Feature engineering, and series application of CatBoost and a PyTorch Convolutional Neural Network: a successful entry for the Kaggle “LEAP - Atmospheric Physics using AI (ClimSim)” competition

Author:	Charlie Wartnaby, Cambridge, UK; charlie(a)wartnaby.org
GitHub for this work:	https://github.com/charlie-wartnaby/kaggle-leap-atmospheric-physics
Kaggle profile:	https://www.kaggle.com/charliewartnaby

1 Abstract

Kaggle hosts machine learning competitions in a wide range of domain areas. This informal report covers my entry for the LEAP Atmospheric Physics competition, where I placed 93 out of 693 entries on the leaderboard[1] for a “bronze medal”. My solution employed the following key techniques:

- obtaining efficient random access to huge CSV source datafiles using novel low-level file handling;
- “vectorisation” of features across atmospheric levels to allow systematic processing and modelling;
- feature engineering using physics-inspired transformations of the raw data provided;
- CatBoost decision tree fitting and generation of predicted outputs;
- implementation of a custom 1-D convolution based (Conv1D) neural network as a second model in PyTorch;
- incorporation of CatBoost outputs as additional input features to the neural network, effectively using the two techniques in series;
- alternatively, using weighted combination of CatBoost and Conv1D predictions, effectively using the two techniques in parallel;
- automated hyperparameter search and feature knockout experiments for both CatBoost and Conv1D models.

The aim of the challenge was to fill in climate grid models with estimated AI-generated data that would be too expensive to compute by using a finer grid of physics-based modelling, to ultimately allow realistic models with good accuracy to be executed more cheaply, enabling more numerous or longer-timespan simulations for the same computing cost. The challenge data was extracted from the large ClimSim dataset [Ref 1].

My code and this report are open source at https://github.com/charlie-wartnaby/kaggle-leap-atmospheric-physics. The main implementation file is `leap_feat_eng.py`.

1 A 'leak' was identified [Ref 4] shortly before the submission deadline allowing teams to score more highly by using information that was not intended to be available; while the prizewinning teams submitted leak-free solutions for review, it is not known how many intermediate teams exploited this leak. This work does not.

2 Contents

3 Input data processing

The sheer size of the training dataset was a cause of some consternation amongst teams in this competition, so particular effort was required in data handling.

3.1 Available computer resources

The methods employed here were moulded by the available disk, RAM, GPU and time available.

Initial experiments were done in the form of a Kaggle Jupyter notebook, and the main code file `leap_feat_eng.py` was maintained with the intention that it could be pasted in its entirety into a single Kaggle notebook cell for execution.

Some early experiments were of necessity performed on a business laptop with only a 4 GB NVIDIA GPU, and debugging in CPU mode on a GPU-free Windows PC.

Most work was done on a PC with the following attributes:

- Intel Core 7 CPU
- 64 GB RAM (but no more than ~10GB was ever obviously consumed)
- 16 GB NVIDIA GTX 3080 GPU (this was a key limitation in choosing batch sizes)
- 1 TB solid-state disk
- Running Ubuntu 24.04
- Miniconda environment installed for Python3 with PyTorch and CatBoost support

A local machine was required to run long modelling phases (e.g. of 36 hours) which would have been impossible within Kaggle's current 30 hr/week GPU quota. Overall the sheer size of the dataset in this competition was a strong determinant of approach.

3.2 Raw data handling

The data for the competition took the form of:

- 9 input "vector" features with values at 60 atmospheric levels
- 16 "scalar" altitude-independent features
- hence (9 x 60 + 16) = 556 input values per record
- 6 "vector" target outputs at 60 atmospheric levels and 8 "scalar" outputs
- hence (6 x 60 + 8) = 368 predicted output values per record
- there was also a string "sample ID" column

The data files provided for the competition comprised:

- a single 170 GB, 10091520 (~10 M) CSV file of training data (inputs and ground-truth outputs hence 556 + 368 = 924 columns), `train.csv`;
- a 6.5 GB, 625000 row CSV file of test data (inputs only, 556 columns), the predicted outputs of which formed the required submission for scoring, `test.csv`;
- a corresponding 446 MB CSV file of corresponding output column submission weights (zeroes or ones), of which only the first row was required, `sample_submission.csv`;
- prior to 19 Jun 2024, a 3.4 GB version of the sample submission weights, with those weights tending to scale the outputs to approximately [-1, 1] range, here `sample_submission_old.csv`;
- the corrsponding output ~4.2 GB, 625000 row, 368 output column CSV file to submit to Kaggle for scoring, here `submission.csv`.

The `train.csv` file in particular was far too large to load into RAM for processing, and by experiment even using the Polars library LazyFrame class and scan_csv() method intended for huge files, access was very slow.

Instead I devised a new approach for working with large CSV input files, implemented in the `HoloFrame` class. This used low-level file operations to give fast random access to anywhere in the CSV input file, as read directly from disk. The key aspects were as follows:

- initially reading through the entire file in binary mode to obtain the byte offset of the end of every line of text (bearing in mind that in CSV, lines do not generally have equal lengths);
- caching those line offsets on disk in pickle format for future reuse, while retaining them in memory for immediate use;
- using Python seek() and read() calls in binary mode to read chunks of the file at arbitrary locations on demand, and in particular groups of rows by index using the predetermined line offsets, which is fast;
- constructing a ‘fake’ or ‘projected’ CSV for an arbitrary slice of file rows, by combining the headings (first file row) with the binary data for a slice of rows into one virtual binary CSV file in memory;
- reading that small CSV file in memory into a Polars dataframe for further processing.

This technique allowed extraction of a small subset dataframe from anywhere in the huge CSV file within milliseconds, fast enough to use "live" during training with randomised training blocks. However, as data processing ultimately required scaling that depended on statistics accumulated over the entire dataset, and resaving in binary format after processing, in the end it was not actually necessary to make random access to the source CSV like this.

3.3 Vectorisation of data by atmospheric level

The data provided in CSV format was necessarily two-dimensional (with different data records spanning the vertical dimension, and values in that record stretching horizontally). However, many of the quantities were really "exploded" vectors across 60 atmospheric levels.

As I intuitively wished to try a convolutional approach that would apply the same kernel to every atmospheric level, the data had to be restructured such that it became 3-dimensional, where the dimensions were:

- 0: the record (each row in original data)
- 1: the physical quantity (feature or output), e.g. temperature or humidity
- 2: spanning the 60 atmospheric levels, from high atmosphere to ground

The `vectorise_data()` method performed this vectorisation. It recognised headings with suffix numbers in the source dataframe (e.g. state_q0002_0, state_q0002_1, state_q0002_2 ...) as referring to different atmospheric levels of the same parameter, which became an array indexed in a dict by its suffix-free name, e.g. "state_q0002" in this example.

