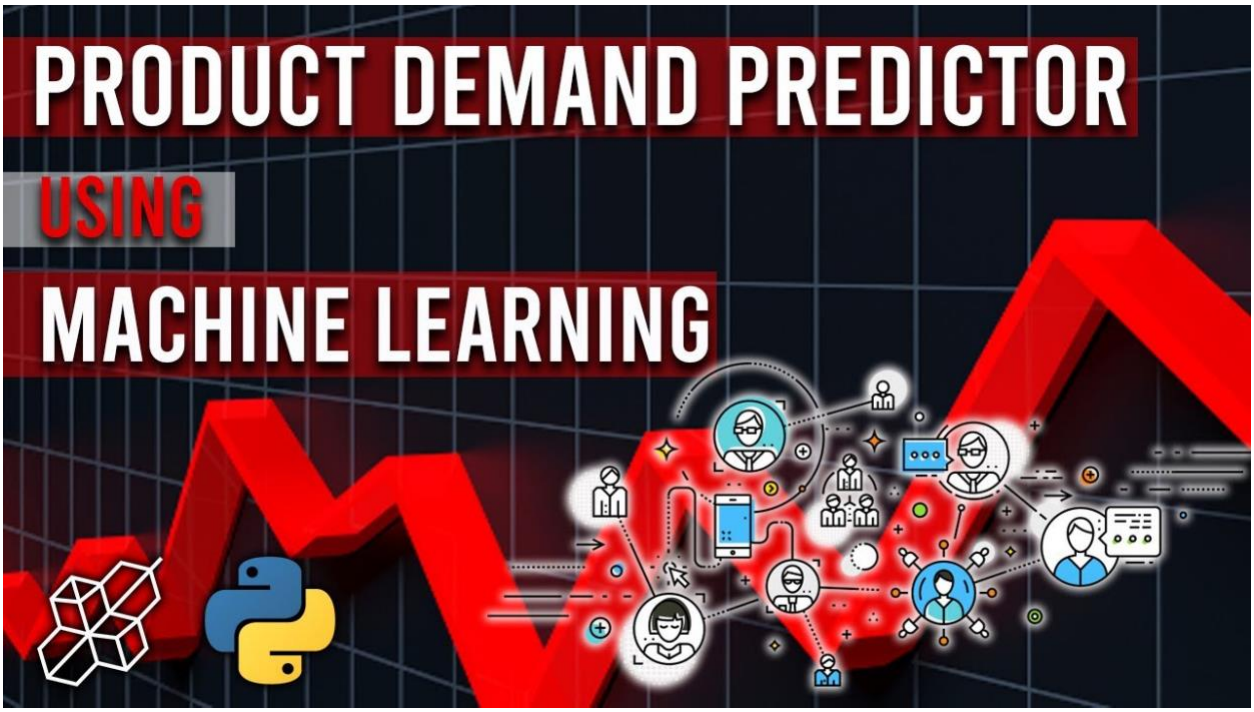
PRODUCT DEMAND PREDICTION WITH MACHINE LEARNINGS

TEAM MEMBER

911721104302 : A.Bosekumar

Phase 2 Submission Document

PROJECT: Product Demand Analysis



INTRODUCTION

In order to provide intelligent and meaningful responses, an in-depthexamination and assessment of various factors such as consumption growthpatterns, income and price elasticity of demand, market composition, nature ofcompetition,

availability of substitutes, and teach of distribution channels isrequired.

Because of the importance of demand analysis, it should be done in amethodical and orderly manner.

The following are the major steps in such ananalysis:

- Situational analysis and goal-setting
- Secondary data collection
- Market survey
- Market characterisation
- Demand forecasting
- Market planning

Content for Phase 2 project

Consider incorporating time series forecasting techniques like ARIMA or Prophet to capture temporal patterns in demand data.

Overview

Time series forecasting is an important topic for machine learning to predict future outcomes or extrapolate data such as forecasting sale targets, product inventories, or electricity consumptions. I found many general ML classes that cover specific context in detail but not in a full scope. This post hope to cover a wider scope by highlighting the different approaches to time series forecasting from statistical methods to a more recent state of the arts deep learning algorithms .

The topics will be as follow:

1. Statistical technique: ARIMA
2. Exponential smoothing approaches (MA, Holt ES, Double ES, Holt-Winter)
3. ML algorithms with Linear Regression and XGBoost
4. Deep Learn approaches with CNN, RNN, and Transformer (LSTM, CNN with LSTM, TCN, LSTNet, N-BEATS, TFT)
5. Commercial applications: Facebook Prophet and Amazon DeepAR

The full working code for each of these algorithms is in the Python notebooks here: https://github.com/phylypo/TimeSeriesPrediction.

In the notebook, we also cover other algorithms not mentioned here like FFT, HMM, and other State Space approaches.

Introduction to Time Series

Time series is a sequence of data that has some order usually with a time component in a set interval. A typical example is the stock daily closing price over time or DNA sequences.

Typical tasks involve times series includes:

- **Data Analysis**: also known as exploratory data analysis (EDA) of a time series involves analyzing the data for its structure such as seasonality or nature of its temporal process.
- **Prediction/Forecasting**: Prediction of future values of time series involves using past data to predict the future.
- **Classification**: You can classify time series into different types such as detection of an ECG heartbeat graph is a normal type or not.
- **Anomaly Detection**: Another task is detecting values that are not within the expected range in the time series.

We will be mainly focusing on a predicting task but partly cover some analysis in order to better understand the time series dataset.

Types of time series

There are two types of time series:

1. univariate: time series with a single observation per time increments.
2. multivariate: time series that has more than one observation per time increments. These observations or time-dependent variables can capture the dynamic of multiple time series.

We will be focusing on univariate in this article for simplicity. There are some example codes that use multivariate.

Dataset

To model or make prediction, we first need the dataset. These dataset are self explanatory with link to download. There are a few publicly available datasets.

Univariate Dataset:

- Air passengers: number of air passenger per month over 12 years (1949–1960) https://assets.digitalocean.com/articles/eng_python/prophet/AirPassengers.csv
- Sunspot: monthly count of the number of observed sunspots for just over 230 years (1749–1983) https://storage.googleapis.com/laurencemoroney-blog.appspot.com/Sunspots.csv
- Shampoo Sales: monthly sales of shampoo over a 3 year period (https://raw.githubusercontent.com/jbrownlee/Datasets/master/shampoo.csv)
- Milk production: average monthly milk production (in lbs) of cows from Jan/1962 to Dec/1975. (https://raw.githubusercontent.com/plotly/datasets/master/monthly-milk-production-pounds.csv)

Multivariate Dataset:

- Traffic: A collection of 48 months (2015–2016) hourly data from the California Department of Transportation. The data describes the road occupancy rates (between 0 and 1) measured by different sensors on the San Francisco Bay area freeways.
- Solar-Energy: the solar power production records in the year 2006, which is sampled every 10 minutes from 137 PV plants in Alabama State.
- Electricity: The electricity consumption in kWh was recorded every 15 minutes from 2012 to 2014
- Exchange-Rate: the collection of the daily exchange rates of eight foreign countries including Australia, British, Canada from 1990 to 2016.

All three above datasets can be found here: https://github.com/laiguokun/multivariate-time-series-data.

Kaggle Tourism:

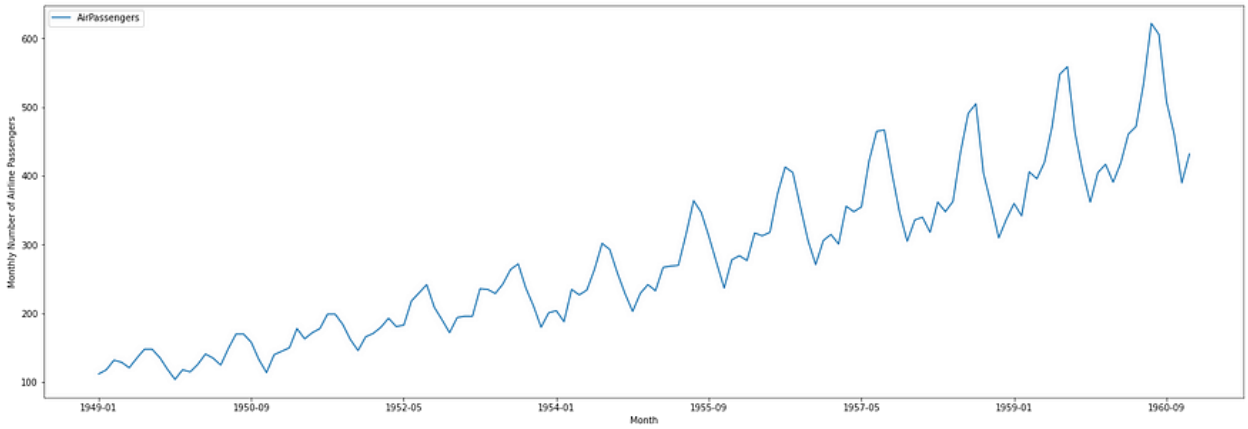
- Part one requires competitors to predict 518 tourism-related time series. It contains 518 yearly time series. (https://www.kaggle.com/c/tourism1)

- Part two requires competitors to predict 793 tourism-related time series. It contains 793 different time series. The first 366 columns contain monthly time series. The next 427 time series contain quarterly time series. (https://www.kaggle.com/c/tourism2)

Makridakis Competitions:

- M3: A total of 3003 different time series was used.
- M4: 100,000 real-life series https://mofc.unic.ac.cy/the-dataset/, https://github.com/Mcompetitions/M4-methods/tree/master/Dataset

We will be using the AirPassenger dataset throughout this article. Here is what the monthly count of air passengers looks like from 1949 to 1960.



