TIME SERIES FORECASTING WITH DEEP LEARNING

Implementing Seq2Seq Models for Efficient Time Series Forecasting

Including attention, covariates, probabilistic forecasting, scheduled sampling, and more

Max Brenner · Follow

14 min read · Jul 18, 2023

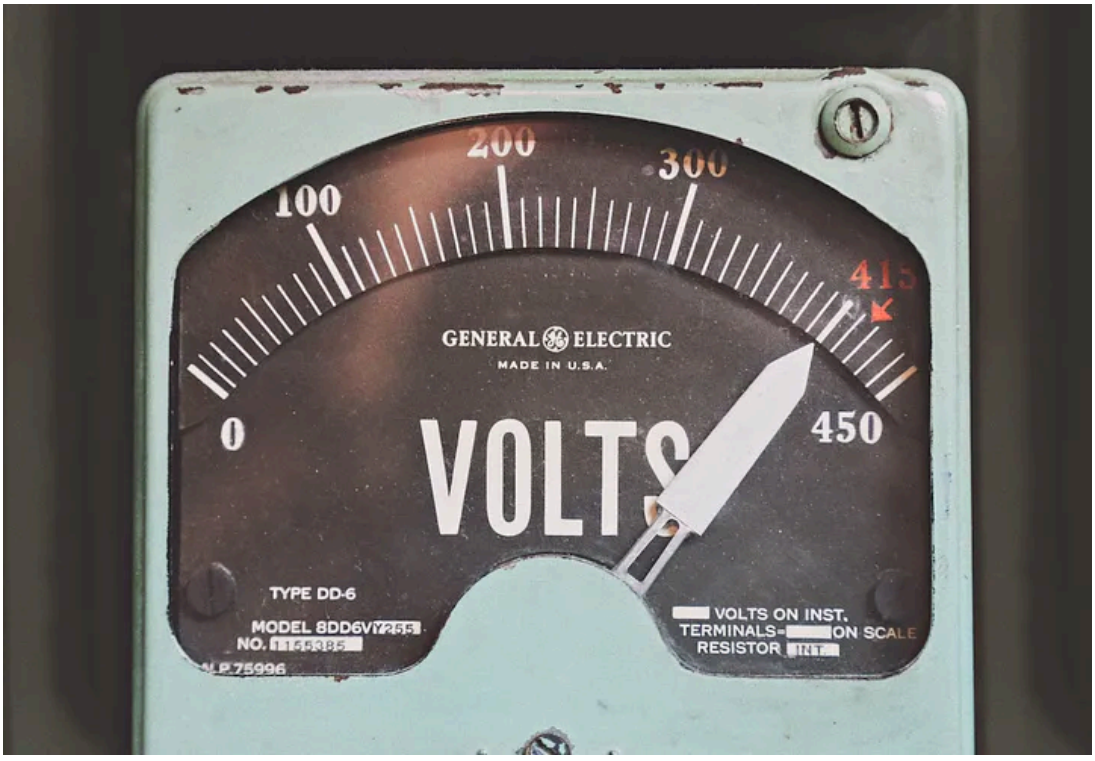


Photo by Thomas Kelley on Unsplash

Introduction to Time Series Forecasting with Deep Learning

In this article we will explore the design of deep learning sequence-to-sequence (seq2seq) models for time series forecasting. This is a popular structure for dealing with the notoriously difficult problem of accurate time series forecasting. Challenging properties of time series can include low autocorrelation, non-stationarity, heteroskedasticity, change points, low signal-to-noise ratio, etc.

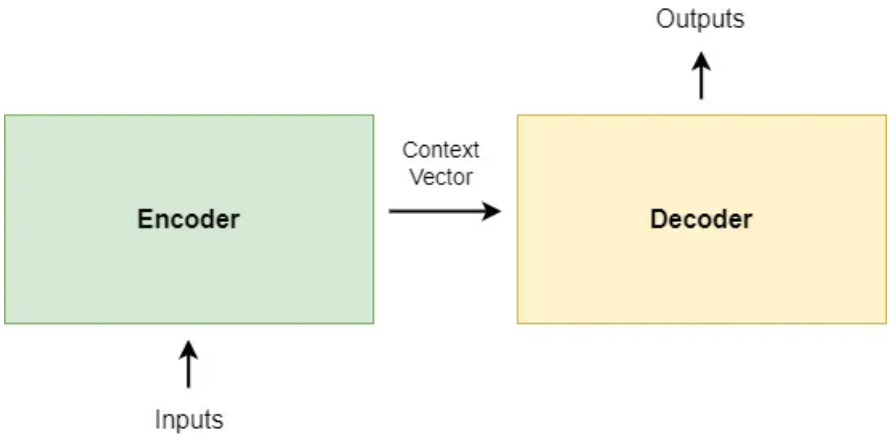
This repo goes along with this article and includes applying the seq2seq models discussed to the Ercot electricity grid load data in the final section.

Seq2Seq Models (Encoder & Decoder)

Seq2seq models have grown more and more popular for dealing with sequential or temporal data in deep learning. They primarily originated with NLP problems such as machine translation and text generation. As the name implies, they are a type of model that simply takes as input a sequence and outputs a sequence in response. However, the internals of the model can vary from recurrent, to convolutional, to transformers (specifically encoder/decoder transformers), to hybrids. So it can be thought of as more of a paradigm or generic structure for dealing with temporal data. Seq2seq models are usually made up of an encoder, which compresses the input sequence into a latent *context vector*, and a decoder, which learns to understand this context vector and produces an appropriate output sequence.

Now we will get into the details of these models along with other useful aspects of dealing with time series data.

Abstract Seq2Seq Structure

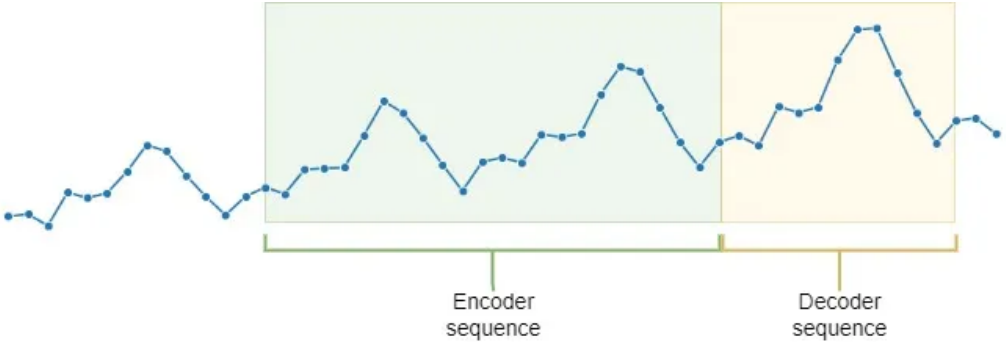


High-level abstraction of the seq2seq structure

Generically, the encoder takes the sequential input, encodes it into a context vector, or latent representations useful to the decoder and the problem at hand, and then the decoder outputs the proper response.