3.4 Custom physics-based feature addition

While I did not attempt to actually simulate any physics – the whole point of the challenge, after all, is to avoid microscopic physics-based simulation in favour of a cheaper AI approximation – I did attempt to devise transformations of original input data that I hoped would be more physically

meaningful and hence better predictors of the desired outputs. These "engineered" features were added alongside the original vector features (of which some were then removed as hopefully redundant, to improve performance). In brief these features were:

- Relative humidity in place of specific humidity, following Beucler et al [Ref 2]
- A buoyancy metric also used by Beucler et al, intended to be more meaningful than absolute temperature
- Absolute magnitudes for wind speed and surface stress to supplement the zonal (E-W) and meridional (N-S) components provided
- Absolute air momentum to supplement velocity (or speed)
- Total cloud (as ice and water components were provided separately)
- Cloud integrated sky-down, and separately ground-up, in the air column, considering that would affect sunshine reaching down to some level, or ground-emitted infrared reaching up to some level
- Total instantaneous global warming potential (GWP) of the greenhouse gas concentrations provided (methane, ozone and NO_2)
- Product of total GWP and longwave infrared flux from the ground, which should give an overall degree of heating
- Air pressure, which was not provided, but available elsewhere
- Air density, and various quantities scaled by density in the hope of getting meaningful rates of change that depended less on atmospheric level

See the `add_vector_features()` method for the creation of these features.

The best CNN added 22 custom vector features and omitted 3 provided features which were effectively superseded, and similarly added 5 scalar features and omitted 5 provided ones. As 9 vector and 16 scalar inputs were provided, overall the CNN processed 28 vector and 16 scalar features (excluding the CatBoost output features).

3.5 Prenormalised matrix batch cache file creation

After the addition of custom features, the resulting unscaled matrices (in float64 format due to the very small magnitudes of some quantities) were written in batches to disk, to avoid excessive RAM use during processing.

At this stage statistics were maintained on the minimum, maximum and mean of each parameter in each batch to support subsequent normalisation.

3.6 Normalisation and final batch cache file creation

The values in this challenge varied enormously in magnitude, with temperatures of ~300K but some trend values of (say) $\sim 10^{-40}$. As is usual for machine learning, normalisation is required such that all values are in a reasonable range, ideally around [-1, 1]. Two approaches to normalisation were tried:

- Offsetting each column by its mean (or not in the case of output data), and then scaling by its standard deviation to give values in an approximate [-1, 1] range; this is conventional.

- Scaling taking account of the total range found to give a [-1, 1] range exactly; this was because I feared that some parameters had quite large values (e.g. tens or even 100) when normalised by the standard deviation, and I feared a neural network would not reach such outliers well. (See section 5.5 below for examples.)

Only when the entire dataset was processed were overall means and ranges available. And to normalise each quantity by the standard deviation of that parameter, the sum variance (squared residual difference from mean) was required for each column. Likewise to scale by the range, the most extreme values had to be found in the first pass. Therefore a second pass through the entire dataset was required to compute the standard deviations used for scaling, once the means were available.

The same scaling was used for any one quantity (e.g. temperature) regardless of atmospheric altitude level, so that a common function could be applied to the data at any level (see 5 below). However, that meant that after scaling the magnitude of some variables was very small at some atmospheric levels compared with others, e.g. absolute pressure (defining the levels) was ~0.1 hPa at the highest altitude but ~1000 hPa at ground level, spanning four orders of magnitude.

After normalising, the data was again written to batch disk files, each small enough to consume in one go during training.

By experiment, I found:

- Scaling by the standard deviation worked better than using the absolute total range
- Omitting the subtraction of the mean in the output data gave better results than offsetting according to the mean. This was not expected, though the output data was dominated by trend values that were nominally symmetric about zero anyway.

The scalings obtained from the training data were reused unchanged when processing the test dataset.

4 CatBoost modelling

CatBoost is an open-source library for gradient-boosted decision tree fitting. This made a good complement to the neural network approach. The CatBoostRegressor variant was used in an attempt to fit the output values quantitatively, training for all columns simultaneously.

There were some limitations in using CatBoost:

- It could handle 2D data (i.e. columns of data), but not multidimensional arrays or tensors. The data that was previously vectorised by atmospheric level (see 3.3 above) therefore had to be "exploded" back into discrete columns, treated as independent inputs.
- Features that were scalar in any case were input only once, to avoid duplication.
- Generous values of the options iterations, depth and border_count were optimal for achieving best fit, but these resulted in a model exceeding 2GB. Unfortunately the save/load cycle in CatBoost did not work for such large models, so it was necessary to use the model whilst still in RAM to predict output features or a submission output, and impossible to retrieve the model later, despite lengthy training time.

- The same altitude-independent normalisation (see 3.6 above) was used as for the CNN approach, but as the CatBoost model treated the values at different atmospheric levels as unrelated variables, its performance probably would have improved if each column in the original input table was normalised individually, to avoid very small input values. There was not enough time to try this before the end of the competition.

See `train_catboost_model()` for implementation.

5 PyTorch convolutional neural network (CNN) modelling

Intuitively, the behaviour of each parcel of atmosphere will depend upon its own state, its inputs and outputs, and its interactions with its immediate neighbours. In this competition we do not have state information from horizontally neighbouring parcels, but we do have it for vertical neighbours in the same vertical column.

A two-dimensional image can be processed by applying the same function over a small patch of pixels, which is then translated across the image to give a new layer of processed output information; this is a 2-D convolution, used extensively in image processing. Likewise here we can run the same function at different layers of atmosphere in the column, taking as input the state information for that layer and its close vertical neighbours, and outputting derived values for that layer. This 1-D convolution (Conv1D) approach is what I have used here.

To support the Conv1D method, the discrete values provided for different atmospheric layers had to be organised into vectors (see 3.3 above), and all values normalised using the same scaling factors regardless of level (see 3.6 above) so that they could be meaningfully operated on by the same function at all altitude levels.

See section 6.4 below for a diagram representing the final model architecture.

5.1 "1x1" input layer

In a two-dimensional convolutional neural network applied to image processing, a 1x1 convolution may be applied initially to each pixel. This gives a transformation of the channel values (e.g. Red/Green/Blue assuming RGB format) to yield some new vector of values for input to the deeper layers, without mixing information from neighbouring pixels. The training process learns what linear combination of source values produces the best result in this "1x1" layer, as in all layers. For example, a network trained to recognise traffic light states may learn that a channel yielding a value which strongly reflects the presence of yellow (red + green) gives better prediction results overall.

By analogy with this, an optional first layer of the CNN constituted a "1x1" layer (now a misnomer but helpful as a name!) that in this case was only 1-dimensional, which mixed the input feature values at each individual atmospheric level into a new vector of transformed values for input into the remainder of the network. This was found to be beneficial.

