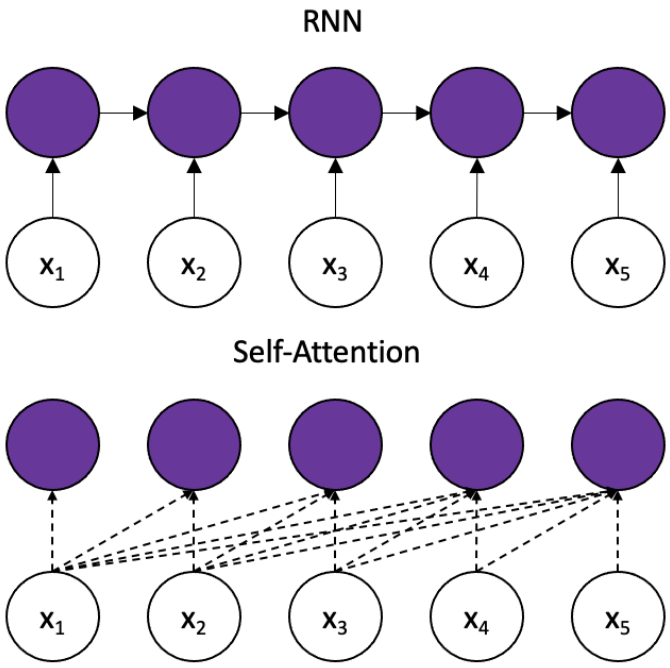
Published in MLearning.ai · Follow

Natasha Klingenbrunn · Follow
Feb 19, 2021 · 10 min read · Listen

Transformers for Time-series Forecasting

Code: https://github.com/nklingen/Transformer-Time-Series-Forecasting

This article will present a Transformer-decoder architecture for forecasting on a humidity time-series data-set provided by Woodsense . This project is a follow-up on a previous project that involved training an LSTM on the same data-set. The LSTM was seen to suffer from "short-term memory" over long sequences. Consequently, a Transformer will be used in this project, which outperforms the LSTM on the same data-set.



Inspired by the graphic in D2L[1]

Why use a Transformer ?

LSTMs process tokens sequentially, as shown above. This architecture maintains a hidden state that is updated with every new input token, representing the entire sequence it has seen. Theoretically, very important information can propagate over infinitely long sequences. However, in practice, this it not the case. Due to the vanishing gradient problem, the LSTM will eventually forget earlier tokens.

In comparison, Transformers retain direct connections to all previous timestamps, allowing information to propagate over much longer sequences. However, this entails a new challenge: the model will be directly connected to an exploding amount of input. In order to filter the important from the unimportant, Transformers use an algorithm called self-attention.

Self-Attention

In psychology, attention is the concentration of awareness on one stimuli while excluding others. Similarly, attention mechanisms are designed to focus on only the most important subsets of arbitrarily long sequences, that are relevant to accomplish a given task.[1]

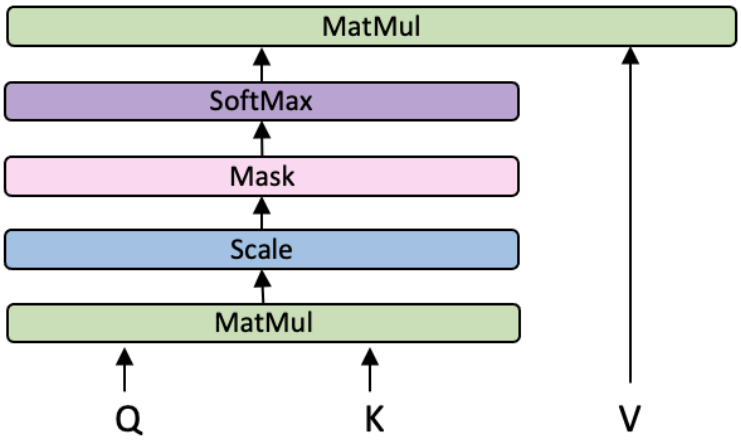
the current token.

Self-attention is explained in detail in Jay Alammars' Illustrated GP2 and Illustrated Transformer articles, with excellent visuals. Additionally, the original paper can be found here. A brief overview is provided below.

