DEEP LEARNING FOR BIAS CORRECTION OF ENSEMBLE WEATHER FORECAST POST-PROCESSING

By

Viv Atureta

ABSTRACT

This study was aimed at establishing the use of deep learning techniques in ensemble weather post-processing task of bias correction and uncertainty estimation by using very high dimensional temperature forecast data for Jan 2019 provided by the UK Met office. The ingenuity of this study is in extending a single dataset to be used to obtain both forecast data and ground *truth* where previous attempts have required reforecasts and reanalysis data for post-processing tasks. The architecture which was explored used a computer vision technique adapted for an image-to-image regression using a combination of convolutions which applied downsampling and upsampling in the model architecture. The report details the steps which have been taken to process the high-dimensional data set to produce valid input into a deep learning network. A very thorough review of previous studies using machine learning / deep learning techniques for ensemble weather post processing tasks has also been provided.

DECLARATION

I hereby certify that this dissertation constitutes my own product, that where the language of others is set forth, quotation marks so indicate, and that appropriate credit is given where I have used the language, ideas, expressions, or writings of another.

I declare that the dissertation describes original work that has not previously been presented for the award of any other degree of any institution.

Signed,

Viv Atureta

TABLE OF CONTENTS

Table of Figures

1. INTRODUCTION

Modelling Weather requires capturing atmospheric dynamics (motion) and physical processes including clouds, radiation, convection, and turbulent diffusion. However, forecasts are somewhat limited by inability of the model to accurately capture these mechanisms present in the atmosphere as well as the errors in the observations which are required to initialize the models. The approximations made in trying to represent the physical processes, parametrization of sub-grid processes and using best-guess estimate of data assimilation process as the initial conditions, result in uncertainty in the model. A further impact on the forecast is tied to the growth rate of the initial errors which over time is neither slow nor uniform but instead diverge with varying trajectories despite starting from very close initial conditions a.k.a. the “Butterfly Effect” – tiny and even seemingly insignificant changes could result in very large consequences. This is drawn from a publication by Lorenz, 1963 “Predictability: Does the flap of a Butterfly’s wings in Brazil set off a Tornado in Texas?” - and is the hallmark of chaos. This inherent sensitive dependence on initial conditions that causes even the smallest changes to result in very widely differing outcomes, resulted in the introduction of probabilistic methods for forecasting which are known as ensemble weather forecasts. Deterministic forecasts make assumptions about the knowledge of the initial state and consider a “perfect” model; both of which fail to recognize the points mentioned above. Hence, the use of probabilistic forecasts provided by ensemble systems. Using this approach, instead of a single forecast, multiple simulations are produced; each initialized with slightly different values resulting in a collection of predictions.

The above is the approach implemented by the UK Met office ensemble forecasting system MOGREPS-UK model. The parameter of interest, Temperature is forecast at 1.5km grid point resolution. To account for errors in both the initial conditions and forecast model, MOGREPS-UK model runs using three perturbed ensemble members with a new cycle starting every hour from each member and forecasts up to 126 hours into the future – T + 126 (approximately five days ahead). A further innovative approach is taken to generate an 18-member ensemble by blending across 6 cycles with the assumption that over the 6-hour window, errors are roughly similar. (*The Met Office ensemble system*, no date).

1.1 Problem Statement

The focus of the study is on understanding the bias associated with ensemble members and correcting same. The ensemble members exhibit bias and are overconfident (under dispersed) in probabilistic forecasts issued. This also helps us in understanding uncertainty associated with ensemble weather forecast initial condition uncertainty and potentially quantifying it. Due to the perturbations made in the initial conditions, and parametrization of physical processes within the model, postprocessing methods are required to correct the systematic errors which are exhibited by raw ensemble forecasts. Furthermore, errors grow as a function of the lead time i.e. a forecast which is one day in advance is more accurate than a forecast seven days in advance and ensemble systems tend to be *under-dispersive* which means that the observations have the tendency to fall outside the forecast range of the ensemble system. *To increase confidence in our predictions, we must have a sense of the uncertainty associated with the ensemble forecast*.

In figure 1 Representation of Under-dispersion (over-confident system) below, we are showing the several different trajectories followed by the separate forecasts with perturbed initial conditions. The UK Met Office time-lagged 18-member ensemble is shown at initialization time each with slightly perturbed initial conditions and high-resolution trajectory. The ensembles represent temperature forecast starting with an initial distribution and running several trajectories to produce the output. The circle is capturing all the possible ranges which the model has predicted while the green spot which is outside the circle represents the case where the system has not covered the entire phase space of possible states for the real atmosphere. This causes the observed value to tend to fall outside the forecast (Slingo and Palmer, 2011). As is clear, the actual distribution produced from running the ensemble forecast is not the final point mainly because of this characteristic of the ensemble system to display under-dispersion. This is where post processing comes in for example bias correction, uncertainty estimation, probability distribution function (PDF) calibration and related techniques which aim to produce a final distribution which is now more useful. This makes Ensemble calibration and Bias Correction some of the more important aspects of ensemble weather forecast post-processing.

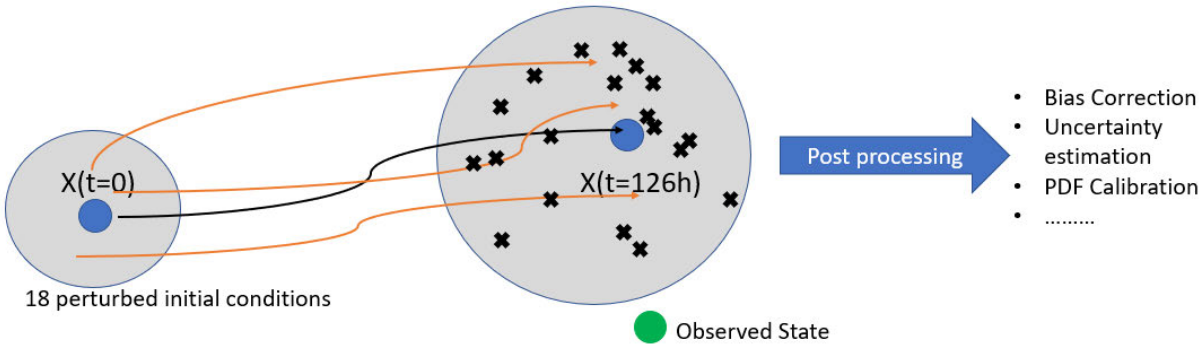


Figure 1 - Representation of under-dispersion (over-confident forecasting system) – observation lies outside ensemble

If the ensemble forecast predicts that the probability of seeing T°C temperature at a given time would be 90%, we expect that 90% of the time, our observation of the temperature at the time would match the forecast temperature. This makes sense in practice for industries like transportation, sports, mining, and tourism among others, where decisions are made based on for example the probability of temperatures going below a certain threshold. While the ensemble forecast produces a probabilistic forecast, assumptions are still made about the initial conditions and model errors, this requires that post-processing techniques are employed as a means of obtaining improved probabilistic forecasts that can address forecast bias and under-dispersion. The common methods for postprocessing ensembles which are prevalent in literature include among others: bias correction, regression methods like non-homogeneous regression, ensemble dressing, Bayesian model averaging (BMA) and non-parametric methods like quantile regression and rank histogram recalibration. (Wilks, 2018) and (Roman and Annette, 2018). The goal is to build models based on the raw ensemble such that forecast bias and under-dispersion may be corrected and is “statistically indistinguishable from the corresponding sample observations” (Buizza, 2018).

1.1.1 Approach and methodology

This study takes the approach of investigating some more recent machine learning techniques in ensemble weather forecast postprocessing tasks and related studies. Deep learning application in weather prediction or postprocessing; is not at advanced stages and presents an active area of on-going research as detailed in (Xiaoli *et al.*, 2021). In this report, we present a unique angle by experimenting with a high dimensional image data set plus working with a much smaller number of ensemble

members than is typical of such studies. It is worthy of note that the optimal use of a small number of ensemble members is complex and being actively explored in research (Buizza *et al.*, 2005). Implementation of a bias correction model is done using Keras API framework which is built on top of TensorFlow 2; an open-source platform for machine learning. Keras API is also not extensively applied in existing studies on ensemble weather postprocessing techniques. Keras has built-in support for Convolutional neural networks (CNNs) which have lower pre-processing requirements due to their ability to learn filters while reducing parameters and reusing weights.

The training data set is composed of historic data (data for the full month of January 2019) and justification for the validity of this fixed data set is supported by approach in (Rasp and Lerch, 2018) where there was no significant difference found in predictive ability between fixed or most recent data being used. In addition, in seeking to be flexible on the choice of the distribution (i.e. avoid the preselection of for example a typical Gaussian distribution for our model estimation); the nonparametric approaches considered are a variation of Neural Networks (Grönquist *et al.*, 2021), (Rasp and Lerch, 2018) and (Peng *et al.*, 2020). Final implementation approach differs from these studies in our use of the deep learning techniques provided Keras API framework. The ambition was a possible extension of the study through Gaussian processes (Wang, Zhang and Villarini, 2021) and (Cai *et al.*, 2019) with our potential implementation differing in the use of open source GPyTorch which is built on top of the PyTorch deep learning framework. This proposed extension of the study using GpyTorch which though not implemented; was considered a potentially beneficial approach due to speed, ease and efficiency of building Gaussian process models and the provision of better support for prototyping through its modular design.

For this study, it is also important to note that we are considering forecast which is categorized in the medium-range which means predictions made are valid for up to two weeks. (Vannitsem, Wilkis and Messner, 2018)

The following are caveats regarding model reliability:

i. Analysis (initial conditions) which have been adopted as the truth are only a best-case estimate of the true observations.

ii. Trade-offs related to spatial resolution, time frame and region covered. For example, having a high-resolution data for the UK but only for the month of January. Model performance on summer months' data may be unreliable.

iii. The process of converting the original forecast data to images causes some loss in precision and may also influence model results. (Details in Ch. 3)

iv. Training has been carried out on archived model data however, in practice, adjustments are made to operational Numerical Weather Prediction (NWP) models frequently, i.e. system is not static and does not provide a continuous set of homogeneous forecasts.

Challenges related to the project include working with and representing very high-dimensional data; computationally intensive processes and time constraints, all of which significantly impact the final output.

1.2 Outcome

The MET Office implements probabilistic weather forecasting which depends on the reliability of ensemble forecasts. To put this into context, consider the output from the ensemble forecasts which has not filled the phase space of possible states for the real atmosphere (i.e. not sufficiently dispersed initial conditions); the forecast tends to be under spread because the entire distribution of all possible conditions has not been captured; resulting in an over-confident ensemble forecast. Due to the complexity of the data set and computational cost, this project considers bias correction as a first order calibration. Still, this process is non-trivial and serves to present our choice of technique as a viable option specifically as relates to date range, data type and ensemble size. The study presents initial potential for extension to more complex postprocessing tasks with data in image (png) format, fewer ensemble members and date range. The selection of temperature as predictor in parametric distributional regression models is common because conditional distribution of temperature can be approximated by a Gaussian distribution which is a natural choice of parametric family; however, nonparametric distributional regression approaches provide approaches that help circumvent the choice of a parametric family; and this is equally extensible to other weather variables like wind speed, precipitation where choice of distribution is less obvious.

1.3 Document Structure

The document is organized into five chapters summarized as follows:

The next chapter, Chapter 2 – *Background and review of literature* provides a review of the commonly used postprocessing techniques in practice and explored in literature as relates to Bias Correction and Uncertainty estimation for ensemble weather forecast post-processing. It also provides insight into the Machine Learning / Deep Learning formulations proposed in some more current research. In Chapter 3 – *Methodology, Design, and Implementation* we describe the ensemble data and preprocessing steps taken and present the bias correction model architecture and implementation. Chapter 4 – *Results and evaluation* is where main results are shown and all evaluation which was carried out, is discussed; followed by Chapter 5 – *Conclusions* which summarizes findings and explores some possible extension to the study for future research.

1.4 Nomenclature and Notations

Abbreviations		
	BMA	Bayesian Model Averaging
	CGP	Convolutional Gaussian Process
	ECMWF	European Centre for Medium-Range Weather Forecasts
	EMOS	Ensemble Model Output Statistics
	ENS	Ensemble
	ENS10	ECMWF 10-member re-forecast ensemble experiment
	ERA5	Fifth major ECMWF Re-analysis data (1979 to present)
	FCN	Fully Connected Neural Network
	GEFS	Global Ensemble Forecasting System
	LDAPS	Local Data Assimilation and Prediction System
	MOGREPS	Met Office Global and Regional Ensemble Prediction System
	MSC	Meteorological Service of Canada
	MTGP	Multi-Task Gaussian Process
	NCEP	National Centres for Environmental Prediction
	NMME	North American Multi-Model Ensemble
	NWP	Numerical Weather Prediction
	PEARP	Météo-France 35-member ensemble forecast
	QRF	Quantile Regression Forest
	SVR	Support Vector Regression
	TIGGE	THORPEX Interactive Grand Global Ensemble

CHAPTER 2. BACKGROUND AND REVIEW OF LITERATURE

Running several forecast model simulations even in the age of 21st century computing technology is computationally intensive and expensive – for this reason, there is significant focus on postprocessing techniques which use "ensemble forecasts to provide a complete picture of the risks and uncertainties in the weather forecast" (*The Met Office ensemble system*, no date). At the UK MET office there are multiple current projects to this end one of which is the IMPROVER (Integrated Model postprocessing and VERification) which is a new postprocessing and verification system. There are other prominent examples of active research and development efforts in Postprocessing of model output in operational weather forecasting which include the National Centers for Environmental Prediction (NCEP) Unified Post Processing System (UPP) (*dtcenter.org*, no date); the Meteorological Service of Canada (MSC) PROGNOS initiative to renew its postprocessing infrastructure (Antonopoulos *et al.*, 2020), and the MISTRAL project which uses ECMWF and COSMO ensemble for post-processing (León, 2021). These efforts are geared towards continually implementing successful postprocessing techniques which address issues with resolution and model formulation issues; potentially can reduce the need for increased ensemble members and consequently conserve computing power.

Prior research has sought to make better informed inference from ensemble forecasts such that the uncertainty is adequately captured as a probability density function "whose implied probabilities for different outcomes can be expected to have a desired level of reliability" (Parker, 2010). This has been approached using historic information from ensemble forecasts to correct for bias and under/over dispersion through probability distributions of the results. Consequently, while we cannot say that our observed variable will always fall within the range of values predicted by the ensemble system, could we instead be more certain that when a variable is predicted as some specific lead time, that we can expect the observed value of the variable will fall within the range of some bias-corrected results produced by the ensemble on about p% of occasions?

2.1 Postprocessing for Interpreting Ensemble Forecasts

As cited above, efforts at post-processing are already ongoing at operational forecasting centers, often using relatively simple methods, e.g. applying a bias

correction in light of recent errors, with probabilistic forecasts then produced according to the fraction of bias-corrected ensemble members that predict the outcome of interest. More complex post-processing approaches consider the spread, building a similar spread to that of the ensemble and the top of the stack are others which deliver full Probability Density Functions (PDFs) to produce what is called uncertainty calibration.

The objective of interpreting ensembles is to successfully construct a probability distribution which connects the initial conditions and truth, such probabilistic relationship does not a priori make any assumptions about the underlying distribution by specifying the mean and variance as parameters to be estimated. Instead, the aim is to estimate the distribution of the error based on the ensemble output.