5.2 Middle layers

A stack of Conv1D layers allowed the model to learn behaviour that depended on near-neighbour atmospheric levels without loss of layer resolution: each layer took values for 60 atmospheric levels, and output 60 levels, though with more variables per level than provided as input.

In general “larger” parameters gave better results, at the cost of increased computation.

The optimal number of layers was determined through automated hyperparameter search (see section 7.4 below).

5.3 Encoder/decoder

An encoder-decoder architecture is often used in tasks such as semantic segmentation image processing. It consists of:

1. An encoder which reduces resolution but generally increases information “depth”, allowing the model to learn broad characteristics across the whole image;
2. A decoder which expands resolution back to the original size, e.g. to predict a segementation mask;
3. Optionally, “skip connections” which link encoder and decoder layers of the same resolution, to preserve detail in the resulting output.

In this project, the idea of using this architecture was to allow the model to learn patterns from the whole atmospheric column (instead of just near-neighbour atmospheric levels).

The final implementation used PyTorch nn.Conv1d layers with non-unity “stride” to compress to lower resolution (60 to 20 to 5 altitude levels), and then corresponding nn.ConvTranspose1d layers to expand back again (from 5 to 20 to 60). Skip connections were tried but did not improve performance; the parallel “middle” Conv1D layers will have achieved the same effect of maintaining detail anyway.

5.4 Output layers

The convolution layers of the model produced more features (values) per atmospheric altitude level than were required at output, such that the model had more freedom to optimise its output, so a reduction down to exactly the number of required outputs for scoring was required.

Also, all values were vectorised across altitude levels for the bulk of the CNN model as the data percolated through convolution layers. But for output, we required 6 vectorised values (60 altitude levels each) and 8 scalar values (one altitude-independent value per climate model grid square).

To achieve these transformations:

1. A final Conv1D layer ingested only a subset of the previous layer’s values per altitude level, and transformed them into exactly the required number of output variables for each level.
2. An AvgPool1D layer averaged each remaining feature value over all atmospheric levels for the remaining subset, followed by a Linear layer which had arbitrary freedom to rearrange the surplus of values for each output into a linear combination giving exactly the required number of output scalars.

3. The vector values were then flattened (to give one output column per atmospheric altitude level) and concatenated with the scalar values, to give the final output shape required.

5.5 Polynomial experiment

Neural networks best with values normalised (by convention) to an approximate [-1, 1] range. Indeed the model here included normalisation layers to constrain the propagating values thus until the output layer, where we necessarily wish the outputs to span the range of the target predictions which may (and in this case do) exceed that range, despite their own normalisation, due to their statistical distribution: some parameters have values many standard deviations from their mean, both at input and output (as noted in section 3.6 above for input scaling).

For example, the input `state_v` (meridional wind speed) has a mean of ~0 m/s and a standard deviation of ~ 7 m/s in the training data, but with outliers of 200 m/s (perhaps implausible ClimSim model artefacts) which are thus ~30 standard deviations from the mean.

On the output size, the y-values here were multiplied by the “old” submission weights, effectively normalising them anyway by their standard deviations. Yet ptend_q0001_* values (moistening tendency) had limit values of around 10 to 20 standard deviations from the mean even at lower altitudes, and ~1000 at the highest altitude levels which were not zeroed out.

For scaling summary data see: `results/2024_07_20_e1bc1e1_scaling_range_stats`

So in an attempt to better reach outlier values, I experimented with a final polynomial output layer with individual learnable coefficients for each ("devectorised") output channel. The zeroth and first coefficients were omitted, because the previous layer already had unconstrained linear scaling of the output, so it only remained to add terms for higher powers of the values. The multitrain functionality was used to iterate over different polynomial orders from 2nd to 4th (i.e. $y_{out} = y_{in} + c_2 y_{in}^2 + c_3 y_{in}^3 + c_4 y_{in}^4$). But then as a precaution against bad reasoning I also tried including zeroth and first order terms, but without improvement.

See section 7.5 below for results. The polynomial layer did not improve performance, so it was left disabled in the final submission.

5.6 Automated multi-parameter experiments

Part of achieving the best results out of a machine learning model is to experiment with its structure (e.g. number or depth of layers) and hyperparameters (e.g. learning rate, number of iterations).

Also, where the number of inputs is constrained due to memory or speed limitations, as was the case with CatBoost in particular, it is useful (but computationally expensive) to discover which input features can be omitted without affecting the results.

For both of these reasons, a “multitrain” mode was implemented that worked in one of two ways:

1. For hyperparameter and structure experiments, a dictionary multitrain_params could be provided listing which named model parameters should be automatically adjusted, and what values should be tried. Where multiple model parameters were specified, a full permutation of every specified value of every varied parameter was built and executed.
2. For feature significance experiments, one feature at a time could be “knocked out” (removed from the input set).

In both cases, the program automatically ran the whole training/validation loop with each generated variant, saving loss metrics and the trained model in each case. These could then be assessed to choose which hyperparameter values to use going forward, and which features could be safely omitted from the input set.

In practice feature knockout was so computationally expensive that it was used only quite early in the development.

6 Final prediction process

6.1 R^2 sensitivity and cloud trick

This competition was judged by the average R^2 (coefficient of determination) score [Ref 5] across all output columns:

$$R^2 = 1 - \sum_i \frac{\left(y_i - y_{target}\right)^2}{\left(y_i - \bar{y}\right)^2}$$

However, some columns (e.g. changes in cloud mass ratio at high altitude) had tiny values with very small variances in the training data, as all values were near zero, giving a very small denominator. A model-predicted output value could easily vary from the target by many standard deviations, resulting in a very large negative R^2 score for that column, and for the submission overall. Avoiding bad scores due to these very problematic columns was a topic of great debate in the competition forum.

A workaround for the most difficult columns was found and publicly discussed [Ref 6]. Essentially, it depended on the cloud mass ratio value at altitude usually being zero in the source data. As a time series, if it was nonzero at one iteration, it was very likely zero at the next iteration, where each iteration was over 1200 s in the data. Therefore a strong prediction of the rate of change was simply to use the current value divided by -1200, i.e. predicting a rate of change that would result in zero "this time".

This trick was used here for `ptend_q0002`[12..28] inclusive (liquid cloud mixing ratio tendency).

See `replace_cloud_tendency_trick()` called from `postprocess_predictions()` in `leap_feat_eng.py`.

Separately, a technique suggested by Hu et al [Ref 3] was tried: instead of predicting water and ice cloud independently, the two were summed, and then attributed all to water above a threshold temperature, all to ice below a lower threshold, and to a weighted proportion between those thresholds. In practice, that was found (commit f9266af6) to worsen the results, so it was not pursued.

See `hu_fit_cloud_types_to_temperature()` for that implementation.