1. **Create Query, Key, and Value Vectors:**
   Each token generates an associated query, key, and value. The **query** of the current token is compared to the **keys** of all other tokens, in order to decide their relevance with respect to the current token. The **value** vector is the true representation of a given token, that is used in creating the new encoding. During training, the model gradually learns these three weight matrices that each token is multiplied by to generate its corresponding key, query, and value.

2. **Calculate the self-attention score:**
   The self-attention score is the measure of relevance between the current token and any other token that has previously been seen in the sequence. It is calculated by computing the dot product between the query vector of the current token with the key vector of the token being scored. High scores indicate high relevancy, while low scores indicate the opposite. The scores are then passed through a softmax function, such that they are all positive and add to 1. At this step, the self-attention score can be seen as a percentage of total focus that is given to a token in the sequence, in encoding the current token.

3. **Encoding the current token accordingly:**
   Words that had a high softmax value in the previous step should contribute more highly to the encoding of the current token, while words with a low score should contribute very little. To accomplish this, each value vector is multiplied by its softmax score. This keeps the original value intact, but scales the overall vector in line with its relative importance for the current token. Finally, all of the scaled values are summed together to produce the encoding of the current token.



Graphic of self-attention calculations, inspired by Attention is All You Need

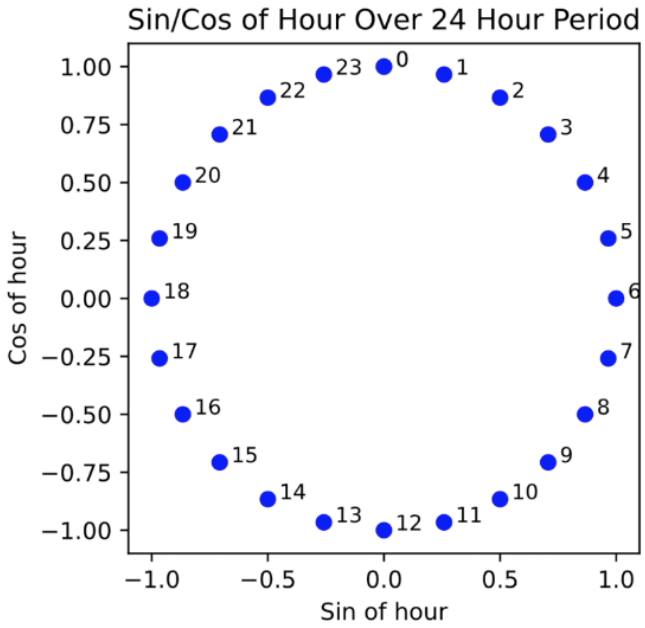
As the three steps consist of matrix operations, they can be optimised as follows:

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

The term in the denominator is used to stabilise the gradient. Other values are also possible.

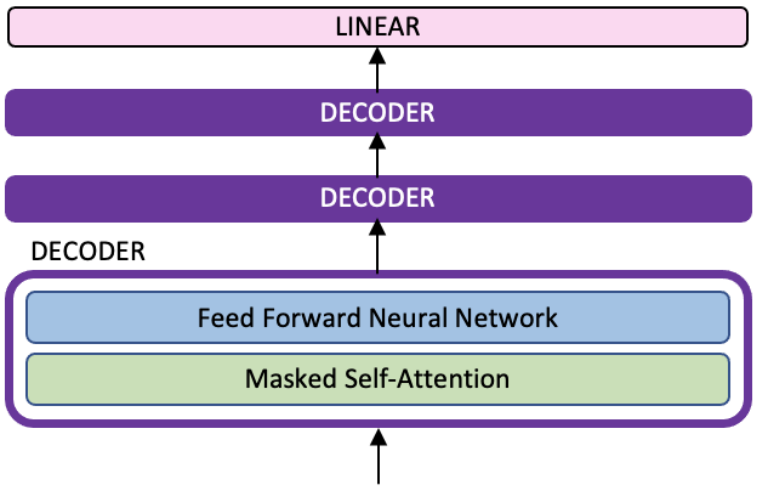
Positional Encoding

For this project, the positional encoding was achieved as follows: the timestamp was represented as three elements — hour, day, and month. To represent each datatype truthfully, each element was decomposed into a sine and cosine component. In this way, December and January are close spatially, just as the months occur close temporally. This same concept is applied to hours and days, such that all elements are represented cyclically.