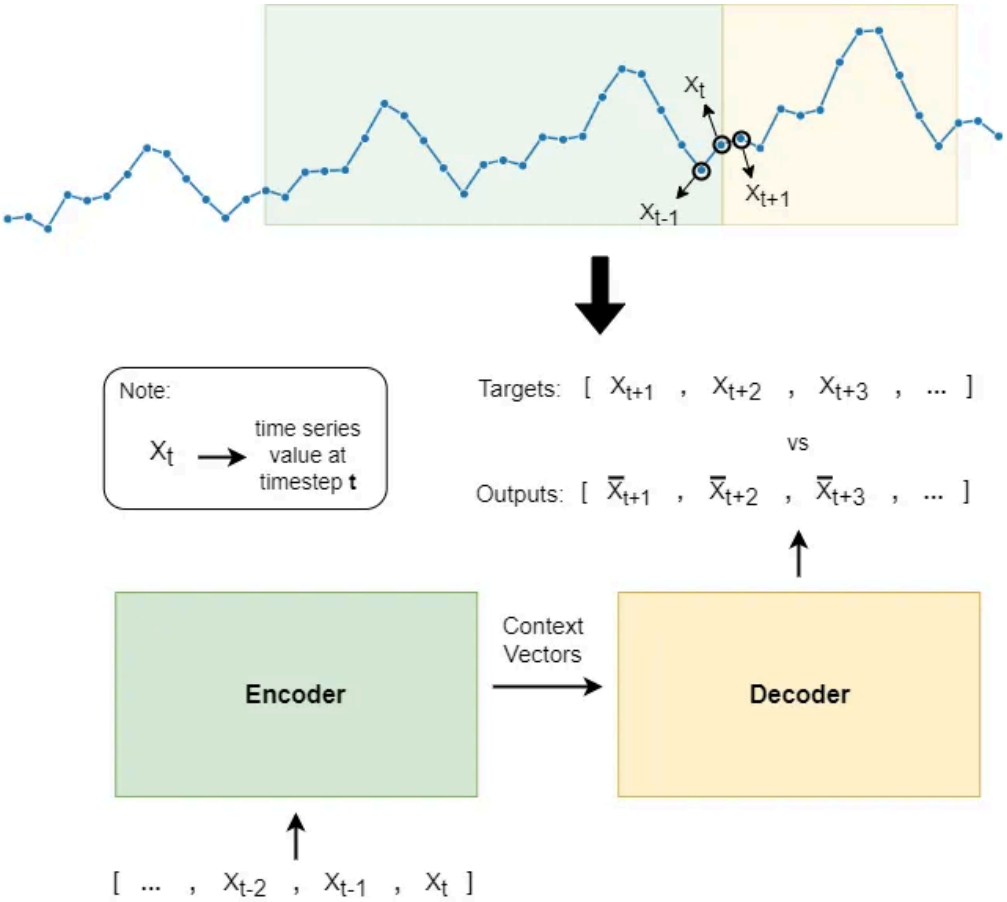
```
import torch.nn as nn
import torch

# Abstract seq2seq model in Python (using Pytorch)
class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder

    def forward(self, inputs):
        context_vector = self.encoder(inputs)
        outputs = self.decoder(context_vector)
        return outputs
```

For forecasting this means the encoder takes a sequence, or window of time series data, and the decoder attempts to forecast multiple steps ahead (known as multi-step or N-step forecasting).



Segment of AirPassenger data (x values are months, y values are # of plane passengers) that shows what a single encoder sequence to decoder sequence could look like, i.e. using 24 months to predict 12 months

In a little more detail:



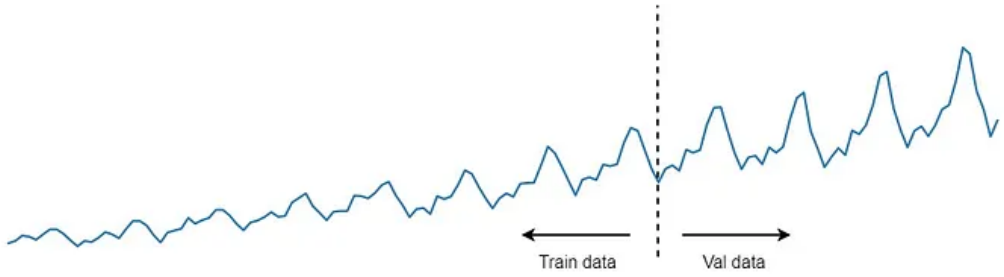
Shows how the specific time series values are fed into the encoder for the decoder to makes its best predictions on; this is compared to the targets during training and evaluating

The above diagrams are an example of a single sequence. A set of sequences (either for training or evaluation) can be created by simply sliding this window over one timestep at a time, as shown below.

This sliding window can be applied to the entire time series to convert it to a set of sequences

Normalizing and Splitting Time Series Data

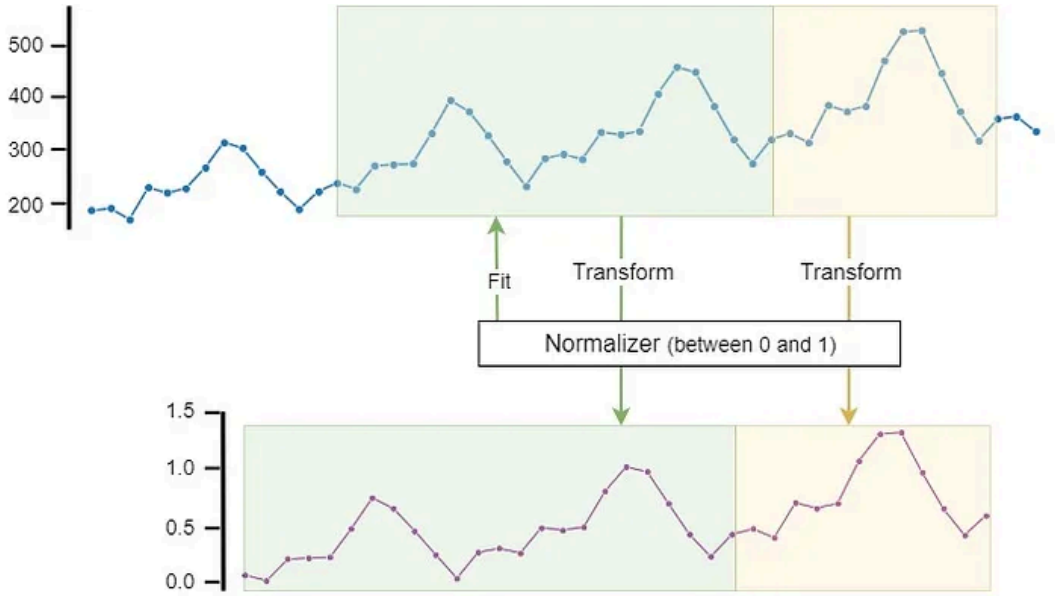
Preprocessing time series data can be complicated as the temporal dependencies of the data constrains what can and should be done. Like normal tabular data, you can split the data into train & validation data or use a type of cross-validation, called walk-forward validation, that adheres to the temporal dimension. For simplicity, we will just cover splitting the time series data into train and val.



Simple train/val split; no shuffling allowed since time series are temporally dependent

However, in the above diagram the distribution of the time series is changing over time (non-stationarity) which makes the train and val data quite different. So it would be useful in this case to apply walk-forward validation to avoid any issues this could cause the model.

