Stock Price prediction using NBeats Model

Gnanachandru.K[1] and Satyajit Jena[2]

[1]3rd year Undergraduate student, Department of Physical Sciences, IISER Mohali
[2]1F5 - AB2, Department of Physical Sciences, IISER Mohali
[1] Mail ID: ms21031@iisermohali.ac.in
[2] Mail ID: sjena@iisermohali.ac.in

Keywords:
Deep learning with tensorflow
NBEATS Model
Conclusion

Abstract I have made this report based on my preparation on deep learning concepts with TensorFlow. I have used these concepts to forecast(Time Series Forecasting) stock price on NIFTY50 data using NBEATS algorithm. This report has three sections. Section 1 contains concepts I have learnt on Machine Learning with tensorflow, section 2 contains steps I followed for replicating the NBEATS model and section 3 summarises the report as conclusion . The primary aim of the project is to forecast stock price with maximum accuracy. I have also added snippets of my code in respective sections.

 Submitted: 11 August,2024 as the Summer Project.

1. Deep learning with tensorflow

1.1 Fundamentals

What is Tensorflow?
Tensorflow is machine learning library for preprocessing data, modelling data and creating models. Rather building models from scratch we can use the common machine learning functions available in tensorflow library to build models. We build machine learning algorithms to which data is fed(converted to numbers or tensors) and the alogorithm finds patterns in them.

1.1.1 Tensors:

Tensors are multi-dimensional(similar to NumPy array) representation of our data similar to matrix with arbitrary number of dimensions. The data can be numbers, images, text or something else which can be represented as numbers. Different between NumPy arrays and tensors is that tensors run of GPUs and TPUs which offer good compuatation speed

Creating tensors with tf.constant() method: Tensors with dimension 0 are called scalars,with dimension 1 are called vectors and tensors with dimension 2 or more are called matrix. In general all these can be called as tensors with thier respective dimensions(tensor with n dimensions are called as n-dimensional tensor). We can change the data type of our tensor by assigning dtype equals to the dtype you wish to change to.

```
import tensorflow as tf
scalar=tf.constant(4)
scalar
print("scalar dimension",scalar.ndim)
vector=tf.constant([10,10])
vector
print("vector dimension",vector.ndim)
matrix=tf.constant([[10,7],[7,10]])
matrix
print("matrix dimension",matrix.ndim)
```

Creating tensors with tf.Variable() method: We can create tensors in similar way as previous method but tensors created with tf.constant() are immutable and tensors created with tf.Variable() are mutable. That is tensors created by this method can be changed by using assign() method mentioning the index of the element to change.

```
changeable_tensor=tf.Variable([10,7])
unchangeable_tensor=tf.constant([10,7])
changeable_tensor,unchangeable_tensor
```

```
(<tf.Variable 'Variable:0' shape=(2,) dtype=int32, numpy=array([10,  7], dtype=int32)>,
 <tf.Tensor: shape=(2,), dtype=int32, numpy=array([10,  7], dtype=int32)>)
```

```
[11] changeable_tensor[0].assign(7)
     changeable_tensor
```

```
<tf.Variable 'Variable:0' shape=(2,) dtype=int32, numpy=array([7, 7], dtype=int32)>
```

Creating random tensors: Creating tensors of arbitrary size with random numbers. We can create random tensors by using tf.random.Generator class. If we set seed we'll get the same random numbers.

```
[13] random=tf.random.Generator.from_seed(42)
     random=random.normal(shape=(5,2))
     random
```

```
<tf.Tensor: shape=(5, 2), dtype=float32, numpy=
array([[-0.7565803 , -0.06854702],
       [ 0.07595026, -1.2573844 ],
       [-0.23193763, -1.8107855 ],
       [ 0.09988727, -0.50998646],
       [-0.7535805 , -0.57166284]], dtype=float32)>
```

Creating using tf.ones() and tf.zeros(): We can specify the shape of the tensor with tf.ones() and tf.zeros().
tf.ones() gives us the tensor of all ones and tf.zeros gives us the tensor of all zeros.

```
[6] tf.ones(shape=(4,2))

<tf.Tensor: shape=(4, 2), dtype=float32, numpy=
array([[1., 1.],
       [1., 1.],
       [1., 1.],
       [1., 1.]], dtype=float32)>
```

```
[7] tf.zeros(shape=(4,2))

<tf.Tensor: shape=(4, 2), dtype=float32, numpy=
array([[0., 0.],
       [0., 0.],
       [0., 0.],
       [0., 0.]], dtype=float32)>
```

Tensors from NumPy arrays: We can also turn NumPy arrays into tensors with any desired shape provided sum of dimensions of the tensor add up to the number of elements of array.

```
[11] import numpy as np
     numpy_A=np.arange(0,20,dtype=np.int32)
     A=tf.constant(numpy_A,
                   shape=[5,4])
     numpy_A,A

(array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
        17, 18, 19], dtype=int32),
 <tf.Tensor: shape=(5, 4), dtype=int32, numpy=
 array([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11],
        [12, 13, 14, 15],
        [16, 17, 18, 19]], dtype=int32)>)
```

Different characteristics of a tensor:

- Shape - The length of each dimension of a tensor
- Rank - The number of tensor dimensions
- Axis - A particular dimension of a tensor
- Size - The total number of items in the tensor

tf.newaxis() and tf.expand_dims(): We can add new axis or dimension to our tensor with keeping the same information with tf.newaxis() and tf.expand_dims().