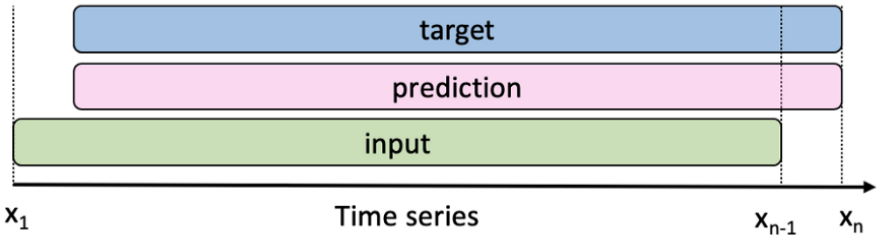
Implementing the Transformer-decoder

In a vanilla transformer, the decoder consists of the following three blocks: first a masked self-attention block, then an encoder-decoder block, and finally a Feed Forward block. The implementation in this paper draws inspiration from GPT2's implementation of a decoder-only Transformer, as seen in the figure below. Three of these modified decoder blocks are stacked on top of each other, passing the encoding from the previous block as the input to the subsequent block.

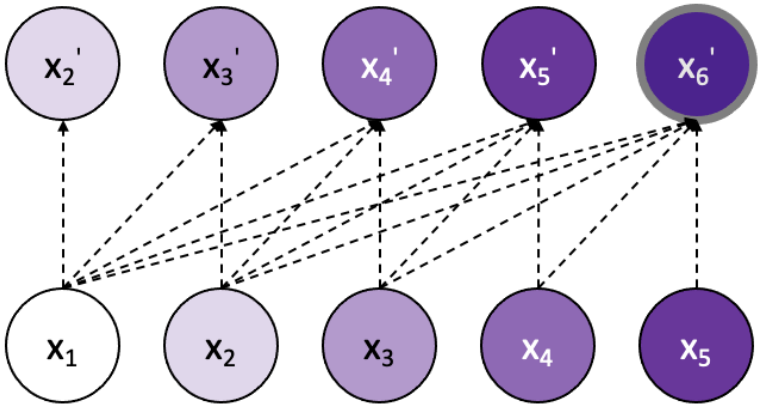


Transformer-decoder Architecture

The input to the transformer is a given time series (either univariate or multivariate), shown in green below. The target is then the



To represent this on a sequence of length 5, for the first input x1, the model will output its prediction for the upcoming token: x2'. Next, it is given the true x1 and x2, and predicts x3', and so forth. At each each new step, it receives all of the true inputs prior in the sequence, to predict the next step. Hereby, the output vector of the model will be the predicted tokens x2', x3', x4', x5', x6'. This is then compared to the true values x2, x3, x4, x5, x6 to train the model, wherein each output token contributes equally to the loss.



Graphic of how the inputs are attended to in calculating each output element.

Masked Self-Attention

To achieve the model demonstrated above, a mask must be used to ensure that the model only has access to the tokens coming prior in the sequence at each step. Concretely, before the softmax function is applied, all tokens coming after the token currently being attended to are masked, to prevent the model from cheating by looking ahead. When applying the soft-max, these future values will get 0% importance, consequently preventing any information from bleeding through.

	x_1	x_2	x_3	x_4	x_5	
1	0	-inf	-inf	-inf	-inf	→ x_2'
2	0	0	-inf	-inf	-inf	→ x_3'
3	0	0	0	-inf	-inf	→ x_4'
4	0	0	0	0	-inf	→ x_5'
5	0	0	0	0	0	→ x_6'

Graphic of the mask applied for an input of size 5. In Step 1, the model only has access to x1 when predicting x2'. Hereby, x1 will have 100% importance. In Step 2, it has access to x1 and x2 when predicting x'3. These inputs could now be allocated 40% and 60% respectively, as an example. At each of the following stages, the mask likewise ensures it is not possible for any token ahead of the current token to have any importance in the current calculation.