6.2 Combining CatBoost and CNN as weighted sum (parallel combination)

One approach to combining the best of the CatBoost and CNN methods was to simply form a weighted sum of the two output files.

As the model validation process gave an R^2 score per output column, it was possible to weight that sum column by column according to which model had achieved a better score.

6.3 Using CatBoost predictions as CNN input features (series combination)

An alternative approach to combining CatBoost and CNN models was to feed the inferior CatBoost predicted outputs as new input "features" for the CNN model to learn from, effectively using the two techniques in series.

This required the CatBoost predictions for all 10M+ training rows of `train.csv`, and all 625K rows of `test.csv`. Yet due to slow execution, the CatBoost model was only trained on a 1M row subset of the training data. It was therefore run in prediction mode (after training) to cover the whole training and test datasets, such that predictions were available both at training time for the CNN model, and in the test/submission process. These predictions were saved to disk as batch files to use in subsequent training and prediction by the CNN.

The predictions were scaled and normalised identically to the "real" y-output target values as described in section 3.6 above to make them suitable for ingestion into the CNN model, using the scale factors determined from the actual output training data.

This general flow is shown in the process diagram (Figure 2 in section 6.6 below).

6.4 Final model architecture

The model structure used for my final submission, combining CatBoost predictions with Conv1D CNN processing, is shown in Figure 1 below.

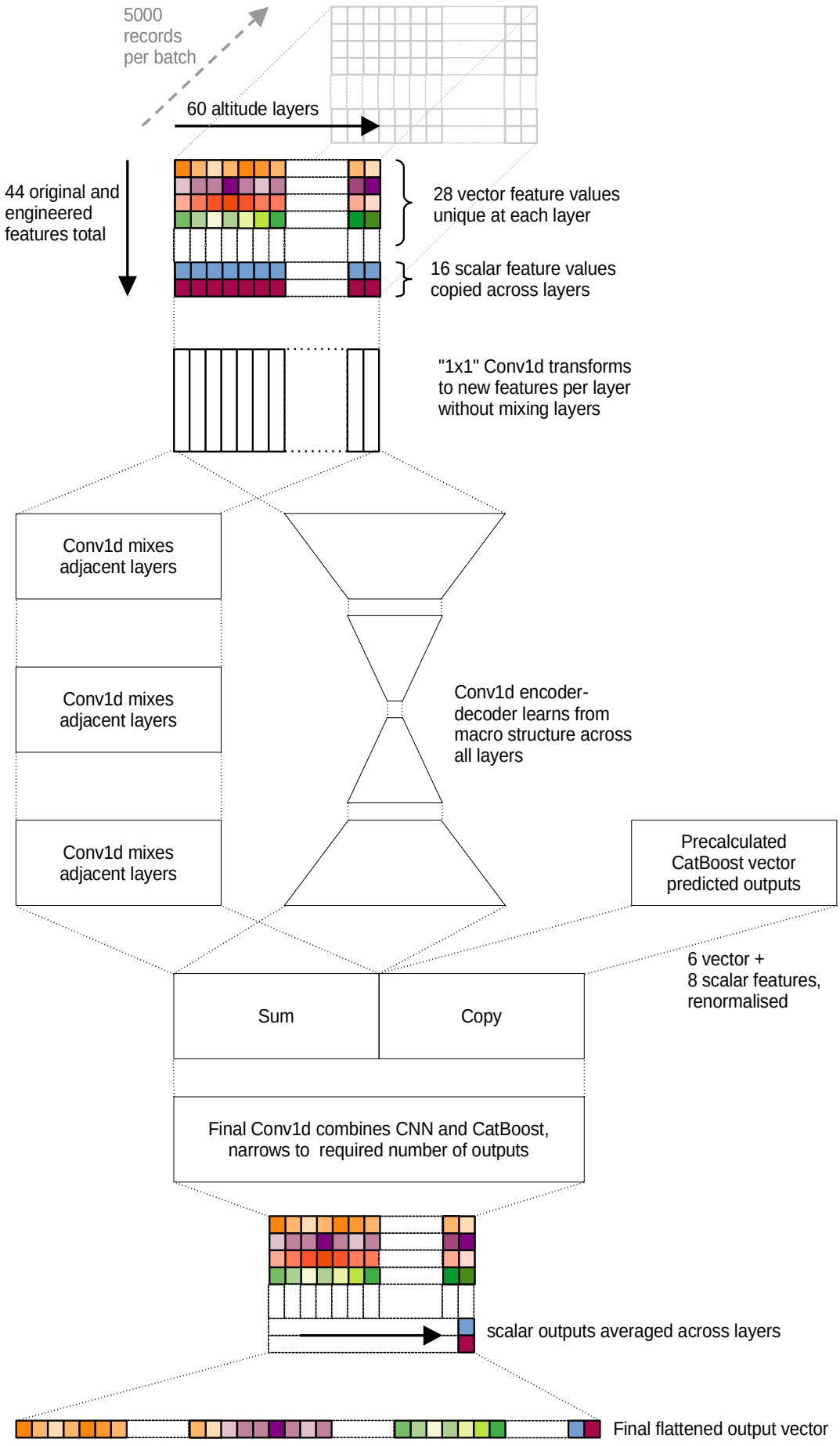

Figure 1: Final model architecture

6.5 Unscaling and submission

The test data provided, `test.csv`, was at 625,000 rows much smaller than the provided training data, and necessarily contained only inputs (x-values).

These values were subject to normalisation using the scale factors determined for the training data (see section 3.6 above) before being fed to the model in prediction (inference) mode. (While new scale factors for normalisation could have been obtained only from the test set, these would make the values somewhat inconsistent with those used for training.)

The y-values output by the model were still a) normalised and b) prescaled by the "old" submission weights provided which put them in a manageable numerical range in the first place. These operations were reversed in the output processing to give a final dataframe containing values in the correct physical units, and as float64 to avoid loss of precision in tiny values.

The dataframe was written to disk as a file, `submission.csv`, and then manually submitted to Kaggle using the command-line API provided. My submission comments always included the git hash of the code used to produce that result, to make it possible to return exactly to the code which produced it.

See `test_submission()` in `leap_feat_eng.py`.

6.6 Final process flow

The final data processing overview is shown in Figure 2.

In practice two runs were required:

1. the first to perform CatBoost modelling, which saved output predictions for both training and test data to disk;
2. the second to perform CNN modelling and final predictions, using competition-provided data, engineered features, and CatBoost predictions as inputs.