Next we consider normalizing the data. We could just fit a normalizer to the train split and apply it to both the train and val data. However, more stable performance can be achieved with temporal data if we normalize at a “local” scale. Our seq2seq paradigm makes this easy since for each sequence of the data we can just fit a normalizer to the encoder data and apply it to both the encoder and decoder data. This also generally avoids issues with long-term trends creating extreme values after normalization.

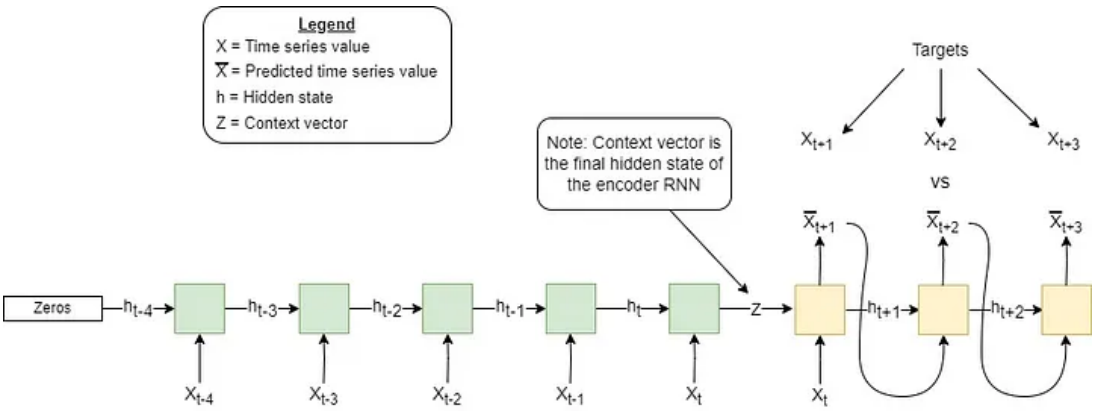


Each sequence is normalized in this way; this essentially mimics the decoder being used for inference on future data for which the exact scale is unknown beforehand

Additionally, you could attempt to detrend and deseasonalize the time series as part of preprocessing but 1) deep learning models tend to be less negatively effected by these properties than traditional auto-regressive models, 2) this would require adding back these properties during prediction time which can be complicated and possibly have a negative effect on performance, and 3) it has been shown through hybrid models (traditional auto-regressive modeling + deep learning) that including trend and seasonality processing *within* the model itself, instead of beforehand as a part of preprocessing, achieves better results and simplifies normalization; the ES-RNN model that won the all-important M4 competition is a hybrid model.

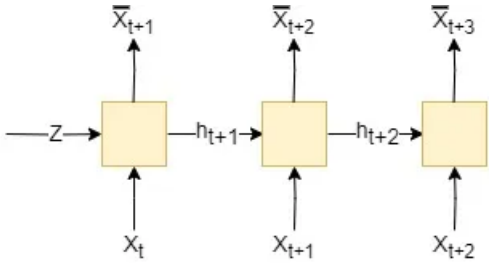
Recurrent Seq2Seq Model

Here we will focus on the earliest modern seq2seq model type to be popularized: RNN Encoder to RNN Decoder. Additionally, we will add a simple but powerful attention module to the decoder to enhance performance.



Fully recurrent seq2seq model where the decoder is free running; note that the last encoder input is the same as the first decoder input

From this we can see the internals of the encoder/decoder black boxes. Importantly, the decoder here is “free running” or feeding its previous output back into itself at each timestep. This is used for inference since we would not know the future values (and evaluation since it is meant to mimic inference). However, for training we can take advantage of having this “future” data to help make the training more stable with teacher forcing.



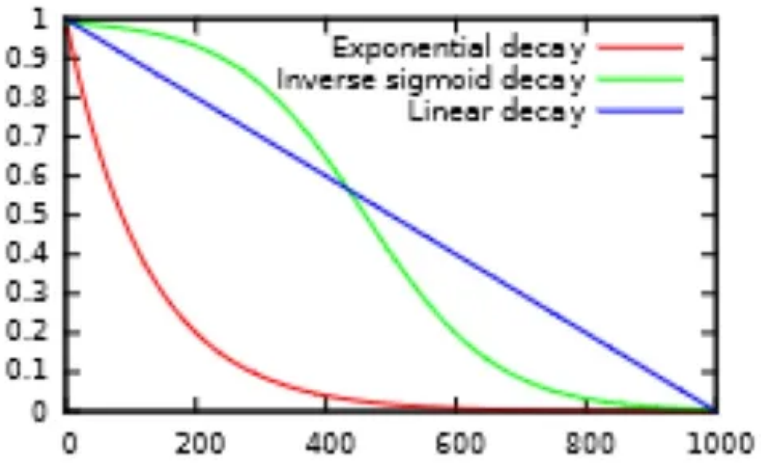
Decoder using teacher forcing: each of the inputs are the previous targets

Note: for a non-recurrent model like a transformer, padding is used for the decoder inputs to avoid giving it future information at each timestep.

Scheduled Sampling

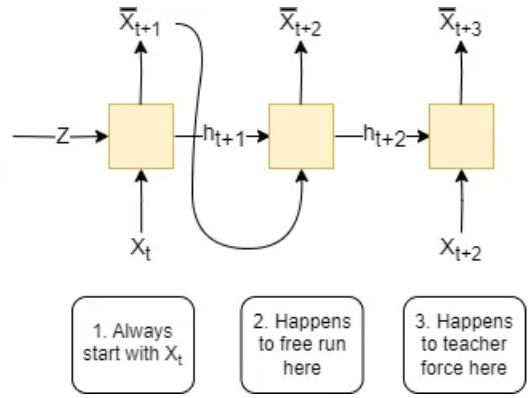
Teacher forcing is much easier than free running given that no matter how long the forecast horizon is, each decoder input keeps it on track. To counteract this difficulty gap we can use scheduled sampling. This simply means we use a decay function parameterized by time (either current training epoch or batch) to decide the probability of teacher forcing during training.

A few popular decay functions are shown below.



From "Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks" Bengio et. al.