As a final note, the above description has indicated sequential steps in the calculations, for simplicity. In actuality, the calculations are done

as a student preparing for an exam with his teacher looking over his shoulder, the model learns quickly as its mistakes are instantly corrected. In other words, it never goes "too far off track" before it is corrected.[2]

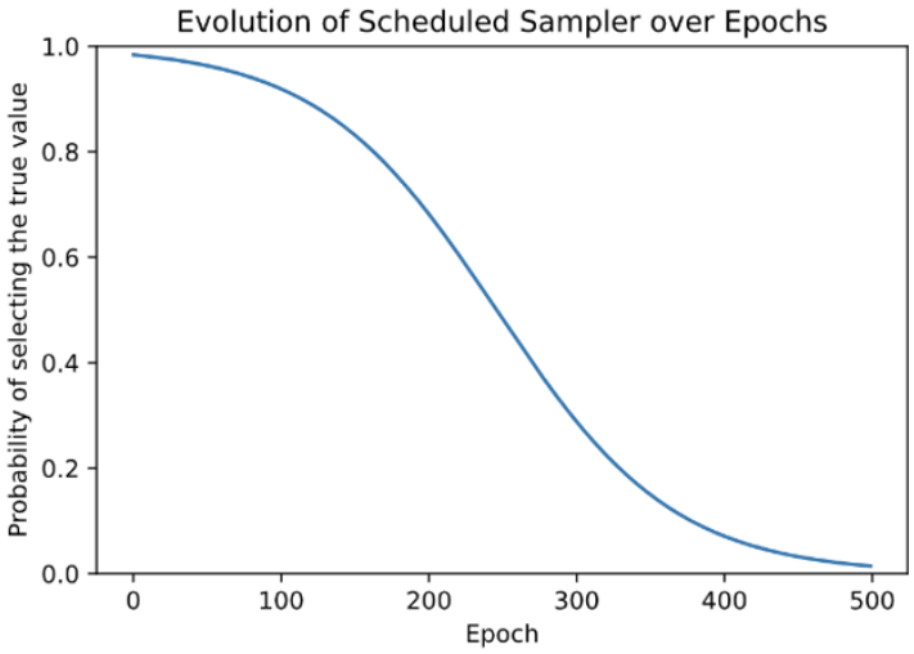
The drawback with teacher forcing is that at each new prediction, the model may make minor mistakes, but it will in any case receive the true value in the next step, meaning that these mistakes never contribute significantly to the loss. The model only has to learn how to predict one time step in advance.

However, during inference, the model now must predict longer sequences, and can no longer rely on the frequent corrections. In each step, the last prediction is appended as new input for the next step. Hereby, minor mistakes that were not critical during training quickly become amplified over longer sequences during inference.[3] In the previous analogy, this can be likened to the student going to the exam, having never taken a practice test without the aide of his teacher beside him.

A basic problem in teacher forcing emerges: training becomes a much different task than inference. To bridge this gap between training and inference, the model needs to slowly learn to correct its mistakes. The task of transitioning the model between training and inference thus becomes to gradually feed the model more of its predicted outputs, rather than the true values.

Moving too quickly towards inference in the early epochs — sampling too heavily from the model's predicted output — would result in random tokens, since the model has not yet had the opportunity to train, thereby causing a slow convergence. On the other hand, not moving quickly enough towards inference in the last epochs — sampling too little — will make the gap too large between training and inference, meaning that a model that performs well during training will drastically drop in performance during inference.