Within a postprocessing context, the more common approaches and somewhat baseline techniques are non-homogenous Gaussian regression (also known as Ensemble Model Output Statistics EMOS) and Bayesian model averaging – BMA. These methods, their applications and formulations in literature have been discussed extensively in (Wilks, 2018). EMOS formulation is regression-based with predictive distributions specified as Gaussian $y_t \sim N[\mu_t, \sigma_t^2]$ for each forecasting occasion t. Bias correction is modeled with predictive mean equal to regression estimate and uncertainty is modelled as a linear function of ensemble spread (variance). Equation 1 specifies the three expressions which are representative of EMOS implementation in (Gneiting *et al.*, 2005) where y is the weather quantity of interest being predicted; a, b, c, d are regression coefficients; s_t^2 is the ensemble variance and $x_1, \ldots, x_m$ are ensemble members:

Equation 1

$$\mu_t = a + b_1 x_1 + b_2 x_2 + \cdots + b_m x_m \qquad \text{predictive distribution mean}$$

$$\sigma_t^2 = c + d s_t^2 \qquad \text{Var}_{(\epsilon)} \text{ - function of ensemble spread}$$

$$N(\mathrm{a} + \mathrm{b_1 x_1} + \cdots + \mathrm{b_m x_m}, c + d s_t^2) \qquad \text{Gaussian predictive distribution}$$

BMA was applied to temperature and sea level pressure postprocessing of uncertainty in (Raftery *et al.*, 2005). BMA is similar to EMOS only in the sense that it yields a continuous predictive distribution. However, it doesn't have one PDF approximated by a normal distribution, instead the BMA PDF consists of a mixture of probability

distributions. There is an emphasis on conditioning the best member or model such that the underlying BMA predictive model is a “weighted average of the conditional PDFs given each of the individual models weighted by their posterior model probabilities” (Raftery *et al.*, 2005); the construction of which is represented as:

Equation 2

$$p(y|f_1, \dots f_k) = \sum_{k=1}^{K} w_k g_k(y|f_k) \qquad \text{BMA Predictive model}$$

Interpreted as the conditional PDF of y conditional on f_k. As usual y is the weather quantity of interest to be forecast; bias-corrected forecast f_k; each w_k is the nonnegative weight (probabilities) associated with the component probability density; $\sum_{k=1}^{K} w_k = 1$.

Additional statistical techniques worth mentioning explored kernel dressed ensemble such as *Affine Kernel dressing* (AKD) (Bröcker and Smith, 2008) in which the probability density function given by $p(y; \boldsymbol{x}, \theta)$ is a representation of the interpreted ensemble. AKD was considered a plausible improvement on previous *Gaussian Distribution Fit interpretation* (GDF), a variant of which we have recently explained using EMOS implementation in (Gneiting *et al.*, 2005) using regression models to predict uncertainty around a weather quantity via a linear relationship with the standard deviation. While both approaches allow for the specification of parameter vector $\boldsymbol{\theta}$, the limitation of GDF is that the interpretation of the ensemble forecast is Gaussian while AKD provides a more versatile method by allowing the shape of the distribution to be determined by the kernel. The aforementioned could be considered as relatively *cheap* post processing methods that compensate for model bias and uncertainties. These notwithstanding there is constant demand for optimizing postprocessing for ensemble weather forecasts, such that can provide additional precision and cost savings. This is where more recent advancements in machine learning have a significant role to play.

2.2 Machine Learning Methods in Ensemble Forecast postprocessing

Speed, accuracy and ability to “learn” the data are desirable features in the application of machine learning in the weather forecasting. *Learning* means that no assumptions are made about the distribution of the variable; specifically, by adopting a nonparametric approach which is quite unlike EMOS and BMA that consider a

Gaussian predictive distribution as discussed in section 2.1. Given such flexibility with Machine Learning techniques, there have been continuous studies aimed at applying such methods to the NWP pipeline.

The subsequent discussion is on postprocessing tasks where there are increasing number of studies aimed at “allowing” the forecast distribution to be learned based on the ensemble forecast data. The model architecture is based on ability to capture nonlinear relationships in a data-driven manner and thus achieve bias correction and other post processing tasks using Machine Learning techniques. A comparison follows of some of the methods analyzed as part of the report and aims to highlight the progression through to the more advanced deep learning techniques which is the focus of this study. First, using 35-member ensemble from French operational Ensemble System (PEARP), (Taillardat *et al.*, 2016) implemented a Quantile Regression Forest (QRF) in a post-processing technique for forecast calibration. The emphasis of the approach is the notion of detecting nonlinear phenomena by using classification and regression (CARTs) and a generalization of random forests with QRF. In the models, the number of trees and leaves are set; for example for temperature variable, number of trees and leaves set is 300 and 10 respectively. The range for the CARTs was mostly 300-400 trees. The study results presented EMOS and QRF rank histograms and CRPS values which indicated an improved performance of QRF over EMOS.

In the (Cho *et al.*, 2020) study, multiple Machine Learning models were analyzed: Artificial Neural Networks (ANN) using a multi-layer perceptron (MLP) architecture, Random Forest (RF) and Support Vector Regression (SVR) to correct the LDAPS Numerical Weather Model (NWP) in South Korea on model outputs of minimum and maximum temperatures. The aim of this study was an output which combined all the machine learning models. In comparison with (Taillardat *et al.*, 2016), CARTs in this study were in a 500-1000 range and the RF model had the highest RMSE in comparison with ANN and SVR. A limitation in the study and which is consistent with (Taillardat *et al.*, 2016) is the RF’s overestimation of lower temperature values and underestimation of higher temperature values which is due to the nature of the RF’s construction where the CDF is strictly between the highest and lowest values in the training data. Thus, the RF is unable to extrapolate outside of this data. Overall, the proposed models showed improved bias of between -0.16 to -0.07 °C for maximum

temperature from original LDAPS model bias of -0.85 °C. Similarly for minimum temperature, the models showed improved bias between -0.03 to 0.03 °C against 0.51 °C. Of further relevance to our study and in the context of neural networks specifically, is the ANN model which used an MLP architecture with two hidden layers which was found to be very accurate, except in its estimates of low temperatures but which did outperform the RF. The following three literature will highlight further experimentation with these neural networks.

In the context of a deep neural network implementation (Scher and Messori, 2018) experimented with various Convolutional Neural Network CNN architectures. The study considered three weather variables: geopotential height (has a high correlation with surface temperature), meridional wind and zonal wind from GEFS reforecasts. Each of the input variables is represented as a channel (i.e. three channels RGB in a CNN convolution layer). The adopted network architecture had two convolutional blocks (see section 3.3), a fully-connected hidden layer and a single output neuron to provide the uncertainty forecast as a scalar quantity representing confidence level. This study also performed comparisons to SVM and RF which it turns out did not perform as well as the CNN model. The network implemented is seemingly basic compared to (Grönquist *et al.*, 2021) where more sophisticated architectures are implemented: ResNet for uncertainty quantification and U-Net structure (with a reduced number of levels) for bias correction. A locally connected network (LCN) was used in the last layer for bias correction. This is because, although the LCN performance was comparable to the CNN, it offered some improvements in the ability to use independent filters as opposed to shared filters; more plainly; the translational invariance feature of CNNs was found to negatively affect the performance of the bias correction model on a global scale. This approach thus used a combination of regular convolution in its *hidden* layers plus LCN in the last layer. The uncertainty estimation and bias correction tasks were treated as regression problems. The study recorded improvements in relative RMSE of 16% and 7.9% respectively on ECMWF ensemble output data. In both cases, the authors did not see improvement with additional layers (ResNet) or additional levels (U-Net) and did resort to reducing these in their networks relative to the layers in the reference architectures. The approach seems consistent with (Rasp and Lerch, 2018) (in terms of hidden layers); where their simple feed-forward network is implemented with only one hidden layer. The study utilized as

ground truth, temperature forecasts at 537 surface stations in Germany (this study was station-specific in contrast with (Grönquist *et al.*, 2021) global approach); along with the ECMWF 50-member ensemble daily forecast data from TIGGE dataset, in an '*embeddings* + *Fully connected network* + *neural network*' architecture. Similarly of interest in this case was the lack of improvement with additional complexity – i.e., increasing the number of hidden layers did not improve the results of the neural network model, thus in the final model, only one hidden layer was adopted. The fully connected neural network model which included one hidden layer achieved the best results at 30% of the 537 weather stations where forecasts were evaluated (this is against a benchmark EMOS technique). The neural network also achieved the lowest CRPS of all the other models evaluated. All of these studies discussed; (Cho *et al.*, 2020), (Scher and Messori, 2018), (Rasp and Lerch, 2018), and (Grönquist *et al.*, 2021) used the same activation function – Rectified Linear unit (ReLU) and Adam optimization algorithm in the implementation of the neural networks; - consequently these metrics are also found to be the preferred for bias correction of temperature forecast. Although there are difference was in the approach to the construction of Neural network architecture in each case; most aim for less complexity in the architectures finally adopted. (Cho *et al.*, 2020) and (Rasp and Lerch, 2018) in addition to Temperature as the weather quantity of interest to be estimated, included auxiliary variables, (5 and 21 of these respectively). Auxiliary variables included such weather quantities as longitude, latitude, solar radiation, heat flux and elevation. The results from both studies does indicate that information from additional predictors was a key component to improvements seen in the models over the raw ensemble (and 29% respectively).

The table below summarizes the machine learning / deep learning techniques which have been discussed in immediate past paragraphs as part of this study and which are found to be more relevant in our related task of ensemble weather postprocessing. It should be noted that **Performance* specified in terms of percentage is not a comparison between the techniques but rather each percent value is specific to the model implemented in that study and represents comparison between the performance of the model implemented and some benchmark technique or raw ensemble used in the specific study. Also included in the table are some of the key attributes of the study, the model implemented, and associated dataset used plus the

improvements recorded in ensemble forecast skill based on performance measure applied and the metrics utilized. This is by no means all-encompassing nor is it an exhaustive list but it serves to highlight some important points especially as relates to the ensemble-member size (mostly 10 and above) and time range (mostly in years); which has used in the training of the networks implemented. Furthermore, in terms of dataset, it is usual to obtain **GT ground truth* and **EF – Ensemble forecasts* from separate systems – surface stations vs. ECMWF 50-member ensemble (Rasp and Lerch, 2018); ERA5 vs. ENS10 (Grönquist *et al.*, 2021); 25 AWSs vs LDAPS (Cho *et al.*, 2020) - this is significantly different to our approach where both of these have been obtained from one dataset.

This and other relevant similarities/differences to our study will be discussed in the following Chapter.

Method	**QRF**	**FCN**	**CNN**	**ANN**	**U-net**
Authors	(Taillardat et al., 2016)	(Rasp and Lerch, 2018)	(Scher and Messori, 2018)	(Cho et al., 2020)	(Grönquist et al., 2021)
Features	Forecast Calibration	Forecasts Calibration	Uncertainty estimation	Bias Correction	Multiple*
Location	France	Germany	Europe	Seoul S.Korea	Global
Dataset	PEARP	TIGGE Dataset	GEFS dataset	LDAPS	ECMWF ENS10
Sample size	NA*	training 1,626,724 test 180,849 validation 182,218	10,000 samples	NA*	NA*
Time frame	2011 – 2014 Daily forecast	2007 – 2016 Daily forecast	1985 – 2016 Daily forecast	2013 – 2017 6-hourly forecast	1999 – 2017 2ce weekly forecast
Grid spacing	NA*	35-km	40-km	40-km	18-km
Variable	Surface Temperature & Wind speed	Temperature	Geopotential height & Zonal wind	Temperature	Temperature
Lead Time	3-54 hours	48-hours	72-hours	36-hours	48-hours
Ensemble	35-member	50-member	10-member	14-member	10-member
Scoring	CRPS	CRPS		RMSE	RMSE/CRPS
***Performance**		65.9 -73.5%		28%+	7.9%-16%

Table 1 - Comparison of Neural Networks approaches employed in ensemble weather postprocessing.

*Multiple means several postprocessing tasks were carried out: uncertainty quantification, bias correction, and then PDF calibration on the results of previous two.

*Performance is measured in terms of the metrics (CRPS/RMSE) and results compared against benchmark techniques or raw ensemble used in each study.

*NA – Not Available *GT – Ground Truth *EF – Ensemble Forecasts

2.3 Deep Neural Networks

Following the discussions from the previous sections, we present the motivation for the use of deep neural networks in this study. In summary, regression techniques in ensemble weather forecast postprocessing would estimate the conditional mean of the response variable, quantile regression would estimate the median or some other quantile but there is a gap in accounting for relationships which cannot be captured using these postprocessing methods which requires the consideration of a non-linear approach. Using deep neural networks, we can model these arbitrary non-linear functions. Widely applied in computer vision problems, a few studies show that the deep learning techniques in postprocessing can improve forecast skill through tasks such as bias correction and uncertainty calibration which when successfully implemented show improved results over the benchmark ensemble forecast and other techniques (section 2.2). This can potentially reduce the number of ensemble input forecasts required saving computational cost of the models (Grönquist *et al.*, 2021). This makes the choice of applying a deep learning technique for this project reasonable.

2.4 Convolutional Neural Network Architecture

The term Convolutional Neural Network is commonly referred to as CNN or CovNet. These will be used interchangeably for the remainder of the report. CNNs are common in computer vision applications where generally the input into the network are images. The architecture is such that the neurons are arranged in 3 dimensions of width, height and depth and thus works well will image data. As an example, for an input specified as [970x1042x3]; the network will expect an image with width 970, height 1042 and with three color channels Red, Green, Blue (R,B,G). This allows manipulation of the architecture to contain additional properties such as weather quantities or ensemble members: (Scher and Messori, 2018) encoded three different weather variables as separate channels and in this study, three ensemble members are encoded as RGB channels.

A CNN is typically fed via an *Input Layer* and the target or output value is represented through the final layer of the network, *Output Layer.* The image example above will be

represented in the CNN by 970x1042x3 neurons in the input layer each with an activation number, and associated weight and a constant bias:

$$\sum(weight * activation + bias)$$

And applying the activation function gives:

$$FUNCTION(\sum(weight * activation + bias)).$$

If we denote the activation (value of a given node) by a_i; its associated weight by w_i; with constant bias b; then the output for each layer is given by: $f(\sum w_i a_i + b)$. The consistent choice of an activation function in our above literature survey is the ReLU activation function. This is a non-linear activation function of the form $ReLU(x) = \max(0, x)$. Such that our function becomes:

Equation 3

$$f(a, W) = ReLU\left(\sum w_i a_i + b\right)$$

ReLU has become the common choice because it addresses the vanishing gradient problem – a situation where the gradients continue to decrease approaching zero causing the gradient descent not to converge. Due to this feature, ReLU activation function should also speed up learning and as a result it is critical in convergence and speed of the network.

The network learns the right weights and biases which give the most accurate output by optimizing these in such a way as to reduce a loss function. In ensemble weather forecast post-processing tasks, the loss functions commonly used are the Root Mean Squared Error (RMSE) and Continuous Ranked Probability Score (CRPS) – this has been highlighted in section 2.2. The optimization of a function is achieved by a process known as stochastic gradient descent (SGD) and a popular choice for its implementation (also previously described in earlier section) is the ADAM optimization method. The reader is referred to (Chollet, 2017) for detailed descriptions.