This function is sampled every decoder step, which can result in both free running and teacher forcing within the same decoder sequence rollout:



Shows that within a single decoder rollout both free running and teacher forcing can occur

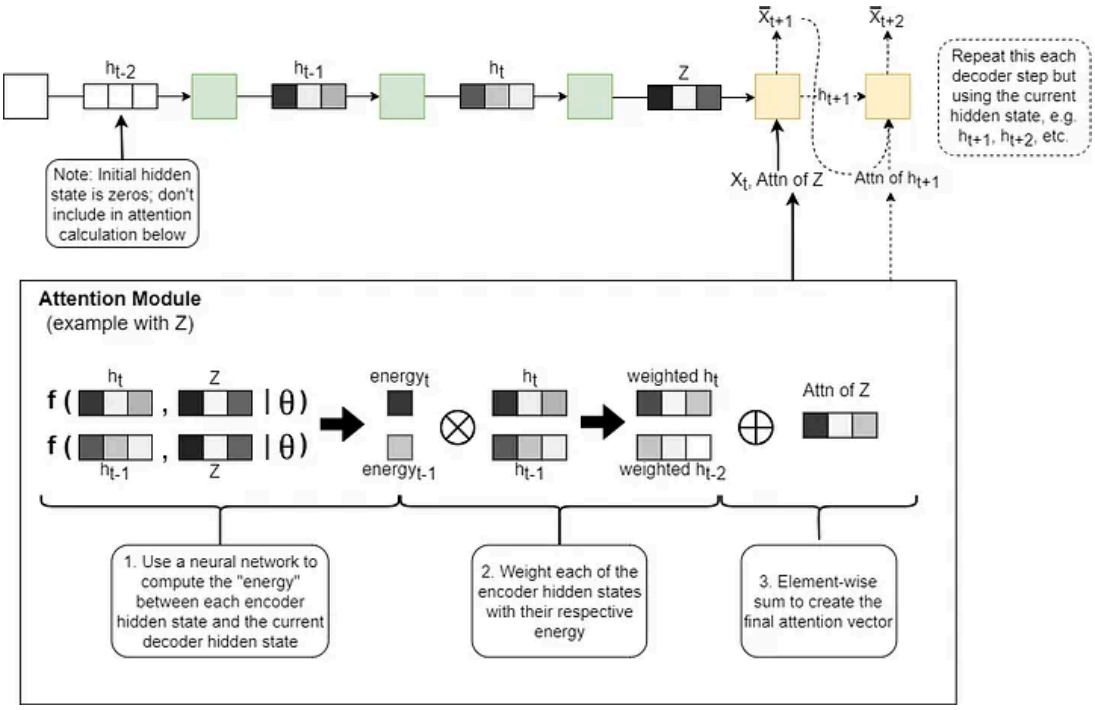
```
# A more detailed recurrent encoder & decoder seq2seq model:

class Encoder(nn.Module):
    def __init__(self, hidden_size):
        super().__init__()
        # Takes 1 time series value in this example, to hidden size
        self.rnn = nn.RNN(1, hidden_size)

    def forward(self, encoder_inputs):
        # NOTE: encoder_inputs looks like:
        # [..., X_t-4, X_t-3, X_t-2, X_t-1, X_t]

        # We let Pytorch handle the rollout behind-the-scenes,
        # so just feed in the whole encoder sequence.
        # And all we need is the final hidden vector as Z
        outputs, hidden = self.rnn(encoder_inputs)

        return outputs, hidden
```

```
class Decoder(nn.Module):
    def __init__(self, hidden_size):
        super().__init__()
        # Also takes 1 time series value
        self.rnn = nn.RNN(1, hidden_size)
        # The output layer transforms the latent representation
        # back to a single prediction
        self.out = nn.Linear(hidden_size, 1)

    def forward(self, initial_input, encoder_outputs, hidden, targets,
                teacher_force_probability):
        # NOTE:
        # initial_input is X_t
        # hidden is Z
        # targets looks like: [X_t+1, X_t+2, X_t+3, ...]
        # encoder_outputs are not used, but will be for attention later

        decoder_sequence_length = len(targets)

        # Store decoder outputs
        outputs = [None for _ in range(decoder_sequence_length)]

        input_at_t = initial_input

        # Here we have to roll out the decoder sequence ourselves because of
        # sometimes teacher forcing
        for t in range(decoder_sequence_length):
            output, hidden = self.rnn(input_at_t, hidden)
            outputs[t] = self.out(output)

            # Set-up input for next timestep
            teacher_force = random() < teacher_force_probability
            # The next timestep's input will either be this timestep's
            # target or output
            input_at_t = targets[t] if teacher_force else outputs[t]

        return outputs

class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder, lr):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder

        self.optimizer = torch.optim.Adam(self.parameters(), lr)
        # The best loss function to use depends on the problem.
        # We will see a different loss function later for probabilistic
        # forecasting
        self.loss_function = nn.L1Loss()

    def forward(self, encoder_inputs, targets, teacher_force_probability):
        encoder_outputs, hidden = self.encoder(encoder_inputs)
        outputs = self.decoder(encoder_inputs[-1], encoder_outputs,
                               hidden, targets, teacher_force_probability)
        return outputs

    def compute_loss(self, outputs, targets):
        loss = self.loss_function(outputs, targets)
        return loss

    def optimize(self, outputs, targets):
        self.optimizer.zero_grad()
```

```
            loss = self.compute_loss(outputs, targets)
            loss.backward()
            self.optimizer.step()

seq2seq = Seq2Seq(Encoder(hdiden_size), Decoder(hdiden_size), lr)
```

Attention

First introduced for machine translation in this paper, attention allows the decoder to look back more directly at encoder hidden states to figure out what the current prediction should be. This frees up the responsibility of the compressed context vector needing to represent all of the encoded information that the decoder might need to see, similar to the reasoning of using skip connections in deep networks.



Shows how the attention vector is computed for each decoder step

```
class Attention(nn.Module):
    def __init__(self, hidden_size):
        super().__init__(hidden_size)
        # Two layers are used so a non-linear act function can be placed
        # in between
        self.initial_layer = nn.Linear(2 * hidden_size, hidden_size)
        self.final_layer = nn.Linear(hidden_size, 1)

    def forward(self, current_decoder_hidden, encoder_outputs):
        # NOTE: encoder_outputs is the same thing as the encoder hidden states
        # that are output at each timestep since the encoder is only an rnn
```