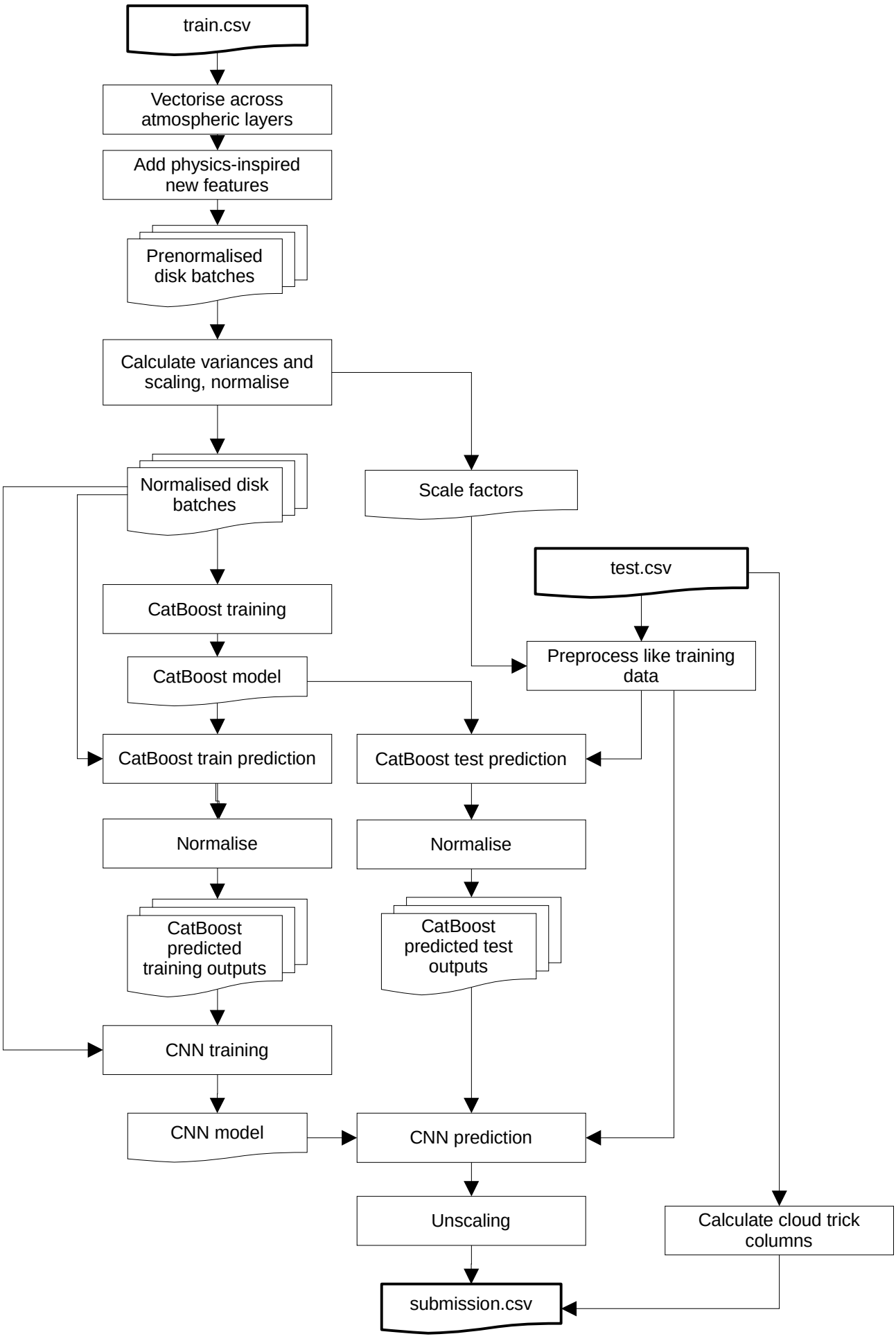

Figure 2: Process used for final competition submission

7 Results

This was a fast-moving competition against a pressing deadline, not a formal research project. However, to aid subsequent understanding (not least for this report), raw results (loss metrics by model epoch, etc) were saved in a folder name indicating the date of completion (sometimes a day or two after starting) and the Git commit hash of the version of `leap_feat_eng.py` used to produce them. Thus it is possible to refer back in version history to obtain the details of the model used and program code for any particular result. The repository path pattern for results is:

`results/[date in YYYY_MM_DD format]_[git commit hash]_[brief description]`

e.g. `results/2024_07_05_ae965128_conv_layers_expts`

7.1 Feature knockout experiments

Due to the high computational cost of repeatedly running training with just one input removed, feature knockout experiments were used only briefly, early in the competition development. With more time, it could well be that equally good or better model results could be obtained with the removal of many input features, which would be important for a production-deployed model.

However, the results of one run are shown in Figure 3 below, for the current CNN model at that time trained on 3M rows but for only 6 epochs. It clearly shows the great dominance of cloud mixing ratios and relative humidity above all else, i.e. atmospheric wetness.

See also `results/2024_05_30_52a1718_larger_feature_knockout`

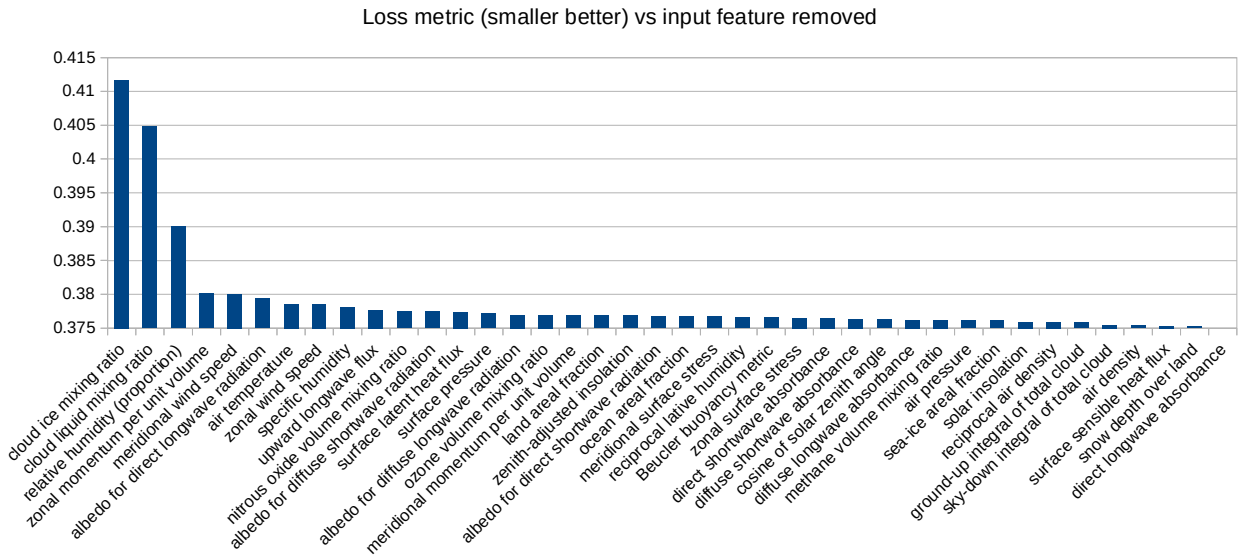

Figure 3: Effect of removing one input feature at a time on model performance

7.2 CatBoost hyperparameters and prediction

The main raw data for several experiments with CatBoost hyperparameters are here in the repository:

`results/2024_06_14_63d2cf4729_catboost_hyperparam_search_1`

`results/2024_06_14_2fb1a9f_hyperparams_2`

`results/2024_06_15_48659d0_catboost_hyperparams_3`

`results/2024_06_15_6096698_catboost_hyperparams_4_its_learning_rate`

Overall it was found that maximising the number of learning iterations and using a reasonably high learning rate produced the best validation scores. Unfortunately attempting a larger number of iterations caused the CatBoost regressor to crash, while other parameters were constrained somewhat to remain within the available 16 GB GPU RAM. The final parameters used were:

Table 1: Optimal CatBoost training parameters

CatBoost parameter	Value
iterations	499
learning_rate	0.25
depth	8
border_count	32
l2_leaf_reg	5

The CatBoost model used to provide output features for the CNN in the final submission itself achieved a score of R^2=0.44262 on the public leaderboard and its output is recorded here:

`results/2024_07_05_a86308a9_catboost_1m_499_its`

7.3 CNN with and without 1x1 input layer or encoder/decoder

The optional 1x1 input layer and encoder/decoder structure are pictured in section 6.4 above, Figure 1. During the competition, Kaggle submission scores (training over the full 10M+ row dataset) were marginally better with these included in the model, so they were retained.

However, a subsequent smaller test run over the first 1M rows (~10%) of the dataset, run specifically to assess the affect of those model structures to support this report here, shows (Figure 4) that the benefits are marginal. The loss and R^2 curves are in fact all broadly similar, within noise.

At about 20 - 25 model epochs, the larger model with both 1x1 input and encoder/decoder is indeed performing slightly better than those with only one of those options, which in turn beat the model with neither. But after that point the larger model in this run apparently became overtrained, with declining performance, while the others “caught up”; after 30 epochs there is nothing significant to choose between models.

In most competition runs however, the training process either selected the best model (as judged by validation score) automatically to guard against overtraining, or available time prevented execution to the point of overtraining anyway.

For raw data see: `results/2024_07_21_014fc08f_model_structure_expts`

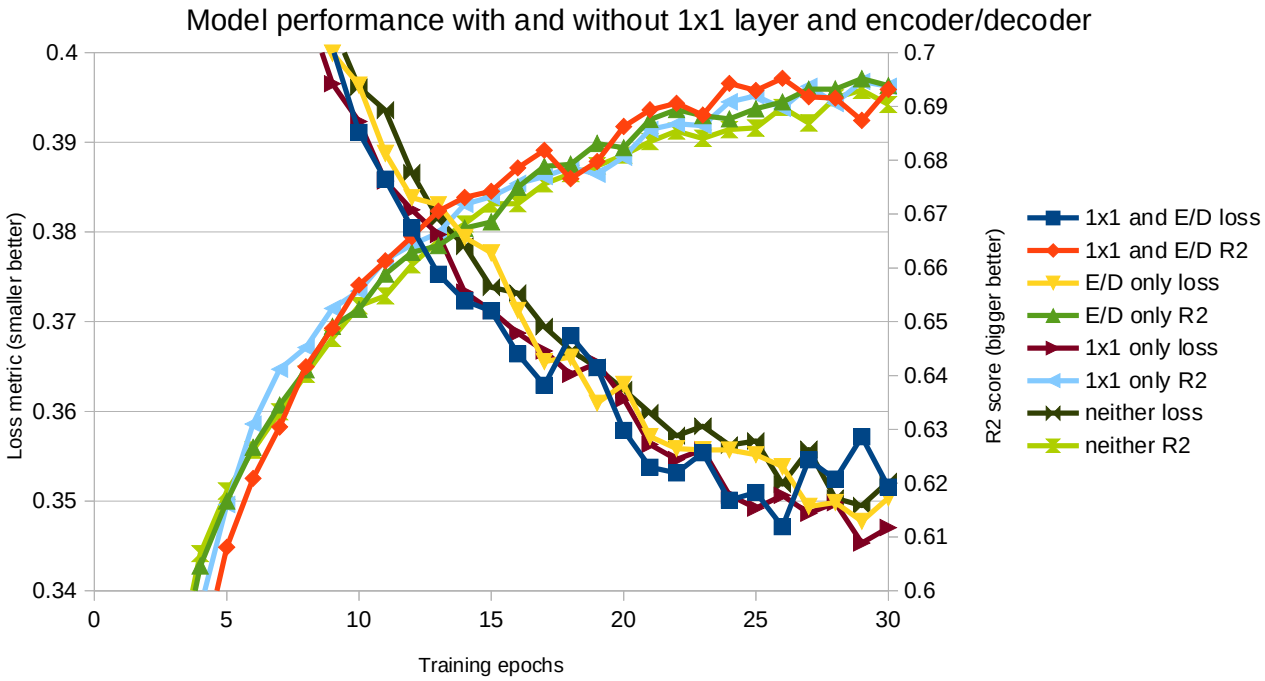

Figure 4: CNN performance in small test run with and without 1x1 layer, and encoder/decoder

7.4 CNN Conv1D number of layers

Using automated hyperparameter search, it was found that adding one “middle” convolutional layer provided a great improvement, and adding further layers gave progressively less improvement, until there was no significant difference between 3 and 4 layers. To save computational expense, 3 layers were therefore used in the final submission.

The raw results for this are in `results/2024_07_05_ae965128_conv_layers_expts`. For this comparison the same subset of 1,000,000 training rows were used for each variation, over a maximum of 30 epochs.