1.1.2 Tensor Operations:

We can also perform basic math operations with $+, -, *$ directly on tensors. When operations are performed on tensor which are created by tf.constant() the result is produced on the copy and the original tensor remains the same.

```
tensor=tf.constant([[10,6],[2,5]])
tensor+10

<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[20, 16],
       [12, 15]], dtype=int32)>
```

```
[13] tensor*10

<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[100,  60],
       [ 20,  50]], dtype=int32)>
```

```
[14] tensor-10

<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[ 0, -4],
       [-8, -5]], dtype=int32)>
```

Matrix multiplication: We can use symbol * or tensorflow function tf.matmul for matrix multiplication of tensors. The basic matrix multiplication condition to be fulfilled that is inner dimensions should match and the resultant has the shape of the outer dimensions. To make the tensor satisy above rule we can use tf.reshape() or tf.transpose wherever necessary.

```
tensor

<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[10,  6],
       [ 2,  5]], dtype=int32)>
```

```
[16] tf.matmul(tensor,tensor)

<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[112,  90],
       [ 30,  37]], dtype=int32)>
```

```
[17] tensor@tensor

<tf.Tensor: shape=(2, 2), dtype=int32, numpy=
array([[112,  90],
       [ 30,  37]], dtype=int32)>
```

tf.tensordot() can also be used for matrix multiplication.

tf.cast() - Can be used to alter the default datatype of our tensor

```
A=tf.constant([[1,4,2],[5,7,2]])
A

<tf.Tensor: shape=(2, 3), dtype=int32, numpy=
array([[1, 4, 2],
       [5, 7, 2]], dtype=int32)>
```

```
[21] A=tf.cast(A,dtype=tf.float16)
     A

<tf.Tensor: shape=(2, 3), dtype=float16, numpy=
array([[1., 4., 2.],
       [5., 7., 2.]], dtype=float16)>
```

tf.abs() - Return the tensor with absolute value of the elements of the original tensor

```
D=tf.constant([-2,-20])
D

<tf.Tensor: shape=(2,), dtype=int32, numpy=array([ -2, -20], dtype=int32)>
```

```
[23] tf.abs(D)

<tf.Tensor: shape=(2,), dtype=int32, numpy=array([ 2, 20], dtype=int32)>
```

tf.reduce_min() - finds the minimum value in a tensor
tf.reduce_max() - finds the maximum value in a tensor
tf.reduce_mean() - finds the mean of all elements in a tensor
tf.reduce_sum() - finds the sum of all elements in a tensor

```
[25] M=tf.constant(np.random.randint(0,100,40))
     M

<tf.Tensor: shape=(40,), dtype=int64, numpy=
array([63, 66, 29, 28, 73, 99,  7,  2, 64, 84, 12,  2,  4, 98, 94, 53, 21,
        4, 28, 36, 76, 44, 17, 26, 73, 99, 66, 62, 16, 61, 99, 67, 67, 10,
       99, 97, 31, 39, 43, 16])>
```

```
[26] tf.reduce_min(M)

<tf.Tensor: shape=(), dtype=int64, numpy=2>
```

```
[27] tf.reduce_max(M)

<tf.Tensor: shape=(), dtype=int64, numpy=99>
```

```
[28] tf.reduce_mean(M)

<tf.Tensor: shape=(), dtype=int64, numpy=49>
```

```
[29] tf.reduce_sum(M)

<tf.Tensor: shape=(), dtype=int64, numpy=1975>
```

tf.argmax() - finds the position of the maximum element in a

given tensor
tf.argmin() - find the position of the minimum element in a given tensor

```
tf.argmax(M)
```
```
<tf.Tensor: shape=(), dtype=int64, numpy=5>
```
```
[31] tf.argmin(M)
```
```
<tf.Tensor: shape=(), dtype=int64, numpy=7>
```

tf.squeeze() - removes all dimensions of 1 from a tensor

```
[32] G=tf.constant(np.random.randint(0,100,50),shape=(1,1,1,1,50))
     G.shape,G.ndim
```
```
(TensorShape([1, 1, 1, 1, 50]), 5)
```
```
[33] G_squeezed=tf.squeeze(G)
     G_squeezed.shape,G_squeezed.ndim
```
```
(TensorShape([50]), 1)
```

tf.one_hot() - return the tensor after one-hot encoding it

```
[36] list=[0,3,1,2]
     tf.one_hot(list,depth=4)
```
```
<tf.Tensor: shape=(4, 4), dtype=float32, numpy=
array([[1., 0., 0., 0.],
       [0., 0., 0., 1.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.]], dtype=float32)>
```

tf.square() - returns tensor with all elements squared
tf.sqrt() - returns tensor after taking square root of every value
tf.math.log() - returns tensor after taking natural log of every element

```
H=tf.constant(np.arange(1,10))
H
```
```
<tf.Tensor: shape=(9,), dtype=int64, numpy=array([1, 2, 3, 4, 5, 6, 7, 8, 9])>
```
```
[38] tf.square(H)
```
```
<tf.Tensor: shape=(9,), dtype=int64, numpy=array([ 1,  4,  9, 16, 25, 36, 49, 64, 81])>
```
```
[39] H=tf.cast(H,dtype=tf.float32)
     tf.sqrt(H)
```
```
<tf.Tensor: shape=(9,), dtype=float32, numpy=
array([1.       , 1.4142135, 1.7320508, 2.       , 2.236068 , 2.4494898,
       2.6457512, 2.828427 , 3.       ], dtype=float32)>
```
```
[41] tf.math.log(H)
```
```
<tf.Tensor: shape=(9,), dtype=float32, numpy=
array([0.       , 0.6931472, 1.0986123, 1.3862944, 1.609438 , 1.7917595,
       1.9459102, 2.0794415, 2.1972246], dtype=float32)>
```