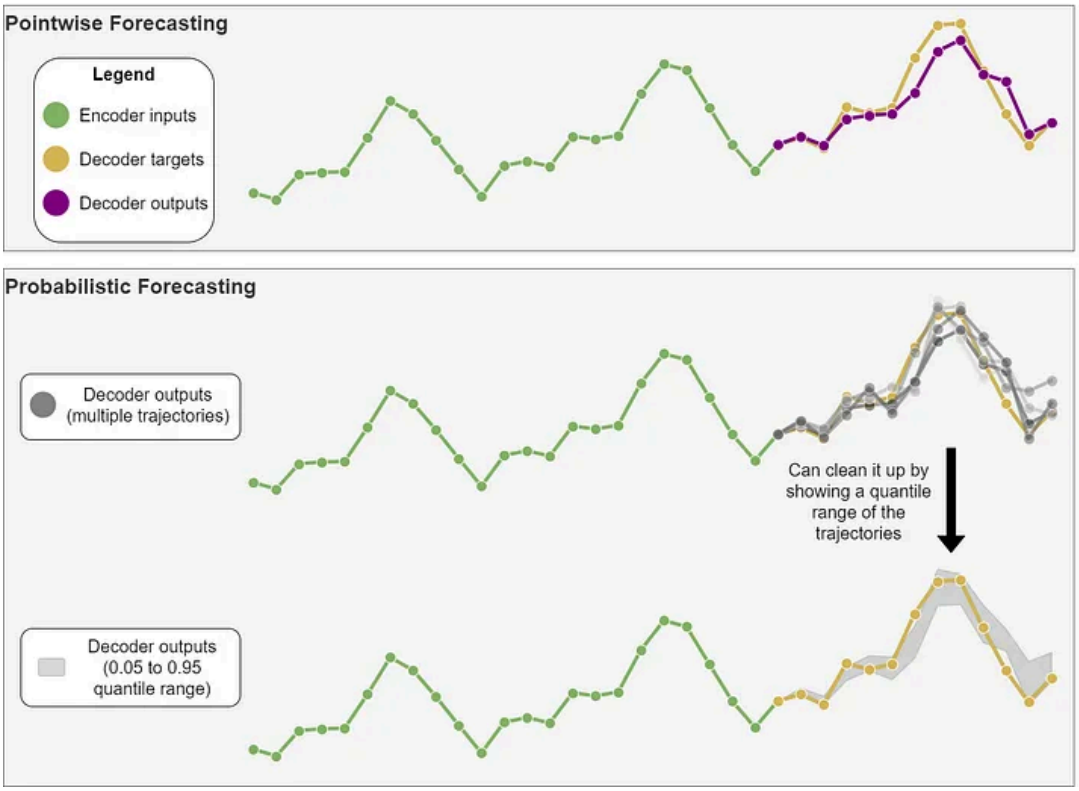
```
        # Step 1 from diagram)

        # Stack decoder hidden state on top of each other encoder_outputs
        # number of times to be able to feed into the neural network
        # all at once
        decoder_hidden_stack = stack_decoder_hidden(current_decoder_hidden,
                                                    len(encoder_outputs))

        # Compare each encoder hidden state with the current hidden state
        energy = self.initial_layer(torch.cat((encoder_outputs,
                                               decoder_hidden_stack)))

        # Apply tanh (used in the paper) to introduce non-linearity
        energy = torch.tanh(energy)

        # Apply final layer to compress to a single value for each comparison
        energy = self.final_layer(energy)

        # Softmax the energy to sum them up to 1
        energy = torch.softmax(energy)

        # Steps 2 & 3 from diagram)

        # The multiplication and sum are the same as a matrix multiplication
        attn = torch.mm(weights, encoder_outputs)

        return attn

# This is similar to the vanilla decoder,
# except it takes an attention vector as input as well.
# Any ... below means same as vanilla decoder
class DecoderWithAttention(nn.Module):
    def __init__(...):
            self.attention_module = Attention(hidden_size)
            # First change is that the decoder takes X_t and
            # the attention vector of size hidden_size
            self.rnn = nn.RNN(1 + hidden_size, hidden_size)
            ...

    def forward(...):
        ...
        for t in range(decoder_sequence_length):
            # Only other change is to compute the attention and
            # feed it, along with X_t, into the decoder at each timstep t
            attention_vector = self.attention_module(hidden, encoder_outputs)
            cat_input = torch.cat((input_at_t, attention_vector))
            output, hidden = self.rnn(cat_input, hidden)
            ...
```

Probabilistic Forecasting

So far we have dealt with “pointwise” predictions, where the model directly predicts each next time series value. A popular alternative is “probabilistic forecasting” which has the model learn the distribution to sample each point, or prediction, from. This is done by having the model output the

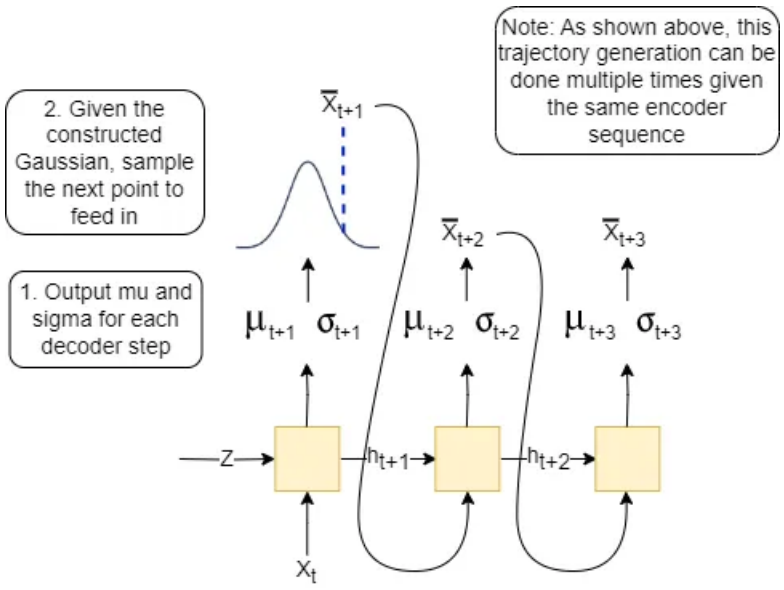
parameters to a distribution such as the Gaussian, which would be the mu (mean) and sigma (standard deviation, or std). And then the actual point is sampled from this distribution. This is an efficient way to avoid the problem of learning instability caused by noise. In pointwise forecasting the model will attempt to predict this noise, which by definition is not predictable.

Probabilistic, or stochastic, forecasting also allows for sampling multiple varying trajectories as shown below.



Example of different outputs between pointwise (single trajectory) and probabilistic (distribution which allows for multiple trajectories) forecasting

And this is how the model would change to allow for this functionality:



Probabilistic forecasting for the decoder (encoder stays the same)

```
# Assumes a Guassian distribution, but other distributions can be chosen

class DecoderForProbabilistic(nn.Module):
    def __init__(...):
        ...
        # First change: instead of a point output
        # for the final layer, it's two for mu and sigma
        self.out = nn.Linear(hidden_size, 2)

    def forward(...):
        ...
        for t in range(decoder_sequence_length):
            ...
            # Second change is to sample from the dist (mu and sigma)
            # to get the next point to feed in if necessary
            input_at_t = targets[t] if teacher_force else sample(outputs[t])
            ...

        # The outputs returned will contain the mus and sigmas,
        # which are used by the loss function described below
        return outputs

# Any ... below means same as vanilla Seq2Seq
class Seq2SeqForProbabilistic(nn.Module):
    def __init__(...):
        ...
        # Seq2Seq will use the Gaussian negative log
        # likelihood loss which takes mus, sigmas (actually variances),
        # and pointwise targets
        self.loss_function = nn.GaussianNLLLoss()

  def compute_loss(self, prediction, target):
      # Need to split prediction tensor into mus and sigmas
      mus = prediction[:, 0]
      sigmas = prediction[:, 1]
      # GaussianNLLLoss takes variance so square std
```

```
        loss = self.loss_function(mus, target, sigmas ** 2)
        return loss
```

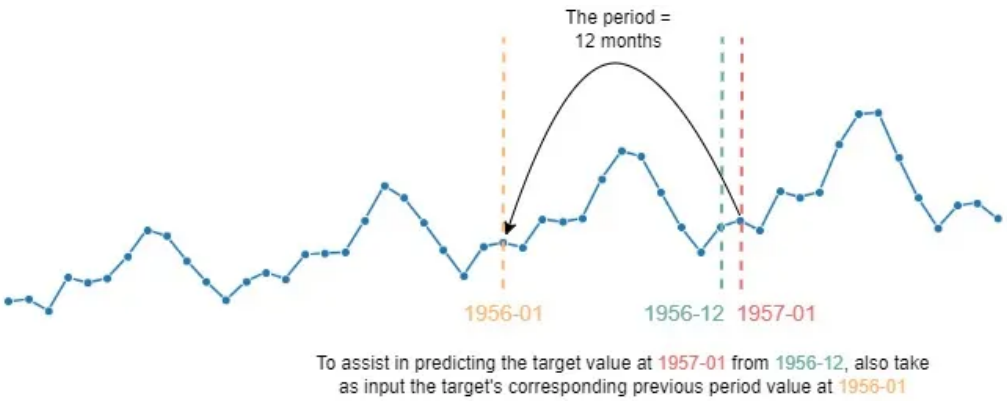
Turning this into a Multivariate and Multiple Time Series Problem

So far our sequence data is univariate and only includes the time series data as input at each timestep. While this is all that is required to forecast, other useful information called covariates can be included to enhance performance (and turn it into a multivariate problem).