On a final note from an architecture standpoint, the choice of certain hyperparameters are critical to the success of the model in terms of producing reliable output. Going back to our earlier dimensions for the input layer, [970 x 1024 x 3] represents the width, height and depth of the input shape and a corresponding output shape needs to be set as well, which depends on the target value. For instance, a classification problem

where a percent likelihood that a certain type of object is detected, will require an output layer depth which is mapped to the number of objects. To creatively achieve the required output shape, a combination of strides and zero-padding is applied to the Input. This is especially important when the spatial dimensions $[\mathrm{W\ x\ H\ x\ D}]$ of the output shape need to be preserved such that they match the input. (Stanford, 2021). Note that the spatial size changes during the training when one or more $pooling\ layers$ are applied as the algorithm moves from layer to layer through the architecture. ($pooling$ is described further in chapter 3). Not much detail has been provided regarding specific output layer shape construction in the literature reviewed, it is possible to infer based on the architectures presented that an output volume size with a depth $D = 1$ is common as an example in (Scher and Messori, 2018). To provide for the representation of a scalar value that predicts forecast uncertainty, it would require simply $flattening$ the final layer into the required dimension. A set of more complex calculations are required to maintain the original dimensions for the output shape and is discussed further in the following chapter.

CHAPTER 3. METHODOLOGY AND DESIGN

The data set is provided by the MET office UK as output from the Met Office Global and Regional Ensemble Prediction System (MOGREPS-UK). It is hosted in an AWS S3 bucket online storage system and contains a month's worth of forecast temperature data obtained at 1.5m grid point spacing across the UK for January 2019. The MOGREPS-UK ensemble system runs with three ensemble members each producing a new forecast starting every hour and predicting out to 5 days in advance (T+126 hours). The temperature output is recorded each hour. The model forecast is stored in image format. Each 970 x 1042-pixel image represents the temperature at each time point across the UK from three ensemble members. Each ensemble member is encoded as a different color channel, giving three channels RGB with three sets of numbers. A single png image thus has dimensions of 970 x 1042 x 3 and this is a representation of a temperature value at a particular point in time for locations across the UK. i.e at 12:00hours we have a single image representative of the temperature at UK locations.

3.1 Image Pre-processing

The data was accessed from Amazon S3 bucket where it is hosted using credentials and Jupyter Notebook access steps provided. The process for converting the data was as follows:

- For each ensemble member, in each forecast output (file), the minimum and maximum values in the original data array was calculated.
- The data is rescaled such that the minimum value is 0, and the maximum is 255.
- The rescaled data array is converted to a PNG image.
- The previous minimum and maximum values are recorded in a json file, this will allow for reversal of the rescaling process.

3.1.1. Dimensionality

Modelling this data with a neural network is non-trivial due to the high dimensionality of the data set. Each record is stored as a 970 x 1042-pixel image. For each image to be fed into a neural network, an input node is required for every number in the array. For instance, each image requires 970 x 1042 x 3 = 3,032,220 nodes in the input layer ONLY. For each additional layer in the network, this number is multiplied by weights and biases therefore 3,032,220 nodes per images can easily grow into tens of millions making this a very computationally expensive process. Furthermore, the dataset contains hourly values (i.e. 24 cycles each day) from 3 ensemble members for 31 days and 126 lead times for each forecast; giving us a total of 282,232 samples in the dataset.

Given that the dataset is too large to be loaded into memory therefore, data is retrieved in batches of 10 -12 on-the-fly (each batch is 485MB) and this is achieved using Keras Sequence class (*tf.keras.utils.Sequence | TensorFlow Core v2.5.0*, no date). This Sequence is employed not only in the model but also in other significant computations which are carried out as part of this analysis and the details explaining specific computational steps taken, are covered in the relevant sections.

3.1.3. Feature Transformation

To feed complex data like images into the Neural Network in an acceptable format, a general requirement for Machine Learning algorithms is the need to scale the data.

The original forecast data in Kelvin with values ~280 to 300k were converted to image format with minimum temperature value set to 0 and maximum set to 255 (black and white respectively). This allows for maximum use of the integer range, without losing significant amounts of precision. To perform transformation on the entire dataset, the original temperature values in Kelvin needed to be obtained. These values had been stored in a json file during the conversion-to-image stage to enable a reversal of the process i.e. rescaling the data back to their original values (noting some loss of precision because of the conversion process (highlighted as caveat in Ch. 1). Both Standardization (mean removal and variance scaling) and Normalization (scaling between a range of 0 and 1) were applied to the data. At this stage, both approaches were sampled to provide for options for training process since each method presents alternative advantages and provide a way to preserve zero values in the data.

Again, due to the high dimensionality of this dataset (a single array shape is 970, 1042, 3), standardization of the data is non-trivial. For example, the standard MinMaxScaler and StandardScaler utility classes in the Python sklearn.preprocessing package do not work in this case because the entire data set cannot be loaded into memory and transformed in one step by simply implementing either of these functions. A creative approach was adopted to compute the mean and standard deviation in a stepwise way. The process required calculating and storing the sums and total number of elements in a piecewise manner in batches of 10 to 12 images per batch using the Sequence. Then, collating these stored sums and calculating the mean and standard deviation. This process was carried out on both the forecast data and observation data and took more than 4 hours in total. The standardization of the entire dataset is achieved during model execution within the network model by inserting the values computed for mean and standard deviation into the Sequence subclass.

3.1.4. Forecast-Observation pairs

This section covers further image analysis on the data to compare samples of the input data which are used as forecast-observation(*truth)* pairs. Observation or *truth* will be used interchangeably in the report. The following steps taken are aimed at highlighting the difference in the image representations of the data and to show their validity to be used in the network model as input and target variables respectively. Further analysis is carried out post standardizing the data to confirm the differences between the

forecast and observation pairs by comparing their representative images. This is visible in Figure 2 where a visual inspection shows the variation that exists between forecast data and observation data. This difference was initially not as significant visually when the original images were plotted but is clearer plotting with the standardized data. Calculating mean square error (MSE) between the forecast - observation image pairs for two time steps selected at random: 12:00 on Jan 1st 2019 and 13:00 for Jan 1st 2019, returned MSE values of 0.306 and 0.311 respectively, further confirming the difference in the two images contained in the forecast-observation pairs. For our case here, lead time has been fixed at 12hours to obtain the forecast – observation pairs. This intuition is relevant to the bias correction method which accepts pairs of the forecast and observation images fed through the sequence and outputs a measure of the error between these for a certain lead time.

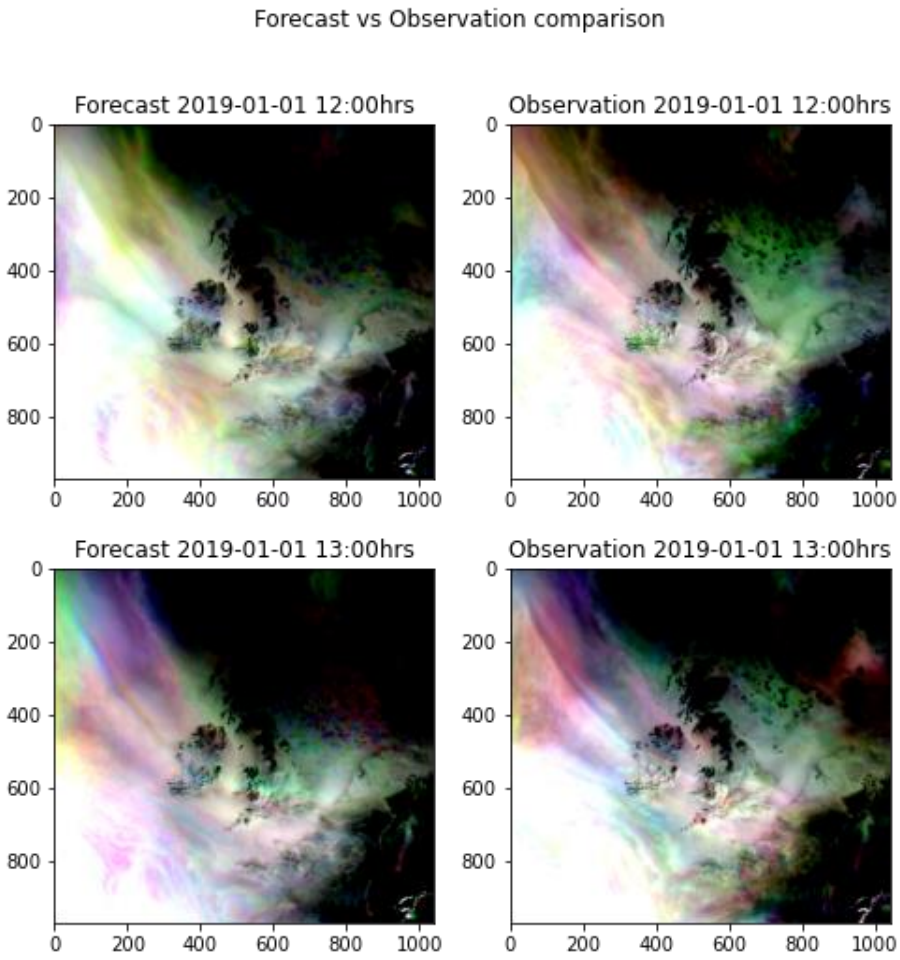


Figure 2 - A visualization of forecast vs observation. This is at hourly intervals for 12:00 and 13:00 same day

3.1.5. Sample Forecast Case

As an example, we consider the ensemble forecast of temperature for Wales on Jan 01, 2019 0:00 hours selected at random for each of three ensemble members which is shown in Figure 3. The forecast images are shown side-by-side with the corresponding observations for the selected example date and time. The variation appears clearer in this representation with temperature bar which shows how the temperatures are calibrated on forecast vs observation representations. This confirms visually the bias present within each of the ensemble forecasts.

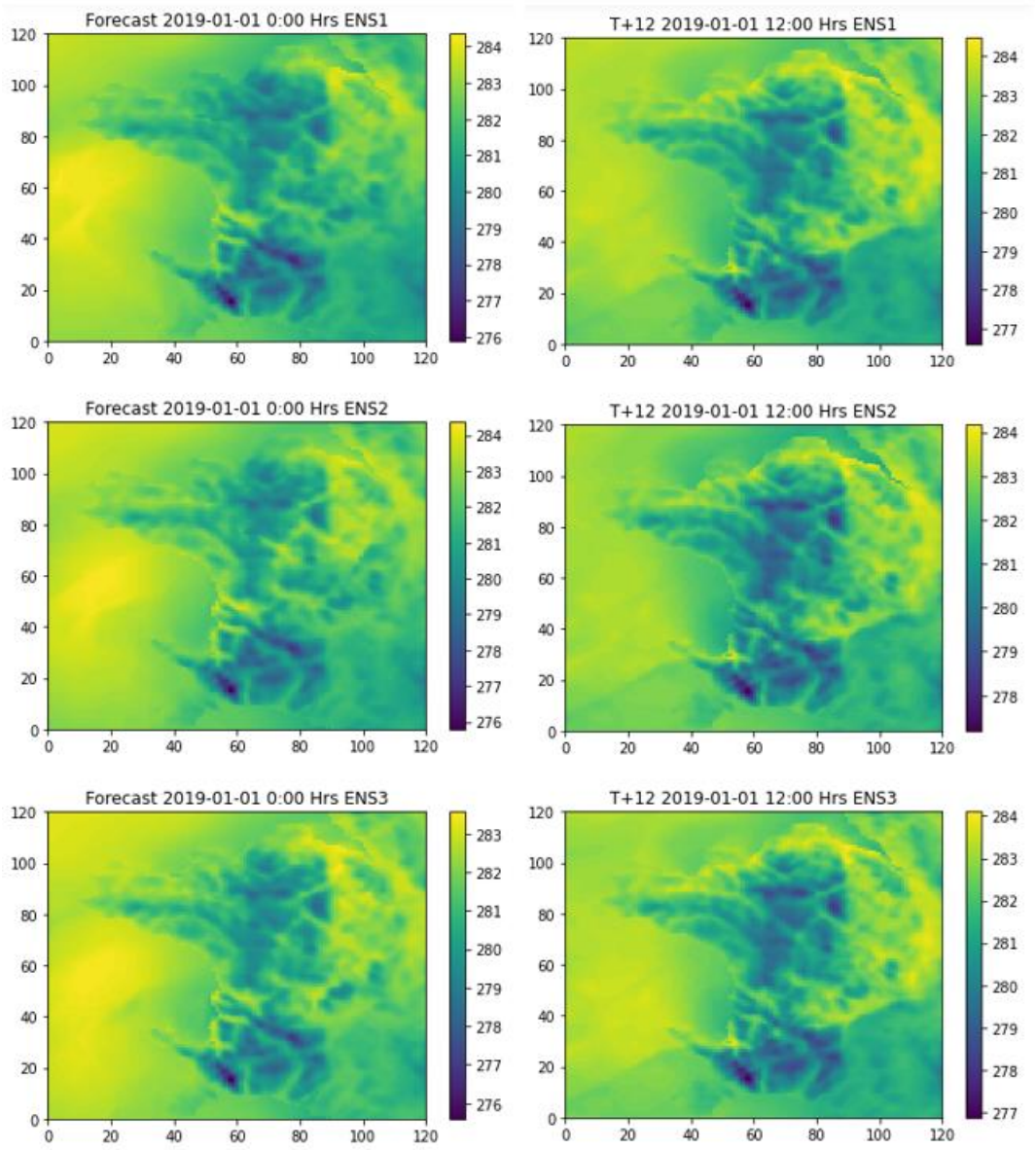


Figure 3 - Three-member ensemble raw data for cut-out of Wales with forecast vs observation

3.2 Neural Networks models

Neural networks have been in use for years across a wide range of research although not explored extensively in meteorological applications and specifically for post processing problems. Therefore, this remains an area of active research focus for successful application of deep learning techniques to weather and climate data. The application of convolutional neural networks to this problem is novel in this sense given the nature of the data set we are working with, time range of a month and the ensemble member size. This is in comparison with surveyed literature where reforecast/re-analysis/weather-station data over a range of several years and with ensemble member size greater than 10 are the norm. Neural networks make very efficient use of modern computing hardware like GPUs. The modular nature of neural network enables models to be built by assembling layers in a building-block style pattern which makes development in Keras more straightforward. The choice of convolutional neural networks for this problem is the advantage of the ability of the network to reduce dimensions of images but still retain the critical information required to perform optimally. This approach is expected to increase the speed of the training process and reduces computational power.

3.2.1 Bias correction model

For our bias correction case, we consider the difference between observed temperature values and the ensemble mean forecast of the temperature, the average of which gives the magnitude of the ensemble mean forecast error. For *N* ensemble members where m_i is the ensemble mean for the i^{th} forecast and o_i is i^{th} observation:

Equation 4

$$c = -\frac{1}{N}\sum_{i=1}^{N} O_i + \frac{1}{N}\sum_{i=1}^{N} m_i$$

The following times describes our representation of the temperature data:

Cycle Time: The time the forecast started running e.g. 01:00 hours on Jan 1st.

Validity Time: Time for which the forecast is valid e.g., 13:00 hours on Jan 1st.

Lead Time: How far in advance the forecast is for e.g., 6 hours, 12 hours, 1 week, etc.

To simplify the algorithm training process, we have chosen to work with the ensemble mean temperature for the bias correction model. To account for the bias associated

with the mean, we have considered the *truth* value for a forecast starting at T + 0 with 12 hours lead time to be the forecast which starts at T + 12 (lead time will be 0 hours). Note cycle time and validity time are the same at a lead time of 0 hours and this serves as the observations (*truth)*.

Cycle Time + Lead Time = Validity Time.

The model will therefore compare forecast -- observation pairs as (C-0:00 + L-12hrs = V-12:00; C-12:00 + L-0hrs = V-12:00). The goal is to match the actual errors between the *Truth* and Forecasts with model predictions of this error.

A specific lead time fixed at 12 hours is selected for the forecasts – observation pairs. The data is read in as pairs of *ground truth* and forecast values. To accurately create the pairs, validity time should be the same for *truth* value and forecast value.