tf.Variable.assign() - assign a different value to a particular index of a variable tensor
tf.Variable.add_assign() - add to an existing value and reassign it at a particular index of a variable tensor

```
[42] I=tf.Variable(np.arange(0,5))
     I
```
```
<tf.Variable 'Variable:0' shape=(5,) dtype=int64, numpy=array([0, 1, 2, 3, 4])>
```
```
[43] I.assign([0,1,2,3,50])
```
```
<tf.Variable 'UnreadVariable' shape=(5,) dtype=int64, numpy=array([ 0,  1,  2,  3, 50])>
```
```
[44] I
```
```
<tf.Variable 'Variable:0' shape=(5,) dtype=int64, numpy=array([ 0,  1,  2,  3, 50])>
```
```
[46] I.assign_add([5,5,5,5,5])
```
```
<tf.Variable 'UnreadVariable' shape=(5,) dtype=int64, numpy=array([ 5,  6,  7,  8, 55])>
```
```
[47] I
```
```
<tf.Variable 'Variable:0' shape=(5,) dtype=int64, numpy=array([ 5,  6,  7,  8, 55])>
```

np.array() or tensor.numpy() - converts tensor to an ndarray

```
J=tf.constant(np.array([2.,4.,10.]))
J
```
```
<tf.Tensor: shape=(3,), dtype=float64, numpy=array([ 2.,  4., 10.])>
```
```
[50] np.array(J),type(np.array(J))
```
```
(array([ 2.,  4., 10.]), numpy.ndarray)
```
```
[51] J.numpy(),type(J.numpy())
```
```
(array([ 2.,  4., 10.]), numpy.ndarray)
```

1.2 Regression problem

This involves estimating the relationship between dependent variable and one or more independant variable and with that relationship we try to predict a number.

Basic architecture of regression neural network:

TABLE 1. Basic archtecture of regression network

Hyperparameter	Typical value
Input layer shape	Same shape as number of features
Hidden layers	Problem specific (min 1 and max unlimited)
Neurons per hidden layer	Problem specific,usually 10 to 100
Output layer shape	Same shape as desired prediction shape
Hidden activation	Usually ReLu
Output activation	None,ReLu,logistic/tanh
Loss function	MSE or MAE
Optimizer	SGD,Adam

We can build regression problem to predict cost of medical insurance for individuals. We use Medical Cost dataset available in kaggle.

1.2.1 Steps in modelling with TensorFlow

Getting our data ready

```
import tensorflow as tf
import pandas as pd
import matplotlib.pyplot as plt
```

```
insurance=pd.read_csv("https://raw.githubusercontent.com/stedy/Machine-Learning-with-R-datasets/master/insurance.csv")
```

```
[42] insurance.head()
```

	age	sex	bmi	children	smoker	region	charges
0	19	female	27.900	0	yes	southwest	16884.92400
1	18	male	33.770	1	no	southeast	1725.55230
2	28	male	33.000	3	no	southeast	4449.46200
3	33	male	22.705	0	no	northwest	21984.47061
4	32	male	28.880	0	no	northwest	3866.85520

We also convert the non-numerical features to numerical representation by onehot encoding it.

```
insurance_one_hot=pd.get_dummies(insurance,dtype=int)
insurance_one_hot.head()
```

	age	bmi	children	charges	sex_female	sex_male	smoker_no	smoker_yes	region_northeast	region_northwest	region_southeast	region_southwest
0	19	27.900	0	16884.92400	1	0	0	1	0	0	0	1
1	18	33.770	1	1725.55230	0	1	1	0	0	0	1	0
2	28	33.000	3	4449.46200	0	1	1	0	0	0	1	0
3	33	22.705	0	21984.47061	0	1	1	0	0	1	0	0
4	32	28.880	0	3866.85520	0	1	1	0	0	1	0	0

We create features set X and labels set y

```
X=insurance_one_hot.drop("charges",axis=1)
y=insurance_one_hot["charges"]
```

```
[45] X.head()
```

	age	bmi	children	sex_female	sex_male	smoker_no	smoker_yes	region_northeast	region_northwest	region_southeast	region_southwest
0	19	27.900	0	1	0	0	1	0	0	0	1
1	18	33.770	1	0	1	1	0	0	0	1	0
2	28	33.000	3	0	1	1	0	0	0	1	0
3	33	22.705	0	0	1	1	0	0	1	0	0
4	32	28.880	0	0	1	1	0	0	1	0	0

Next steps: Generate code with X | View recommended plots | New interactive sheet

```
[50] y.head()
```

	charges
0	16884.92400
1	1725.55230
2	4449.46200
3	21984.47061
4	3866.85520

dtype: float64

Spliting data into training/test set This step includes spliting the data into train data, test data, validation data(sometimes) each serving its own purposes.

- Training set - this consists 70 to 80 percent of the data on which the model learns
- Validation set - this consists of 10 to 15 percent of the data on which the model gets tuned
- Test set - this consists the remaing data on which the model gets evaluated to test what it has learned

```
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test=train_test_split(X,
                                               y,
                                               test_size=0.2,
                                               random_state=42)
```

We for now consider train and test sets only.

Visualising the data Now we've got our training and testing data, let's visualize it.

```
X_train.head(),X_test.head(),y_train.head(),y_test.head()
```

```
(      age   bmi  children  sex_female  sex_male  smoker_no  smoker_yes  \
560     46 19.95         2           1         0          1           0
1285    47 24.32         0           1         0          1           0
1142    52 24.86         0           1         0          1           0
969     39 34.32         5           1         0          1           0
486     54 21.47         3           1         0          1           0

      region_northeast  region_northwest  region_southeast  region_southwest
560                  0                 1                 0                 0
1285                 1                 0                 0                 0
1142                 0                 0                 1                 0
969                  0                 0                 1                 0
486                  0                 1                 0                 0 ,
      age     bmi  children  sex_female  sex_male  smoker_no  smoker_yes  \
764    45  25.175         2           1         0          1           0
887    36  30.020         0           1         0          1           0
890    64  26.885         0           1         0          0           1
1293   46  25.745         3           0         1          1           0
259    19  31.920         0           0         1          0           1

      region_northeast  region_northwest  region_southeast  region_southwest
764                  1                 0                 0                 0
887                  0                 1                 0                 0
890                  0                 1                 0                 0
1293                 0                 1                 0                 0
259                  0                 1                 0                 0 ,
560     9193.83850
1285    8534.67180
1142   27117.99378
969     8596.82780
486    12475.35130
Name: charges, dtype: float64,
764     9095.06825
887     5272.17580
890    29330.98315
1293    9301.89355
259    33750.29180
Name: charges, dtype: float64)
```