Covariates, also known as exogenous variables, are external variables used as additional input. For plane passenger forecasting this could include weather, plane type or even the date. Covariates aren't forecasted, so they aren't used as targets, just inputs.

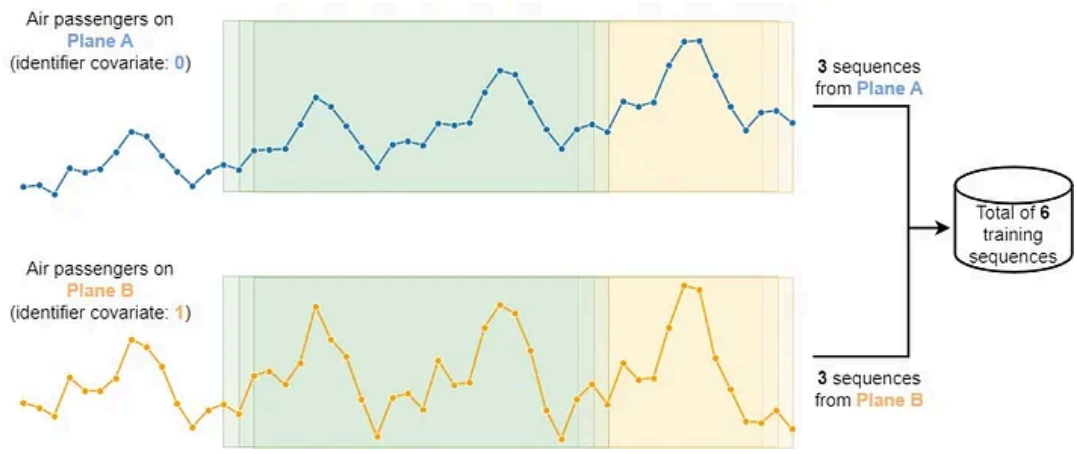
Its important to frame a forecasting problem in terms of applying the trained model to real inference. In inference, the purpose of the decoder is to forecast unknown data, so it can only receive "future" covariates, or features that are known in the future, such as the date at each timestep (the encoder could also receive this). A covariate such as weather which is only known for certain "historically" can only be included as input into the encoder.

Given the strong seasonality of the air passenger data we can actually use the value of the *previous* period, called a lag feature, as both a historic and future covariate. However, to use this as a future covariate the decoder must be restricted to forecasting only *one* period in the future, so that it always *knows* the actual lag value.



Example showing what a lag feature is given seasonality

Finally, many models nowadays are trained on multiple time series in an attempt to make them more “general”. For example, the Ercot data described below has 4 time series where each is electricity grid load data from different regions (North, West, Coast, etc.). These time series should be related (and use the same kinds of inputs and targets) otherwise a single model will have a difficult time forecasting them. The sequence generation operates on each time series separately and then just throws them together for training. This also means we can add a new covariate which is just a categorical identifier for the time series that a sequence was generated from. This can help the model know which time series its dealing with in case they behave a little differently from each other.



Example on dealing with multiple time series; Note: the time series of Plane B is just Plane A but detrended and noised for the sake of this example

At this point the encoder and decoder inputs would have three features per timestep: # passengers time series value, lag covariate and identifier covariate. Once again, they might not have the same number of features, but do in this example. The targets remain unchanged.

```
class Encoder(nn.Module):
    def __init__(self, encoder_input_size, hidden_size):
        super().__init__()
        # Now it can take more than 1 value, if historical covariates
        # are involved. In our example this would be 3.
        self.rnn = nn.RNN(encoder_input_size, hidden_size)
        ...

class Decoder(nn.Module):
```

```
    def __init__(self, decoder_input_size, hidden_size):
        super().__init__()
        # Takes 3 as well, but could be different
```



```
                teacher_force_probability):
        # decoder_inputs for a single time series sequence lookes like:
        # [(X_t, lag_cov_t, id_cov_t), (X_t+1, lag_cov_t+1, id_cov_t+1), ...]
        ...
        for t in range(decoder_sequence_length):
            output, hidden = self.rnn(input_at_t, hidden)
            outputs[t] = self.out(output)
            ...
            # With decoder_inputs can just select the next one if
            # teacher forcing, otherwise concat output with covariates
            # as the output is just the time series value
            input_at_t = decoder_inputs[t+1] if teacher_force else \
                            outputs[t] + decoder_inputs[t+1][1:]
            ...

class Seq2Seq(nn.Module):
    def forward(self, encoder_inputs, decoder_inputs, targets,
                teacher_force_probability):
        ...
        # Send in decoder_inputs
        outputs = self.decoder(decoder_inputs, encoder_outputs,
                               hidden, targets, teacher_force_probability)
        return outputs
```

Note that the alterations above (attention, probabilistic forecasting, covariates) can be applied all at once, or in any combination, e.g. attention + probabilistic, covariates + attention, etc, which is shown in the accompanying jupyter notebook.

That is it on the details of seq2seq models for time series forecasting. Next is a practical example of the model + its alterations applied to a popular forecasting dataset.

Quick Application of Seq2Seq Forecasting to Energy Load Data

Ercot Data

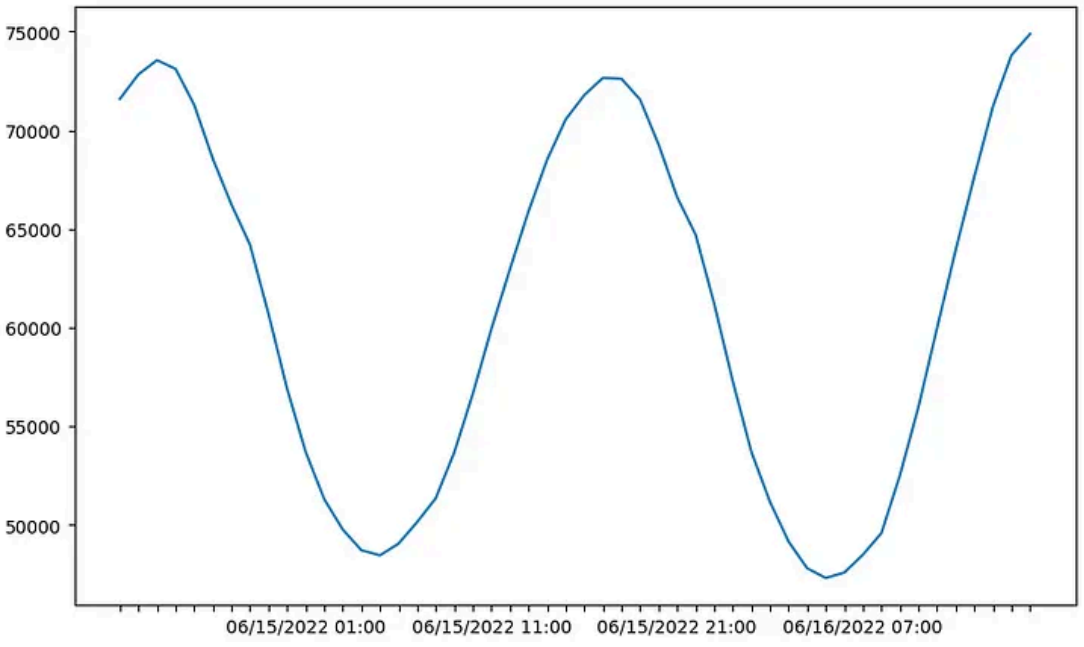
Ercot is the organization that manages Texas's power supply. Their hourly load data can be found here, dating back to 1995 and across multiple regions