Steps:

1. Over the three ensemble members take the mean of a particular temperature forecast F_i for a cycle
2. The corresponding mean temperature at T + 0 is considered as the *ground truth* this is the observation O_i (where lead time is equal to zero). It is important to note that the values we are treating as observations are really the results of the data assimilation; technically known as the analysis fields. We can also refer to these as the initial conditions.
3. Correct the mean of the forecast to the mean of the observation such that the actual error $Observation_i - Forecast_i$ i.e. $O_i - F_i$ matches the network's prediction of the error.

The above means that by accurately predicting this difference, we can adjust for bias in the forecast.

With lead time fixed at 12 hours we have N = 732 paired data points for observation and forecast for which we are aiming to approximate a relationship using a CNN by minimizing the error:

Equation 5

$$Q = \sum_{i=1}^{N} \varepsilon_i^2 = \sum_{i=1}^{N} (\hat{O} - O_i)^2$$

Where $\hat{O}$ represents our predicted values from model output. For our bias correction model, the Convolutional Neural Network architecture which was determined after a series of trial and error models, consisted of three hidden layers, each with a Rectified linear unit (ReLU) activation function. The increased number of hidden layers is necessary to allow the network learn the complex representation exhibited by our data set, however we also suffered impact to computing resources as a result. There was no flattening required in our scenario and hence no dense layer in the last layer. The loss function is selected to be the MSE – our regression decision per pixel places an MSE loss function on every pixel of the output such that for every pixel in the output we compute MSE loss between it and ground truth pixels and this value describes how well the network is predicting the mean and the predictions are corrected such that MSE is minimized. Finally, an adaptive moment estimation (Adam) optimization algorithm was adopted in the model.

To perform the machine learning task of training and testing, the N=732 pairs of forecast / observation data is split into three sets for training (456 pairs), validation (132 pairs) and for testing (144 pairs). For this training process, the typical random splitting of datasets common with machine learning is not recommended as autocorrelation evident in the data is non-zero (see Figure 4). In related studies, data sets are typically split by years and full years (or range of years) are used for training, validation and testing which enables the deep learning algorithm to make predictions

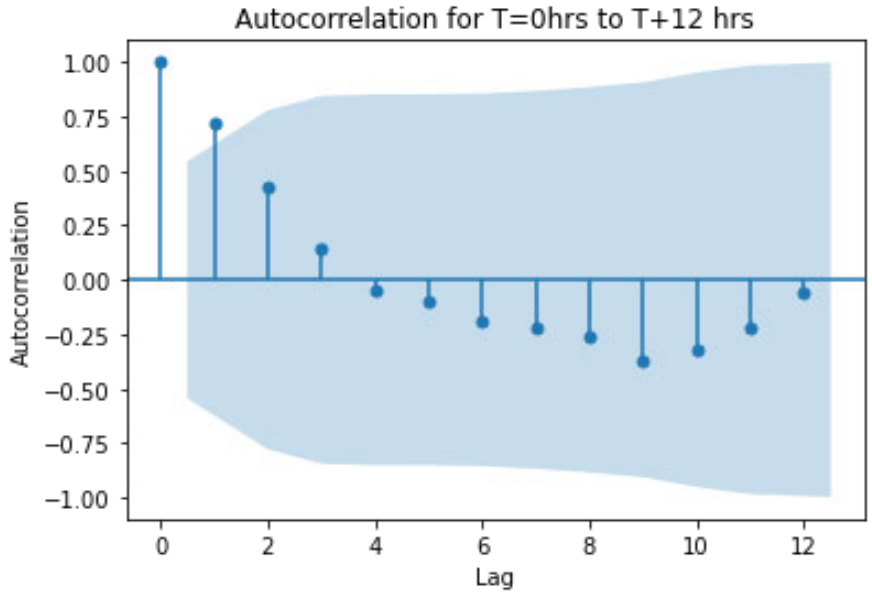


Figure 4 - non-zero autocorrelation in data

of the target values more easily and effectively (Scher and Messori, 2018). Since we have a month's worth of hourly data, the split has been done in a serial manner to ensure that the hours before and after each test case are included within the same set.

Figure 5 provides a graphical representation of the data preparation and loading steps carried out to feed the neural network model.

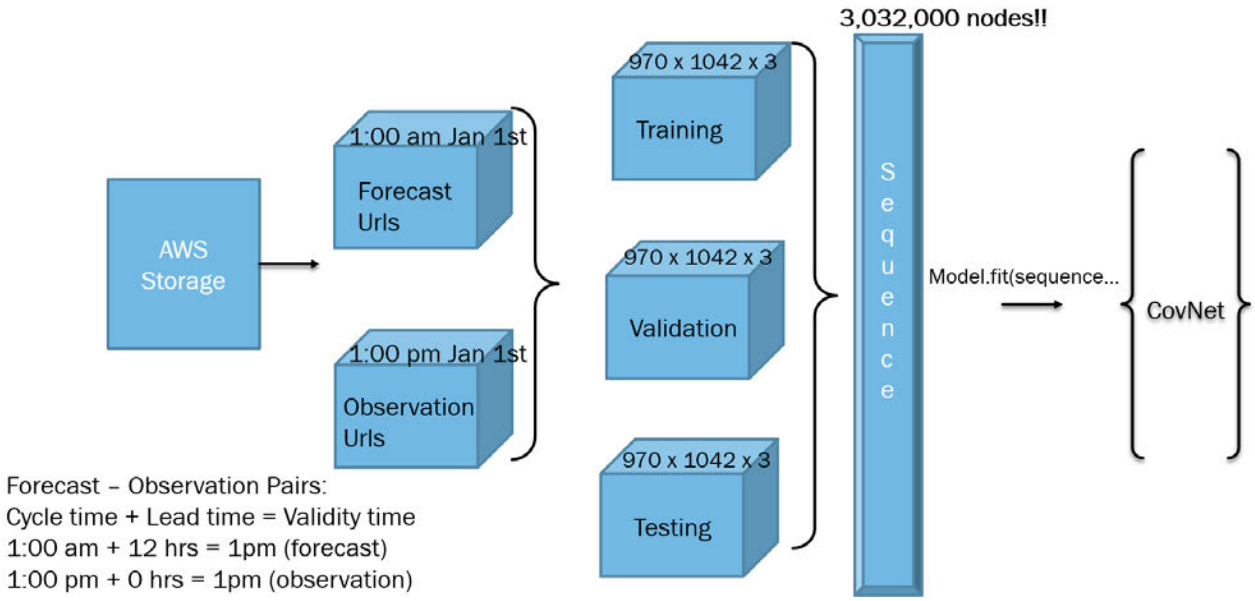


Figure 5 - Data pre-processing summary

3.3 CNN Estimation Procedure

The neural network structure needs to be considered carefully for each choice of technique to be applied. The following notes describe the common layers implemented in a CNN architecture. Following an example of a standard implementation of a CNN architecture as implemented in a regression task from (Chollet, 2017). The network consisted of two hidden layers and a dense layer as the last layer described as follows:

The **Convolutional layer** adds translational invariance to recognize inputs even if translated which allows the network to find the pattern no matter where it appears in the image. The **Max pooling Layer** down samples the data by passing on only the most significant output from the convolutional layer. The **Dropout layer** forces the network to network to learn in a more robust way by randomly dropping some data passing through it by cutting some of the connections in the neural network. This prevents the neural network from memorizing the training data. The convolutional layer, max pooling layer and dropout layer make up the convolutional block (an

example was implemented in (Scher and Messori, 2018). Finally, the **Dense Layer** maps the previous outputs to the output layer. A network of the form described, did not work well for our case and the next two sections describe in detail the approach taken to achieve our

3.3.1. Downsampling and Upsampling

and output because the architecture does not support the preservation of the input dimensions! Dimensions should be same for both input and output. Hence the approach taken to first downsample the data to enable the network perform computations at a lower resolution and subsequently upsampling the network to return the spatial resolution of the output. Strategy for increasing the feature size inside the network

The network learning approach is implemented leveraging an architecture from the computer vision domain which entails:

1. Using regular convolution operations to learn the kernels
2. Then using the learned kernels in a transposed convolution
3. The technique allows us to find the “original data” in this case our forecast images
4. Ideally, the aim is to find something that looks like our original image
5. This provides a basis to compare the original input and the output image (using loss function = ‘mse’) and ascribe associated bias.

A Convolutional Neural Network model has been developed for this problem inspired by architecture as described in Keras autoencoder documentation (Valdarrama, 2021) and U-Net approach in (Grönquist *et al.*, 2021). The steps detailed above are typical of a convolutional autoencoder where the result of encoding/decoding technique is aimed at a classification and as such the architecture would transition from a convolutional layer to a fully connected layer (dense) and the output is a scalar quantity. In contrast, for our model, the architecture is modified for a regression task using the techniques which were explained in section 2.4; we need to maintain the spatial dimensions of the output shape which is a requirement for this problem. Applying convolutions that keeps the size of such a high resolution input image is computationally expensive, so in the first half of the network we use a combination of convolutions and pooling operations to extract features at each level - $downsampling$;

and in the second half of the network, the process features are upscaled again (increasing the spatial resolution of the predictions) to get back the output in the same size as the input shape - $upsampling$. This avoids performing all the convolutions at the full spatial resolution of the image but still being able to make the network deeper and then perform the processing at a lower resolution (Stanford, 2021)

3.3.1. Hyperparameter tuning

Calculations for the $downsampling$ and $upsampling$ layers required the following hyperparameters to be adjusted: Number of filters (or kernel) – k; Spatial extent (or Receptive field) – F; Stride – S; Zero-padding – P. To calculate output volume, it is given that for an input volume defined as $[W_1 * H_1 * D_1]$ then our output volume $[W_2 * H_2 * D_2]$ is given as:

Equation 6

$$W_2 = \frac{[W_1 - F + 2P]}{S} + 1$$

$$H_2 = \frac{[H_1 - F + 2P]}{S} + 1$$

$$D_2 = K$$

Working with above formula from (Stanford, 2021) network was initialized with convolutional layer $32\ \text{channels}$ and Receptive field $(3, 3)$ at the input layer. Followed by a pooling layer $(2, 2)$. One convolutional with same parameters and one pooling layer with same parameters is repeated to complete the $downsampling$.

For the $upsampling$ half of the network, two transpose layers with $32\ \text{channels}$, $spatial\ extent\ (2, 2)$ and $stride = 2$ are added to the network. These layers increase the size of the output from the reduction done in the $downsampling$ step. The final layer is a convolutional layer with $1\ \text{channel}$ and, $spatial\ extent\ (3, 3)$, brings the output to the exact dimensions of the input.

While the full UK region with $970\ \text{x}\ 1042\ \text{x}\ 3$ was attempted during training, the choice of input into the model had to be reconsidered due to time and computational constraints. As final input we have taken a cut-out of Wales from the data which is of the order $120\ \text{x}\ 120\ \text{x}\ 3$. We are thus feeding the network with forecast – observation pairs for this region only – taking the mean reduces the depth to 1 and brings our final input volume to $120\ \text{x}\ 120\ \text{x}\ 1$.

For our final network, the saved structure of the neural network along with weights is recorded. The weights and structure are saved separately because network was trained multiple times with different settings and modifications to the amount of data in training set. This makes it possible to now load different sets of weights using the same neural network structure. Figure 6 shows the final network architecture implemented.

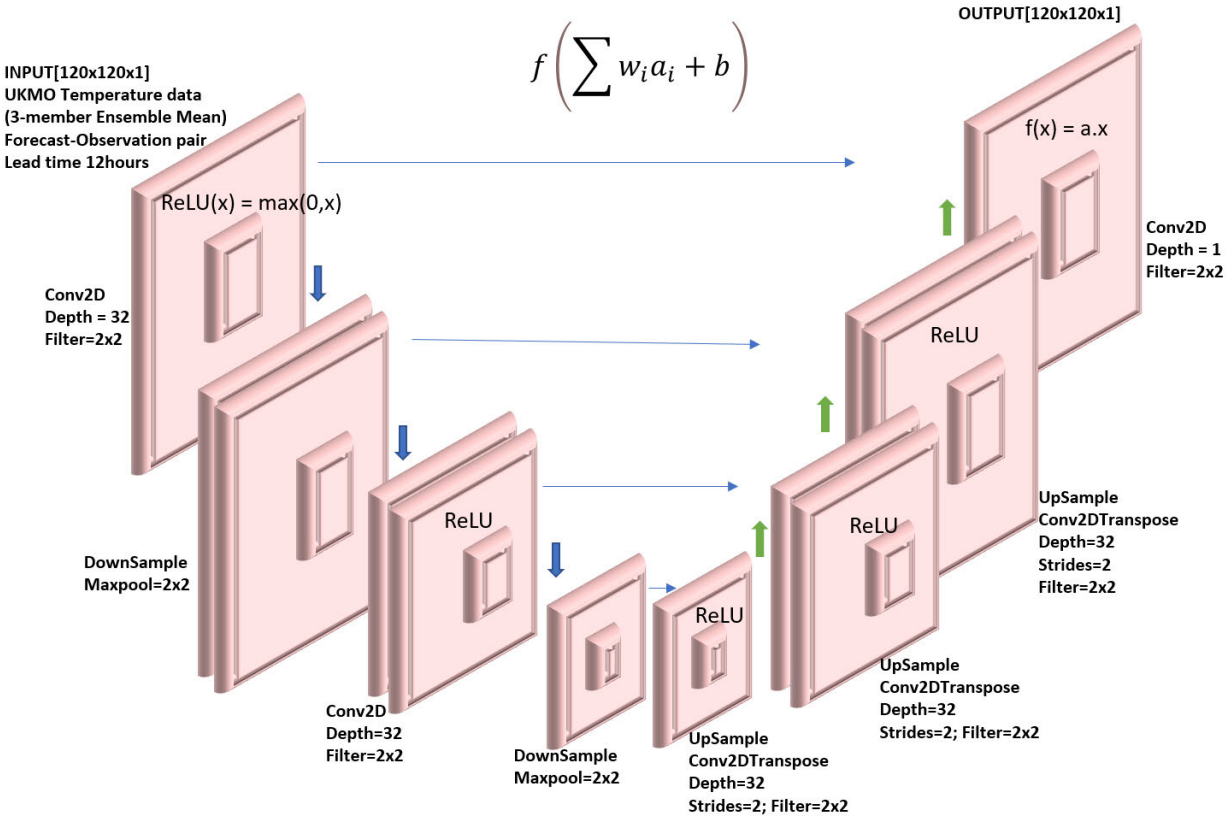


Figure 6 - Bias Correction CNN Network

3.4 Execution

The network described in section 3.3 above is implemented using Keras API with the following metrics specified within the network:

Activation function: Non-Linear Rectified Linear Unit – ReLU.

Loss function: Mean Square Error MSE; computes the difference between network predictions and the target (which is our ground truth / observations).

Stochastic gradient descent: For optimizing weights and biases to reduce loss function Adaptive Moment Estimation ADAM using a learning rate lr=0.001.

Metric monitored: Mean Absolute Error MAE; computes the difference between network predictions and the target (which is our ground truth / observations) and provides an indication of how far off the predictions are from the ground truth.

In terms of computing resources, the training lasted several days on a CPU and did not converge. Google Colab free edition was also insufficient as the time out period is just 12hours. Therefore, a Google Colab Pro subscription was required which provides access to faster GPUs and up to 24 hours connection time and less timeouts. The TPU option was selected at runtime and with this the model eventually reached convergence.