Creating a model - build together the layers of the network (using Functional or Sequential API) or import previously build model (transfer learning)
Compiling a model - defining how a models performance(loss/metrics) is measured as well as defining how it should improve(optimizer)
Fitting a model - letting the model to find patterns in the data

We build our model using keras Sequential API

```
[47] tf.random.set_seed(42)
     insurance_model=tf.keras.Sequential([
         tf.keras.layers.Dense(100),
         tf.keras.layers.Dense(10),
         tf.keras.layers.Dense(1)
     ])
     insurance_model.compile(loss=tf.keras.losses.mae,
                             optimizer=tf.keras.optimizers.Adam(),
                             metrics=["mae"])
     history=insurance_model.fit(X_train,y_train,epochs=200)
```

```
[48] insurance_model.evaluate(X_test,y_test)
```

```
9/9 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 3478.9182 - mae: 3478.9182
[3412.285888671875, 3412.285888671875]
```

Improving a model We can tweak some of the things in modelling steps

1. Creating a model - we can add more layers, increase number of hidden units within each layer, change activation function of each layer.
2. Compiling a model - you can choose optimization function or change learning rate of the optimization funciton
3. Fitting a model - fit a model for more epochs(more time) or on more data (much info)

Fitting our model We fit our model on train data
Visualising the predictions We create plot prediction function to visualise the predictions
Evaluating the predictions Evaluation can be done on two main metrics MAE and MSE.
Further improving our model We can tweak around to improve our model further. Some ways include increasing the number of epochs and increasing number of layers

1.3 Classification Problem

This involves predicting if something is one thing or another.Some different types of classification include,

- Binary classification - the output has only two options ,one or the other
- Multi-class classification - the output has multiple options

TABLE 2

Hyperparameter	Binary Classification	Multiclass classification
Input layer	Same as number of features	Same as binary classification
Hidden layers	Problem specific, min=1 , max=unlimited	Same as binary classification
Neurons per hidden layer	Problem specific, usually 10 to 100	Same as binary classification
Output layer shape	1(one class or the other)	1 per class
Hidden activation	Usually ReLU	Same as binary classification
Output activation	Sigmoid	Softmax
Loss function	BinaryCrossentropy	CategoricalCrossentropy
Optimizer	SGD	Same as binary classification

[a]Example footnote.

1.3.1 Getting the data ready

We make our own classification data using Scikit-Learn's make_circles() function. We see the data features and labels to understand what type of classification problem we are dealing with. If there are only two labels then we're dealing with binary classification and if there are more labels then it would be multiclass classfication. We also plot and visualize the data.

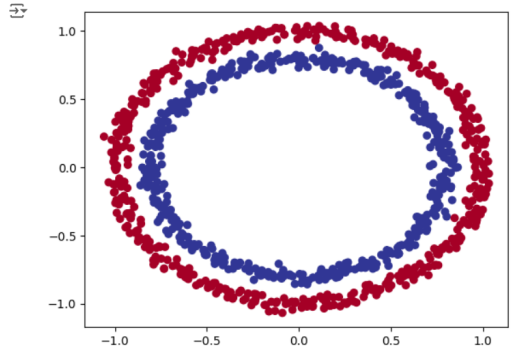
```
from sklearn.datasets import make_circles
n_samples=1000
X,y=make_circles(n_samples,
                 noise=0.03,
                 random_state=42)
```

```
[61] import pandas as pd
     circles=pd.DataFrame({"X0":X[:,0],"X1":X[:,1],"label":y})
     circles.head()
```

	X0	X1	label
0	0.754246	0.231481	1
1	-0.756159	0.153259	1
2	-0.815392	0.173282	1
3	-0.393731	0.692883	1
4	0.442208	-0.896723	0

Plotting the data

```
import matplotlib.pyplot as plt
plt.scatter(X[:,0],X[:,1],c=y,cmap=plt.cm.RdYlBu);
```



1.3.2 Steps in modelling

Creating a model
Compiling a model
Fitting a model
We use the same steps that we followed for regression problem with different hyperparameters.

We follow the steps and improve our results by tweaking our model like adding more layers,increasing the number of hidden units within each layer, change the activation functions of each layer,choose the different optimization function, and fit our model for more epochs.

1.3.3 Non-linearity

We see despite our efforts for improving our model we see our model performs worse. We see when our model is made to predict it give us straight line prediction. So we'll use non-linear activation(relu) parameter in our model.

We see still 50 percent accuracy much like guessing. We now change the activation function on our output layer too. That is for binary classification the output layer activation is Sigmoid activation function.

```
[66] tf.random.set_seed(42)
     model_7=tf.keras.Sequential([
         tf.keras.layers.Dense(4,activation="relu"),
         tf.keras.layers.Dense(4,activation="relu"),
         tf.keras.layers.Dense(1,activation="sigmoid")
     ])

     model_7.compile(loss=tf.keras.losses.binary_crossentropy,
                     optimizer=tf.keras.optimizers.Adam(),
                     metrics=["accuracy"])

     history=model_7.fit(X,y,epochs=100,verbose=0)
```

```
[67] model_7.evaluate(X,y)

     32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9846 - loss: 0.2665
     [0.25948235392570496, 0.984000027179718]
```

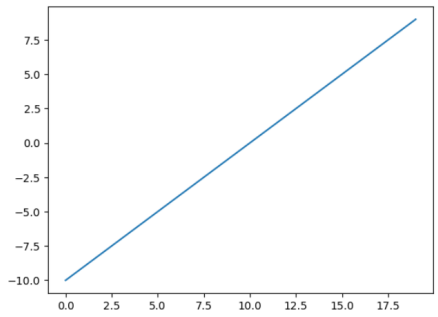
We see we get better results now.