AirPassengers

Components of Time Series

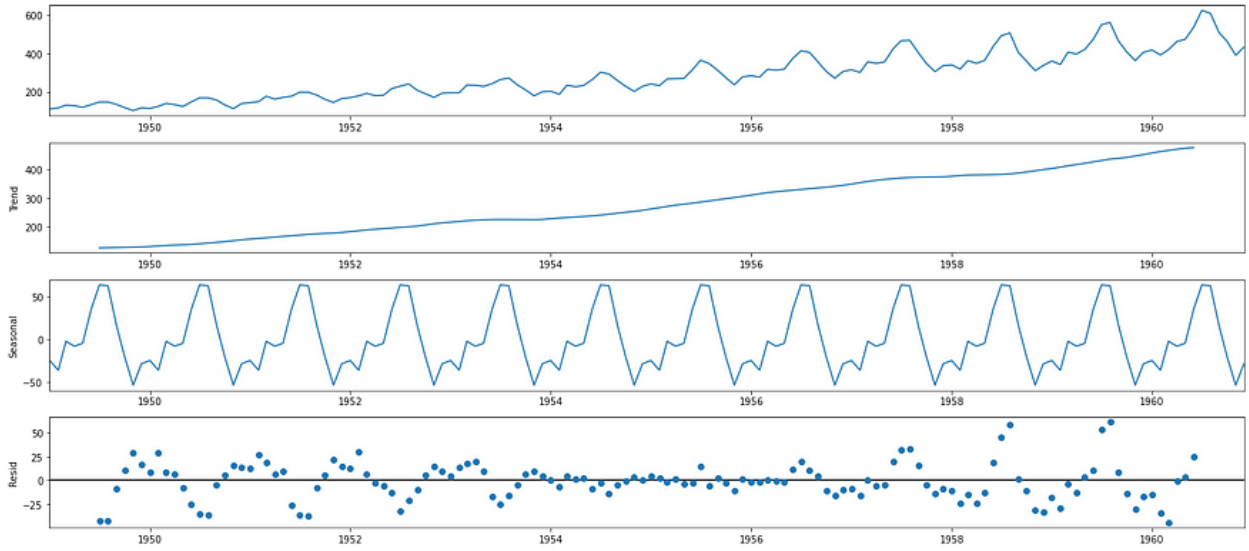
The above chart of the AirPassengers data clearly shows a linear trend and some seasonality. These are example of components in time series. Formally, time series consists of:

1. Level: local mean of the series
2. Trend: the change between successive time points
3. Seasonality: the deviation from the local mean due to seasonality
4. Noise: residual components or random variation

$$y = level + trend + seasonality + noise$$

where y is the observation.

With the AirPassengers dataset above, it can be decomposed into trend, seasonality, and noise as below:



Predictions and Metrics

The goal of our task is to predict the future values. Then we need to measure how our predictions for different approaches performed in comparison. There are two types of prediction that we need to distinguish before we measure it.

1. One-step ahead prediction: predicting the next one step from our last observation
2. Multi-horizons forecast: predicting multiple steps ahead

In the multi-horizon forecast, we can accomplish this through two approaches:

1. Iterated approaches: utilize one-step-ahead prediction and recursively feeding predictions to future inputs.
2. Direct approach: to explicitly generate predictions for multiple time steps at once.

Before we can measure how well our algorithms predict the future values, we need to discuss how we split our data into training and test sets before running the experiment.

Train and Test Split

In machine learning, we need to test our algorithm on the data that we have not seen. This is done by setting aside a portion of the dataset that will not be used in the training. This set is only used for testing the model. So in our dataset, we want to train on the earlier part of the dataset and leave out the

later part of the dataset to evaluate how well the model performs. We don't want to randomly split the dataset since sequence matters. We can split the data as follow:

- Training set: take the early portion of the data (ie. the first 80%)
- Test set: set aside the latest part of the data (ie. last 20% or the last year of monthly data, or last month of the yearly data)

Metrics

To measure how well the algorithms perform, there are several metrics that we can use:

- MSE: Mean Squared Error
  Pandas Series: *np.square(y_hat - y).mean()*
- RMSE: Root Mean Squared Error
  Pandas Series: *np.sqrt(MSE)*
- MAE: mean absolute error
  Pandas Series: *abs(y_hat - y).mean()*
- MAD: mean absolute deviation
  Pandas Series: *abs(y_hat - y.mean()).mean()*
- MAPE: mean absolute percentage error
  Pandas Series: *(abs(y - y_hat) / y).mean() * 100*

- sMAPE: Symmetric mean absolute percentage error (scales the error by the average between the forecast and actual)
  Pandas Series: *abs(y - y_hat).sum()/(y + y_hat).sum( ) * 100*
- MASE: Mean Absolute Scaled Error — scales by the average error of the naive null model

where
y : the ground truth observation value
y_hat : predicted value

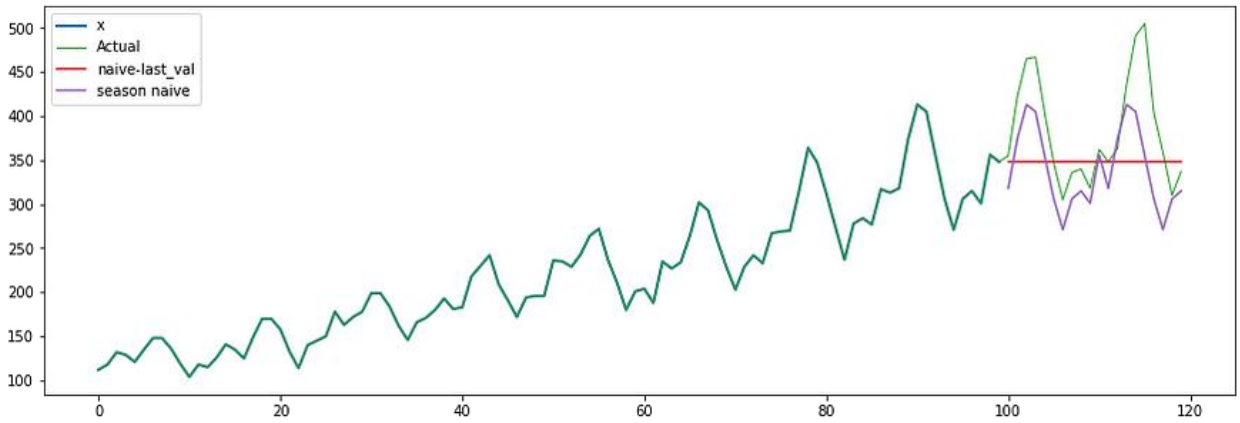
We will be using **MAE** as our metric to see how well our algorithms are doing.

Naive Approaches

We need a baseline approach that we can compare the performance of our algorithms. There are several naïve approaches :

- **Null model**: predict the next value as the previous one (persistence model)
- **Seasonal Naive**: add seasonality to the prediction by using the previous season value
- **Mean**: take an average of all previous values as the forecast value
- **Random Walk**: randomly add noise to the previous value as the next prediction

Here is the illustration of the predictions of the two models (null model and seasonal naive).



The **MAE** for the **Null model** for this dataset to predict the last 12-month is 49.95 and for the **Seasonal Naive model** is 45.60. We will use this as our baseline comparison.

Smoothing

The technique of smoothing is using the moving average or exponential smoothing as a way to filter out some of the noises in the data.

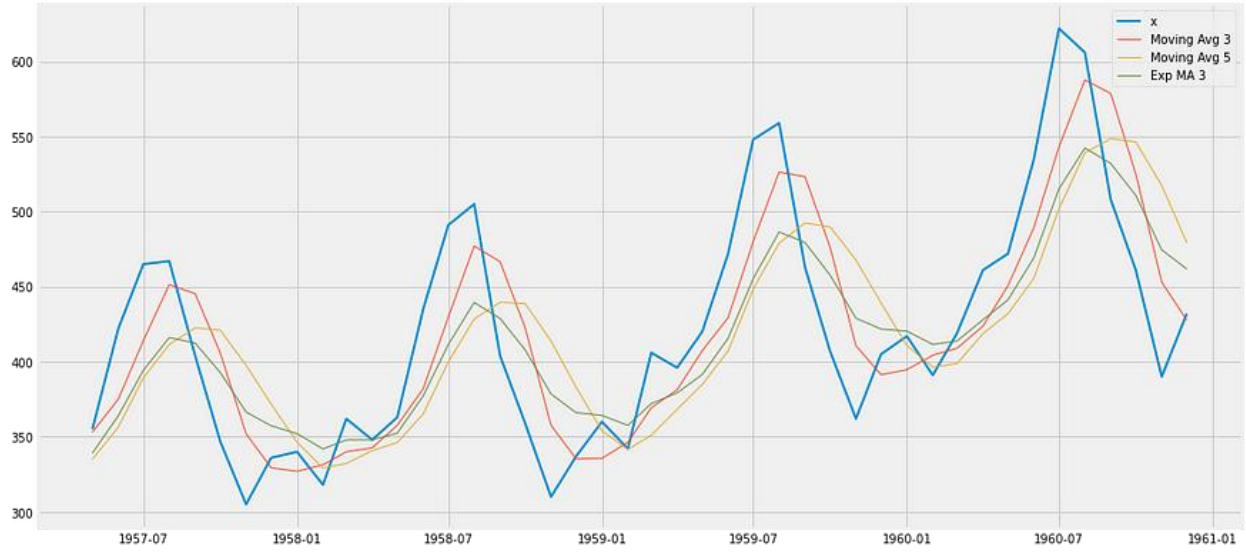
Simple Moving Average

Each data point is an average over *n* data points. The larger n is, the smoother the chart becomes. In *pandas*, you just call:

```
pandas.DataFrame.rolling(n, center=False).mean()
```

The moving average could be in the center and trailing Moving Average by specified the *center* argument

Here are examples of moving average with 3-month, 5-month moving average, and 3-month exponential moving average that will be explained below.



Exponential Smoothing

Linear Exponential Smoothing (Holt ES)

The Linear Exponential Smoothing (Holt ES) is also known as a simple exponential weighted moving average. Whereas the simple moving average weighted the previous points equally, simple exponential MA gives more weight to the more recent values.

$$y_0 = x_0$$

$$y_t = \alpha x_t + (1 - \alpha) y_{t-1}$$

- α : smoothing factor and $0 < \alpha < 1$; lower value means more smoothing
- y_t approximate over $\frac{1}{\alpha}$ data points

ie. α = 0.5 ≈ 2 data points, 0.1 ≈ 10 data points, 0.02 ≈ 50 data points

In *pandas*, we can use:

```
pandas.DataFrame.ewm(alpha=0.12).mean()
```

Double Exponential Smoothing

Also known as **Holt-Winters Double ES**, double exponential smoothing takes into account the trend.