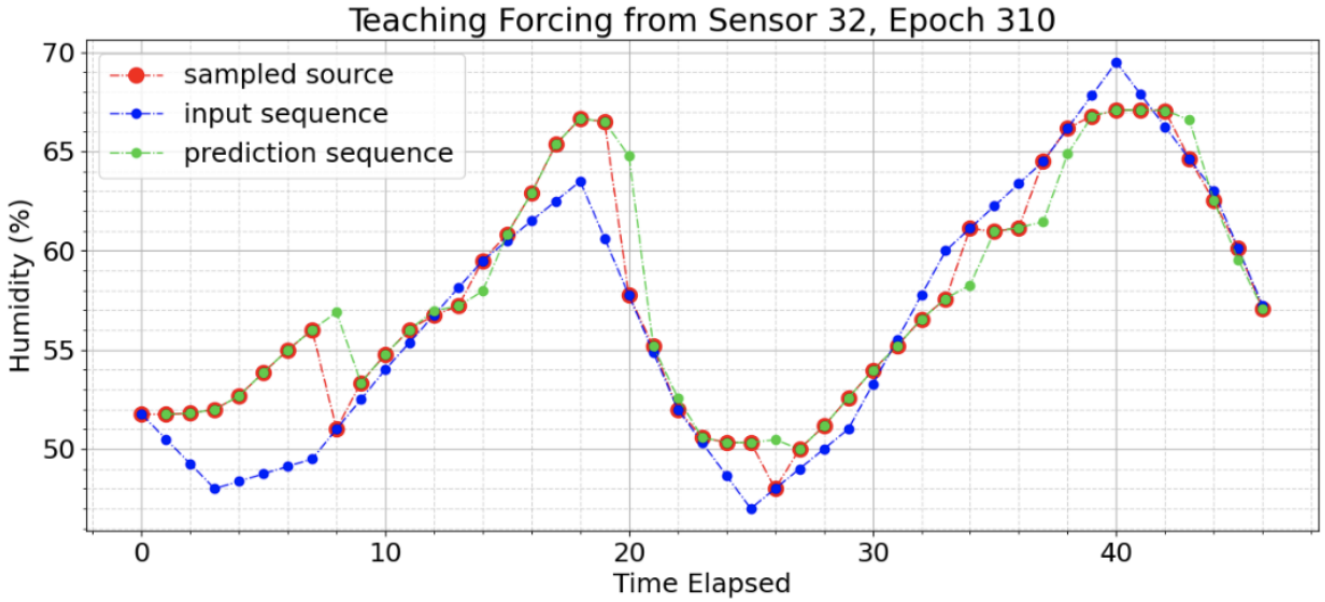
In order to gently bridge this gap, a sampling method is used, inspired by Bengio's "Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks". The sampling rate evolves over time, starting with a high probability of selecting the true value initially, as in classical teacher forcing, and gently converging towards sampling purely from the models output, to simulate the inference task. Different schedules are proposed in the paper, of which inverse sigmoid decay was chosen for this project. In these last epochs, when the model samples many of its own predictive values successively, the model ideally learns to prevent its small mistakes from accumulating, as it now is penalized much more highly.



A constant k is used to fine-tune the rate of decay in the inverse sigmoid decay graph.

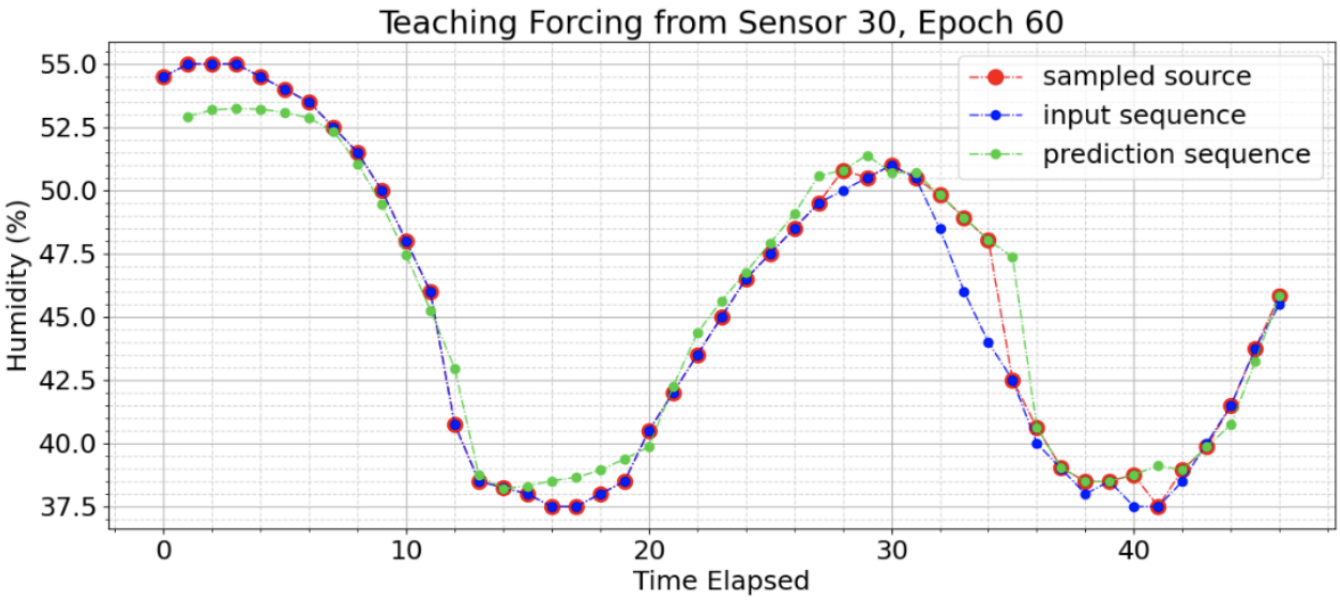
Applying this technique to the model yields results shown below during training at epoch 310 of 500. The blue dots show the true input.

