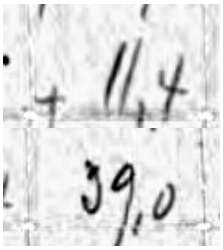
To whom it might concern,

My name is Valter Fallenius. Over the past two years, I have been employed as an Analytics Engineer at Echo State. This blog post is intended to provide insights into my experiences with the development and evaluation of ML-models. Although my exposure to building ML-models at Echo State has been confined to smaller projects, I have acquired valuable skills such as data wrangling, modeling large volumes of data, identifying business processes, and developing production-grade code. These skills are highly beneficial for any ML-engineer.

At Echo State, I have done some projects for clients involving different clustering techniques, for example I reduced a dataset of 100,000 investors to find leads for my client to a much more manageable size of 300 leads with PCA and k-means techniques. I have also learnt how to efficiently access data through VPN-tunnels, REST APIs and web scraping.



In addition to these skills, I have experience building ML-models from scratch. During my master's I worked as a computer vision engineer where I was tasked with classifying handwritten weather-journals from 100 years ago for studying climate change. I did this by building CNN-network that with the right confidence level could classify the handwritten notes with 99.9% accuracy and 93% recall which saved SMHI ~80 years in man-hours to manually enter these numbers.

However, where I learnt the most about Deep Learning, was when I wrote my master thesis at SMHI on Google AI's Deep Learning Model for forecasting rain, MetNet. The following section is a deep dive to what I did, what I learnt along the way and some of the struggles I faced during this challenge.

MetNet blogpost

Numerical Weather Prediction is known to be one of the most challenging problems in mathematics. Today it is done through brute-force approaches where some of the largest computers in the world are used to model the atmosphere and calculate a forecast of the weather, a single forecast can take upwards of 24h to generate. Machine Learning models have in recent years shown very promising results to not only improve the accuracy of the current state-of-the-art physical models, but also reduce the time it takes to generate a forecast to mere seconds. One such model that beat all physical models was released by Google in 2020, called "MetNet". During my master thesis, I aimed to recreate this model on a Swedish dataset.

My MetNet-model was based on an open source pytorch-model from OpenClimateFix that I had to reconfigure for my setup, I downsized the network and removed the satellite input data channels. I also rebuilt it to work with pytorch-lightning to parallelize the training of the model. The model consisted of three parts: a spatial downsampler in the form of a 2D-Convolutional Neural Network, a temporal encoder in the form of a Convolutional GRU network and finally an axial vision-Transformer network as a spatial aggregator.

I asked Jacob Bieker at OpenClimateFix to provide a description of my contribution to their MetNet-model:

"Valter's contributions helped debug and improve our public reimplementation of MetNet, fixing some important bugs that we missed and overall improving the implementation. I had insightful

and useful discussions with Valter on GitHub, and his project provided meaningful contributions to our codebase. We have, subsequently, used the improved MetNet for experimentation as part of our solar forecasting work."- Jacob Bieker, Senior Machine Learning Engineer at OpenClimateFix.

Data discussion

I used a collection of data from 10 radar towers in Sweden supplied by SMHI that I converted to the correct unit for precipitation [mm/h]. The original model for MetNet was trained on both satellite and radar data, this dataset was publicly available but as I wrote my thesis under the Swedish Meteorological Institute (SMHI) we wanted to test the viability of ML-forecasts in Sweden. Because of how geostationary satellites work satellite-data is skewed in the Nordics. That is partly why we opted for using only radar data, but also to reduce the dimensionality of the project. We also used time-of-day and time-of-day encodings, as well as elevation, longitude, and latitude encodings.

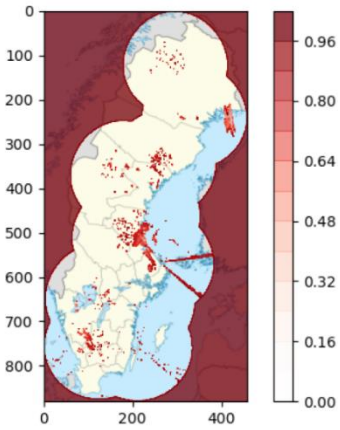


Figure 3.1. Radar data over Sweden (in dBZ) normalized to the range [0,1]. Note that, in this plot, the visible ranges of the radar and that values close to 1 means high precipitation and values of 1 mean "no data available". Before feeding the input to our network we transform all "no data available" to 0 instead.

The first problem I encountered with the data was to load it into my model during training since it was a considerable size 500GB I couldn't load everything at once to the RAM. I solved this by only loading smaller batches of samples from memory at a time, this slowed down training a little bit but not significantly. Given the large dataset, I chose to use a stationary input frame instead of a moving one to simplify the task. However, in hindsight, this decision may have been a mistake. By keeping the input area for elevation and longitude/latitude static, the model couldn't fully learn the impact of these variables.

Another problem with the data was that since rain is a very scarce phenomenon, having a model train on data where 99.9% of pixels are without rain would lead to it always guessing for "no rain", therefore we had to filter out a large chunk of the training data by only selecting samples with at least some rain. We used 15% of the rainiest periods for training.

A final struggle with radar-only data is that radars cannot detect humidity, which is a precursor to rain, which is why satellite data remains vital for a model like this. Within the scope of my master thesis, I couldn't explore the impact of adding satellite data to my model.