> $y_0 = x_0$
>
> $b_0 = x_1 - x_0$

$$y_t = \alpha x_t + (1 - \alpha)(y_{t-1} + b_{t-1})$$

$$b_t = \beta(y_t - y_{t-1}) + (1 - \beta)b_{t-1}$$

where

- $0 \leq \alpha \leq 1$: the data smoothing factor
- $0 \leq \beta \leq 1$: the trend smoothing factor

Triple Exponential Smoothing (Holts-Winters ES)

Also known as **Holts-Winters ES**, Triple Exponential Smoothing take into account the seasonality in addition to the trend.

$$y_0 = x_0$$

$$y_t = \alpha \frac{x_t}{c_t - L} + (1 - \alpha)(y_{t-1} + b_{t-1})$$

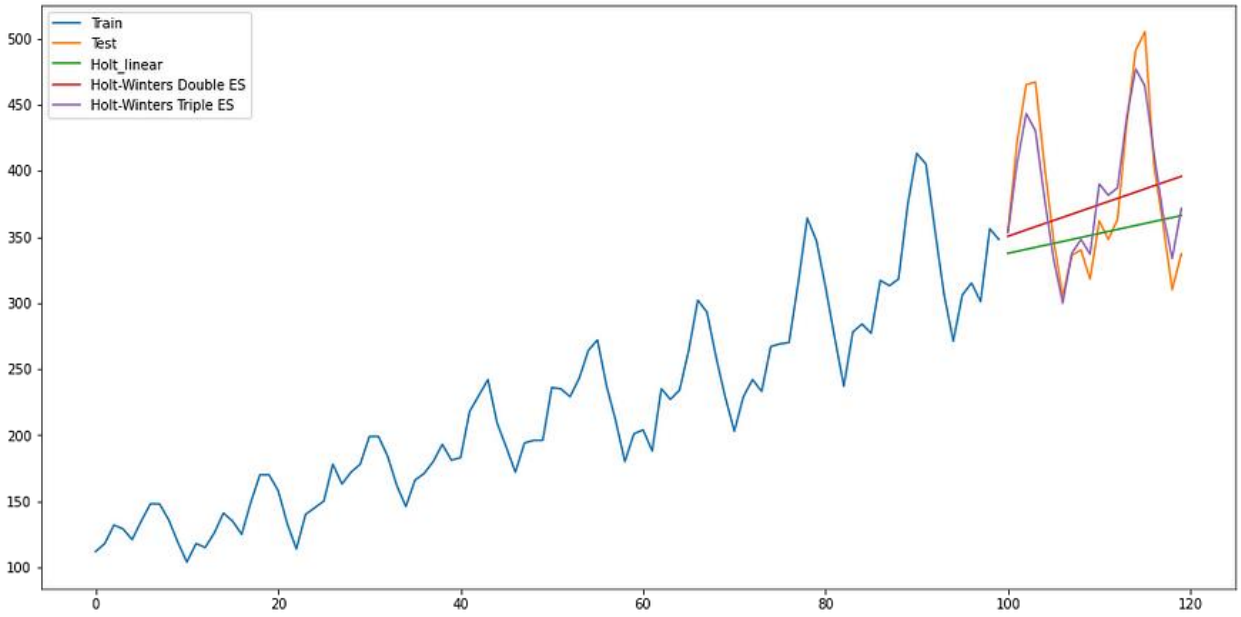
$$b_t = \beta(y_t - y_{t-1}) + (1 - \beta)y_{t-1}$$

$$c_t = \gamma \frac{x_t}{y_t} + (1 - \gamma)c_{t-L}$$

where

- $0 \leq \alpha \leq 1$: the data smoothing factor,
- $0 \leq \beta \leq 1$: the trend smoothing factor
- $0 \leq \gamma \leq 1$: the seasonal change smoothing factor
- L : cycle of seasonal change

Using these exponential smoothing's to predict future value we get the following:



The MAE for these are:

- MAE Simple ES: 50.86

- MAE Double ES: 52.62
- MAE Triple ES: 18.44

Notice that the linear and double exponential smoothing is doing worst than the Null model while the Holt-Winter is doing very well on this dataset.

ARIMA

Also known as the classical Box-Jenkins methodology, ARIMA is the statistical approach created specifically for time series.

ARIMA is part of the state-space model (SSM). SSM also includes the exponential smoothing technique we have seen earlier. It is based on a structural analysis of the problem that can be described by different components like trend, seasonality, cycle. It is suitable for an approach where the structure of the time series is well-understood.

ARIMA has 3 distinct structures as follow:

- **AR** : Auto Regression — uses the dependent relationship between an observation and some number of lagged observations. The lag n value is the previous n value of the observation.
- **I** : Integrated — uses the differencing of raw observations in order to make the time series stationary.

- **MA** : Moving Average — uses the dependency between an observation and a residual error from a moving average model applied to lag observations.

In the library that we will be using, the notation is ARIMA(p,d,q) where:

- p: (AR) The number of lag observations included in the model, also called the lag order.
- d: (I) The number of times that the raw observations are differenced, also called the degree of differencing.
- q: (MA) The size of the moving average window, also called the order of moving average.

Since we will be applying the seasonality, the API we will be using is SARIMAX. The S means Seasonality where X means eXogenous which allows for the inclusion of other data besides the univariate values.

ARIMA Parameters Search

We can import the following library: *sm.tsa.statespace.SARIMAX*. Examples of parameter combinations for Seasonal ARIMA will have p,d,q for ARIMA, then for S in seasonality, there are p,d,q, and seasonal value. For example:

SARIMAX: (0, 1, 0) x (0, 1, 1, 12) where 12 is the seaonsal value.

You may analyze for the *p*, *d*, *q* based on the dataset or use grid search with *pmdarima* library which does the grid search for us more efficiently.

You will need to install the package first then import and run the method as the following:

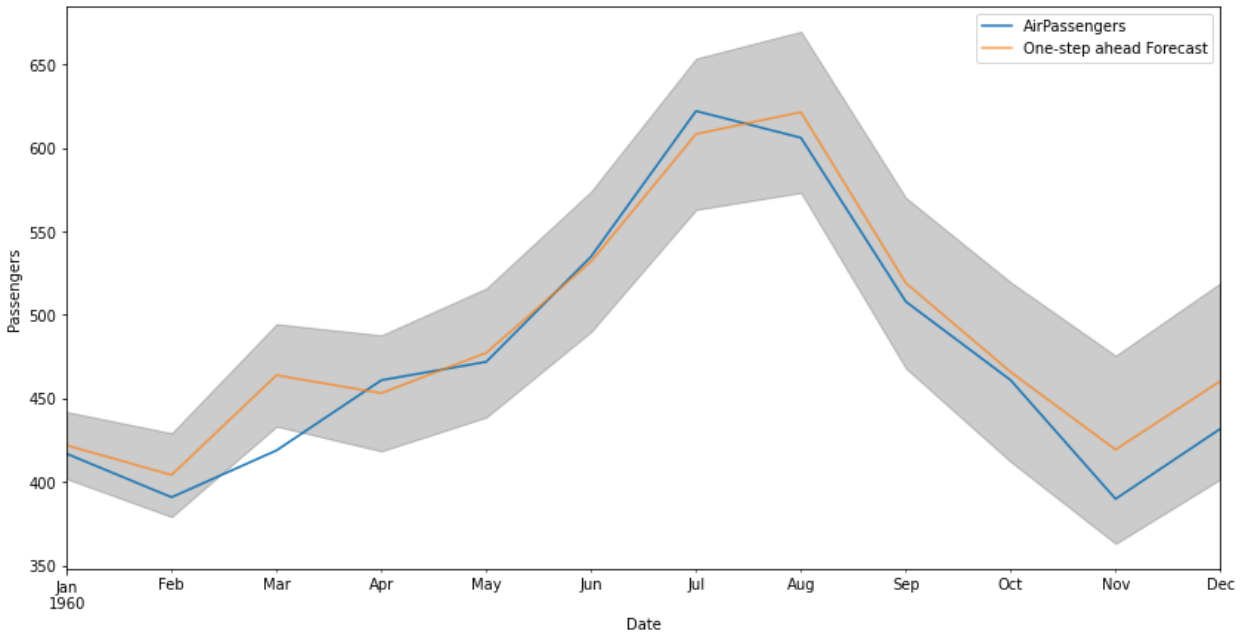
```
# required restart of the runtime
#!pip install pmdarima# Import the library, require install and restart
from pmdarima import auto_arimastepwise_fit = auto_arima(data['AirPassengers'],
  start_p = 1, start_q = 1,
  max_p = 3, max_q = 3, m = 12,
  start_P = 0, seasonal = True, …)
```

You can also do a manual grid search yourself. The code is included in the notebook by finding the smallest AIC value.

We use the following hyperparameters as input to the ARIMA algorithm:

ARIMA(1, 1, 1)x(1, 1, 1, 12)12 — AIC:803.3627295936407

We got the following prediction with the MAE of 15.22.



The gray area indicates the confidence level. As we predict further time steps, the range gets wider. Since this is one-step ahead prediction, at each time step we use the prediction value as the input for the next time step prediction.

ML: Classical Supervised Learning Approaches

We'll explore two basic supervised learning techniques:

1. Linear Regression
2. Decision Tree with XGBoost (Gradient Boosted Tree)

Generating Features

In earlier statistical approaches like ARIMA, the system was built for time series data. We can just feed the data directly. But for these ML approaches,

we need to generate features that the algorithms can use to account for time ordering. Some of the features include:

- lags : value at time step t-N
- amplitude: amplitude from the mean value
- max/min value: max or min value at that window
- value > 1 std : count of value great than 1 standard deviation

For our case, using lag values for our features is sufficient. To generate lag features we just use the data and shift by one value per lag. As an example, we want to create 2 lags feature from this data [2, 4, 6, 8, 7, 9], we would have:

```
lag2  lag1  y
— — — — — — -
nan   nan   2
nan   2     4
2     4     6
4     6     8
6     8     7
8     7     9
```

Linear Regression

We will start with linear regression. We will fit the data with linear regression with 12 lags values as our features since we knew that our seasonality is 12 months. So the lags f01 to f12 are the features and y is the actual value:

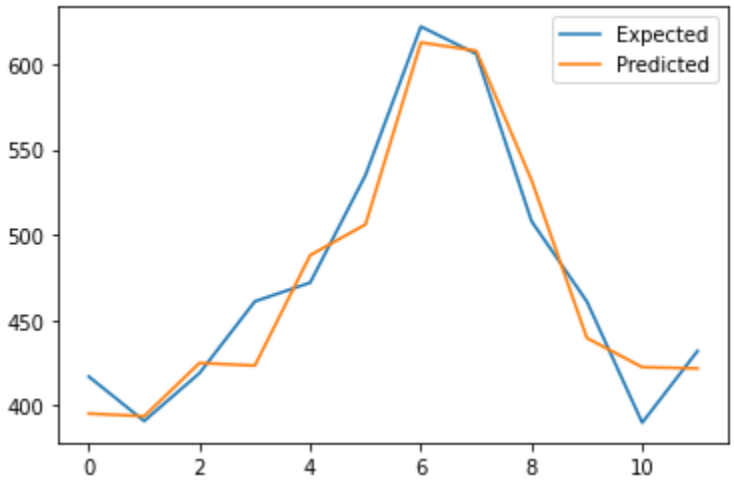
```
 f12  f11  f10  f09  f08  f07  f06  f05  f04  f03  f02  f01    y
-----------------------------------------------------------------
[nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, 112],
[nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, 112, 118],
[nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, 112, 118, 132],
[nan, nan, nan, nan, nan, nan, nan, nan, nan, 112, 118, 132, 129],
[nan, nan, nan, nan, nan, nan, nan, nan, 112, 118, 132, 129, 121],
```

[nan, nan, nan, nan, nan, nan, nan, 112, 118, 132, 129, 121, 135],
[nan, nan, nan, nan, nan, nan, 112, 118, 132, 129, 121, 135, 148],
[nan, nan, nan, nan, nan, 112, 118, 132, 129, 121, 135, 148, 148],
[nan, nan, nan, nan, 112, 118, 132, 129, 121, 135, 148, 148, 136],
[nan, nan, nan, 112, 118, 132, 129, 121, 135, 148, 148, 136, 119],
[nan, nan, 112, 118, 132, 129, 121, 135, 148, 148, 136, 119, 104],
[nan, 112, 118, 132, 129, 121, 135, 148, 148, 136, 119, 104, 118],
[112, 118, 132, 129, 121, 135, 148, 148, 136, 119, 104, 118, 115],
[118, 132, 129, 121, 135, 148, 148, 136, 119, 104, 118, 115, 126],
[132, 129, 121, 135, 148, 148, 136, 119, 104, 118, 115, 126, 141],…

Our first input will start with the 13th row that has the complete 12 feature values without *nan*.

```
from sklearn import linear_modelmodel = linear_model.LinearRegression()
model.fit(trainX, trainy)# make a one-step prediction
yhat = model.predict(asarray([testX]))
```

We see the result with MAE of 17.69.