of Texas:

	Hour Ending	COAST	EAST	FWEST	NORTH	NCENT	SOUTH	SCENT	WEST	ERCOT
0	01/01/2022 01:00	12054.939199	1302.296674	4161.193625	757.843076	9676.300802	3172.878316	5908.031505	973.455700	38006.938896
1	01/01/2022 02:00	11793.290315	1259.355201	4147.907009	737.236591	9307.126712	3123.318608	5708.512022	959.775908	37036.522365
2	01/01/2022 03:00	11460.841252	1210.287905	4156.412366	725.610502	8920.424552	3003.396233	5463.522829	941.112359	35881.607998
3	01/01/2022 04:00	11244.980243	1179.311517	4149.811722	717.420214	8678.807826	2898.097471	5255.252404	920.373708	35044.055105
4	01/01/2022 05:00	11073.085585	1171.841803	4140.619028	719.178247	8573.370608	2825.100402	5164.172158	918.203309	34585.571140
...	...	...	...	...	...	...	...	...	...	...
8755	12/31/2022 20:00	10247.562908	1407.683729	5218.996613	938.901362	11656.958118	3452.868539	6833.564855	1125.338906	40881.875030
8756	12/31/2022 21:00	9887.676773	1362.637978	5217.501326	963.913723	11242.914145	3228.129723	6629.002740	1103.919231	39635.695638
8757	12/31/2022 22:00	9572.382483	1327.298817	5235.313653	949.798259	10944.369192	3078.543411	6413.225604	1084.122587	38605.054006
8758	12/31/2022 23:00	9258.586996	1237.913479	5226.630655	928.929373	10508.050809	3020.175467	6161.413424	1058.601615	37400.301819
8759	12/31/2022 24:00	8999.616362	1216.122804	5193.649360	898.505355	10139.198578	2951.500773	5957.229897	1043.484788	36399.307918

8760 rows × 10 columns

2022 hourly load data across 8 different regions, with the “ERCOT” column identifying the total



A small section of 2022 total plotted

Subset of Ercot Data

We will use the hourly 2022 data for 4 regions: East, West, North and Coast. That means the model will be dealing with 4 different (but related) time series. This is a single target problem (energy load of the next hour), but will be multivariate: current energy load, lag covariate (in this case one period is 24 hours) and region identifier covariate. 144 hours, or 6 days, were encoded for the decoder to forecast 24 hours, or 1 day.

The data is split into train/val/test as described above, at about a 60/20/20 ratio. The val performance was used to search over multiple

hyperparameters: architecture size, learning rate, batch size, etc.

Holdout Test Performance and Baselines

It's important to include simple baselines to compare with model performance. The baselines used here are: 1) have the decoder sample from a Gaussian using the mean and std of the encoded time series sequence, 2) output the corresponding previous period value (i.e. the lag covariate feature), which is a solid predictor.

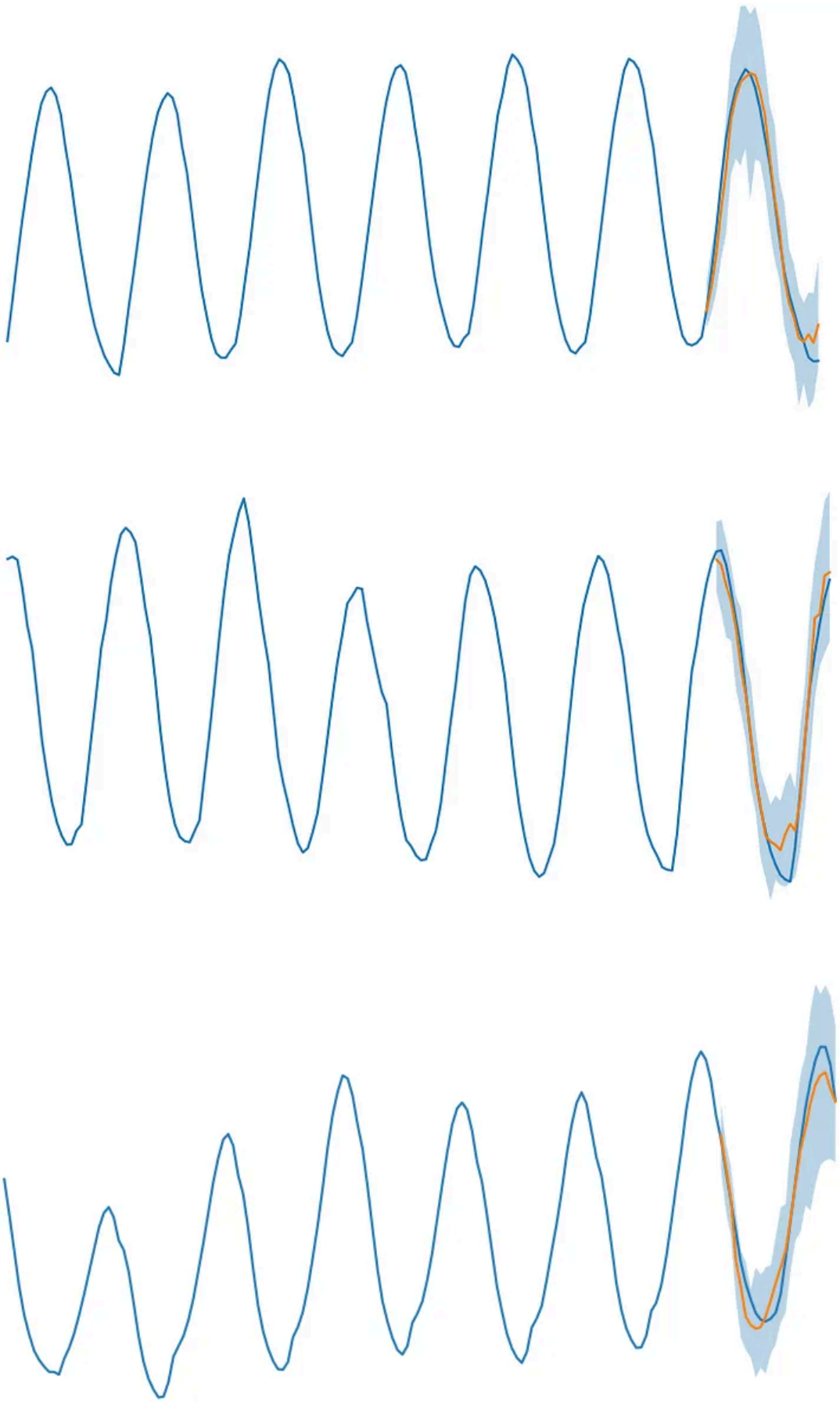
For brevity, we will look at the performance of the model with all of its available alterations (attention + probabilistic), and the baselines. But you can run other combinations from the notebook.

