to assess the training of the network via the attention maps and developed a data loader to abstract the representation of the files. I then benchmarked the *SCRATCH* cache system from Jülich, with the goal of easing the data loading cost. Finally, I experimentally showed that the network was not using the external information, and proposed and experimented with various alternatives. Overall, my contribution helped to design and implement the first prototype of the Multiformer.

We did not find evidence that external information is used, which might become more relevant in a more complete version of the Multiformer. The harmonic positional encoding seems to suffer from aliasing when the number of tokens is high, and does not correctly encode the position for the last part of the embedding dimension of the token, which is why future work should look at deeper analysis of the positional encoding. Regarding the AtmoRep project, we still have to combine the Multiformer with the Embedformer and train them with the whole dataset for the four dynamics fields.



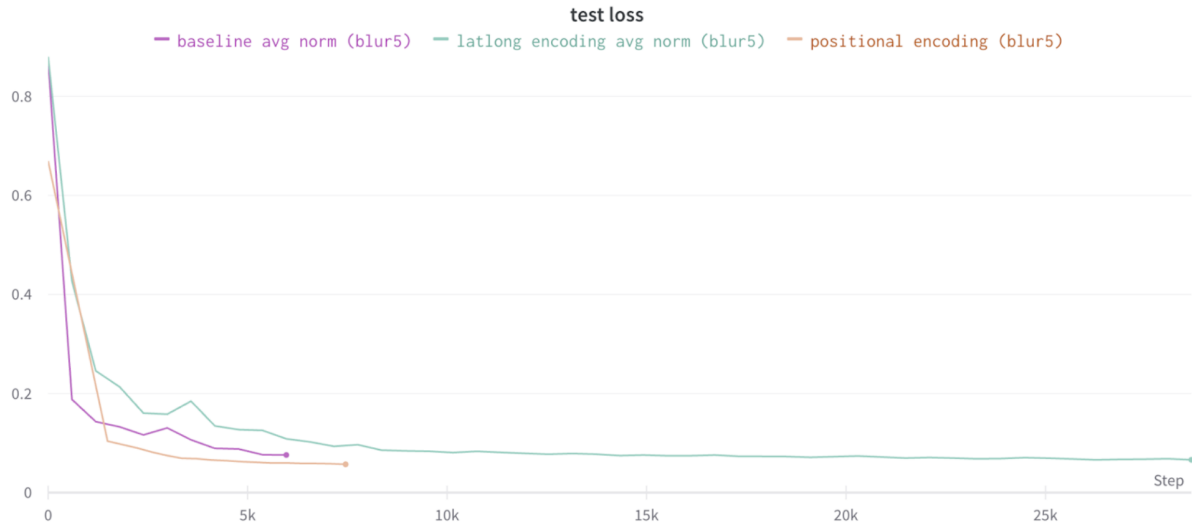


Figure 14: Comparison between the different positional encodings without token normalization.

5 REFERENCES

- [1] Hila Chefer, Shir Gur, and Lior Wolf. “Transformer Interpretability Beyond Attention Visualization”. In: (2020). DOI: <https://arxiv.org/abs/2012.09838>.
- [2] Jacob Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: (2019). DOI: <https://arxiv.org/abs/1810.04805>.
- [3] *GPFS File Systems in the Jülich Environment*. URL: <https://apps.fz-juelich.de/jsc/hps/just/filesystems.html>.
- [4] Andrew Jaegle et al. “Perceiver: General Perception with Iterative Attention”. In: (2021). DOI: <https://arxiv.org/abs/2103.03206>.
- [5] Ashish Vaswani et al. “Attention Is All You Need”. In: (2017). DOI: <https://arxiv.org/abs/1706.03762>.