CHAPTER 4. RESULTS AND EVALUATION

During training, the loss represented by MSE and MAE was monitored as an indication of the performance of the model. After a few epochs (2-5), the values for loss and MAE become NaN. The model was adjusted to include several recommendations to penalize this behavior: standardizing the data; normalizing the data, $l2$ regularization, weight initialization, reducing the batch size during training and, reducing the size of the gradient by clipping. There was little improvement applying the above as NaN values were continually returned. The model architecture – was also simplified and the number of layers reduced but this did not change the results. While it should be possible to fix the issues; however the running time for each execution of the model makes it difficult to resolve within the time frame. Therefore, the next few sections will discuss the expectations in terms of the results and possible problems which might cause the results which have been obtained in the study.

4.1 Challenge of exploding gradients

Further research into the problem suggests that obtaining NaN values is as a result of exploding gradients. Described as a situation where there is a large increase to the norm of the gradient during training due to the exponential growth of long term components compared to short term ones (Pascanu, Mikolov and Bengio, 2012). In this case, the loss values which are returning NaN values during the training and the changes in the loss are quite large going from one update to the next which causes us to conclude the network suffers from this problem of exploding gradients.

The authors present regularization and gradient clipping approaches as steps to resolving this challenge. However these techniques have not worked in this case. Reviewing the network architecture, the design does not appear faulty neither is it overtly complex. It is also worth mentioning that the paper describes the application of these techniques in a recurrent network (ResNet) and thus it would be worthwhile to further investigate the performance of a model with a Recurrent Network architecture which will consider a temporal component. This is not to say that the CNN used in the study is limited in modelling the data, on the contrary, our survey has shown wide success of different CNN architectures applied in post-processing and with more time and further investigation, the issues discussed above can be resolved.

4.2 Expectation

While the task of correcting the forecast ensemble mean to match the observation may seem simple, it has been shown that manipulating the dataset utilized for the task is nontrivial. It was also important to deliver results which showed successful implementation of bias correction using this UK Met Office dataset while highlighting the difficulties of modeling such high dimensional data. It was also important to show the potential of getting reliable results while using a small number of ensemble members. Both of these would serve as an initial result with the potential to expand techniques to each of the ensemble members and further to uncertainty calibration. With more time, further extension using Gaussian processes could have also been explored as well.

In the choice of network architecture, the input and output significantly differ from literature cited. For example (Scher and Messori, 2018) achieved forecast uncertainty using convolution blocks as described in section 3.3 but which cannot address the problem in this study which is an image-to-image challenge requiring output of a prediction which “looks like” the image while previous studies in ensemble post processing have adapted the a computer vision approach to output a probability value between zero and one and more akin to a classification approach.

CHAPTER 5. CONCLUSIONS

5.1 Summary of the findings

The approach in this research was to provide an innovative analysis of high-dimensional data with smaller size of ensemble members for the ensemble post processing task of bias correction. While this has not been entirely successful, it can be seen that the data processing steps are a meaningful addition to the study. Additional time for training and fine-tuning the model should have also produced some meaningful results .

5.2 Novelty and Limitations of the Research

In previous studies, the size of the ensemble members were 10 and higher, in contrast with our study using only 3-ensemble members. The lower number of ensemble members has its limitations, for example, ideally for a more robust output, using increased number of ensemble members makes it possible to select individual members based on their performance. If specific members have consistently lower weighting, it makes sense to eliminate these in order to have a reduced ensemble. Also with more ensemble members, the PDFs of the predictions can be sampled more easily, and inferences made using as an example, standard deviation, would be more useful. However, equally non-trivial is the attempt to obtain results using limited number or smaller sized ensemble. There are advantages to this, a leading one of course being the potential reduction in computational cost of requiring much larger ensembles to perform post-processing tasks. Secondly, datasets from literature surveyed were in GRIB format or other numerical formats which did not require the level of pre-processing implemented in this study where data is very high-dimensional image data. The number of years spanning 5 to several years of data required for deep learning experiments is also shown to be potentially. The study is also creative in obtaining forecast and ground *truth* from the same data set.

In terms of limitations, an important aspect not explored in the study is the different weather patterns which are evident for different seasons in a year. Therefore, to account for other seasons, a dataset containing these annual seasonal patterns could have been more robust.

For simplicity and ease of computation, the ensemble average has been used in the bias correction model. A weighted approach could improve the results and provide a

more sophisticated model which would be better suited to UK MET Office ensemble weather forecast operations and other similar operating centers.

Transfer learning from previous neural network architectures could have been explored, although this was non-trivial to build and adapt to this problem and data set in the given time frame. There is also a limitation in this field in general related to the lack of a framework for applying deep learning in post-processing despite the number of previous studies addressing this. This framework equally impacts the transferability of the techniques *successfully* applied. One could site data sources, structure etc as reason, however, based on the study and reading, it is likely that computer vision provides some reasonable building blocks as a starting point for such a framework to be developed. Finally, due to time constraint, other methods were not tried for comparison and a potential extension to deep learning with Gaussian Processes was not explored

5.3 Future Activity

Probabilistic forecasts which are accurately calibrated remains crucial in many fields. Renewal energy and air traffic control and commonly mentioned in literature (Bauer, Thorpe and Brunet, 2015) and (Vannitsem, Wilkis and Messner, 2018). Therefore extensions of the method to multivariate predictive distributions which can also incorporate intervariable, spatial and temporal dependance structures would make a significant addition to the study.

With previously successful studies, there is to the best of our knowledge no recommended architecture for machine learning implementations on meteorological data. There is room for research which produces a network architecture designed specifically for problems in this space.

REFERENCES CITED

Antonopoulos, S. *et al.* (2020) 'PROGNOS: A Meteorological Service of Canada (MSC) Initiative to Renew the Operational Statistical Post-processing Infrastructure', in Mensink, C., Gong, W., and Hakami, A. (eds) *Air Pollution Modeling and its Application XXVI*. Cham: Springer International Publishing (Springer Proceedings in Complexity), pp. 291–295. doi:10.1007/978-3-030-22055-6_46.

Bauer, P., Thorpe, A. and Brunet, G. (2015) 'The quiet revolution of numerical weather prediction', *Nature*, 525(7567), pp. 47–55. doi:10.1038/nature14956.

Bröcker, J. and Smith, L.A. (2008) 'From ensemble forecasts to predictive distribution functions', *Tellus A*, 60(4), pp. 663–678. doi:10.1111/j.1600-0870.2008.00333.x.

Buizza, R. *et al.* (2005) 'A Comparison of the ECMWF, MSC, and NCEP Global Ensemble Prediction Systems', *Monthly Weather Review*, 133(5), pp. 1076–1097. doi:10.1175/MWR2905.1.

Buizza, R. (2018) 'Chapter 2 - Ensemble Forecasting and the Need for Calibration', in Vannitsem, S., Wilks, D.S., and Messner, J.W. (eds) *Statistical Postprocessing of Ensemble Forecasts*. Elsevier, pp. 15–48. doi:10.1016/B978-0-12-812372-0.00002-9.

Cai, H. *et al.* (2019) 'Gaussian Process Regression for Numerical Wind Speed Prediction Enhancement', *Renewable Energy*, 146. doi:10.1016/j.renene.2019.08.018.

Cho, D. *et al.* (2020) 'Comparative Assessment of Various Machine Learning-Based Bias Correction Methods for Numerical Weather Prediction Model Forecasts of Extreme Air Temperatures in Urban Areas', *Earth and Space Science*, 7(4), p. e2019EA000740. doi:10.1029/2019EA000740.

Chollet, F. (2017) *Chapter 3. Getting started with neural networks · Deep Learning with Python*. Available at: https://livebook.manning.com/book/deep-learning-with-python/chapter-3/ (Accessed: 6 June 2021).

DTC (no date) *dtcenter.org*. Available at: https://dtcenter.org/community-code/unified-post-processor-upp (Accessed: 3 September 2021).

Gneiting, T. *et al.* (2005) 'Calibrated Probabilistic Forecasting Using Ensemble Model Output Statistics and Minimum CRPS Estimation', *Monthly Weather Review*, 133(5), pp. 1098–1118. doi:10.1175/MWR2904.1.

Grönquist, P. *et al.* (2021) 'Deep learning for post-processing ensemble weather forecasts', *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 379(2194), p. 20200092. doi:10.1098/rsta.2020.0092.

León, J.A.P. (2021) *Major flash floods in Sardinia in November 2020*, *ECMWF*. Available at: https://www.ecmwf.int/en/newsletter/167/news/major-flash-floods-sardinia-november-2020 (Accessed: 3 September 2021).

Parker, W.S. (2010) ‘Predicting weather and climate: Uncertainty, ensembles and probability’, *Studies in History and Philosophy of Science Part B: Studies in History and Philosophy of Modern Physics*, 41(3), pp. 263–272. doi:10.1016/j.shpsb.2010.07.006.

Pascanu, R., Mikolov, T. and Bengio, Y. (2012) ‘On the difficulty of training recurrent neural networks’, p. 9. Available at: http://proceedings.mlr.press/v28/pascanu13.pdf.

Peng, T. *et al.* (2020) ‘Prediction Skill of Extended Range 2-m Maximum Air Temperature Probabilistic Forecasts Using Machine Learning Post-Processing Methods’, *Atmosphere*, 11(8), p. 823. doi:10.3390/atmos11080823.

Raftery, A.E. *et al.* (2005) ‘Using Bayesian Model Averaging to Calibrate Forecast Ensembles’, *Monthly Weather Review*, 133(5), pp. 1155–1174. doi:10.1175/MWR2906.1.

Rasp, S. and Lerch, S. (2018) ‘Neural Networks for Postprocessing Ensemble Weather Forecasts’, *Monthly Weather Review*, 146(11), pp. 3885–3900. doi:10.1175/MWR-D-18-0187.1.

Roman, S. and Annette, M. (2018) *Chapter 4 - Ensemble Postprocessing Methods Incorporating Dependence Structures | Elsevier Enhanced Reader*. doi:10.1016/B978-0-12-812372-0.00004-2.

Scher, S. and Messori, G. (2018) ‘Predicting weather forecast uncertainty with machine learning’, *Quarterly Journal of the Royal Meteorological Society*, 144(717), pp. 2830–2841. doi:10.1002/qj.3410.

Slingo, J. and Palmer, T. (2011) ‘Uncertainty in weather and climate prediction’, *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 369(1956), pp. 4751–4767. doi:10.1098/rsta.2011.0161.

Stanford (2021) *CS231n Convolutional Neural Networks for Visual Recognition*. Available at: https://cs231n.github.io/convolutional-networks/#conv (Accessed: 3 September 2021).

Taillardat, M. *et al.* (2016) ‘Calibrated Ensemble Forecasts Using Quantile Regression Forests and Ensemble Model Output Statistics’, *Monthly Weather Review*, 144(6), pp. 2375–2393. doi:10.1175/MWR-D-15-0260.1.

tf.keras.utils.Sequence | TensorFlow Core v2.5.0 (no date) *TensorFlow*. Available at: https://www.tensorflow.org/api_docs/python/tf/keras/utils/Sequence (Accessed: 7 August 2021).

The Met Office ensemble system (no date) *Met Office*. Available at: https://www.metoffice.gov.uk/research/weather/ensemble-forecasting/mogreps (Accessed: 11 June 2021).

Valdarrama, S.L. (2021) *Keras documentation: Convolutional autoencoder for image denoising*. Available at: https://keras.io/examples/vision/autoencoder/#build-the-autoencoder (Accessed: 29 August 2021).

Vannitsem, S., Wilkis, D.S. and Messner, J.W. (2018) *Statistical Postprocessing of Ensemble Forecasts*. Elsevier. doi:10.1016/C2016-0-03244-8.

Wang, C., Zhang, W. and Villarini, G. (2021) 'On the use of convolutional Gaussian processes to improve the seasonal forecasting of precipitation and temperature | Elsevier Enhanced Reader', *Journal of Hydrology*, 593. doi:10.1016/j.jhydrol.2020.125862.

Wilks, D., S. (2018) *Chapter 3 - Univariate Ensemble Postprocessing. Statistical Postprocessing of Ensemble Forecasts*. doi:10.1016/B978-0-12-812372-0.00003-0.

Xiaoli, R. *et al.* (2021) *Deep Learning-Based Weather Prediction: A Survey | Elsevier Enhanced Reader*. doi:10.1016/j.bdr.2020.100178.

Python Code

Steps to access the forecast data - saved as Urls in AWS

In [2]:
```
!pip install s3fs
import fsspec
import s3fs
```

In [3]:
```
BUCKET_NAME = 'mo-exeter-collab'
AWS_KEY_ID = 'AKIAXZUQFG2DOUPXUI5T'
AWS_SECRET = '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'

STORAGE_OPTIONS = {
    'key': AWS_KEY_ID,
    'secret': AWS_SECRET,
}
```

Information about the data: In the dataset, we have 1.5m temperature data, which were collected from forecast runs in January 2019.

These data were collected from the Met Office UK regional ensemble model (MOGREPS-UK, or enukx here) and converted into images in order to make the data easier to work with (without requiring the use of specialist tools).

The model has a new forecast run started every hour (24 "cycles" per day) - forecasting out to ~5 days ahead. (For MOGREPS-UK, each cycle of data has lead times out to T+126 hours).

The temperature are outputted from the model at every hour.

Conversion to images The process for converting the data was as follows:

For each ensemble member, in each forecast output (file), we calculate the minimum and maximum values in the original data array. We rescale the data such that the minimum value is 0, and the maximum is 255. We convert the rescaled data array to a png image. The old minimum and maximum are recorded in a json file, so as to allow us to reverse the rescaling process. This process loses some of the precision of the underlying forecast data – we need to keep this in mind when creating models with these data. If this is likely to be an issue, we should instead use the original forecast data.

(NB: This process relies on the fact that each cycle of MOGREPS-UK produces 3 ensemble members, and so we are able to encode each member as a different colour channel of the images.)

In [4]:
```
import datetime
import dateutil

# this is the data that is available in the S3 bucket
KNOWN_MODELS = ['enukx']
KNOWN_DIAGNOSTICS = [
    'temperature_at_screen_level',
]
START_CYCLE = '20190101T0000Z'
END_CYCLE = '20190131T2300Z'
```

The URLs of the data contain information about the diagnostic and time information (in ISO 8601 format) – below are functions for converting from datetime.datetime and datetime.timedelta objects into compliant string representations.

In [5]:
```
def timedelta_to_duration_str(td):
    minutes, seconds = divmod(td.seconds, 60)
    hours, minutes = divmod(minutes, 60)
    hours += td.days*24
    assert seconds == 0
    return f'PT{hours:04}H{minutes:02}M'

def datetime_to_iso_str(dt):
    return dt.strftime('%Y%m%dT%H%MZ')
```

There are 3 types of time that we care about:

validity time = the time that the forecast is valid for cycle time = the time when the model was initialized lead time = how far ahead we are forecasting (the difference between the two) The calc_cycle_validity_lead_times function is used for ensuring that we have all three times, given exactly two of them:

In [6]:
```
def calc_cycle_validity_lead_times(cycle_time=None, validity_time=None, lead_time=None
    num_inputs = sum(map(lambda x: 1 if x else 0, [cycle_time, validity_time, lead_ti
    if num_inputs != 2:
        # raise RuntimeError('Expected 2 of cycle_time, validity_time and lead_time')
        pass
    if not validity_time:
        validity_time = cycle_time + lead_time
    elif not cycle_time:
        cycle_time = validity_time - lead_time
    elif not lead_time:
        lead_time = validity_time - cycle_time
    return cycle_time, validity_time, lead_time
```

Accessing image data directly:

Using get_image_url we are able to construct the URL to the image directly:

In [7]:
```
def get_image_url(model, diagnostic,
                  cycle_time=None, validity_time=None, lead_time=None):
    # model and diagnostic are strings
    # validity_time (datetime.datetime)
    # lead_time (datetime.timedelta)

    assert model in KNOWN_MODELS
    assert diagnostic in KNOWN_DIAGNOSTICS

    # determine all times
    cycle_time, validity_time, lead_time = calc_cycle_validity_lead_times(
        cycle_time, validity_time, lead_time
    )

    # convert all to strings
    cycle_time = datetime_to_iso_str(cycle_time)
    validity_time = datetime_to_iso_str(validity_time)
    lead_time = timedelta_to_duration_str(lead_time)

    filepath = f'fcst_images/level_1/{model}/{cycle_time}/{validity_time}-{lead_time}-
    return f's3://{BUCKET_NAME}/{filepath}'
```

Function to read from S3 URL:

In [8]:
```
def read_from_url(url, **storage_options):
    # use fsspec to read file into memory
    with fsspec.open(url, 'rb', **storage_options) as of:
        data = of.read()
    return data
```

Rescaling the data:

When the data were converted to image format, they were rescaled in such a way that the minimum value was set to 0 (black), and the maximum value was set to 255 (white). The original values were saved in a json file which is stored for each cycle of the data.

get_json_url function returns the URL for this json file.