Linear,Sigmoid and ReLU function visualization

```
A=tf.cast(tf.range(-10,10),tf.float32)
A
```

```
<tf.Tensor: shape=(20,), dtype=float32, numpy=
array([-10.,  -9.,  -8.,  -7.,  -6.,  -5.,  -4.,  -3.,  -2.,  -1.,   0.,
         1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.],
      dtype=float32)>
```

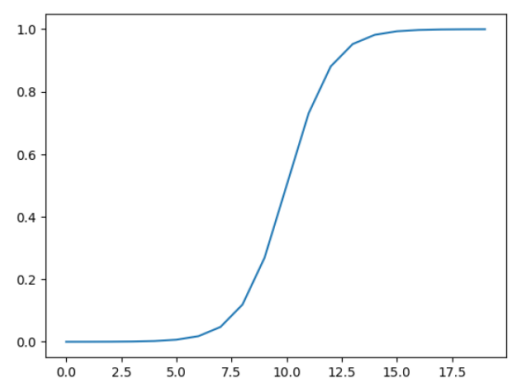
```
[60] plt.plot(A);
```



```
[61] def sigmoid(x):
       return 1/(1+tf.exp(-x))
     sigmoid(A)
```

```
<tf.Tensor: shape=(20,), dtype=float32, numpy=
array([4.5397872e-05, 1.2339458e-04, 3.3535014e-04, 9.1105117e-04,
       2.4726233e-03, 6.6928510e-03, 1.7986210e-02, 4.7425874e-02,
       1.1920292e-01, 2.6894143e-01, 5.0000000e-01, 7.3105860e-01,
       8.8079703e-01, 9.5257413e-01, 9.8201376e-01, 9.9330717e-01,
       9.9752742e-01, 9.9908900e-01, 9.9966466e-01, 9.9987662e-01],
      dtype=float32)>
```

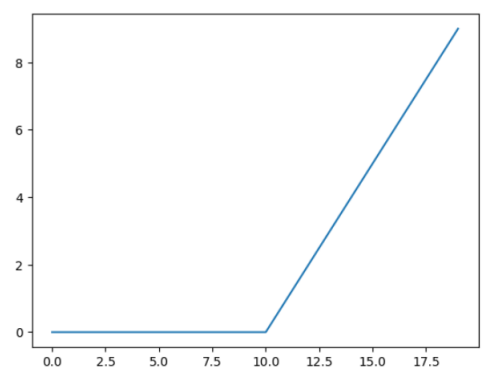
```
[62] plt.plot(sigmoid(A));
```



```
def relu(x):
  return tf.maximum(0,x)
relu(A)
```

```
<tf.Tensor: shape=(20,), dtype=float32, numpy=
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 2., 3., 4., 5., 6.,
       7., 8., 9.], dtype=float32)>
```

```
[64] plt.plot(relu(A));
```



1.4 Convolution Neural Network(CNNs)

We can use CNN,a special kind of neural network for detecting patterns in visual data. That is you can classify wheather a picture belongs to one or the other category.

1.4.1 A typical architecture of a convolution neural network

TABLE 3. A typical architecture of a convolution neural network

Hyperparameter	Function	Typical values
Input image	Image in which patters to be found	Any photo
Input layer	Take the input images and preprocesses for further layers	batch size, image height, image width, color channels
Convolution layer	Learns the most important features from the input image	ConvXD layer where X can be multiple values
Hidden activation	Add non-linearity to learned features	Usually ReLU
Pooling layer	Reduces the dimensionality of learned image features	AvgPool2D or MaxPool2D
Fully connected layer	further refines learned features from convolution layers	Dense layers
Output layer	outputs the learned features in shape required	output shape
Output activation	adds non-linearity to output layer	sigmoid for binary ,softmax for multiclass classification

[a]Example footnote.

1.4.2 Getting the data ready

We download the data of food items having two classes:pizza and steak. Here use tensorflow to classify the food items .

```
import zipfile
!wget https://storage.googleapis.com/ztm_tf_course/food_vision/pizza_steak.zip
zip_ref=zipfile.ZipFile("pizza_steak.zip","r")
zip_ref.extractall()
zip_ref.close()
```

```
--2024-08-11 12:20:44--  https://storage.googleapis.com/ztm_tf_course/food_vision/pizza_steak.zip
Resolving storage.googleapis.com (storage.googleapis.com)... 142.250.99.207, 172.253.117.207, 74.125.199.207, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|142.250.99.207|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 109540975 (104M) [application/zip]
Saving to: 'pizza_steak.zip'

pizza_steak.zip     100%[===================>] 104.47M   118MB/s    in 0.9s

2024-08-11 12:20:45 (118 MB/s) - 'pizza_steak.zip' saved [109540975/109540975]
```

We have 750 training and 250 test images of pizza and steak.

```
[5] import os
    for dirpath, dirnames, filenames in os.walk("pizza_steak"):
      print(f"There are {len(dirnames)} directories and {len(filenames)} images in '{dirpath}'")
```

```
There are 2 directories and 0 images in 'pizza_steak'
There are 2 directories and 0 images in 'pizza_steak/test'
There are 0 directories and 250 images in 'pizza_steak/test/steak'
There are 0 directories and 250 images in 'pizza_steak/test/pizza'
There are 2 directories and 0 images in 'pizza_steak/train'
There are 0 directories and 750 images in 'pizza_steak/train/steak'
There are 0 directories and 750 images in 'pizza_steak/train/pizza'
```

We view random images of our data so as to understand what kind of data we're dealing with.