Test performance (using the probabilistic GaussianNLLLoss, i.e. lower is better)

- Baseline 1: 0.58
- Baseline 2: -0.22
- **Best Model: -0.70**

Note: a negative loss is valid; also the time series values are normalized which increases the chance that the loss dips into the negative (given it's working with small numbers). The simple first baseline doesn't do well. The second baseline is much better given the strong seasonality. But the model still does better than this which means it not only understands the seasonality but also other properties that describe the distribution of the data such as trend and/or variance.

Here are some example forecasts from the best model on the test set. 50 trajectories were sampled per encoder sequence with the orange line as the median forecast, and the shaded region as the quantile range from 0.05 to 0.95.



As we can see, the median output is very close to the targets. Additionally, the shaded region indicates that the model outputs a tighter distribution on the "straightaways" of the time series and then loosens at the crests/troughs, where there is more uncertainty in the forecast.

Conclusion

While we focused on simple fully recurrent networks in this article, most of the details and abstractions apply to various seq2seq models including transformers. There are more concepts that were not included here to understand such as sMAPE, multiple targets, trend and seasonality processing, ensembles, bi-directional RNNs, etc. that can help enhance or aid the forecasting process.

Thanks for reading!

Forecasting | Seq2seq | Time Series Forecasting | Deep Learning | Attention

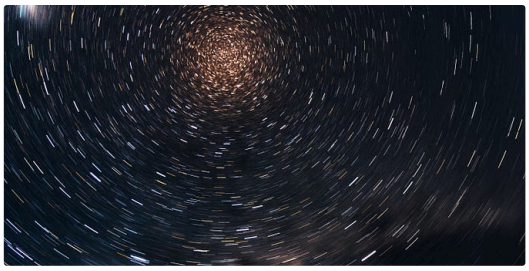
Written by Max Brenner

503 Followers

Interested in all things machine learning, procedural and generative

Follow

More from Max Brenner



Max Brenner

Eigenvalues and Eigenvectors in Data Science: Intuition and...

What does eigenstuff mean for data science?

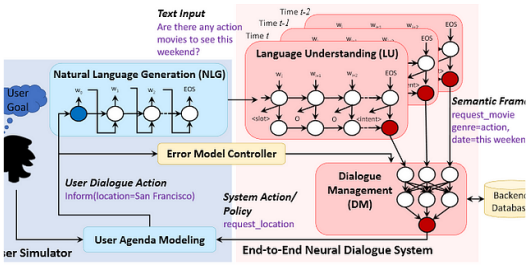
Apr 19, 2023 62 1

Max Brenner

Matrix Factorization for Collaborative Filtering: Linear to...

In this article, we will explore a variety of matrix factorization models, and how to...

Dec 17, 2022 30



Max Brenner in Towards Data Science

Training a Goal-Oriented Chatbot with Deep Reinforcement Learnin...

Part I: Introduction and Training Loop

Dec 1, 2018 589 8

Max Brenner in Towards Data Science

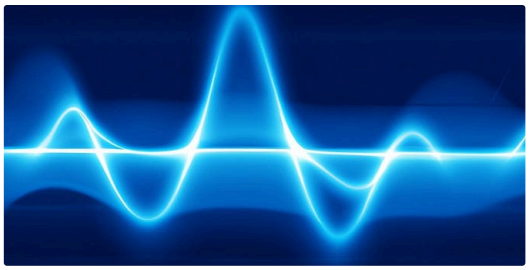
Deploying a Custom Docker Model with SageMaker to a Serverless...

How to effectively deploy a model with AWS

Aug 27, 2020 180 1

See all from Max Brenner

Recommended from Medium

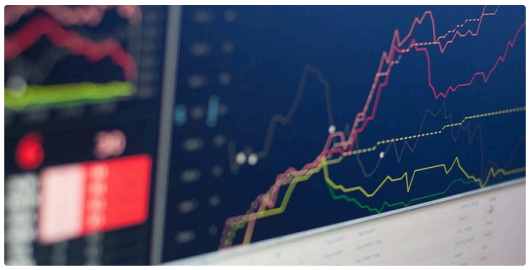


Nikos Kafritsas in Towards Data Science

Temporal Fusion Transformer: Time Series Forecasting with Dee...

Create accurate & interpretable predictions

Nov 4, 2022 2.1K 26



Pranjal Joshi

Tokenized Language Models for Time-Series Forecasting

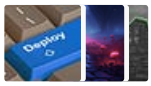
A Zero-Shot Time Series Forecasting with Chronos (LLM-based model)—Python Code...

Apr 22 55

Lists

Practical Guides to Machine Learning

10 stories · 1859 saves



Predictive Modeling w/ Python

20 stories · 1535 saves

Living Well as a Neurodivergent Person

10 stories · 1096 saves

Natural Language Processing

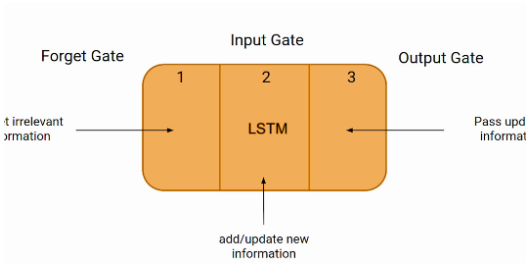
1705 stories · 1271 saves

AI SageScribe

Building a Multi-Head Attention with PyTorch from Scratch—A...

Here, we explore a streamlined implementation of the multi-head attention...

Jun 30 1

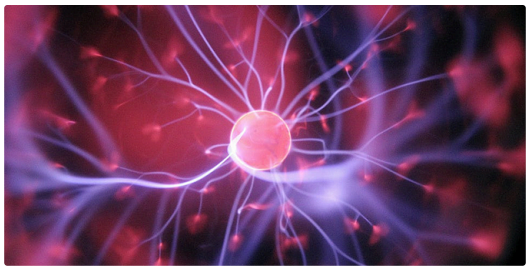


Kevin Akbari

Unveiling the Power of LSTM: A Guide to Text Generation

Introduction

Apr 15



Alexzap

Stock Time Series Forecasting in a Nutshell: AI/ML...

Zero to hero use-case examples of learning-based forecasting future stock prices based...

Sep 9 87 1

Shenggang Li in DataDrivenInvestor

Time Series Forecasting: A Comparative Analysis of SARIMA...

Assessing the Efficiency and Efficacy of Leading Forecasting Algorithms Across...

Apr 13 445 4

See more recommendations