In [9]:
```
def get_json_url(model, diagnostic, cycle_time):
    cycle_time = datetime_to_iso_str(cycle_time)
    filepath = f'fcst_images/level_1/{model}/{cycle_time}/{diagnostic}.json'
    return f's3://{BUCKET_NAME}/{filepath}'
```

In [10]:
```
from PIL import Image
from io import BytesIO

import numpy as np
```

Creating forecast - Observation pairs

Keeping lead time fixed at 12hours. For cycle start time of 01-01-2019 at 12:00noon, corresponding truth value is at 01-01-2019 at 12:00am

There are a total of 24 time steps each day for 30 days = 720

plus 12 time steps on Jan 1st (which started at noon) = 732 total pairs

In [11]:
```
# URL Pairs for forecast and observations

import pandas as pd

#creating datetime object containing all my validity times
validity_times = list(pd.date_range(start = '20190101T1200Z', end = '20190131T2300Z',
forecast_urls = []
truth_urls = []
url_list2 = []

for time_steps in validity_times:
    time_steps = time_steps.to_pydatetime() # convert timestamp to datetime https://w
    image_urlforecast = get_image_url(model='enukx', # generating the forecast Urls
                        diagnostic='temperature_at_screen_level',
                        validity_time=time_steps,
                        lead_time=datetime.timedelta(hours=12))
    image_urltruth = get_image_url(model='enukx', # generating the truth Urls
                        diagnostic='temperature_at_screen_level',
                        validity_time=time_steps,
                        lead_time=datetime.timedelta(hours=0))
    forecast_urls.append(image_urlforecast)
    truth_urls.append(image_urltruth)
    url_list2.append([image_urlforecast, image_urltruth])
```

In [12]:
```
from pandas import DataFrame
df_image_urls = DataFrame (url_list2,columns=['forecast_urls','truth_urls'])
```

In [13]:
```
#SPLITTING THE DATASET FIRST INTO TRAINING AND TEST

training_data = df_image_urls[:int(df_image_urls.shape[0]*0.803279)]
testing_data = df_image_urls[int(df_image_urls.shape[0]*0.803279):]

print(f"No. of training samples: {training_data.shape[0]}") #588 for batch size 12 =
print(f"No. of testing samples: {testing_data.shape[0]}") #144 for batch size 12 = 12
```

```
No. of training samples: 588
No. of testing samples: 144
```

In [14]:
```
#FURTHER SPLITTING TRAINING DATA INTO TRAINNING AND VALIDATION SETS
training_data_split = training_data[:int(training_data.shape[0]*0.77551)]
validation_data = training_data[int(training_data.shape[0]*0.77551):]

print(f"No. of validation samples: {training_data_split.shape[0]}") #455 for batch si
print(f"No. of validation samples: {validation_data.shape[0]}") #133 for batch size 1
```

```
No. of validation samples: 455
No. of validation samples: 133
```

In [15]:
```
#this dataset is used this for calculating mean, SD, Min and Max as required for stan

forecast_for_minmax = training_data['forecast_urls'].values.tolist() #data for foreca
truth_for_minmax = training_data['truth_urls'].values.tolist() #data for observations

# The following datasets are used in the model
forecast_urls_train = training_data_split['forecast_urls'].values.tolist() #training
truth_urls_train = training_data_split['truth_urls'].values.tolist() #training data f

forecast_urls_vald = validation_data['forecast_urls'].values.tolist() #validation dat
truth_urls_vald = validation_data['truth_urls'].values.tolist() #validation data for

forecast_urls_test = testing_data['forecast_urls'].values.tolist() #testing data for
truth_urls_test = testing_data['truth_urls'].values.tolist() #testing data for observ
```

Sequence created using tensorflow.keras.utils import Sequence. Used this sequence to get the data back in original scale. Also used this to calculate mean and SD for standardizing the data and Minimum and Maximum values once the data was in the original scale

In [17]:
```
# SEQUENCE CREATED FOR GETTING IMAGE BACK IN ORIGINAL SCALE WITH IMAGE IN RANGE OF TH
# use this sequence to get the data back in original scale. Also use to calculate mea
import warnings
warnings.filterwarnings('ignore')
from tensorflow.keras.utils import Sequence
import json
import os
class ImgSequence(Sequence):
    def __init__(self, forecast_urls_list, truth_urls_list, batch_size=2, dim=(120,12
        self.forecasts, self.truths = forecast_urls_list, truth_urls_list
        self.batch_size = batch_size
        self.dim = dim
        self.n_channels = n_channels
        self.shuffle = shuffle

    def __len__(self):
        return int(np.ceil(len(self.forecasts) / float(self.batch_size)))
# Counts the number of possible batches that can be made from the total available dat
# Rule of thumb, number of datasets % batch_size = 0, so that every sample is seen
# for 732 sets of data and batch size 10, __len__() will return 73 i.e. 73 unique bat
# each batch is assigned an index from 0 to 72 and this is used in the __getitem__() 

    def __getitem__(self, idx):
        batch_x = self.forecasts[idx * self.batch_size:(idx + 1) * self.batch_size]
        batch_y = self.truths[idx * self.batch_size:(idx + 1) * self.batch_size]

        X = self.__data_generation(batch_x)
        y = self.__data_generation(batch_y)
        return X, y

#__getitem__() is called when model needs a new batch of data. For requested batch wi
# method will extract datasets from forecast_urls_list that have the index 121 * 10 t

    @staticmethod
    def __read_scaled_image(image_url):
        # read image data
        file_data = read_from_url(image_url, **STORAGE_OPTIONS)
        img = Image.open(BytesIO(file_data))
        arr = np.array(img)

        # construct json url
        # split image_url into dirname and basename
        dirname, basename = os.path.split(image_url)
        # remove extension from basename
        basename, _ = os.path.splitext(basename)
        # diagnostic is the component in the string before the extension
        diagnostic = basename.split('-')[-1]
        json_url = f'{dirname}/{diagnostic}.json'

        # read scaling data from json file
        json_data = read_from_url(json_url, **STORAGE_OPTIONS)
        json_data = json.loads(json_data)
        image_name = os.path.basename(image_url)
        image_data_bounds = json_data[image_name]
        # convert values (lists) to numpy arrays
        image_data_bounds = {k: np.array(v)
                             for k, v in image_data_bounds.items()}

        # rescale image
        arr_scaled = arr.astype(np.float32)
        # rescale in range 0 to 1
        arr_scaled = (arr - arr.min(axis=(0,1))) / (arr.max(axis=(0,1)) - arr.min(axi
        # put on range of output data
        arr_scaled = (arr_scaled * (image_data_bounds['max'] - image_data_bounds['min
                      + image_data_bounds['min'])
        arr_scaled = arr_scaled[532:652, 473:593]
        return arr_scaled

    def __data_generation(self, batch_urls):

          # Creates an empty placeholder array that will be populated with data tha
          #X = np.empty((self.batch_size, *self.dim, self.n_channels), dtype=np.flo
          #X = np.empty((self.batch_size, *self.dim, self.n_channels), dtype=np.uin
          X = np.empty((self.batch_size, *self.dim, self.n_channels))
          #outfile = TemporaryFile()

          for i, url in enumerate(batch_urls):
              X[i, ] = self.__read_scaled_image(url)
              #np.savez(outfile, X) # https://numpy.org/doc/stable/reference/genera


          return X#, y
```

Calculating the Mean and Standard Deviation of the Data

This is done using the sequence above. Calculated the values for the training data set for all of UK as well as the training dataset for the Wales cut-out

In [ ]:
```
#calculating the mean and SD for standardizing data
import math

batch_Xsums = []
batch_ysums = []
batch_Xnum_elements = []
batch_ynum_elements = []

for X,y in ImgSequence(forecast_urls_train, truth_urls_train, batch_size=12):
    batch_Xsums.append(X.sum())
    batch_ysums.append(y.sum())

    batch_Xnum_elements.append(np.shape(X)[0] * np.shape(X)[1] * np.shape(X)[2] * np.
    batch_ynum_elements.append(np.shape(X)[0] * np.shape(y)[1] * np.shape(y)[2] * np.

Xmean = np.sum(batch_Xsums)/np.sum(batch_Xnum_elements)
ymean = np.sum(batch_ysums)/np.sum(batch_ynum_elements)

batch_Xsubmean = []
batch_ysubmean = []
for X, y in ImgSequence(test_urls, test_urls2, batch_size=2):
    batch_Xsubmean.append(np.sum((X - Xmean)**2))
    batch_ysubmean.append(np.sum((y - ymean)**2))

Xstdev = math.sqrt(np.sum(batch_Xsubmean) / np.sum(batch_Xnum_images))
ystdev = math.sqrt(np.sum(batch_ysubmean) / np.sum(batch_ynum_images))

Xmean, ymean, Xstdev, ystdev
#(279.2222502788791, 279.14195960409154, 4.864251560679521, 4.915170531644156) values
#(278.40751329128517, 278.54763033652455, 13.506688794260096, 13.471262058577377) val
```

In [ ]:
```
Xmin_batch3=[]
ymin_batch3=[]
Xmax_batch3=[]
ymax_batch3=[]
for X, y in ImgSequence(forecast_for_minmax, truth_for_minmax, batch_size=12):
    Xmin_batch3.append(X.min())
    ymin_batch3.append(y.min())
    Xmax_batch3.append(X.max())
    ymax_batch3.append(y.max())

Xmin3 = min(Xmin_batch3)
ymin3 = min(ymin_batch3)
Xmax3 = max(Xmax_batch3)
ymax3 = max(ymax_batch3)

Xmin3, ymin3, Xmax3, ymax3

#(231.34646606445312, 230.29400634765625, 289.9052734375, 288.4958190917969) min max
#(260.3576956954657, 260.93533456839765, 285.6530643238741, 285.9685453527114) min ma
```

Out[ ]:
```
(260.3576956954657, 260.93533456839765, 285.6530643238741, 285.9685453527114)
```

Saved the values obtained from calculating mean and SD for standardizing. Not necessary to calculate again. The computation took approximately 4hours.

(Xmean, ymean, Xstdev, ystdev) values for all of UK

(279.2222502788791, 279.14195960409154, 4.864251560679521, 4.915170531644156)

(Xmean, ymean, Xstdev, ystdev) values for Wales cut-out

(278.40751329128517, 278.54763033652455, 13.506688794260096, 13.471262058577377)

Also saved the values obtained from calculating min and max values

(Xmin3, ymin3, Xmax3, ymax3) values for all of UK (231.34646606445312, 230.29400634765625, 289.9052734375, 288.4958190917969)

(Xmin3, ymin3, Xmax3, ymax3) values for Wales cut-out (260.3576956954657, 260.93533456839765, 285.6530643238741, 285.9685453527114)

Xmax3-Xmin3, ymax3-ymin3 (25.295368628408426, 25.033210784313724)

In [ ]:
```
Xmax3-Xmin3, ymax3-ymin3
#(25.295368628408426, 25.033210784313724)
```

Out[ ]:
```
(25.295368628408426, 25.033210784313724)
```

In [44]:
```
# 4TH WORKING SEQUENCE WITH SCALED DATA (STANDARDIZED) THIS IS FINAL SEQUENCE WHICH I
from tensorflow.keras.utils import Sequence
import json
import os
class ImgSequence4(Sequence):
    def __init__(self, forecast_urls_list, truth_urls_list, batch_size=12, dim=(120,1
                 #dataset_name=None, rescale_Xmean=279.2222502788791,rescale_Xstdev=4
                 #dataset_name=None, rescale_Xmean=278.40751329128517,rescale_Xstdev=
                 dataset_name=None, rescale_Xmean=260.3576956954657,rescale_Xstdev=25
        self.forecasts, self.truths = forecast_urls_list, truth_urls_list
        self.batch_size = batch_size
        self.dim = dim
        self.n_channels = n_channels
        self.shuffle = shuffle
        #self.on_epoch_end()

        self.dataset_name = dataset_name
        self.rescale_Xmean, self.rescale_Xstdev = rescale_Xmean, rescale_Xstdev
        self.rescale_ymean, self.rescale_ystdev = rescale_ymean, rescale_ystdev

   # def on_epoch_end(self):
      #self.indexes = np.arange(len(self.list_IDs))
      #if self.shuffle == True:
         #np.random.shuffle(self.indexes)

    def __len__(self):
        return int(np.ceil(len(self.forecasts) / float(self.batch_size)))
# Counts the number of possible batches that can be made from the total available dat
# Rule of thumb, number of datasets % batch_size should equal = 0, so that every samp
# for 732 sets of data and batch size 12, __len__() will return 61 i.e. 61 unique bat
# each batch is assigned an index from 0 to 61 and this is used in the __getitem__() 

    def __getitem__(self, idx):
        batch_x = self.forecasts[idx * self.batch_size:(idx + 1) * self.batch_size]
        batch_y = self.truths[idx * self.batch_size:(idx + 1) * self.batch_size]

        X = self.__data_generation(batch_x)
        y = self.__data_generation(batch_y)

        if self.rescale_Xmean is not None and self.rescale_Xstdev is not None:
            X -= self.rescale_Xmean
            X /= self.rescale_Xstdev
        if self.rescale_ymean is not None and self.rescale_ystdev is not None:
            y -= self.rescale_ymean
            y /= self.rescale_ystdev

        return X.mean(axis=3), y.mean(axis=3)

#__getitem__() is called when model needs a new batch of data. For requested batch wi
# method will extract datasets from forecast_urls_list or truth_urls_list that have t

    @staticmethod
    def __read_scaled_image(image_url):
        # read image data
        file_data = read_from_url(image_url, **STORAGE_OPTIONS)
        img = Image.open(BytesIO(file_data))
        arr = np.array(img)

        # construct json url
        # split image_url into dirname and basename
        dirname, basename = os.path.split(image_url)
        # remove extension from basename
        basename, _ = os.path.splitext(basename)
        # diagnostic is the component in the string before the extension
        diagnostic = basename.split('-')[-1]
        json_url = f'{dirname}/{diagnostic}.json'

        # read scaling data from json file
        json_data = read_from_url(json_url, **STORAGE_OPTIONS)
        json_data = json.loads(json_data)
        image_name = os.path.basename(image_url)
        image_data_bounds = json_data[image_name]
        # convert values (lists) to numpy arrays

        image_data_bounds = {k: np.array(v)
                             for k, v in image_data_bounds.items()}

        # rescale image
        arr_scaled = arr.astype(np.float32)
        # rescale in range 0 to 1
        arr_scaled = (arr_scaled - arr_scaled.min(axis=(0,1))) / (arr_scaled.max(axis=
        # put on range of output data
        arr_scaled = (arr_scaled * (image_data_bounds['max'] - image_data_bounds['min
                      + image_data_bounds['min'])
        arr_scaled = arr_scaled[532:652, 473:593]
        return arr_scaled

    def __data_generation(self, batch_urls):

        # Creates an empty placeholder array that will be populated with data that is
        #X = np.empty((self.batch_size, *self.dim, self.n_channels), dtype=np.float32
        #X = np.empty((self.batch_size, *self.dim, self.n_channels), dtype=np.uint8)
        X = np.empty((self.batch_size, *self.dim, self.n_channels))

        for i, url in enumerate(batch_urls):
            X[i, ] = self.__read_scaled_image(url)
            #np.savez(outfile, X) # https://numpy.org/doc/stable/reference/generated/


        return X#, y
```

