N-beats & MLP time-series predictive analysis

18.11.2021

Introduction

The main aim of this project is the analysis of the N-BEATS neural network and MLPmodels used in time-series predictive analysis. The N-BEATS model is usually abbreviated as neural basis expansion analysis for interpretable time-series forecasting. On the other hand, MLP is defined as the multilayer perceptron for feed-forward neural networks. The comprehensive analysis of both models is done using a tourism dataset contained by csv files. The detailed analysis of both the models and the dataset is done in succeeding sections.

Dataset

The tourism dataset as mentioned in the preceding section includes 366 monthly series, 427 quarterly series and 518 yearly series. The series are defined as the columns of the provided csv file. Thus, the monthly dataset contains 366 columns, quarterly dataset contains 427 columns and yearly dataset contains 518 columns in its respective csv files. The dataset is preprocessed and manipulated in a certain way to make it compatible for the model to train at. Data preprocessing steps are described in the next sections.

Data preprocessing

The data preprocessing is described as a bulk action in the code. After reading the data from the directory, the step involves multiple preprocessing steps described below individually. This bulk data preprocessing step is represented below.

```
def data_preprocessing(file_path):
  dataframe = pd.read_csv(file_path)
  data = data_normalization(dataframe)
  data = data_interpolation(data)
  x, y = data_sequencing(data)
  x_train, y_train, x_test, y_test = data_splitting(x, y)

  return x_train, y_train, x_test, y_test
```

Data normalization

This step involves normalizing the dataset values in between the range(0, 1). The normalization is performed using the MinMaxScaler() built-in class provided by the scikit-learn library. After transformation, the array containing the data values is flattened into a 1D array for further steps.

```
def data_normalization(arr):
  scaler = MinMaxScaler()
  scaler.fit(arr)
  arr = scaler.transform(arr)

  return arr.flatten()
```

Data interpolation

This step involves removing the missing values to make the dataset more consistent. The rows containing the empty values are discarded from the flattened array returned from the preceding step.

```
def data_interpolation(data):
  data = data[~(np.isnan(data))]

  return data
```

Data sequencing

After the data is normalized and interpolated, it is divided into input and output sequences. Input sequences are defined as backcast length and output sequences are defined as forecast length.

```
def data_sequencing(data):
  x, y = [], []
  steps = 1
  for epoch in range(backcast_length, len(data)-forecast_length, steps):
    x.append(data[epoch - backcast_length:epoch])
    y.append(data[epoch:epoch + forecast_length])

  return x, y
```

Data splitting

After dividing the whole data into input and output sequences, it is the right time to divide these sequences further into train and test sets. The data splitting is performed following the ratio of 80:20 where 80 percent will be used for training purposes and the remaining 20 percent for testing or validation processes.

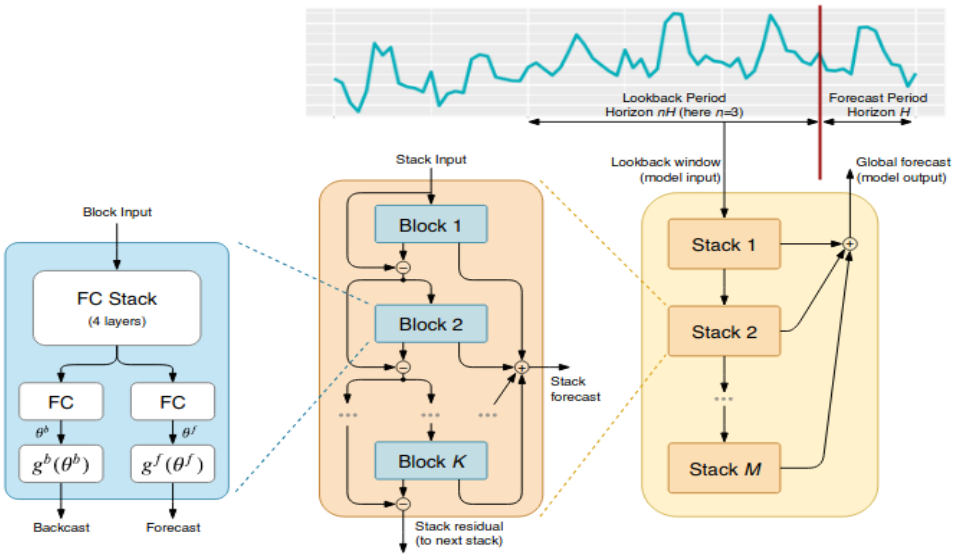
```
def data_splitting(x, y):
  c = int(len(x) * 0.8)
  x_train, y_train = x[:c], y[:c]
  x_test, y_test = x[c:], y[c:]

  return x_train, y_train, x_test, y_test
```

N-beats architecture

N-beats is based on a deep neural network architecture with a backward and forward residual links and a pool of fully-connected layers stacked upon another. This architecture is applicable in several targeted domains without any modification to the network. It possesses a number of desirable properties and is fast to train. N-beats takes the time-series data as an input where input is defined as the data points upto time 't'. Using these data points, it predicts the output 't+1'. It depends upon the lookback period where input values usually range from 2 to 7 data points. The model learns these time-series data points based on the lookback period and predicts the output upto 'H' data points called as forecast period. The architecture is made up of several building blocks based on fully connected layers and other components. These blocks are described briefly in the next section.