For example, at timestamp 20, the model had moved significantly off track and predicted a humidity of 65%. The sampler selected the true output as the next input, and the model succeeded in correcting itself almost perfectly in the subsequent timestamp 21.

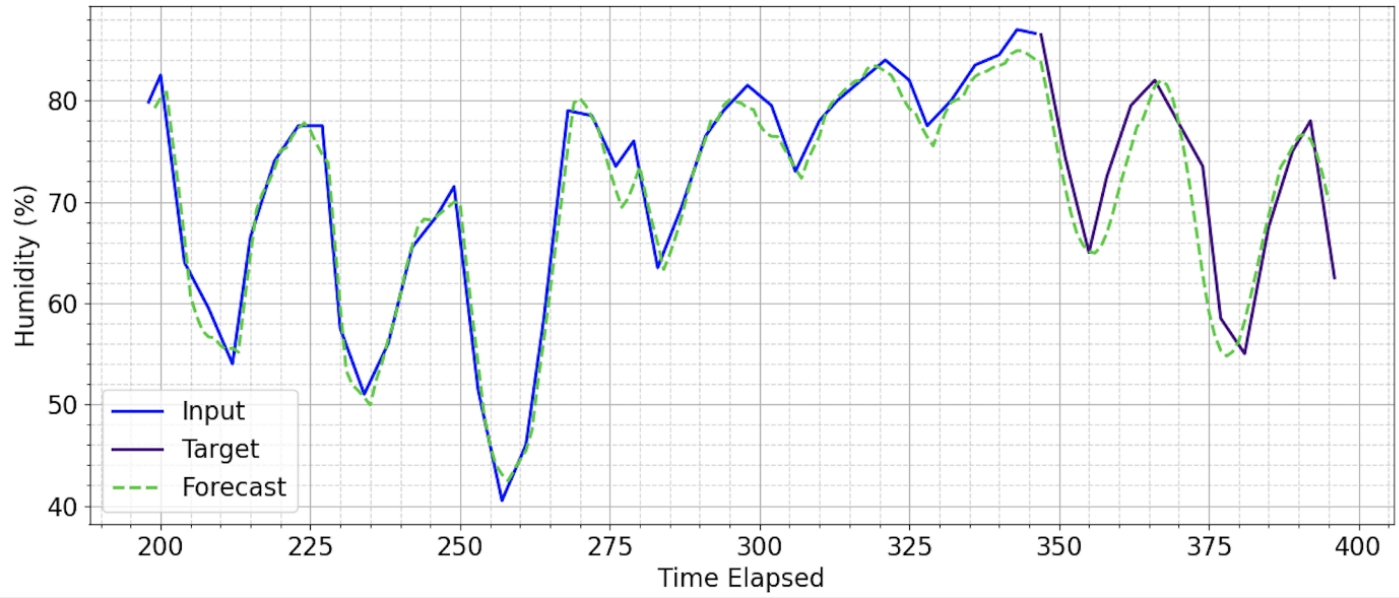


Scheduled sampler without theshold

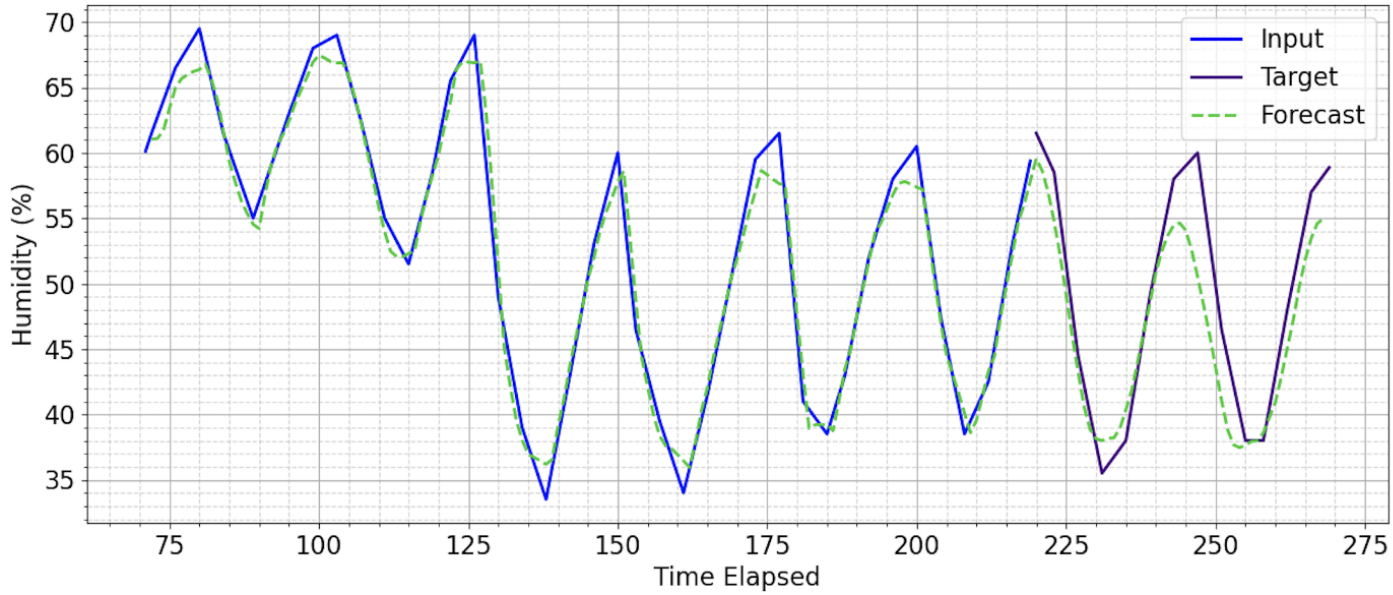
One important note is that the model performs poorly in the beginning of the sequence. This is intuitive, considering the model is predicting with very little knowledge of the specific sequence. Training with the scheduled sampler that kicks in early in the sequence was demonstrated to confuse the