Unable to load this data into memory therefore, using a subset of data used to develop plots and other analysis for example to check that the sequence is working well. ETC

In [19]:
```
#Use the test (FORECAST) URls FOR PLOTS AND ANALYSIS and quick tests on ImgSequence
test_urls = forecast_urls_train[0:6]
test_img_forecast = []
test_arr_forecast = []
for url in test_urls:
    x_file_data = read_from_url(url, **STORAGE_OPTIONS)
    x_img = Image.open(BytesIO(x_file_data))
    test_img_forecast.append(x_img)
    x_arr = np.array(x_img)
    test_arr_forecast.append(x_arr)

#Use the test (TRUTH) URls FOR PLOTS AND ANALYSIS and quick tests on ImgSequence
test_urls2 = truth_urls_train[0:]
test_img_truth = []
test_arr_truth = []
for url in test_urls2:
    y_file_data = read_from_url(url, **STORAGE_OPTIONS)
    y_img = Image.open(BytesIO(y_file_data))
    test_img_truth.append(y_img)
    y_arr = np.array(y_img)
    test_arr_truth.append(y_arr)
```

In [20]:
```
# using the sample of forecast-observation pairs to test the sequence is working. Use
img_test = ImgSequence(test_urls, test_urls2, batch_size=2, dim=(120,120), n_channels
```

In [46]:
```
# using the sample of forecast-observation pairs to test the sequence is working. Use
img_test4 = ImgSequence4(test_urls, test_urls2, batch_size=2, dim=(120,120), n_channe
```

In [47]:
```
#Extracting the first batch from the output of the Imgsequence
batch0 = img_test4[0]
batch_x, batch_y = batch0 #extracting the forecast and observations

#forecast-observation pairs for plots and mse calculations (shows difference between
bx0 = batch_x[0]
by0 = batch_y[0]
bx1 = batch_x[1]
by1 = batch_y[1]
```

In [23]:
```
# code to calculate mse value between randomly selected forecast-observation pairs
def mse(imageA, imageB):
    err = np.sum((imageA.astype("float32") - imageB.astype("float32")) ** 2)
    err /= float(imageA.shape[0] * imageA.shape[1])
    return err

mse(bx0,by0), mse(bx1,by1) #(2.2230031956784138, 2.214571749411322) #(1.381059050794
#calculated mse values confirm images in forecast-observation pairs are indeed differ
```

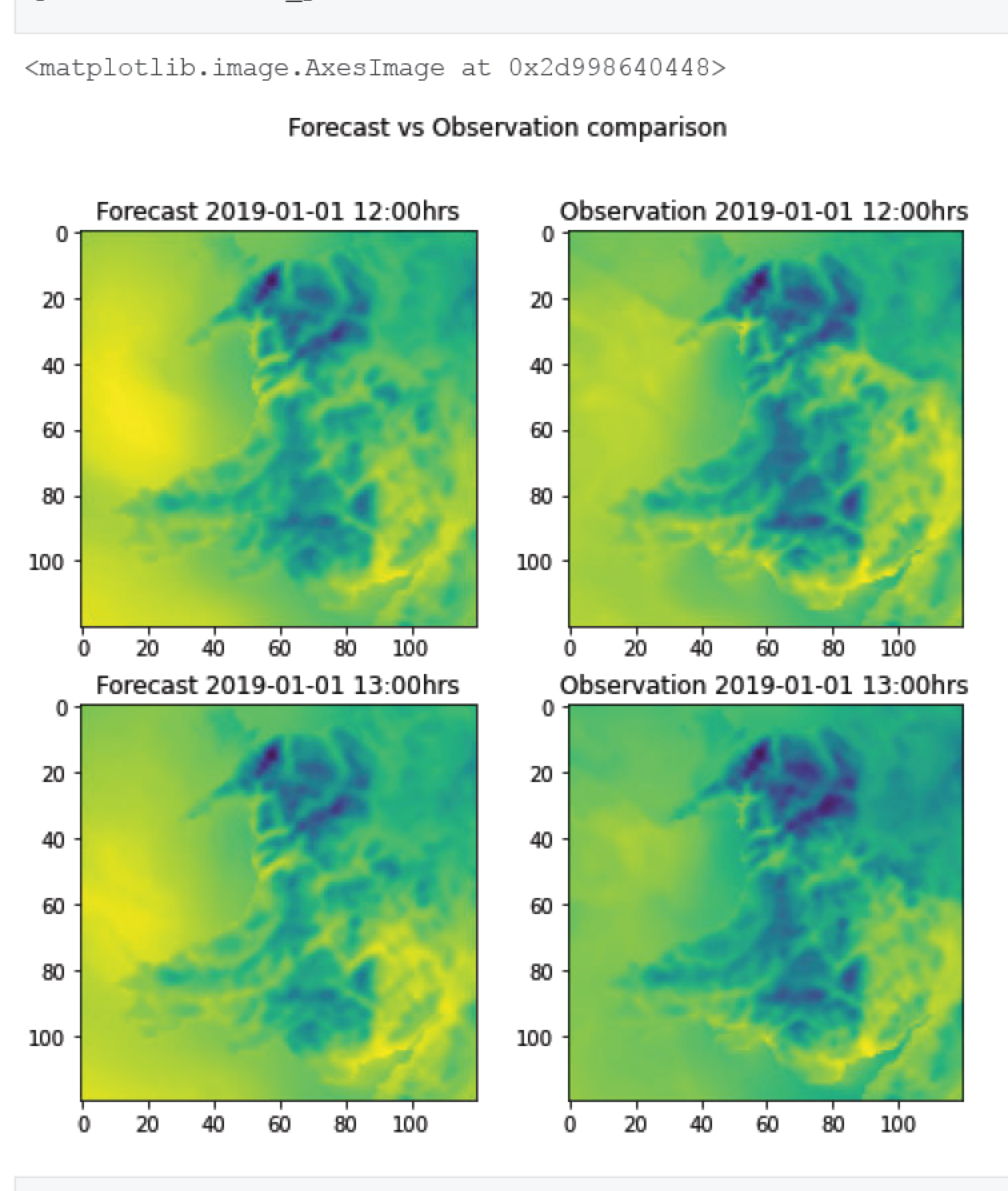
Out[23]:
```
(0.0003879759046766493, 0.0002566655476888021)
```

A plot comparing the forecast-observation pairs

Plotted this after scaling the data using ImgSequence4

In [48]:
```
import matplotlib.pyplot as plt
fig = plt.figure(figsize=(8, 8))
fig.suptitle("Forecast vs Observation comparison")
plt.subplot(2,2,1)
plt.title('Forecast 2019-01-01 12:00hrs')
plt.imshow(batch_x[0])
plt.subplot(2,2,2)
plt.title('Observation 2019-01-01 12:00hrs')
plt.imshow(batch_y[0])
plt.subplot(2,2,3)
plt.title('Forecast 2019-01-01 13:00hrs')
plt.imshow(batch_x[1])
plt.subplot(2,2,4)
plt.title('Observation 2019-01-01 13:00hrs')
plt.imshow(batch_y[1])
```

Out[48]:
```
<matplotlib.image.AxesImage at 0x2d998640448>
```

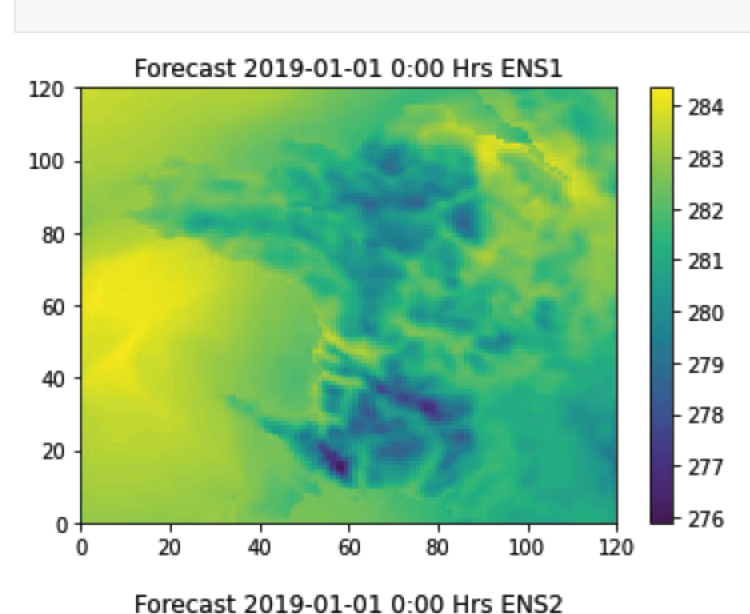


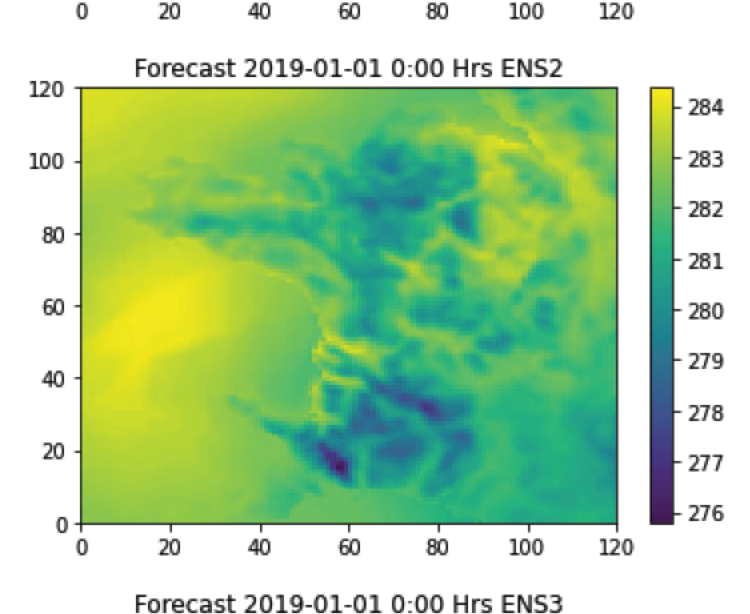
In [25]:
```
#Extracting the first batch from the output of the Imgsequence
batch0 = img_test[0]
batch_x, batch_y = batch0 #extracting the forecast and observations

#forecast-observation pairs for plots and mse calculations (shows difference between
bx0 = batch_x[0]
by0 = batch_y[0]
bx1 = batch_x[1]
by1 = batch_y[1]
```

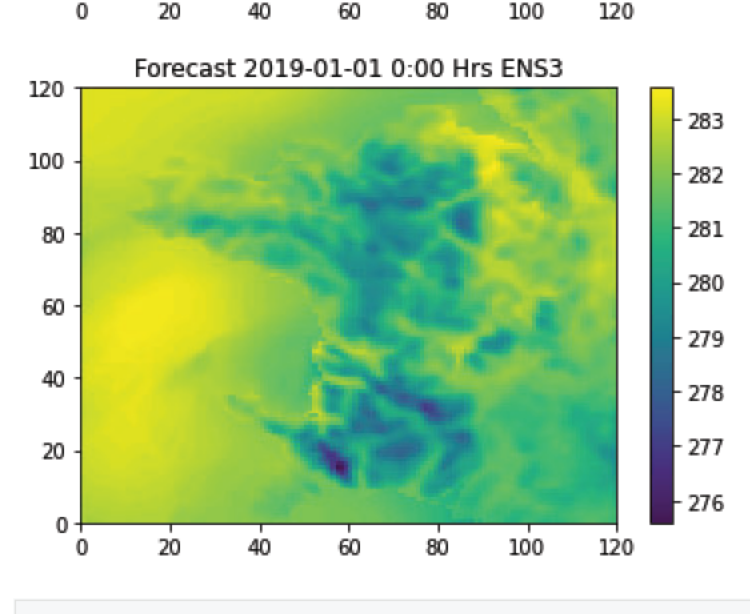
A cut out of Wales showing temperatures forecast for three ensemble members

Plotted this after putting the image back in the original temperature values scale with ImgSequence

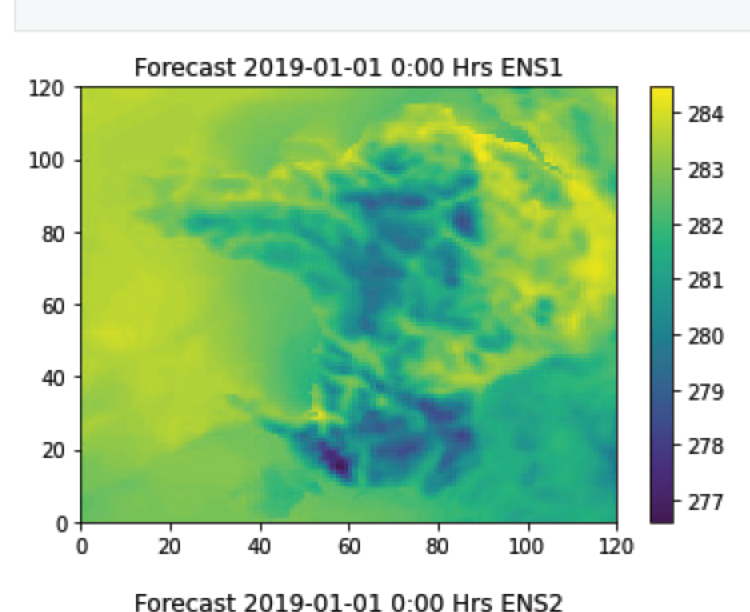
In [26]:
```
#FORECAST
import matplotlib.pyplot as plt
for batch_index in range(3):
    plt.pcolormesh(bx0[:,:,batch_index])
    caption = 'Forecast 2019-01-01 0:00 Hrs ENS'+str(batch_index+1)
    plt.title(caption)
    plt.colorbar()
    plt.show()
```