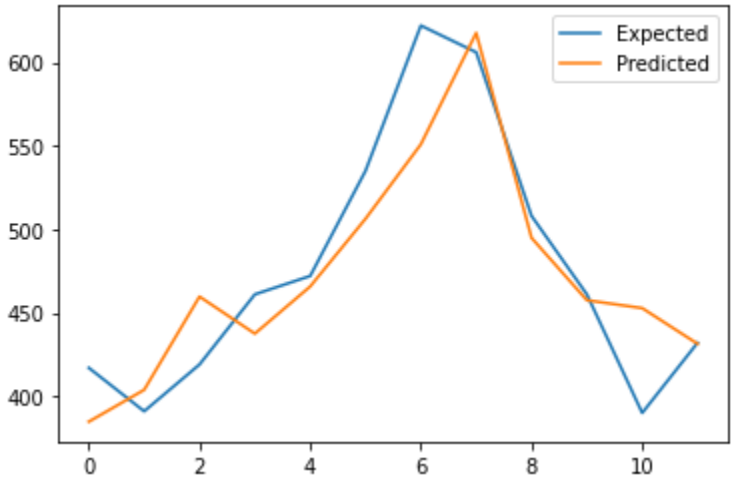


XGBoost

With the same technique, we can replace the algorithm with XGBoost.

```
from xgboost import XGBRegressor
model = XGBRegressor(objective='reg:squarederror', n_estimators=500)
model.fit(trainX, trainy)# make a one-step prediction
yhat = model.predict(asarray([testX]))
```

We get an MAE of 25.58.



This shows that the linear regression is pretty good with this simple dataset. Although XGBoost is known to do very well in many of Kaggle competition datasets.

Deep Learning with RNN — LSTM

We now look at deep learning approaches with RNN. Instead of using the plain RNN, we will use Long Short-Term Memory (LSTM). LSTM improved over the plain RNN by helping to mitigate the vanishing and exploding gradient issue.

In the preprocessing step, we also added MinMaxScaler to ensure our data is in the same range (0–1). This seems to improve our prediction.

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(x_train)x_train = scaler.transform(x_train)
x_valid = scaler.transform(x_valid)
```

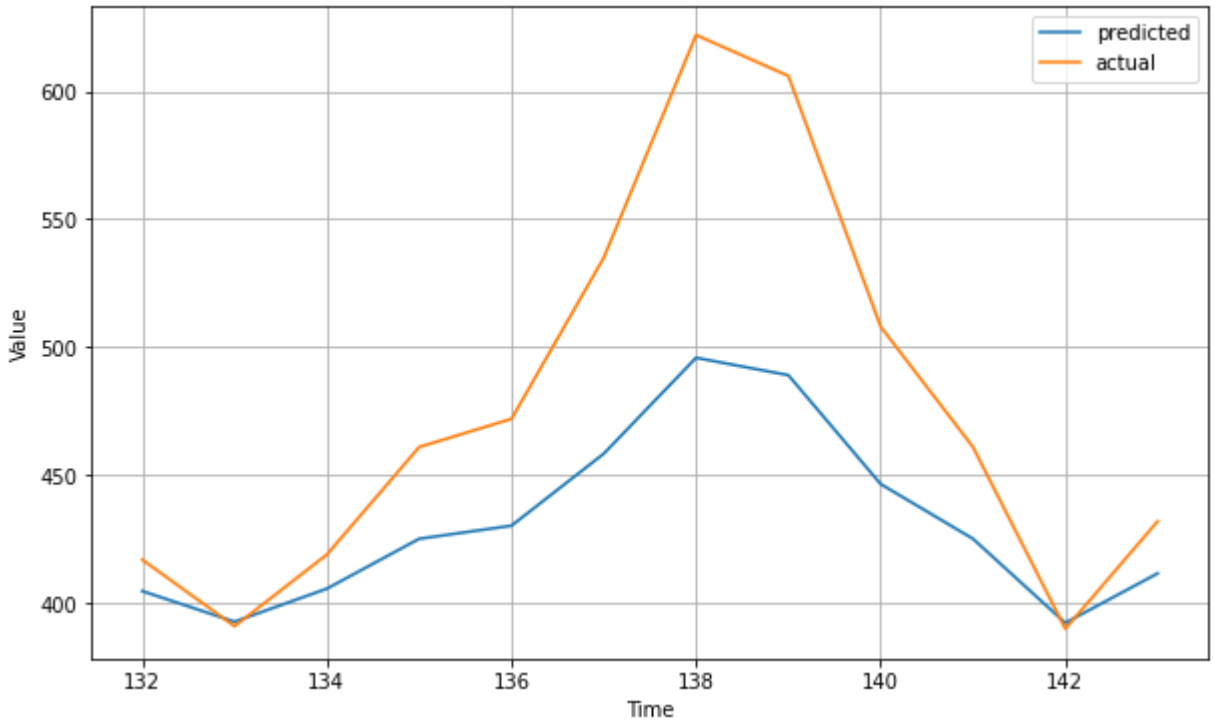
Model:

```
tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(260,
  activation='relu', return_sequences=True)),
  tf.keras.layers.Dropout(0.15),
  tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(248)),
  tf.keras.layers.Dropout(0.05),
  tf.keras.layers.Dense(1), ...)
```

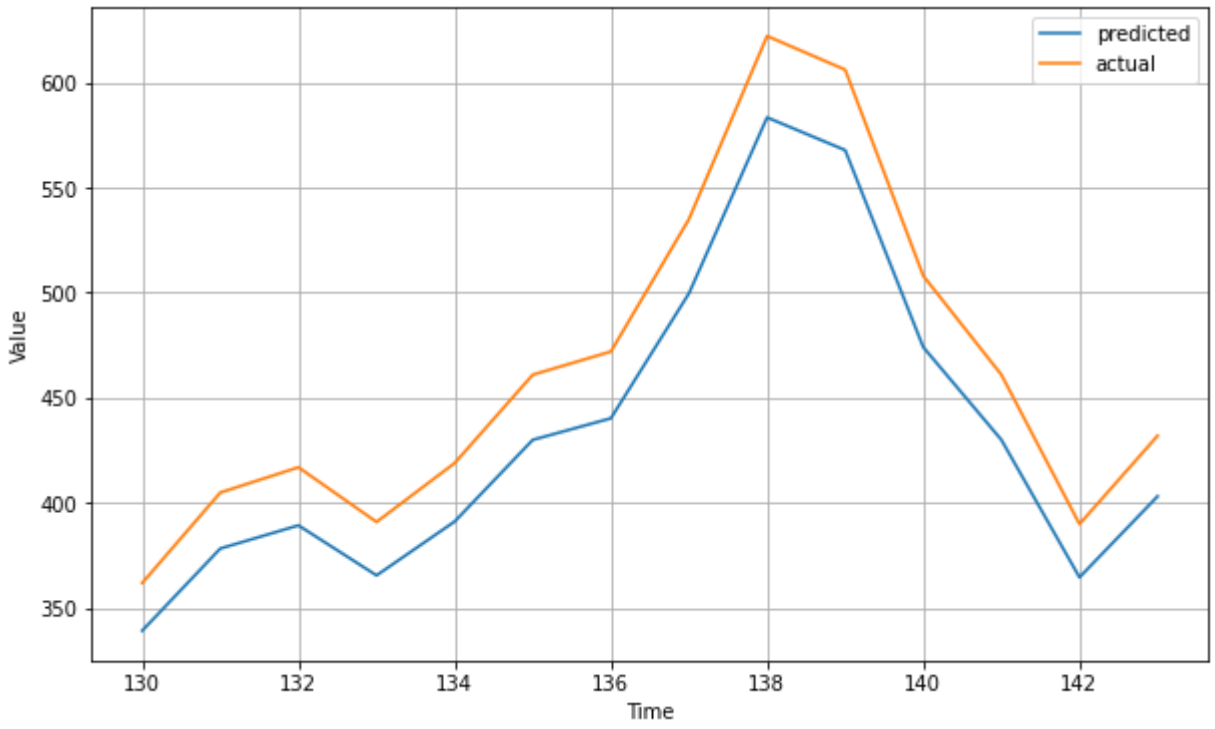
With Adam optimizer:

```
optimizer = tf.keras.optimizers.Adam()
model.compile(loss="mse", optimizer=optimizer, metrics=["mae"])
model.fit(dataset,epochs=100,verbose=1)Epoch 100/10020/20
[==============================] — 3s 134ms/step — loss: 0.0039 — mae: 0.0507
```

The latest result in the notebook, our MAE is 47.37.



After tweaking some parameters and architecture, we were able to get the MAE on the test set to be 33.39.



References:

- https://github.com/gianfelton/12-Month-Forecast-With-LSTM/blob/master/12-Month%20Forecast%20With%20LSTM.ipynb
- https://colab.research.google.com/github/lmoroney/dlaicourse/blob/master/TensorFlow%20In%20Practice/Course%204%20-%20S%2BP/S%2BP%20Week%203%20Lesson%204%20-%20LSTM.ipynb

CNN and LSTM

We can add a CNN layer in front of LSTM to help encode the input. This showed an improvement of LSTM alone.[]

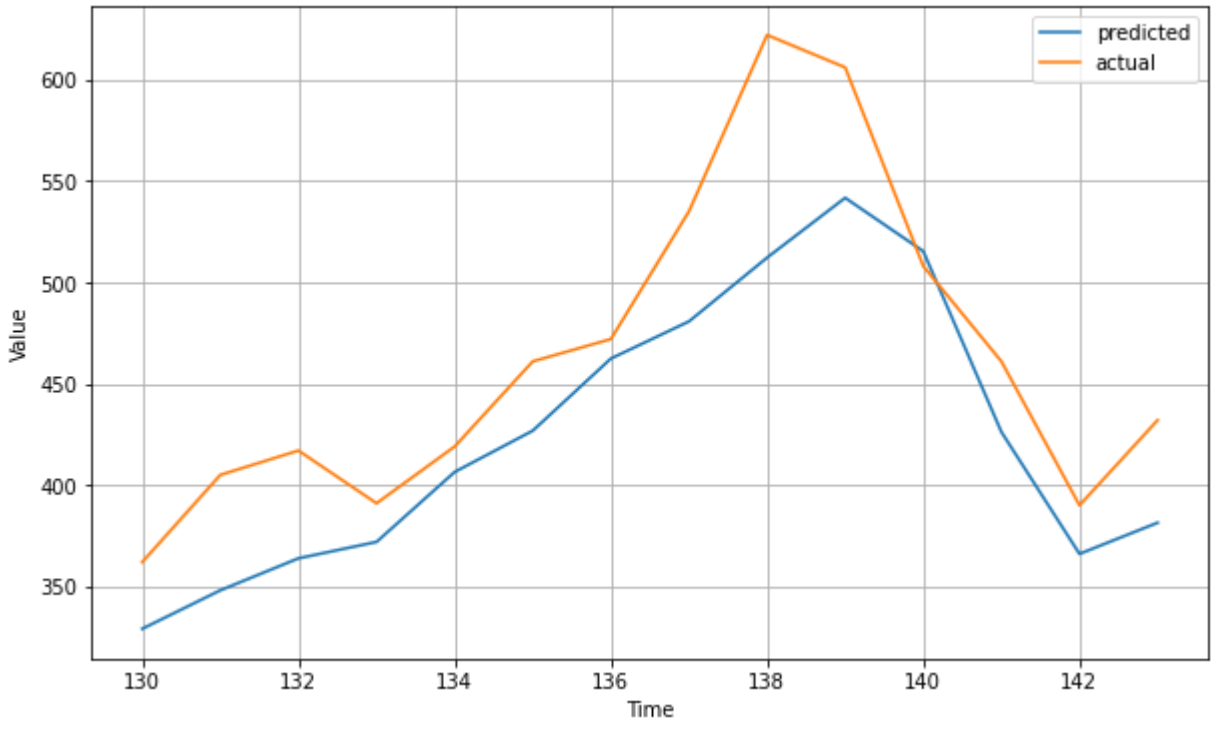
For optimizer, we use momentum.

```
tf.keras.layers.Conv1D(filters=60, kernel_size=5,
  strides=1, padding=”causal”,
  activation=”relu”,
  input_shape=[None, 1]),
  tf.keras.layers.LSTM(200, return_sequences=True), #60
  tf.keras.layers.LSTM(200, return_sequences=True),
  tf.keras.layers.Dropout(0.15),
  tf.keras.layers.Dense(200, activation=”relu”),
  tf.keras.layers.Dense(10, activation=”relu”),
  tf.keras.layers.Dense(1),
```

Output:

```
Epoch 150/1505/5 [==============================] — 0s 53ms/step — loss: 23.7124 — mae: 24.2093
```

The MAE for the test set is 40.25. The error is larger than LSTM above. We used a scaler to improve the performance in the LSTM approach.



Reference: https://colab.research.google.com/github/lmoroney/dlaicourse/blob/master/TensorFlow%20In%20Practice/Course%204%20-%20S%2BP/S%2BP%20Week%204%20Lesson%205.ipynb

TCN

Temporal Convolutional Networks (TCN) is a variation of Convolutional Neural Networks for sequence modeling tasks, by combining aspects of RNN and CNN architectures. This model performs a multi-horizon prediction.

TCN is based on two principles:

1. the network produces an output of the same length as the input
2. there can be no leakage from the future into the past.

To accomplish the first point, the TCN uses a 1D fully-convolutional network (FCN) architecture.

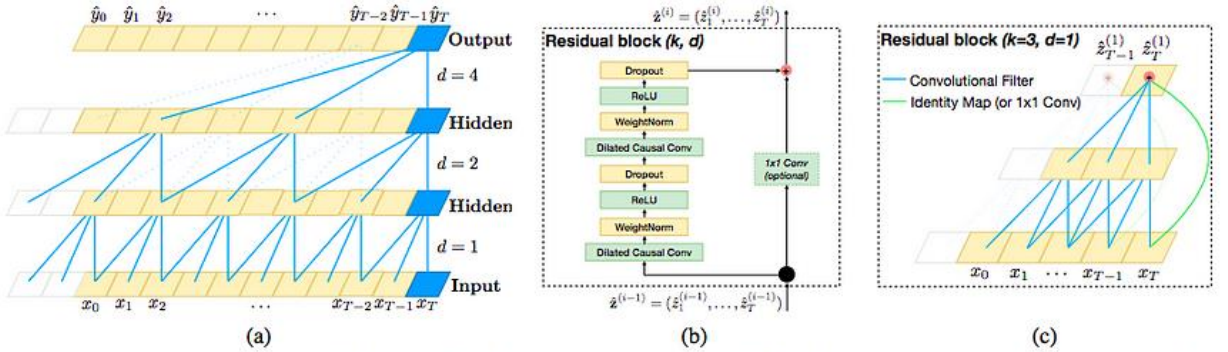


Figure 1. Architectural elements in a TCN. (a) A dilated causal convolution with dilation factors $d = 1, 2, 4$ and filter size $k = 3$. The receptive field is able to cover all values from the input sequence. (b) TCN residual block. An 1x1 convolution is added when residual input and output have different dimensions. (c) An example of residual connection in a TCN. The blue lines are filters in the residual function, and the green lines are identity mappings.

"An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling", Shaojie Bai, J. Zico Kolter, Vladlen Koltun, 2018.

The paper showed improved performance over LSTM, GRU, and RNN in many of the datasets.

Sequence Modeling Task	Model Size (≈)	Models			
		LSTM	GRU	RNN	TCN
Seq. MNIST (accuracy^h)	70K	87.2	96.2	21.5	**99.0**
Permuted MNIST (accuracy)	70K	85.7	87.3	25.3	**97.2**
Adding problem T=600 (loss$^\ell$)	70K	0.164	**5.3e-5**	0.177	**5.8e-5**
Copy memory T=1000 (loss)	16K	0.0204	0.0197	0.0202	**3.5e-5**
Music JSB Chorales (loss)	300K	8.45	8.43	8.91	**8.10**
Music Nottingham (loss)	1M	3.29	3.46	4.05	**3.07**
Word-level PTB (perplexity$^\ell$)	13M	**78.93**	92.48	114.50	88.68
Word-level Wiki-103 (perplexity)	-	48.4	-	-	**45.19**
Word-level LAMBADA (perplexity)	-	4186	-	14725	**1279**
Char-level PTB (bpc$^\ell$)	3M	1.36	1.37	1.48	**1.31**
Char-level text8 (bpc)	5M	1.50	1.53	1.69	**1.45**

TCN implementations for different ML libraries can be found here:

- Pytorch: http://github.com/locuslab/TCN
- Keras: https://github.com/philipperemy/keras-tcn
- Tensorflow: https://github.com/Baichenjia/Tensorflow-TCN

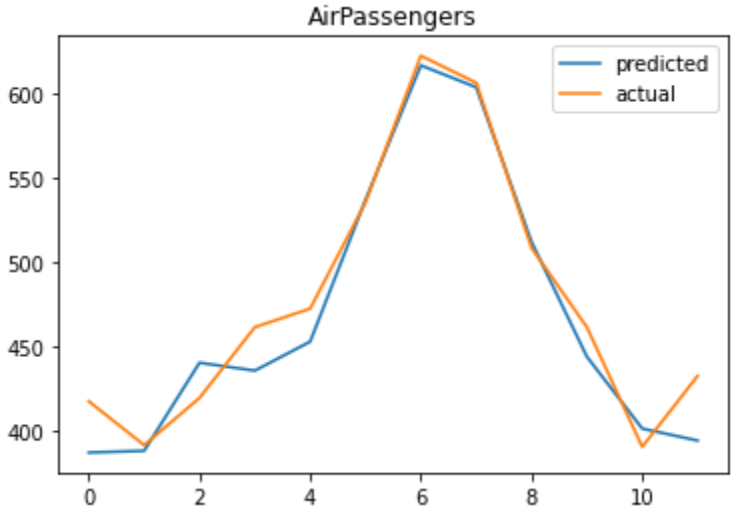
Our code uses the keras-tcn library as follow:

```
# installing tcn
#!pip install keras-tcnfrom tcn import TCN, tcn_full_summaryi = Input(shape=(lookback_window, 1))
m = TCN()(i)
m = Dense(1, activation='linear')(m)model = Model(inputs=[i], outputs=[m])
model.summary()
model.compile('adam', 'mae')model.fit(x_train, y_train, epochs=150, verbose=0)
```

Output:

```
Model: "functional_25"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_13 (InputLayer)        [(None, 12, 1)]           0
_________________________________________________________________
tcn_12 (TCN)                 (None, 64)                91136
_________________________________________________________________
dense_12 (Dense)             (None, 1)                 65
=================================================================Total
params: 91,201
Trainable params: 91,201
Non-trainable params: 0
```

We got the MAE of 15.01 for the AirPassenger dataset.



References:

- "An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling", Shaojie Bai, J. Zico Kolter, Vladlen Koltun (https://arxiv.org/abs/1803.01271)

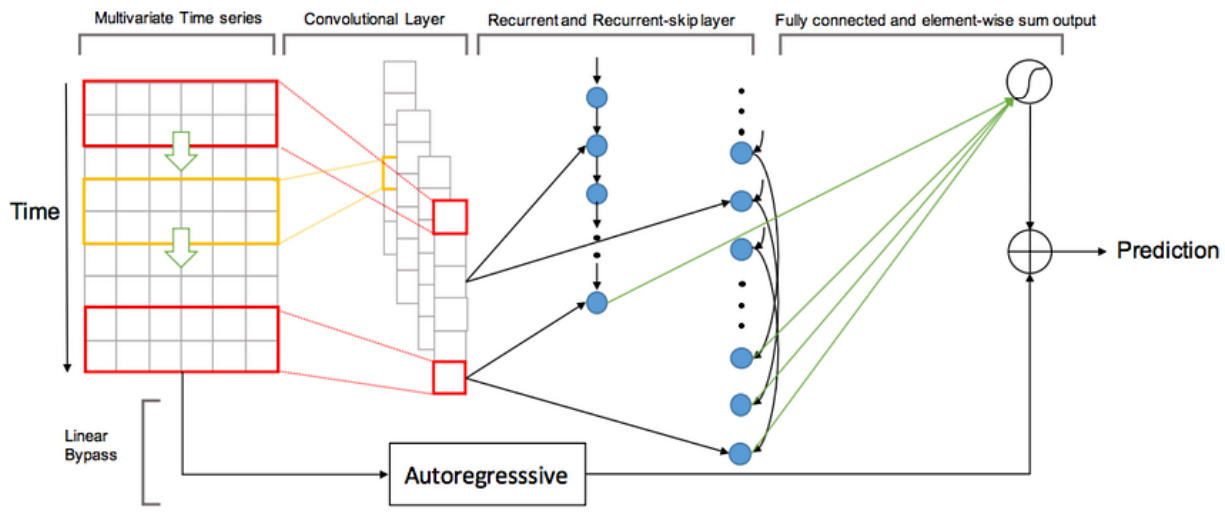
- https://towardsdatascience.com/farewell-rnns-welcome-tcns-dd76674707c8
- https://dida.do/blog/temporal-convolutional-networks-for-sequence-modeling

Long- and Short-Term Temporal Network (LSTNet)

Multivariate time series forecasting often faces a major research challenge, that is, how to capture and leverage the dynamics dependencies among multiple variables.

The shortcoming of classical ARIMA:

- ARIMA models are adaptive to various exponential smoothing techniques and flexible enough to subsume other types of time series models. However, ARIMA models, are rarely used in high dimensional multivariate time series forecasting.
- vector autoregression (VAR) is arguably the most widely used models in multivariate time series due to its simplicity. The model capacity of VAR grows linearly over the temporal window size and quadratically over the number of variables.



"Modeling Long- and Short-Term Temporal Patterns with Deep Neural Networks", Lai et al., SIGIR 2018.

LSTNet Components:

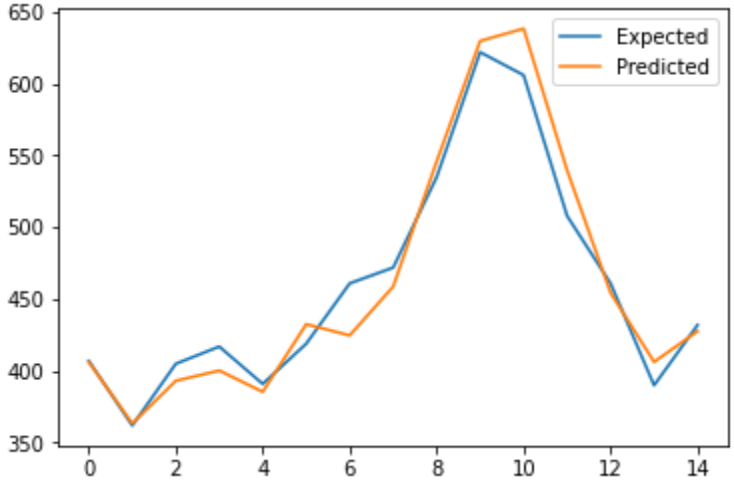
1. CNN with k-th filter with RELU
2. RNN (GRU) — with RELU
3. RNN Skip Connection
4. Temporal Attention Layer
5. Autoregression with highway

Model consisted of: (CNN, and RNN(GRU) and dropout layer)

```
class LSTNet(nn.Module):  def __init__(self, args, data):
    super(LSTNet, self).__init__()
    self.P = args.window;
    self.m = data.m
    self.hidR = args.hidRNN;
    self.hidC = args.hidCNN;
    self.hidS = args.hidSkip;
    self.Ck = args.CNN_kernel;
```

```
…
self.conv1 = nn.Conv2d(1, self.hidC, kernel_size = (self.Ck, self.m));
self.GRU1 = nn.GRU(self.hidC, self.hidR);
self.dropout = nn.Dropout(p = args.dropout);
```

The MAE is 13.96 for the AirPassengers dataset.



References:

- https://modelzoo.co/model/lstnet
- https://opringle.github.io/2018/01/05/deep_learning_multivariate_ts.html
- "Modeling Long- and Short-Term Temporal Patterns with Deep Neural Networks", Lai et al., SIGIR 2018 (https://arxiv.org/pdf/1703.07015.pdf)
  Code: https://github.com/laiguokun/LSTNet

N-BEATS

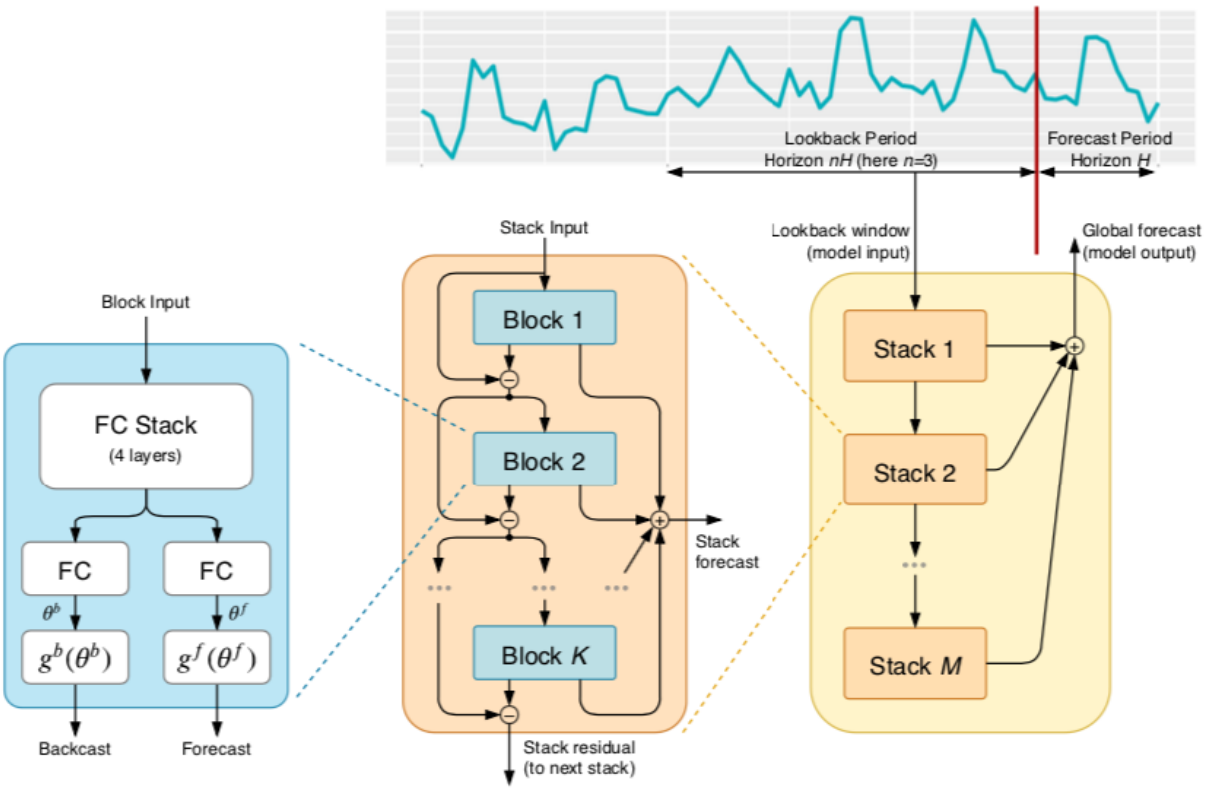
Neural Basis Expansion Analysis for Interpretable Time Series (N-BEATS) is built for:

- multi-horizons
- multiple series
- interpretabilities: seasonality and trend.

“Pure DL using no time-series specific components outperforms well-established statistical approaches on M3, M4 and TOURISM datasets (on M4, by 11% over statistical benchmark, by 7% over the best statistical entry, and by 3% over the M4 competition winner),” N-BEATS.

Architectural design:

- simple and generic, yet expressive (deep).
- not rely on time series specific feature engineering or input scaling for pure DL architecture in time series forecasting.
- explore interpretability, extendable towards making its outputs human interpretable.

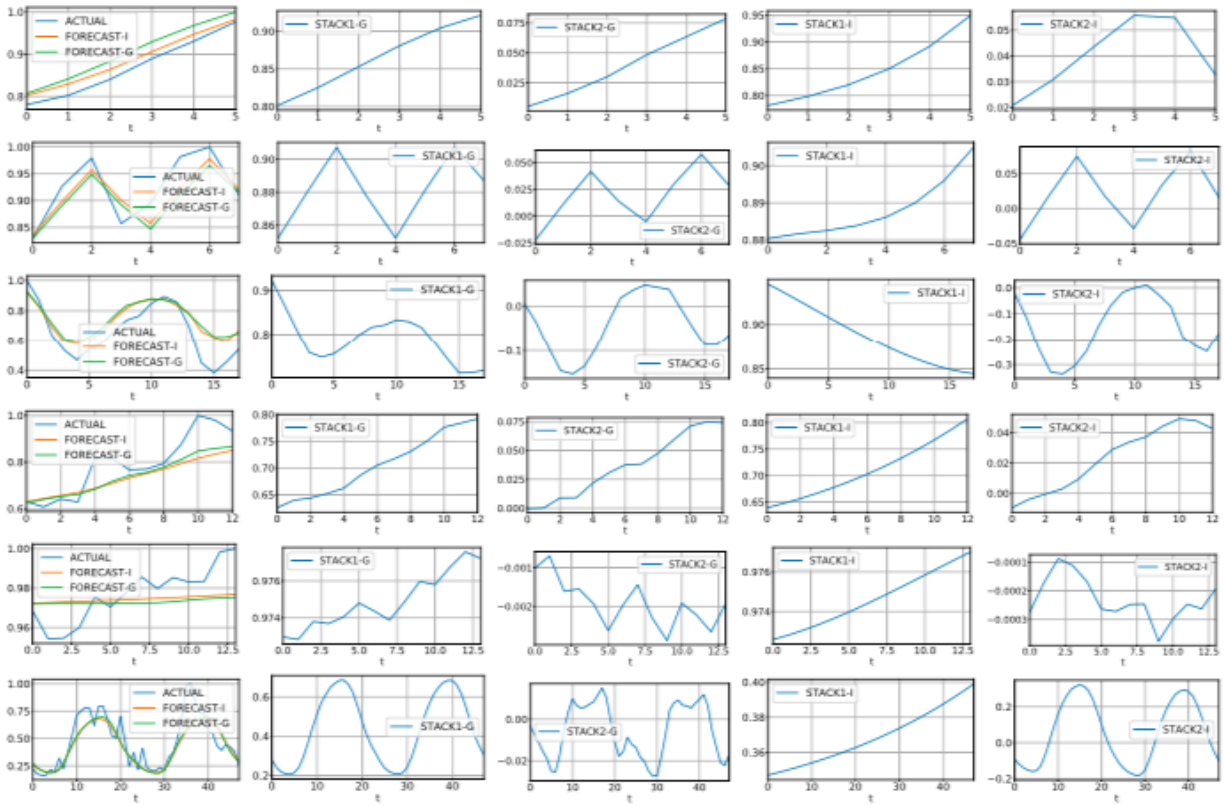


N-BEATS: Neural Basis Expansion Analysis for Interpretable Time Series Forecasting, Oreshkin, et al.,2020.

Model types in the notations:

- I : interpretability
- G : general
- G+I : ensembles with I and G

Interpretabilities is provided with graph of trend and seasonality like below:



(a) Combined (b) Stack1-G (c) Stack2-G (d) StackT-I (e) StackS-I

N-BEATS: Neural Basis Expansion Analysis for Interpretable Time Series Forecasting, Oreshkin, et al.,2020.

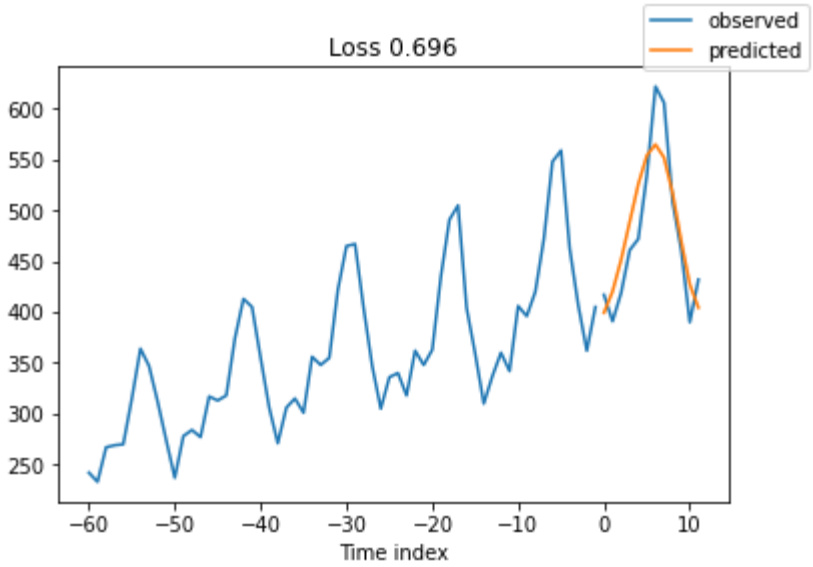
Table 1: Performance on the M4, M3, TOURISM test sets, aggregated over each dataset. Evaluation metrics are specified for each dataset; lower values are better. The number of time series in each dataset is provided in brackets.

M4 Average (100,000)			M3 Average (3,003)		TOURISM Average (1,311)	
	sMAPE	OWA		sMAPE		MAPE
Pure ML	12.894	0.915	Comb S-H-D	13.52	ETS	20.88
Statistical	11.986	0.861	ForecastPro	13.19	Theta	20.88
ProLogistica	11.845	0.841	Theta	13.01	ForePro	19.84
ML/TS combination	11.720	0.838	DOTM	12.90	Stratometrics	19.52
DL/TS hybrid	11.374	0.821	EXP	12.71	LeeCBaker	19.35
N-BEATS-G	11.168	0.797		12.47		**18.47**
N-BEATS-I	11.174	0.798		12.43		18.97
N-BEATS-I+G	**11.135**	**0.795**		**12.37**		18.52

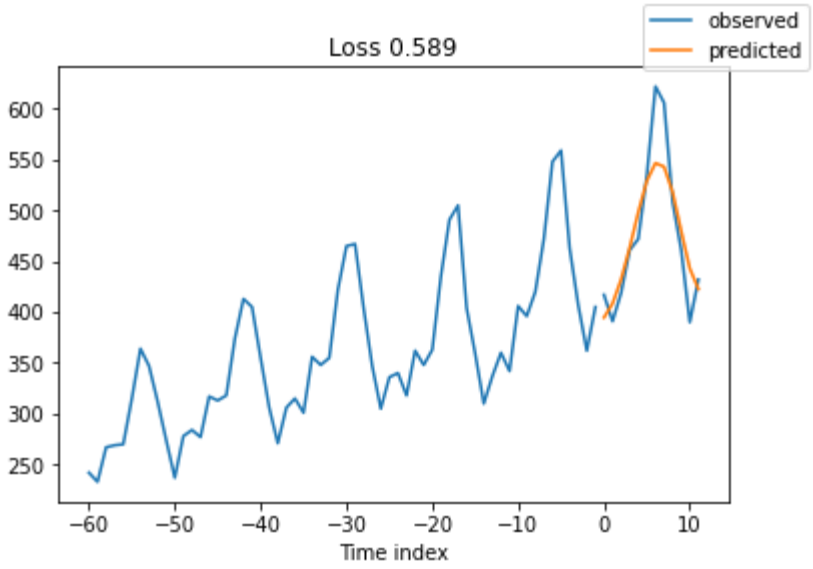
```
# install the libraries
#!pip install pytorch_lightning
#!pip install pytorch_forecastingimport pytorch_lightning as pl
from pytorch_lightning.callbacks import EarlyStoppingfrom pytorch_forecasting import
TimeSeriesDataSet, NBeats, Baseline
```