model, as it is penalized for poor predictions in cases where it does not have sufficient input to understand patterns in the data yet. To correct this, a threshold is set on the scheduled sampler, ensuring that only true values are used for the first 24 time-stamps, representing one full day of data, before the scheduled sampler kicks in at the normal rate demonstrated in the inverse sigmoid graph, correlated with the epoch.



Scheduled Sampler with threshold. The first 24 time stamps come only from the true input, regardless of the

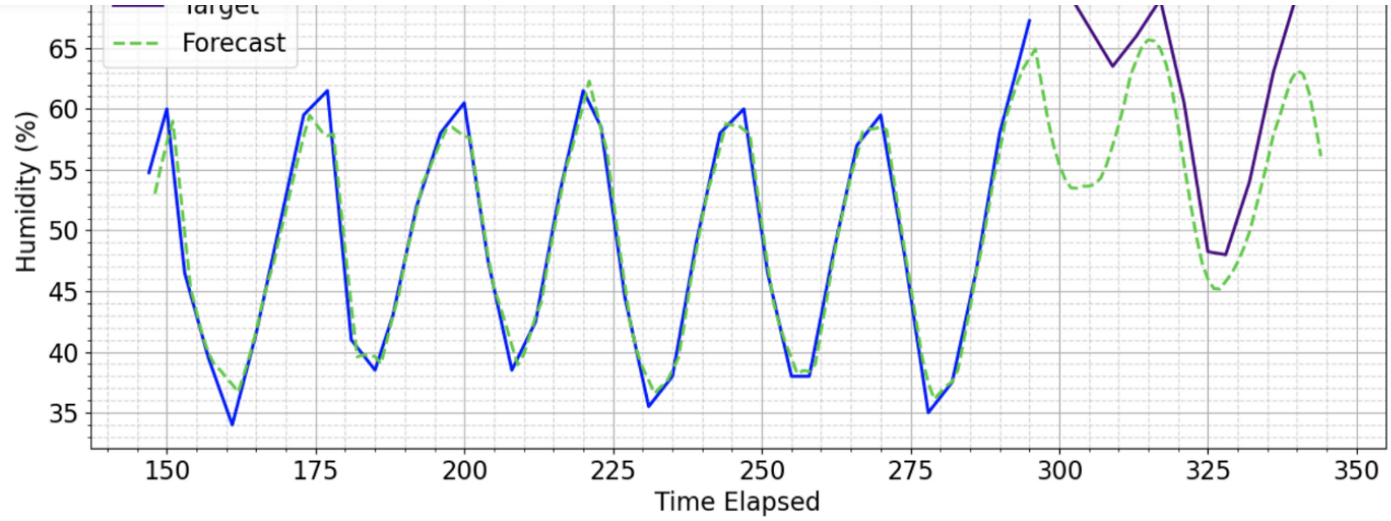


Prediction on an unseen sequence with a forecast window of 50, using a model trained for 500 epochs