Architecture blocks

Input is based on the lookback period using the time-series data. This input is served to stack 1 which is made up of several fully-connected blocks. These input blocks are arranged in a Doubly Residual Stacking manner. These basic blocks are used as an input fashion for the N-beat model. The information is passed to the next connected block for processing the model and so on. At the end, output configuration is based on the forecast period. There are two main operations which are being processed in the whole architecture. Those functions are backcast and forecast; therefore, the configuration is called Double Residual Stacking. The pictorial representation of the model is described below:



After giving some insight of the N-beats model, it is time to be specific to the given case. For the given case, the model is parameterized by following the hyper-parameters as represented in the programmatically.

```
def architecture(backcast_length, forecast_length):
  model = NBeatsNet(
        stack_types=(NBeatsNet.GENERIC_BLOCK, NBeatsNet.GENERIC_BLOCK),
        forecast_length=forecast_length,
        backcast_length=backcast_length,
        hidden_layer_units=128,
    )
  optimiser = optim.Adam(lr=1e-4, params=model.parameters())

  return model, optimiser
```

Forecast and backcast lengths are defined at top of the program file. 'Adam' optimizer is used to try to reduce the loss every each epoch run with a declared learning rate of '1e-4'. After the model is compiled by calling the above method programmatically, the training method is called to train the model following a defined number of epochs. The training is explained in detail in the next section.

N-beats training

After elaborating the architecture of the model and model parameters to use, it is the right time for the training process. For training and validating the model during the whole process, train and test sets are divided into batches following the batch size of 4. The division is performed using the method represented below.

```
def data_generator(x, y, size):
    assert len(x) == len(y)
    batches = []
    for ii in range(0, len(x), size):
        batches.append((x[ii:ii + size], y[ii:ii + size]))
    for batch in batches:
        yield batch
```

Each batch as programmed above contains elements of input features with their corresponding output features. The number of elements in a batch is equivalent to the defined batch size.

The model is trained for 250 epochs, which is the best suited number considering the complexity of the dataset and model. 'Adam' optimizer is used with the learning rate of '1e-4' and 'mean square error' is used as a loss function to calculate the loss after each epoch. The training is all done using the pytorch deep learning framework. During each

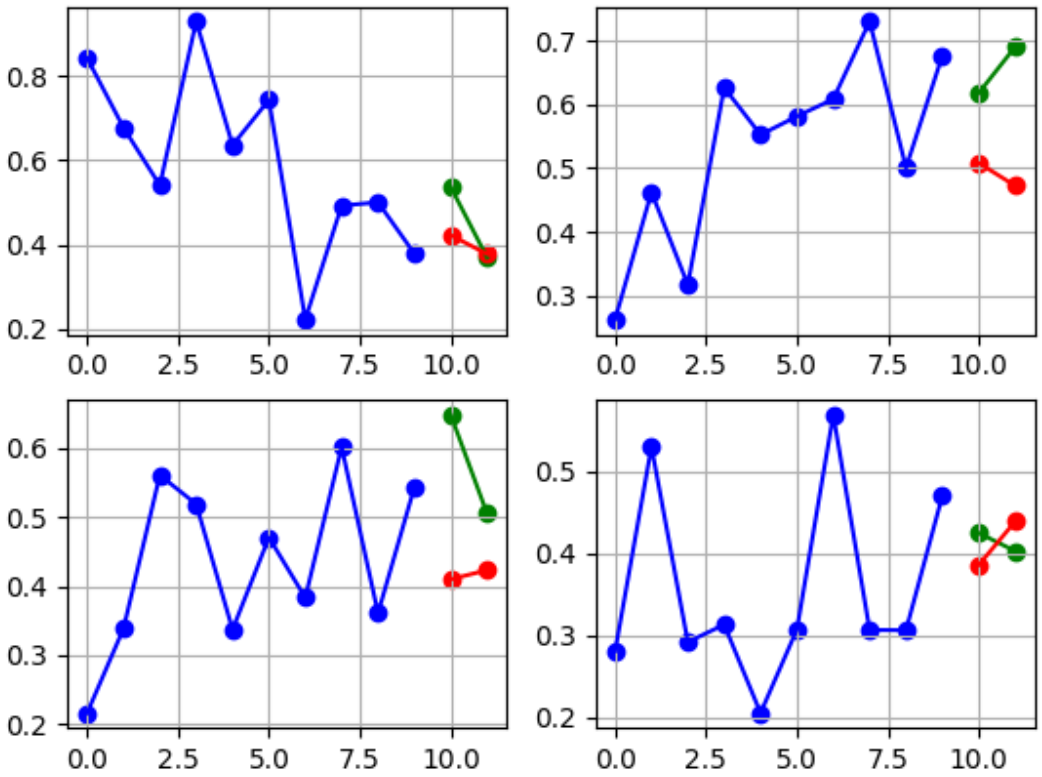
epoch, a set of input features of defined batch size is converted into tensors because pytorch uses tensors for numeric computation.

```
grad_step = 0
for epoch in range(400):
  # train.
  model.train()
  train_loss = []
  for x_train_batch, y_train_batch in data_generator(x_train, y_train, batch_size):
    grad_step += 1
    optimiser.zero_grad()
    _, forecast = model(torch.tensor(x_train_batch, dtype=torch.float).to(model.device))
    loss = F.mse_loss(forecast, torch.tensor(y_train_batch, dtype=torch.float).to(model.device))
    train_loss.append(loss.item())
    loss.backward()
    optimiser.step()
  train_loss = np.mean(train_loss)

  model.eval()
  _, forecast = model(torch.tensor(x_test, dtype=torch.float))
  test_loss = F.mse_loss(forecast, torch.tensor(y_test, dtype=torch.float)).item()
  p = forecast.detach().numpy()
  test_score = r2_score(y_test, p)
```