```
from pytorch_forecasting.data import NaNLabelEncoder
from pytorch_forecasting.data.examples import generate_ar_data
from pytorch_forecasting.metrics import SMAPEtrainer = pl.Trainer(
  max_epochs=100,
  gpus=0,
  weights_summary="top",
  gradient_clip_val=0.1,
  callbacks=[early_stop_callback],
  limit_train_batches=30,
)net = NBeats.from_dataset(training, learning_rate=4e-3,
  log_interval=10, log_val_interval=1, weight_decay=1e-2,
  widths=[32, 512], backcast_loss_ratio=1.0)trainer.fit(net, train_dataloader=train_dataloader,
  val_dataloaders=val_dataloader,
)
```

Performance on AirPassengers dataset,



We have to finetune a lot of hyper parameters. We added monthly hint and show a little improvement.



After some finetune we get the MAE of 26.42.

Temporal Fusion Transformers (TFT)

“By interpreting attention patterns, TFT can provide insightful explanations about temporal dynamics, and do so while maintaining state-of-the-art performance on a variety of datasets,” Lim et al.,2019.

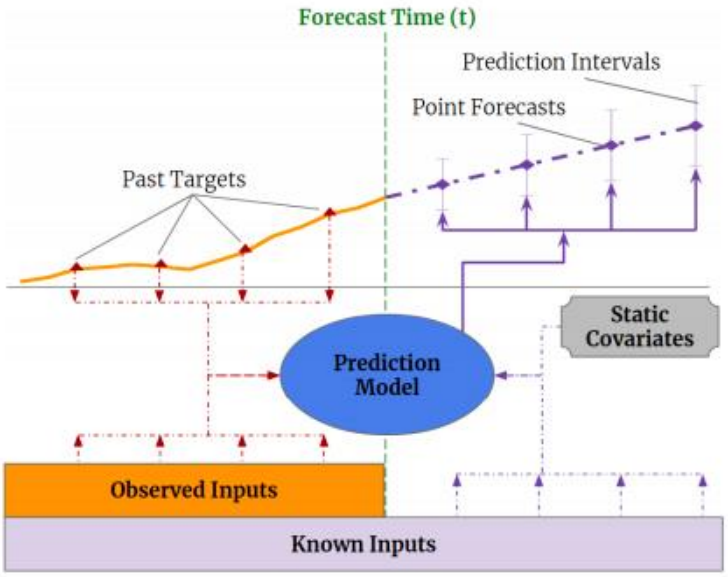


Figure 1: Illustration of multi-horizon forecasting with static covariates, past-observed and apriori-known future time-dependent inputs.

Temporal Fusion Transformers for Interpretable Multi-horizon Time Series Forecasting, Lim et al., 2019.

For interpretability, TFT uses the attention mechanism as follow:

- Attention mechanisms are used to identify salient portions of input for each instance using the magnitude of attention weights.
- TFT uses separate encoder-decoder attention for static features at each time step on top of the self-attention to determine the contribution time-varying inputs.

Here are the performance cited in the paper.

Table 2: P50 and P90 quantile losses on a range of real-world datasets. Percentages in brackets reflect the increase in quantile loss versus TFT (lower q-Risk better), with TFT outperforming competing methods across all experiments, improving on the next best alternative method (underlined) between 3% and 26%.

	ARIMA	ETS	TRMF	DeepAR	DSSM
Electricity	0.154 (*+180%*)	0.102 (*+85%*)	0.084 (*+53%*)	0.075 (*+36%*)	0.083 (*+51%*)
Traffic	0.223 (*+135%*)	0.236 (*+148%*)	0.186 (*+96%*)	0.161 (*+69%*)	0.167 (*+76%*)
	ConvTrans	**Seq2Seq**	**MQRNN**	**TFT**	
Electricity	0.059 (*+7%*)	0.067 (*+22%*)	0.077 (*+40%*)	**0.055***	
Traffic	0.122 (*+28%*)	0.105 (*+11%*)	0.117 (*+23%*)	**0.095***	

(a) P50 losses on simpler univariate datasets.

	ARIMA	ETS	TRMF	DeepAR	DSSM
Electricity	0.102 (*+278%*)	0.077 (*+185%*)	-	0.040 (*+48%*)	0.056 (*+107%*)
Traffic	0.137 (*+94%*)	0.148 (*+110%*)	-	0.099 (*+40%*)	0.113 (*+60%*)
	ConvTrans	**Seq2Seq**	**MQRNN**	**TFT**	
Electricity	0.034 (*+26%*)	0.036 (*+33%*)	0.036 (*+33%*)	**0.027***	
Traffic	0.081 (*+15%*)	0.075 (*+6%*)	0.082 (*+16%*)	**0.070***	

(b) P90 losses on simpler univariate datasets.

The paper compares to other models (ARIMA, ETS, TRMF, DeepAR, DSSM, ConvTrans, Seq2Seq, and MQRNN). Here is some information on the different models.

- ETS (Error, Trend, Seasonal) method is an approach method for forecasting time series univariate. (https://otexts.com/fpp2/arima-ets.html)
- DeepAR uses stacked LSTM layers to generate parameters of one-step-ahead Gaussian predictive distributions. This will be described later.
- Deep State-Space Models (DSSM) utilize LSTMs to generate parameters of a predefined linear state-space model with predictive

distributions produced via Kalman filtering — with extensions for multivariate time series data.

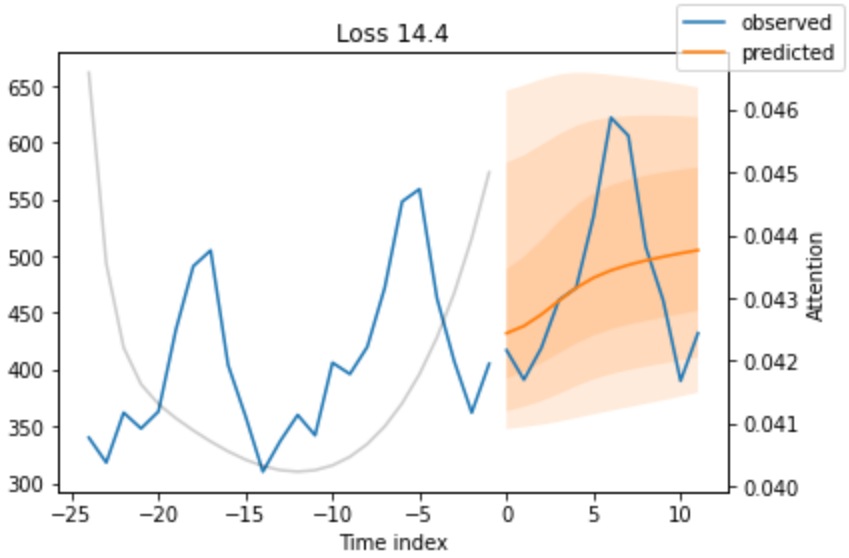
- ConvTrans: the Transformer-based architecture with the use of convolutional layers for local processing and a sparse attention mechanism to increase the size of the receptive field during forecasting. (https://arxiv.org/abs/1907.00235)

- The Multi-horizon Quantile Recurrent Forecaster (MQRNN) uses LSTM as convolutional encoders to generate context vectors that are fed into multi-layer perceptrons (MLPs) for each horizon.

Here is the main part of the code. See the notebook for a complete working code.