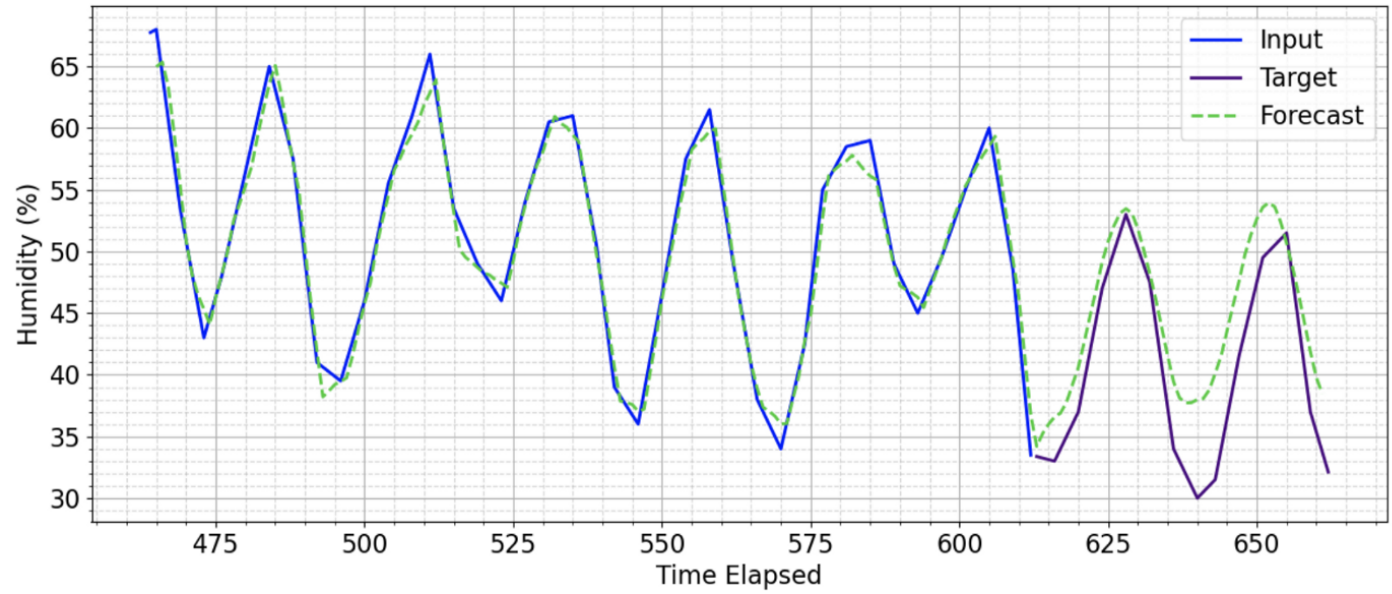


Prediction on an unseen sequence with a forecast window of 50, using a model trained for 500 epochs

Moreover, in the cases where it does shift off track over time, it produces convincing predictions that show a deeper understanding of the data.



Prediction on an unseen sequence with a forecast window of 50, using a model trained for 500 epochs



Prediction on an unseen sequence with a forecast window of 50, using a model trained for 500 epochs

The final model shows excellent results on the data-set, outperforming the previous implementation using an LSTM, which was seen to suffer from short-term memory. Moreover, the architecture allows for much more rapid training, as the computations during training are done concurrently rather than sequentially. It can be concluded that the Transformer architecture, which is traditionally applied to NLP problems, has large potential in time series forecasting.

[1] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola, Dive into Deep Learning, 2019 https://d2l.ai/chapter_attention-mechanisms/index.html

[2] Jason Brownlee,"What is teacher forcing for recurrent neural networks?," https://machinelearningmastery.com/teacher-forcing-for-recurrent-neural-networks/, Dec 2017

[3] Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer, "Scheduled sampling for sequence prediction with recurrent neural networks," CoRR, vol. abs/1506.03099, 2015. https://arxiv.org/pdf/1506.03099.pdf