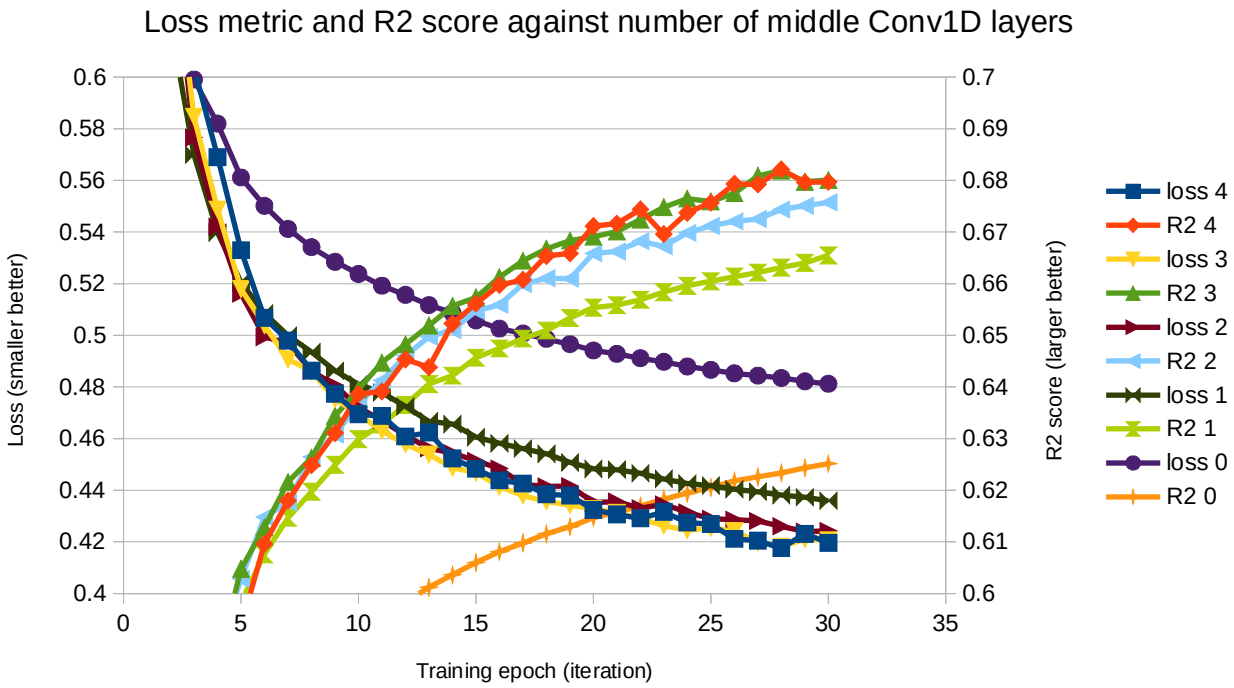

Figure 5: Determination of optimal number of "middle" Conv1D layers

7.5 CNN with and without polynomial output layers

By experiment, adding the polynomial layer slowed model training without improving the final score achieved, and so it was not used in the final submission. This is shown in Figure 6 below. Raw results can be found in:

`results/2024_07_02_1ca4a85_poly_output_more_degrees_worse`

`results/2024_07_02_6fa93abe_poly_output_with_0_1_coeffs_still_worse`

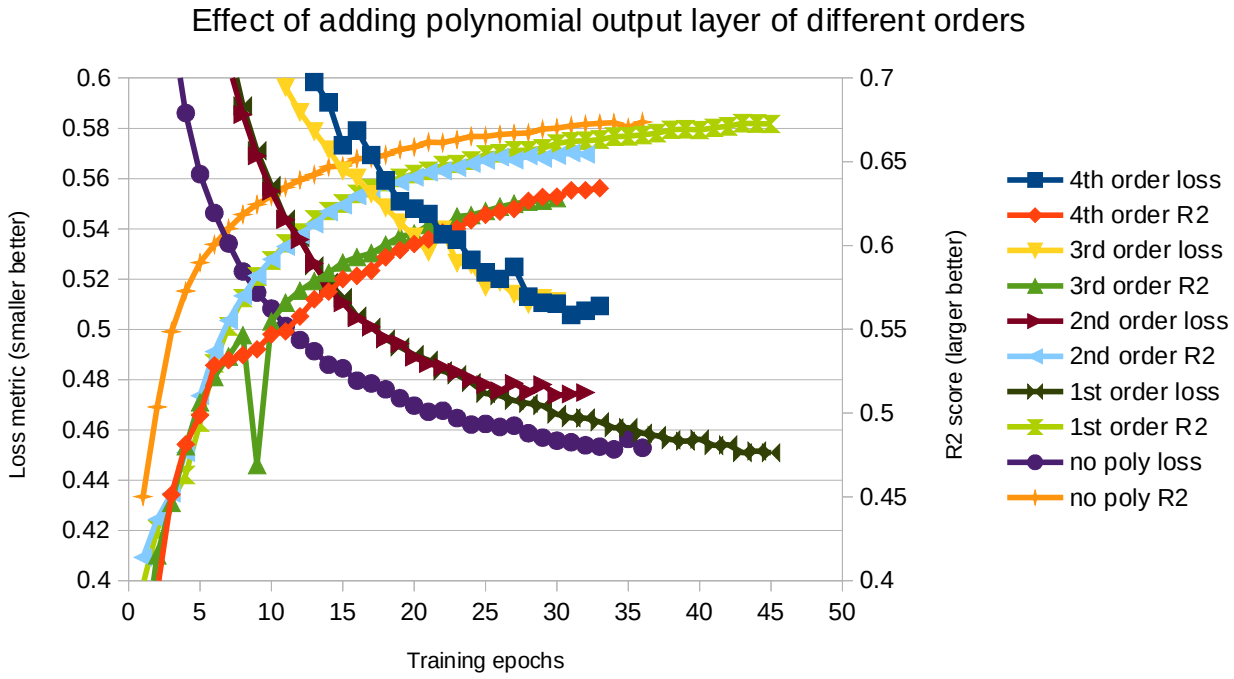

Figure 6: Model performance with different orders of polynomial output layer in use

7.6 Weighted sum of CNN and CatBoost predictions

Intuitively, I expected mixing the predictions of two independent methods to produce a better result than either technique alone. However, the "parallel" approach of producing a weighted sum of the final output (see section 6.2 above) did not in fact improve the Kaggle score. The CNN, being intrinsically better, was always weakened by mixing with CatBoost outputs, as tabulated here:

Table 2: Scores resulting from weighted combinations of CatBoost and CNN final predictions

Commit	Kaggle R^2 score	Description
fa7e995a	0.69652	CNN trained for only 7 epochs on full 10M+ rows
867b3ea4	0.44679	CatBoost trained on 1M rows
2d3f0e47	0.66874	Weighted sum of predictions, linearly weighted by R^2 scores
1dbed19f	0.67852	Sum of predictions weighted by square of R^2 scores
6b46658	0.68357	Sum of predictions weighted by fourth power of R^2 scores
b713d09	0.68306	Each column chosen from CNN or CatBoost according to which had larger R^2 score

7.7 CNN with and without CatBoost prediction features

A full-dataset run with CatBoost features ingested near the input of the CNN (commit 917f41b3) scored R^2=0.71349 on the public leaderboard, compared with R^2=0.71705 without CatBoost features (commit 8a50b544). Without performing more systematic experiments (which would have been too slow), this suggested early CatBoost feature ingestion did not improve performance.

A more direct comparison was made between the use of late CatBoost prediction injection into the model (as shown in Figure 1, section 6.4 above) showed a clear improvement however, as pictured here in Figure 7 below.