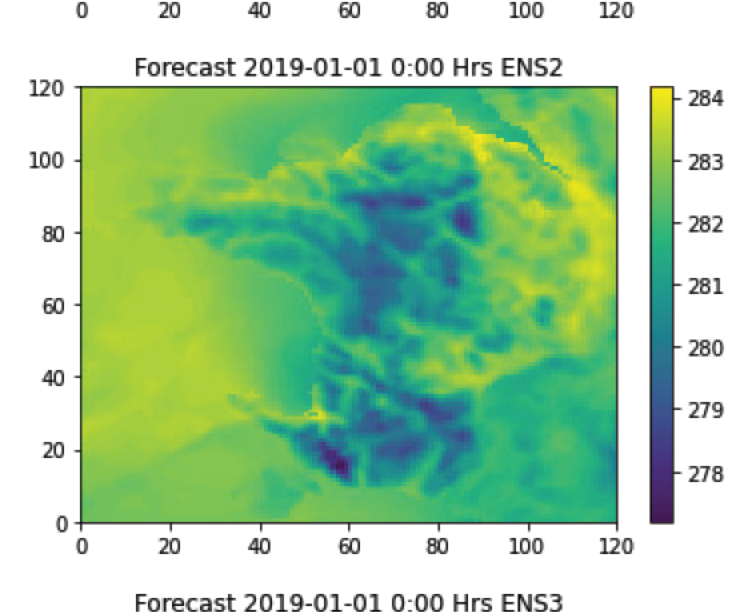


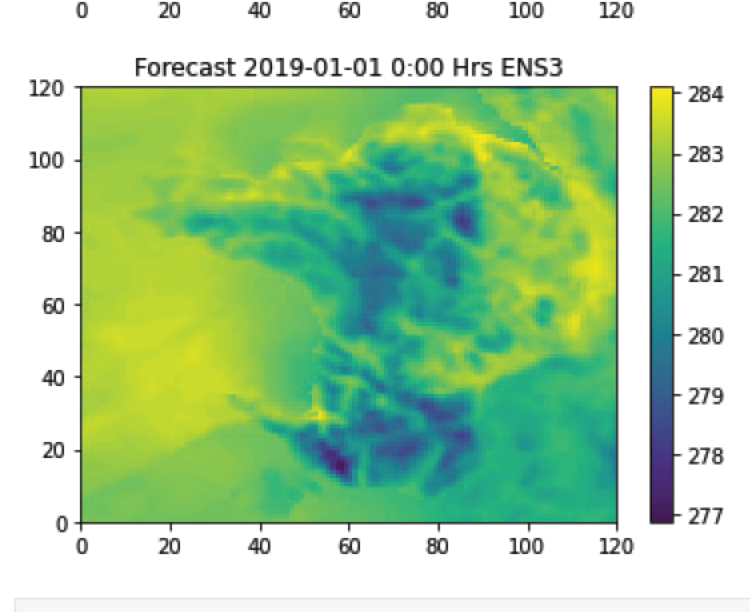




In [27]:
```
#TRUTH
#wales = batch_y[0][532:652, 473:593]
for batch_index in range(3):
    plt.pcolormesh(by0[:,:,batch_index])
    caption = 'Forecast 2019-01-01 0:00 Hrs ENS'+str(batch_index+1)
    plt.title(caption)
    plt.colorbar()
    plt.show()
```







In [28]:
```
# BIAS CORRECTION MODEL
import warnings
warnings.filterwarnings('ignore')
import tensorflow as tf
from tensorflow.keras import models
from tensorflow.keras import layers
from tensorflow.keras.models import Model
from tensorflow.keras.models import Sequential
from tensorflow.keras import optimizers
from tensorflow.keras.layers import Conv2D, MaxPooling2D

input = layers.Input(shape=(120, 120, 1))

# downsample
x = layers.Conv2D(32, (2, 2),activation="relu", padding="same",kernel_regularizer="l2
x = layers.MaxPooling2D((2, 2), padding="same")(x)
x = layers.Conv2D(32, (2, 2), activation="relu", padding="same",kernel_regularizer="l
x = layers.MaxPooling2D((2, 2), padding="same")(x)

# upsample
x = layers.Conv2DTranspose(32, (2, 2), strides=2, activation="relu", padding="same",k
x = layers.Conv2DTranspose(32, (2, 2), strides=2, activation="relu", padding="same",k
x = layers.Conv2D(1, (2, 2), padding="same",kernel_regularizer="l2",kernel_initialize

# final model
model = Model(input, x)
model.compile(optimizer=optimizers.Adam(lr=0.001, clipvalue=1.0),
    loss='mse',
    metrics=['mae']
)
model.summary()
```

```
WARNING:tensorflow:From C:\Users\vatur\anaconda3\envs\MTHM504-tf-2.0\lib\site-packages
\tensorflow\python\ops\init_ops.py:1251: calling VarianceScaling.__init__ (from tensor
flow.python.ops.init_ops) with dtype is deprecated and will be removed in a future ver
sion.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the constru
ctor
Model: "model"
```

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 120, 120, 1)]	0
conv2d (Conv2D)	(None, 120, 120, 32)	160
max_pooling2d (MaxPooling2D)	(None, 60, 60, 32)	0
conv2d_1 (Conv2D)	(None, 60, 60, 32)	4128
max_pooling2d_1 (MaxPooling2	(None, 30, 30, 32)	0
conv2d_transpose (Conv2DTran	(None, 60, 60, 32)	4128
conv2d_transpose_1 (Conv2DTr	(None, 120, 120, 32)	4128
conv2d_2 (Conv2D)	(None, 120, 120, 1)	129

```
Total params: 12,673
Trainable params: 12,673
Non-trainable params: 0
```

In [ ]:
```
# code for saving the weights
checkpoint_path = "training_1/cp-{epoch:04d}.ckpt"
checkpoint_dir = os.path.dirname(checkpoint_path)
batch_size=512
cp_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath=checkpoint_path,
    verbose=1,
    save_weights_only=True,
    save_freq=5*batch_size)
```

In [ ]:
```
#SET UP MODEL

#TRAINING with Wales cut-out with standardized data (z-scaling)
training_sequence = ImgSequence4(forecast_urls_train, truth_urls_train, batch_size=12,
validation_sequence = ImgSequence4(forecast_urls_vald, truth_urls_vald, batch_size=12,

#TRAINING with Wales cut-out with normalized data (min-max scaling)
training_sequence = ImgSequence4(forecast_urls_train, truth_urls_train, batch_size=12,
validation_sequence = ImgSequence4(forecast_urls_vald, truth_urls_vald, batch_size=12,

#TRAINING with full UK image representation
#training_sequence = ImgSequence4(forecast_urls_train, truth_urls_train, batch_size=1
#validation_sequence = ImgSequence4(forecast_urls_vald, truth_urls_vald, batch_size=1
```

In [ ]:
```
# CONFIRM SEQUENCE SHAPE
for data_batch, labels_batch in training_sequence:
    print('data batch shape:', data_batch.shape)
    print('labels batch shape:', labels_batch.shape)
    break
#data batch shape: (12, 120, 120)
#labels batch shape: (12, 120, 120)
```

In [ ]:
```
#RUN MODEL
#history = model.fit(training_sequence, epochs=20, callbacks=[cp_callback],validation
                    # workers=4)

history =model2.fit(training_sequence, epochs=5, validation_data=validation_sequence)
```

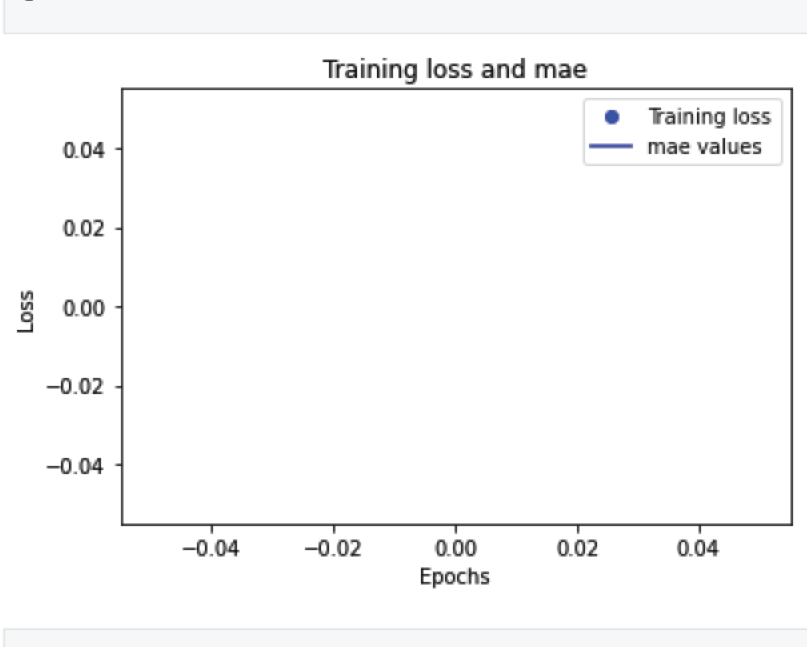
```
Epoch 1/5
 1/38 [..............................] - ETA: 46:37 - loss: 1.9536 - mae: 0.8071
```

In [ ]:
```
history_dict = history.history
history_dict.keys()
```

Out[ ]:
```
dict_keys(['loss', 'mae'])
```

In [ ]:
```
import matplotlib.pyplot as plt

history_dict = history.history
loss_values = history_dict['loss']
mae_values = history_dict['mae']
epochs = np.arange(1, len(loss_values) + 1)
plt.plot(epochs, loss_values, 'bo', label='Training loss')
plt.plot(epochs, mae_values, 'b', label='mae values')
plt.title('Training loss and mae')
plt.xlabel('Epochs')
```

```
plt.ylabel('Loss')
plt.legend()

plt.show()
```



In [ ]:

```
loss_values, mae_values

weights= model2.get_weights()
weights

np.max(weights[0])
```

Out[ ]: nan

Regression Model Tests

As an experiment, tried a regression model on the data:

In [ ]:

```
#Extracting regression model.
reg_test_urls = forecast_urls_train
arr_scaled_forecast = []
for url in reg_test_urls:
    reg_x_file_data = read_from_url(url, **STORAGE_OPTIONS)
    reg_x_img = Image.open(BytesIO(reg_x_file_data))
    reg_x_arr = np.array(reg_x_img)

    dirnameX, basenameX = os.path.split(url)
    basenameX, _ = os.path.splitext(basenameX)
    diagnosticX = basenameX.split('-')[-1]
    json_urlX = f'{dirnameX}/{diagnosticX}.json'
    json_dataX = read_from_url(json_urlX, **STORAGE_OPTIONS)
    json_dataX = json.loads(json_dataX)
    image_nameX = os.path.basename(url)
    image_data_boundsX = json_dataX[image_nameX]
    image_data_boundsX = {k: np.array(v)
                          for k, v in image_data_boundsX.items()}

    arr_scaledX = reg_x_arr.astype(np.float32)
    arr_scaledX = (arr_scaledX - arr_scaledX.min(axis=(0,1))) / (arr_scaledX.max(axis=
    arr_scaledX = (arr_scaledX * (image_data_boundsX['max'] - image_data_boundsX['min
                  + image_data_boundsX['min'])
    arr_scaledX = arr_scaledX[532:652, 473:593]
    arr_scaledX = arr_scaledX.mean(axis=2)
    arr_scaled_forecast.append(arr_scaledX)

reg_test_urls2 = truth_urls_train
arr_scaled_truth = []
for url in reg_test_urls2:
    reg_y_file_data = read_from_url(url, **STORAGE_OPTIONS)
    reg_y_img = Image.open(BytesIO(reg_y_file_data))
    reg_y_arr = np.array(reg_y_img)

    dirnameY, basenameY = os.path.split(url)
    basenameY, _ = os.path.splitext(basenameY)
    diagnosticY = basenameY.split('-')[-1]
    json_urlY = f'{dirnameY}/{diagnosticY}.json'
    json_dataY = read_from_url(json_urlY, **STORAGE_OPTIONS)
    json_dataY = json.loads(json_dataY)
    image_nameY = os.path.basename(url)
    image_data_boundsY = json_dataY[image_nameY]
    image_data_boundsY = {k: np.array(v)
                          for k, v in image_data_boundsY.items()}

    arr_scaledY = reg_y_arr.astype(np.float32)
    arr_scaledY = (arr_scaledY - arr_scaledY.min(axis=(0,1))) / (arr_scaledY.max(axis=
    arr_scaledY = (arr_scaledY * (image_data_boundsY['max'] - image_data_boundsY['min
                  + image_data_boundsY['min'])
    arr_scaledY = arr_scaledY[532:652, 473:593]
    arr_scaledY = arr_scaledY.mean(axis=2)
    arr_scaled_truth.append(arr_scaledY)
```

In [ ]:

```
#change to array
arr_scaled_forecast = np.array(arr_scaled_forecast)
arr_scaled_truth = np.array(arr_scaled_truth)
```

In [ ]:

```
#reshape data
nx_samples = len(arr_scaled_forecast)
dataX = arr_scaled_forecast.reshape((nx_samples, -1))

ny_samples = len(arr_scaled_truth)
dataY = arr_scaled_truth.reshape((ny_samples, -1))
```

In [ ]:

```
#fit regression model
regressor.fit(dataX, dataY)
```

In [ ]:

```
#data for testing
#get x_test data on which to test the regressor (linear regression model)
x_test_urls = forecast_urls_vald
x_test_list = []
for url in x_test_urls:
    xtest_file_data = read_from_url(url, **STORAGE_OPTIONS)
    xtest_img = Image.open(BytesIO(xtest_file_data))
    xtest_arr = np.array(xtest_img)

    dirnameX, basenameX = os.path.split(url)
    basenameX, _ = os.path.splitext(basenameX)
    diagnosticX = basenameX.split('-')[-1]
    json_urlX = f'{dirnameX}/{diagnosticX}.json'
    json_dataX = read_from_url(json_urlX, **STORAGE_OPTIONS)
    json_dataX = json.loads(json_dataX)
    image_nameX = os.path.basename(url)
    image_data_boundsX = json_dataX[image_nameX]
    image_data_boundsX = {k: np.array(v)
                          for k, v in image_data_boundsX.items()}

    arr_scaledX = xtest_arr.astype(np.float32)
    arr_scaledX = (arr_scaledX - arr_scaledX.min(axis=(0,1))) / (arr_scaledX.max(axis=
    arr_scaledX = (arr_scaledX * (image_data_boundsX['max'] - image_data_boundsX['min
                  + image_data_boundsX['min'])
    arr_scaledX = arr_scaledX[532:652, 473:593]
    arr_scaledX = arr_scaledX.mean(axis=2)
    x_test_list.append(arr_scaledX)


#get y_test
y_test_urls = truth_urls_vald
y_test_list = []
for url in y_test_urls:
    ytest_file_data = read_from_url(url, **STORAGE_OPTIONS)
    ytest_img = Image.open(BytesIO(ytest_file_data))
    ytest_arr = np.array(ytest_img)

    dirnameY, basenameY = os.path.split(url)
    basenameY, _ = os.path.splitext(basenameY)
    diagnosticY = basenameY.split('-')[-1]
    json_urlY = f'{dirnameY}/{diagnosticY}.json'
    json_dataY = read_from_url(json_urlY, **STORAGE_OPTIONS)
    json_dataY = json.loads(json_dataY)
    image_nameY = os.path.basename(url)
    image_data_boundsY = json_dataY[image_nameY]
    image_data_boundsY = {k: np.array(v)
                          for k, v in image_data_boundsY.items()}

    arr_scaledY = ytest_arr.astype(np.float32)
    arr_scaledY = (arr_scaledY - arr_scaledY.min(axis=(0,1))) / (arr_scaledY.max(axis=
    arr_scaledY = (arr_scaledY * (image_data_boundsY['max'] - image_data_boundsY['min
                  + image_data_boundsY['min'])
    arr_scaledY = arr_scaledY[532:652, 473:593]
    arr_scaledY = arr_scaledY.mean(axis=2)
    y_test_list.append(arr_scaledY)
```

In [ ]:

```
#change to array
x_test_arr = np.array(x_test_list)
y_test_arr = np.array(y_test_list)

#reshape data
nx_samples = len(x_test_arr)
x_test = x_test_arr.reshape((nx_samples, -1))

ny_samples = len(y_test_arr)
y_test = y_test_arr.reshape((ny_samples, -1))
```

In [ ]:

```
y_pred = regressor.predict(x_test)

print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(y_test, y_pred)))

#Mean Absolute Error: 0.7703156630266417
#Mean Squared Error: 1.014896073056825
#Root Mean Squared Error: 1.00742050458427
```