As represented above, the model is trained by looping over the number of epochs. The train() method is called on the model object to start the training. In each epoch, the model goes through all the batches of the training dataset. The Optimizer step() method is used to reduce the loss after each epoch run.

N-beats evaluation

After passing through the training dataset once, the model is evaluated by calling the eval() method on the model object. Predicted and ground values are compared to fine test loss of the model. Test accuracy is calculated using the r2_score class provided by the scikit-learn library. The class is used for calculating the accuracy of regression or time-series models. While training the model, the model continuously becomes under fitted because training loss continuously decreases and the testing loss increases gradually. The underfitting is caused because of inconsistencies in the dataset. Resultantly, the training accuracy ought to be 98 percent but the testing accuracy goes downwards. For better insight, visualizations are made using the matplotlib library used to plot various visualizations. The following plots gives better understanding related to performance of the trained model.

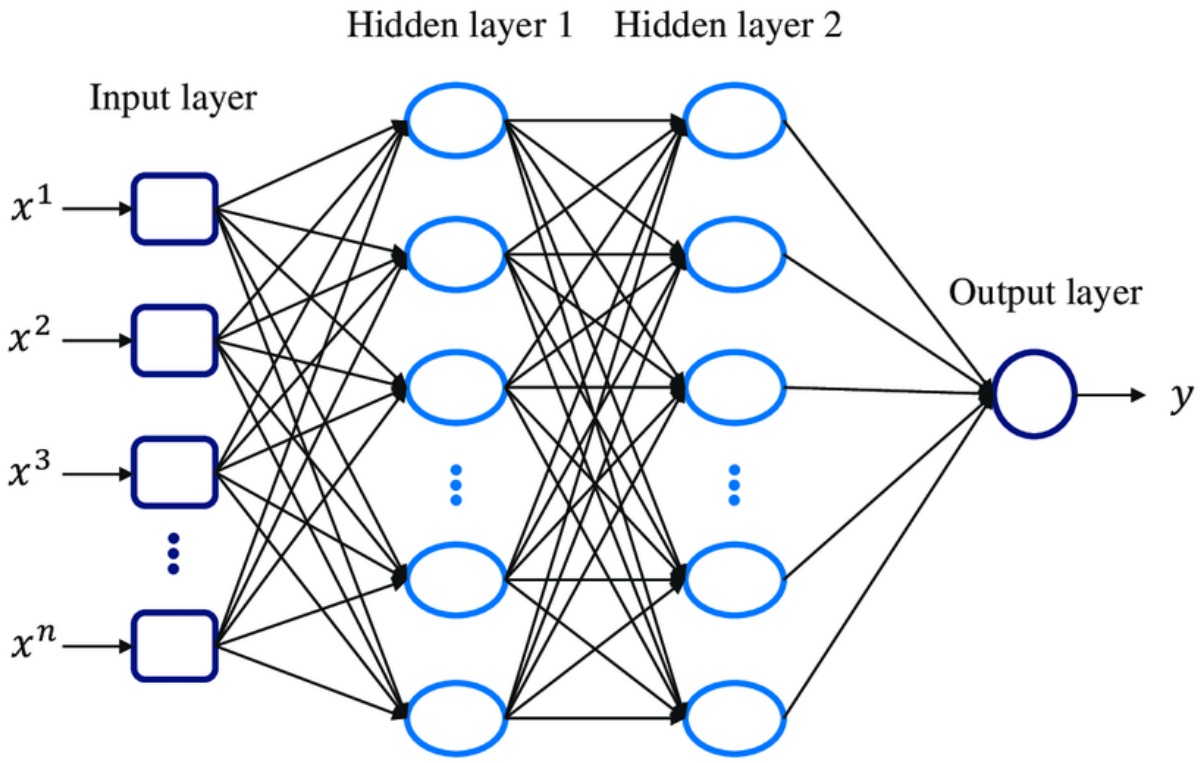


Considering the above evaluation plots, blue line indicates the training data points, green line is an indication of validation data points and red line indicates the test data points. In each plot, there are 10 training data points which means that there are 10 input data points in a training input feature set. Both the validation and test data points are of length 2 which means that there are two data points in both the validation feature set and test feature set which are predicted. So, the plots give an insight that the model is underfitted because the validation and test data points are distant from each other.

MLP architecture

Multilayer Perceptron models can be applied to time-series forecasting. For the given case, mlp is used for modelling the multi-step time-series forecasting problem. Multilayer Perceptron is the most common addition to feed-forward neural networks. The architecture is built on the composition of perceptrons. The network layers are made with a combination of perceptrons. The network is based on three types of layers: input layer, hidden layer and the output layer. These three layers are required to build the architecture. The input layer is used to receive the input dataset to be processed. HIdden layers are used

to process the input information. The prediction of the task depends upon the configuration of the output layer. The information flows in the forward direction from the input layer to the output layer as in the feed-forward neural network. Each node in the architecture is a neuron which is facilitated by the need of a nonlinear activation function. Since MLP is a fully-connected architecture, each node in the preceding layer is connected to every node of the succeeding layer with a certain tendency of weight. It utilizes the technique of back-propagation to facilitate the training process. The pictorial representation of the architecture is summarized below:



MLP architecture is used for the time-series predictive analysis in this case. It is used to follow the process of multivariate or multistep time-series analysis for tourism dataset predictions.

Now to be specific for the given case, three linear layers are used for building the model, The first linear layer acts as the input layer, second linear layer as the only hidden layer in the whole model, and the last linear layer as the output or predictive layer of the model. The configurations of the input and the output layers depend upon the dimensionalities of the input and the output features. The 'Relu' activation function is used after the input and the hidden layer. So, the model is parameterized in this way following 3-layer architecture which is represented in the diagram below.

```
class MLP(nn.Module):
  def __init__(self, input_dim, output_dim):
      super().__init__()

      self.input_fc = nn.Linear(input_dim, 250)
      self.hidden_fc = nn.Linear(250, 100)
      self.output_fc = nn.Linear(100, output_dim)

  def forward(self, x):
      #x = [batch size, height, width]
      h_1 = F.relu(self.input_fc(x))
      #h_1 = [batch size, 250]
      h_2 = F.relu(self.hidden_fc(h_1))
      #h_2 = [batch size, 100]
      y_pred = self.output_fc(h_2)
      #y_pred = [batch size, output dim]
      return y_pred, h_2
```

MLP training & evaluation

After briefing the general architecture and the model parameters of the multilayer perceptron, the training continues. The model chosen for training is based on three linear layers following the relu activation function. 'Adam' optimiser is used for trying to reduce the loss calculated after each epoch. Unfortunately, this multilayer perceptron model does not ought to be a good fit for the tourism dataset. The reason is the model is too immature and small to handle this complex inconsistent dataset. The training is represented in the following picture.

```
grad_step = 0
for epoch in range(400):
  # train.
  model.train()
  train_loss = []
  for x_train_batch, y_train_batch in data_generator(x_train, y_train, batch_size):
    grad_step += 1
    optimiser.zero_grad()
    _, forecast = model(torch.tensor(x_train_batch, dtype=torch.float))
    loss = F.mse_loss(forecast, torch.tensor(y_train_batch, dtype=torch.float))
    train_loss.append(loss.item())
    loss.backward()
    optimiser.step()
    print(np.mean(train_loss))
```

References

https://kshavg.medium.com/n-beats-neural-basis-expansion-analysis-for-interpretable-time-series-forecasting-91e94c830393

https://www.sciencedirect.com/topics/computer-science/multilayer-perceptron

https://machinelearningmastery.com/how-to-develop-multilayer-perceptron-models-for-time-series-forecasting/

https://www.deepdetect.com/blog/11-ts-forecast-nbeats/