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import random
def view_random_image(target_dir,target_class):
  target_folder=target_dir+target_class
  random_image=random.sample(os.listdir(target_folder),1)

  img=mpimg.imread(target_folder+"/"+random_image[0])
  plt.imshow(img)
  plt.title(target_class)
  plt.axis("off");
  print(f"Image shape:{img.shape}")
  return img
```

```
img=view_random_image(target_dir="pizza_steak/train/",
                      target_class="steak")
```

```
Image shape:(512, 512, 3)
```

Now we use Convolution layers and MaxPooling layers to create our CNN.
Some parameters used include,

- **filters** these are the number of feature extractors that will be moving over our images
- **kernel_size** size of our filters
- **stride** the number of pixels a filter will move across as it covers the image.
- **padding** this is can be 'same'(the output is same as input as the zeros are added to outsides of the image) or 'valid'(cuts off the excess pixels where filter doesn't fit)

```
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
tf.random.set_seed(42)
train_datagen=ImageDataGenerator(rescale=1./255)
valid_datagen=ImageDataGenerator(rescale=1./255)
train_dir="pizza_steak/train/"
test_dir="pizza_steak/test/"

train_data=train_datagen.flow_from_directory(train_dir,
                                             batch_size=32,
                                             target_size=(224,224),
                                             class_mode="binary",
                                             seed=42)
valid_data=valid_datagen.flow_from_directory(test_dir,
                                             batch_size=32,
                                             target_size=(224,224),
                                             class_mode="binary",
                                             seed=42)

model_1=tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(filters=10,
                           kernel_size=3,
                           activation="relu",
                           input_shape=(224,224,3)),
    tf.keras.layers.Conv2D(10,3,activation="relu"),
    tf.keras.layers.MaxPool2D(pool_size=2,
                              padding="valid"),
    tf.keras.layers.Conv2D(10,3,activation="relu"),
    tf.keras.layers.Conv2D(10,3,activation="relu"),
    tf.keras.layers.MaxPool2D(2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(1,activation="sigmoid")
])
model_1.compile(loss="binary_crossentropy",
                optimizer=tf.keras.optimizers.Adam(),
                metrics=["accuracy"])
history_1=model_1.fit(train_data,
                      epochs=5,
                      validation_data=valid_data)
```

```
47/47 ━━━━━━━━━━━━━━━━━━━━ 10s 155ms/step - accuracy: 0.6093 - loss: 0.6399 - val_accuracy: 0.7340 - val_loss: 0.5151
Epoch 2/5
47/47 ━━━━━━━━━━━━━━━━━━━━ 5s 102ms/step - accuracy: 0.7762 - loss: 0.5027 - val_accuracy: 0.8520 - val_loss: 0.3890
Epoch 3/5
47/47 ━━━━━━━━━━━━━━━━━━━━ 10s 99ms/step - accuracy: 0.8351 - loss: 0.3960 - val_accuracy: 0.8500 - val_loss: 0.3513
Epoch 4/5
47/47 ━━━━━━━━━━━━━━━━━━━━ 6s 112ms/step - accuracy: 0.8589 - loss: 0.3401 - val_accuracy: 0.8420 - val_loss: 0.3572
Epoch 5/5
47/47 ━━━━━━━━━━━━━━━━━━━━ 5s 102ms/step - accuracy: 0.8788 - loss: 0.3125 - val_accuracy: 0.8800 - val_loss: 0.2882
```

We get accuracy of 88 percent and validation accuracy of 89 percent which is good. And to further improve we can tweak the parameters and to reduce the overfitting we can use data augmentation which generalizes the learning.

1.5 Transfer Learning

Transfer learning involves taking the patterns another model has learned from another problem and using them for our own problem. That is instead of building models from scratch we can use model which others have created to solve similar problem. This helps in situations when we have less data to build from scratch.

1.5.1 Transfer learning with tensorflow hub: Getting great results with less data

Tensorflow hub is a repository which contains models made for many problems which can be imported and use which is already fully trained.
Transfer learning often allows you to get great results with less data

2. Time series forecasting with NBEATS

Time series problems deal with data over time. This comes with two types:**Classification** and **Forecasting**. Classification problem include anomaly detection and forecasting involves predicting the stock price.
Types of time series:

- Trend - time series has a clear long- term increase or decrease
- Seasonal - time series affected by time of the year or day of week(increased sales during weekends)
- Cyclic - time series shows rises and falls over an unfixed period

2.1 NBEATS MODEL

2.1.1 Introduction

The NBeats paper proposes an architecture which gives state of the art performance in univariate time series forecasting by improving the forecast accuracy by 11 percent over a statistical benchmark and by 3 percent over winner of M4 competition which is a hybrid model. The paper proposes two cofigurations for NBeats Model which includes Generic and Interpretable model.

2.1.2 N-BEATS

As given in the figure the architecture is structured as several blocks grouped together whose forecast is summed up in a stack and similarly different stacks grouped together whose forecast is summed up to give our model output or Global forecast.

Considering a random single l^{th} block of the stack, we see that it accepts input x_l and gives two output namely $\hat{x}_l$ and $\hat{y}_l$ .x_l is the residual outputs of the previous block except for the very first block where x_l is the overall model input. x_l can be any multiple of forecast horizon H. We can take horizon to be 2H to 7H. $\hat{x}_l$ is the block's best estimate of x_l and $\hat{y}_l$ is the block's forward forecast of length H.

The block internally has full connected layers which gives us the forward θ_l^f and θ_l^b expansion coefficients.And then these coefficients are accepted and projected on the set of basis functions by backward and forward basis layers(g_l^b and g_l^f) and produce the backcast $\hat{x}_l$ and forecast $\hat{y}_l$ outputs.

Double Residual Stacking:
This architecture has two residual branches in which one runs over backcast prediction of each layer and other runs over forecast prediction of each layer.

$$x_l = x_{l-1} - \hat{x}_{l-1}$$

,

$$\hat{y} = \sum_l \hat{y}_l$$

Block receives the input x_l from the previous block input with some of it removed which it is easily able to predict($\hat{x}_l$).And the forecast output of each block is added to produce overall stack forecast which then each forecast output of stack is added to produce the overall output.

The two configurations of the architecture differs from the choice of the basis functions we choose(g_l^b and g_l^f). In generic architecture these functions are set to be linear projection of the previous layer output. In interpretable architecture we can decompose time series into trend and seasonality components to increase more interpretability.

Trend model We here take basis functions to be slowly varying monotonic polynomial function with small degree p. Trend forecast in matrix form:

$$\hat{y_{s,l}^{tr}} = T\theta_{s,l}^f$$

where,
T=[1,t,...t^p]
Seasonality model We here take basis functions to be fourier series as to satisfy the criteria of regular, cyclical, recurring fluctuation.

$$\hat{y_{s,l}} = \sum_{i=0}^{[H/2-1]} \theta_{s,l,i}^f \cos 2\pi it + \theta_{s,l,i+[H/2]}^f \sin 2\pi it$$

Seasonality forecast in matrix form:

$$\hat{y}_{s,l}^{seas} = S\theta_{s,l}^f$$

where,
S=[1,cos(2πt),...cos(2π[H/2-1]t),sin(2πt),...,sin(2π[H/2-1]t)]

TABLE 4. Generic and Interpretable Model

S.No	Generic	Interpretable
1	Does not impose any specific structure on model's output or the way it processes input data	Can be used to interpret the forecast by using trend and seasonality components
2	High accuracy	Less accuracy
3	Not interpretable	Interpretable
4	Used for high accuracy in forecasts	Used for understanding the logic behind forecast at the expense of accuracy

[a]Example footnote.

2.1.3 Get the data

I downloaded the data from kaggle on Nifty-50 stock market data(2000-2021) and created a list object "company" by which we can access each company's data seperately for our forecasting.