```
!pip install pytorch_lightning
!pip install pytorch_forecastingimport pytorch_lightning as pl
from pytorch_forecasting import TimeSeriesDataSet, TemporalFusionTransformer, Baseline
from pytorch_forecasting.data import GroupNormalizer
from pytorch_forecasting.metrics import PoissonLoss, QuantileLoss, SMAPE
from pytorch_forecasting.models.temporal_fusion_transformer.tuning import
optimize_hyperparameters…trainer = pl.Trainer( gpus=0, gradient_clip_val=0.1)tft =
TemporalFusionTransformer.from_dataset(
 training,
 learning_rate=0.03,
 hidden_size=16,  # most important hyperparameter
 attention_head_size=1,
 dropout=0.1,  # between 0.1 and 0.3 are good value
 hidden_continuous_size=8,  # set to <= hidden_size
 output_size=7,  # 7 quantiles by default
 loss=QuantileLoss(),
 # reduce learning rate if no improvement in validation loss
 reduce_on_plateau_patience=4,
)
```

I tried to use AirPassengers dataset but it didn't turn out well. There are several parameters that might have not tuned properly.



Commercial Applications

Facebook Prophet

In 2017, Facebook open sourced a project called Prophet. "Prophet is a modular regression model with interpretable parameters that can be intuitively adjusted by analysts". It decomposed the series into three main model components:

1. trend
2. seasonality
3. holidays

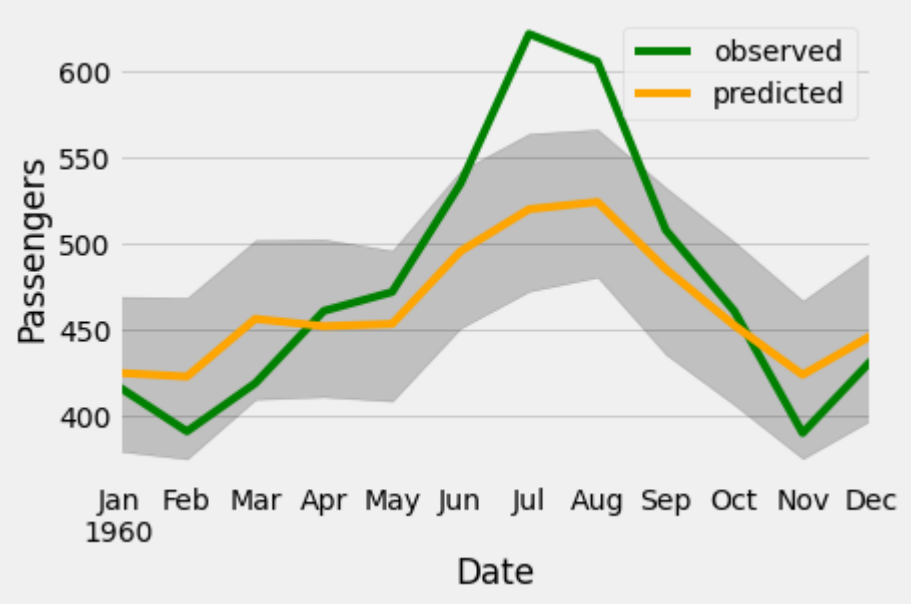
Use only time as a regressor but possibly several linear and nonlinear functions of time as components.

Has a number of advantages:

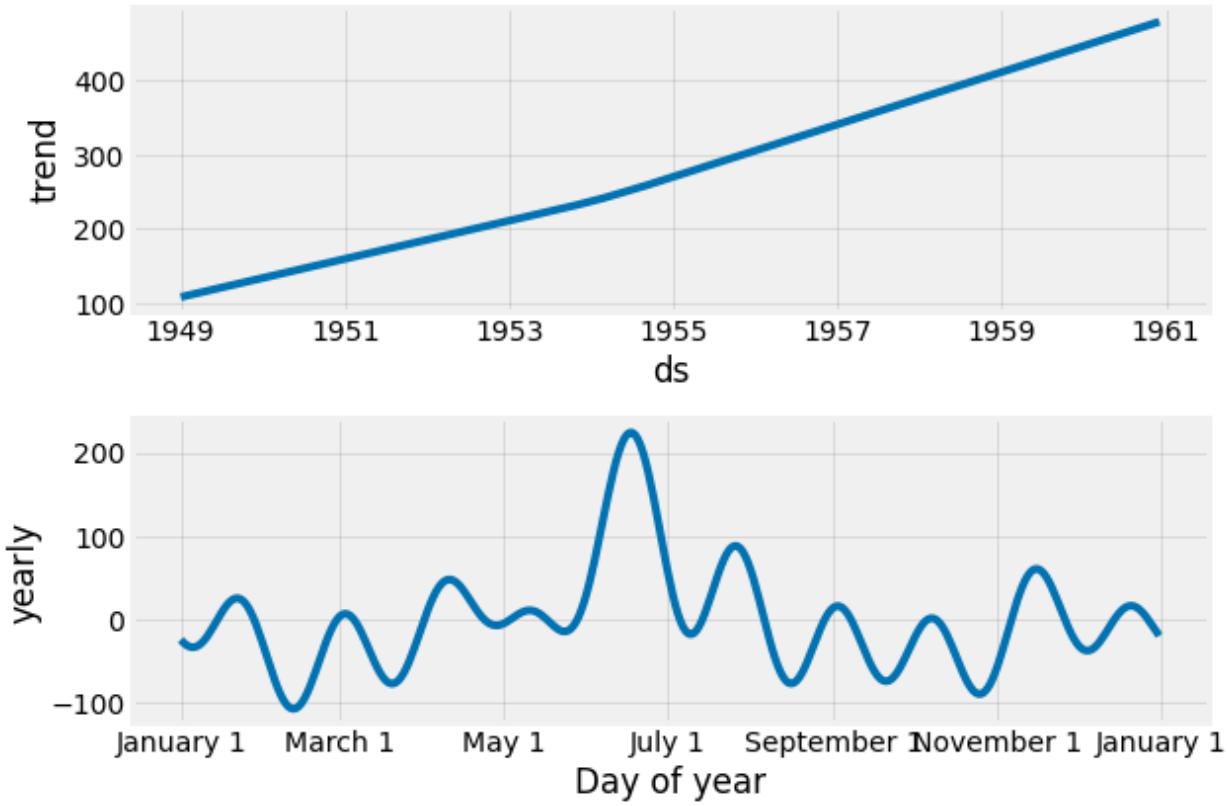
- Flexibility: We can easily accommodate seasonality with multiple periods
- Fitting is very fast
- Has easily interpretable parameters

```
from fbprophet import Prophetmy_model = Prophet(interval_width=0.98, daily_seasonality=False, weekly_seasonality=False, yearly_seasonality=True,)my_model.fit(train_data)future_dates = my_model.make_future_dataframe(periods=12, freq='MS')forecast = my_model.predict(future_dates)
```

With the AirPassenger dataset, we get an MAE of 33.92.



You can see the explainability aspect with these:



Amazon DeepAR

“DeepAR, a forecasting method based on autoregressive recurrent networks, which learns such a global model from historical data of all the time series in the data set,” Salinas et al., 2019. It uses a similar LSTM-based recurrent neural network architecture. This only runs on Amazon Sagemaker but there is an implementation in PyTorch in the link below. We did not get to try out the PyTorch code.

Classical Techniques Problem:

- there is a need for forecasting thousands or millions of related time series

- Examples: forecasting the load for servers in a data center, or forecasting the demand for all products of a large retailer
- related time series can be leveraged for making a forecast for an individual time series.
- allows fitting more complex (and hence potentially more accurate) models without overfitting
- eliminate manual feature engineering and model selection steps

Features:

- the model learns seasonal behavior and dependencies on given covariates across time series
- by learning from similar items, it can provide forecasts for items with little or no history at all

Conclusion

Time series fits well with a structural approach such as ARIMA which shows the interpretability with trend and seasonality. Holt-Winters triple exponential smoothing can account for trend and seasonality and give a similar result for the simple dataset we used.

Then in the machine learning approach, it required the input to be formatted with features. The typical linear regression is not that bad for our dataset. We

also tried XGBoost which is frequently used by the Kaggle competition winners.

In deep learning, the sequence to sequence approaches like RNN and LSTM does show some promise. New architectures started to emerge just beyond CNN and RNN. TCN and LSTNet showed great performance surpassing ARIMA.

Then TFT and N-BEATS are the latest approaches that are the current state of the arts. While TFT outperformed previous approaches, N-BEATS outperformed previous winner approaches in the M4 competition.