Raw data with late CatBoost features used:
`results/2024_07_07_71741533_full_cnn_3_midlayers_late_catboost`

Comparison with no late CatBoost features including this graph:
`results/2024_07_08_2c2c3bb18_full_cnn_no_catboost`

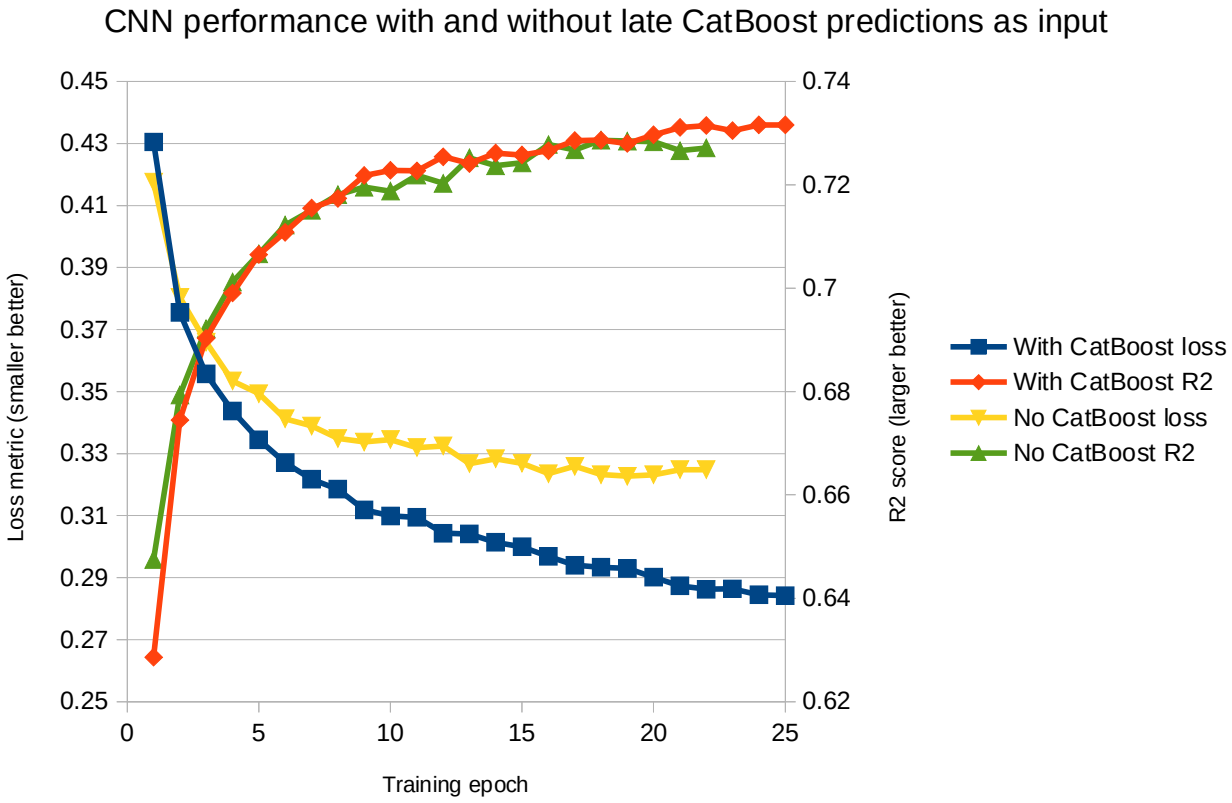

Figure 7: CNN performance with and without late CatBoost predictions as input

7.8 Execution metrics

The largest CNN model (with CatBoost features included both early and late in the stack of layers, but omitting the optional polynomial output) had 33.8 M trainable parameters. This took ~ 2.6 hr to iterate one epoch on the available hardware in batches of 2500 dataset rows, to avoid exceeding the available 16 GB of GPU RAM.

The CNN model yielding the best submitted score (with CatBoost features included only late) had 19.8 M trainable parameters and could be run in batches of 5000 dataset rows.

Training the CatBoost model with the maximum 499 iterations took 15-20 min per batch of 20000 dataset rows. Only 1 M rows were used due to the slowness of this training.

7.9 Final optimal submission results

My best score was $R^2 = 0.72998$ (on Kaggle's private leaderboard, withheld until the end of the competition) and 0.73373 (public leaderboard, shown at the time of submission). This was generated by commit 76b5c4fa of `leap_feat_eng.py`. By comparison, the winning entry scored 0.79.

To extract the most value out of the training dataset, training was interrupted before completion and the code changed to use 99.9% for training and only a (somewhat meaningless) 0.1% for local validation.

That optimal run featured the following, in summary:

- CatBoost predictions ingested 'late' (only)
- 1x1 input layer enabled
- encoder-decoder enabled
- 3 Conv1D mid-layers used
- Conv1D widths of 7 altitude levels used
- Conv1D depths of 15 x number of input features used
- no polynomial output layer

This is reflected in the architecture diagram (Figure 1 in section 6.4 above).

8 Conclusions

While this entry was some way from the top of the leaderboard, it was nevertheless a submission which achieved a creditable score despite some distinct challenges posed by the competition, which necessarily led to some innovative methods, namely:

- Using low-level file operations to gain fast random access to huge CSV files
- Combining two completely different techniques (CatBoost gradient-boosted decision tree and PyTorch neural network) in series
- Automated techniques to optimise structure options and hyperparameters

9 Acknowledgments

This competition stimulated lively discussion on the accompanying forum, with many users generously sharing starter notebooks and ideas. I would like to thank the following users in particular:

- Jano123 for identifying the "cloud trick" [Ref 6]
- Andoni Irazusta for a Pytorch neural network starter: https://www.kaggle.com/code/airazusta014/pytorch-nn/notebook
- Lonnie for showing how CatBoost could be used in this competition: https://www.kaggle.com/code/lonnieqin/leap-catboost-baseline

- Yirun Zhang for providing an example of using CatBoost in multiregression mode: https://www.kaggle.com/code/gogo827jz/multiregression-catboost-1-model-for-206-targets

10 References

[Ref 1] Sungduck Yu et al., "ClimSim: A large multi-scale dataset for hybrid physics-ML climate emulation", https://arxiv.org/pdf/2306.08754v5 (v6 already available)

[Ref 2] Tom Beucler et al., "Climate-Invariant Machine Learning", https://arxiv.org/pdf/2112.08440

[Ref 3] Zeyuan Hu et al., "Stable Machine-Learning Parameterization of Subgrid Processes with Real Geography and Full-physics Emulation", https://arxiv.org/pdf/2407.00124

[Ref 4] "Leak" found allowing extraction of geographically correlated data, https://www.kaggle.com/competitions/leap-atmospheric-physics-ai-climsim/discussion/519184

[Ref 5] R^2 (coefficient of determination) explanation on Wikipedia: https://en.wikipedia.org/wiki/Coefficient_of_determination

[Ref 6] Discussion of “cloud trick” to predict difficult output columns: https://www.kaggle.com/competitions/leap-atmospheric-physics-ai-climsim/discussion/502484