```
!pip install kaggle
```

```
Requirement already satisfied: kaggle in /usr/local/lib/python3.10/dist-packages (1.6.14)
Requirement already satisfied: six>=1.10 in /usr/local/lib/python3.10/dist-packages (from kaggle) (1.16.0)
Requirement already satisfied: certifi>=2023.7.22 in /usr/local/lib/python3.10/dist-packages (from kaggle) (2024.7.4)
Requirement already satisfied: python-dateutil in /usr/local/lib/python3.10/dist-packages (from kaggle) (2.8.2)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from kaggle) (2.31.0)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages (from kaggle) (4.66.4)
Requirement already satisfied: python-slugify in /usr/local/lib/python3.10/dist-packages (from kaggle) (8.0.4)
Requirement already satisfied: urllib3 in /usr/local/lib/python3.10/dist-packages (from kaggle) (2.0.7)
Requirement already satisfied: bleach in /usr/local/lib/python3.10/dist-packages (from kaggle) (6.1.0)
Requirement already satisfied: webencodings in /usr/local/lib/python3.10/dist-packages (from bleach->kaggle) (0.5.1)
Requirement already satisfied: text-unidecode>=1.3 in /usr/local/lib/python3.10/dist-packages (from python-slugify->kaggle) (1.3)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests->kaggle) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->kaggle) (3.7)
```

```
!kaggle datasets download -d rohanrao/nifty50-stock-market-data
```

```
Warning: Looks like you're using an outdated API Version, please consider updating (server 1.6.17 / client 1.6.14)
Dataset URL: https://www.kaggle.com/datasets/rohanrao/nifty50-stock-market-data
License(s): CC0-1.0
Downloading nifty50-stock-market-data.zip to /content
100% 18.4M/18.4M [00:01<00:00, 23.4MB/s]
100% 18.4M/18.4M [00:01<00:00, 13.4MB/s]
```

```
import zipfile
zip_ref=zipfile.ZipFile("/content/nifty50-stock-market-data.zip")
zip_ref.extractall()
```

```
import os
company=[]
for i in os.listdir("/content/"):
  company.append(i)
company.remove(".config")
company.remove("nifty50-stock-market-data.zip")
company.remove("stock_metadata.csv")
company.remove("sample_data")
company.remove("NIFTY50_all.csv")
print(len(company))
company
```

2.1.4 Becoming one with data

Visulising the data: Once the data is downloaded we notice various price fluctuations given. Lets consider market closing price data and ignore the remaining for our forecasting model. The data is visualised by plotting after loading the data through python's Pandas/CSV module.

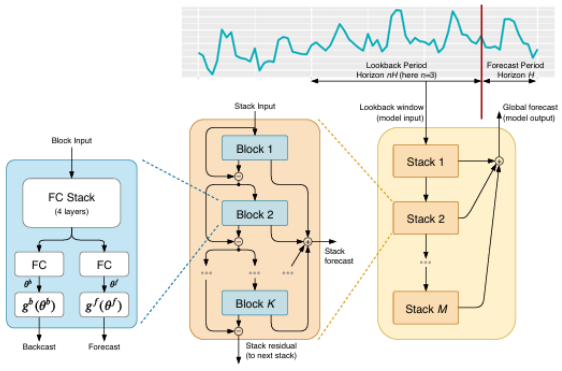


FIGURE 1. Proposed architecture