Since rain clouds travel with the windspeed, to ensure predictability, the input area must be bigger than the prediction area. Here is a sketch of how the radar data was formatted:

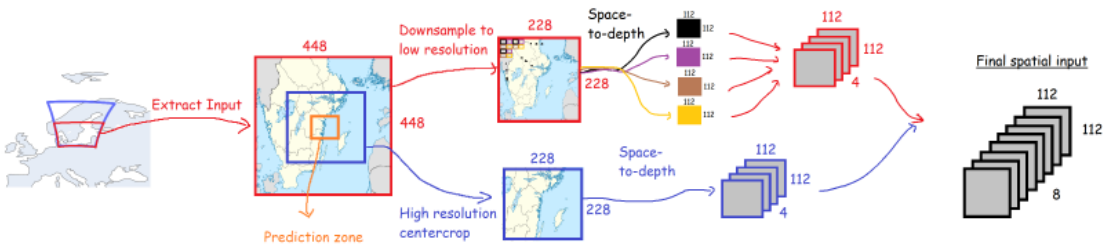


Figure 3.2. Input patch for radar data and illustration of space-to-depth transformation.

Evaluation of the model

The ground truth was 128 one-hot encoded vectors describing the rate of precipitation. During training the loss-function we used *categorical cross entropy* as this is known to work well with one-hot encoded data. However, I was curious to see how it would perform if you introduced some weight in the loss function for “how bad” a guess was. With categorical cross entropy, a guess of 0 and 88 mm/h is weighted as equally bad when the correct value is 90 mm/h. I didn’t get to explore this within the scope of my thesis, maybe other loss functions could speed up training.

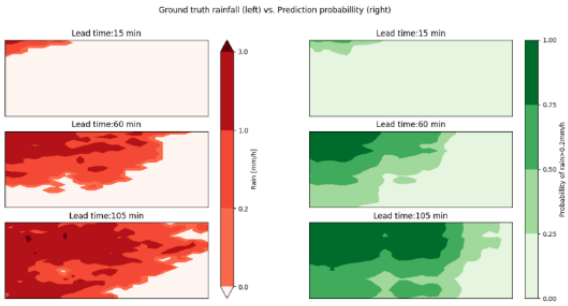


Figure 5.6. Example of successful prediction with model. As can be seen for all three lead times: the model outputs a similar shape as the ground truth.

For evaluation we used an *F1-score* which addresses the problem of having a tail-heavy dataset by being directly proportional to the number of true positives, in comparison with a more conventional accuracy score a guess of “no rain” would most of the time give a high accuracy. To convert the score to a binary representation, I summed the bins above and below a certain threshold. During validation and testing the model was evaluated with the F1-score at different rates of precipitation (light, medium and heavy rainfall), and the evaluation was done on pixel-by-pixel basis over all predictions.

$$\mathrm{F1} = \frac{TP}{TP + 0.5(FP + FN)}$$

Comparison with some baseline models

To compare my model, I used a simple persistence benchmark that guesses the rain will continue as it does now. While my results were less than what I had hoped for, I still managed to beat the persistence benchmark with my model. Considering the scope of my master thesis compared to Google AI and the resources they had available, I was happy to at least beat the persistence benchmark. The model beat persistence benchmark for light rainfall for lead times >30min:

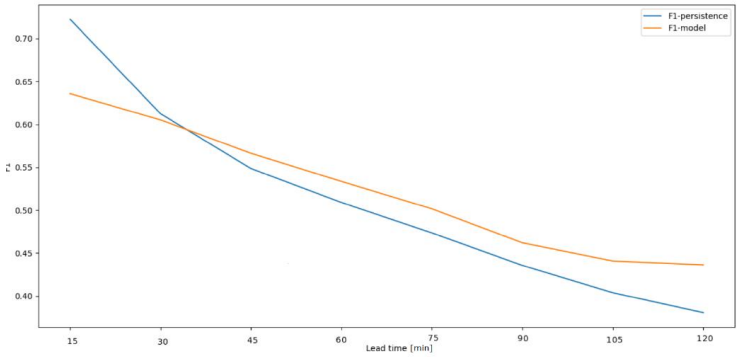


Figure 5.5. F1-score for light rainfall > 0.2mm/h for different lead times compared with persistence benchmark.

I also tried training a simpler model by omitting the third part of my Network (which was the cause for the slow training) the spatial transformer, I did this to study the effects of the spatial

transformer. We found that the spatial aggregator did what it was supposed to and differentiated the different lead times. Without the spatial aggregator, the network seemed to guess on a mean for all lead times.

My model was too large for the training data I supplied and started overfitting during the fine-tuning step, before reaching an acceptable F1-score. In hindsight I should've used a moving input-frame during training, which would have allowed a lot more data samples. I would also change the approach of how to build the model by increasing its size incrementally along with increasing the training-time, making sure to increase the amount of data so that it doesn't overfit.

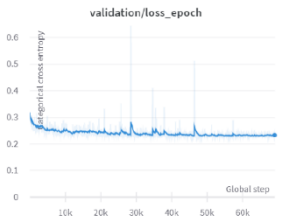


Figure 5.1. Smoothed validation loss during 1,000 epochs or ∼70 thousand global steps with a learning rate $\eta = 0.01$.

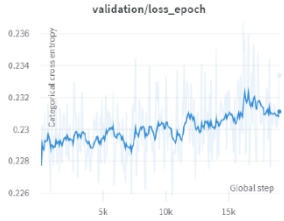


Figure 5.2. Continuation of validation loss for another ∼20 thousand global steps with a lower learning rate $\eta = 0.001$. Overfitting is observed.

Final words

As you can see, I do have some experience with working on ML-models even if my recent work has been limited in the amount of time put on ML-projects. My work on the MetNet-model was the most fun I have ever had in a 'work'-environment, and to be able to go back to this type of work would mean my complete dedication and commitment.

Thank you for reading!