```
[ ] import csv
    import datetime as dt
    timesteps=[]
    price=[]
    with open(f"/content/"+choice_of_company.upper()+".csv") as f:
      csv_reader=csv.reader(f,delimiter=",")
      next(csv_reader)
      for line in csv_reader:
        timesteps.append(dt.datetime.strptime(line[0],"%Y-%m-%d"))
        price.append(float(line[8]))
    timesteps[:10],price[:10]
```

```
([datetime.datetime(2000, 1, 3, 0, 0),
  datetime.datetime(2000, 1, 4, 0, 0),
  datetime.datetime(2000, 1, 5, 0, 0),
  datetime.datetime(2000, 1, 6, 0, 0),
  datetime.datetime(2000, 1, 7, 0, 0),
  datetime.datetime(2000, 1, 10, 0, 0),
  datetime.datetime(2000, 1, 11, 0, 0),
  datetime.datetime(2000, 1, 12, 0, 0),
  datetime.datetime(2000, 1, 13, 0, 0),
  datetime.datetime(2000, 1, 14, 0, 0)],
 [2724.2,
  2942.15,
  2990.1,
  2932.25,
  2697.7,
  2701.35,
  2485.45,
  2562.9,
  2480.25,
  2497.85])
```

```
import matplotlib.pyplot as plt
plt.plot(timesteps,price)
plt.xlabel("Date")
plt.ylabel("Closing Price")
plt.title(f"Stock Price of {choice_of_company}")
plt.show()
```

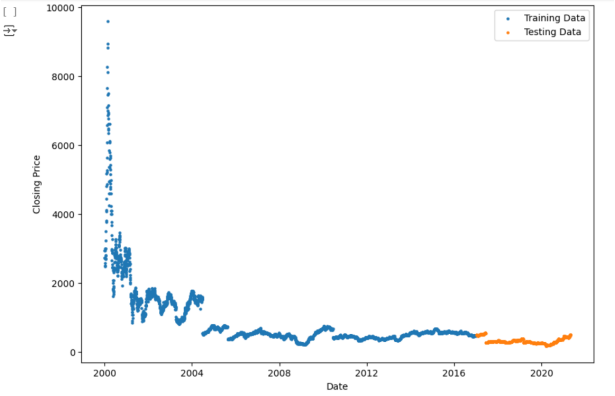
2.1.5 Getting the data ready

Creating train and test data: We need our model to be trained on train data and to be validated on test data. So we split our data to 80 percent as train data and rest as test data.We plot the data after our data is split.

```
[ ] split_size=int(len(stock_prices)*0.8)
    X_train,y_train=timesteps[:split_size],price[:split_size]
    X_test,y_test=timesteps[split_size:],price[split_size:]
    len(X_train),len(X_test),len(y_train),len(y_test)
```

```
(4244, 1062, 4244, 1062)
```

```
plt.figure(figsize=(10,7))
plt.scatter(X_train,y_train,s=5,label="Training Data")
plt.scatter(X_test,y_test,s=5,label="Testing Data")
plt.xlabel("Date")
plt.ylabel("Closing Price")
plt.legend()
plt.show()
```



Creating lookback window and forecast horizon: We now prepare our data into lookback window of size 7 which we will be using to train and forecast horizon of size 1. For this we shift the data and remove the non-numerical characters.We then split our new data to train and test data.

```
stock_price_nbeats=stock_prices.copy()
for i in range(WINDOW_SIZE):
  stock_price_nbeats[f"Closing Price{i+1}"]=stock_price_nbeats["Closing Price"].shift(i+1)
stock_price_nbeats.head(15)
```

Date	Closing Price	Closing Price1	Closing Price2	Closing Price3	Closing Price4	Closing Price5	Closing Price6	Closing Price7
2000-01-03	2724.20	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2000-01-04	2942.15	2724.20	NaN	NaN	NaN	NaN	NaN	NaN
2000-01-05	2990.10	2942.15	2724.20	NaN	NaN	NaN	NaN	NaN
2000-01-06	2932.25	2990.10	2942.15	2724.20	NaN	NaN	NaN	NaN
2000-01-07	2697.70	2932.25	2990.10	2942.15	2724.20	NaN	NaN	NaN
2000-01-10	2701.35	2697.70	2932.25	2990.10	2942.15	2724.20	NaN	NaN
2000-01-11	2485.45	2701.35	2697.70	2932.25	2990.10	2942.15	2724.20	NaN
2000-01-12	2562.90	2485.45	2701.35	2697.70	2932.25	2990.10	2942.15	2724.20
2000-01-13	2480.25	2562.90	2485.45	2701.35	2697.70	2932.25	2990.10	2942.15
2000-01-14	2497.85	2480.25	2562.90	2485.45	2701.35	2697.70	2932.25	2990.10
2000-01-17	2697.70	2497.85	2480.25	2562.90	2485.45	2701.35	2697.70	2932.25
2000-01-18	2677.15	2697.70	2497.85	2480.25	2562.90	2485.45	2701.35	2697.70
2000-01-19	2784.50	2677.15	2697.70	2497.85	2480.25	2562.90	2485.45	2701.35

```
X=stock_price_nbeats.dropna().drop(columns=["Closing Price"])
y=stock_price_nbeats.dropna()["Closing Price"]
split_size=int(len(X)*0.8)
X_train,y_train=X[:split_size],y[:split_size]
X_test,y_test=X[split_size:],y[split_size:]
len(X_train),len(X_test),len(y_train),len(y_test)
X_train.shape,y_train.shape
```

```
((4239, 7), (4239,))
```

Data Prefetching and Preprocessing: As we are going to use larger model architecture we prefetch the data using tf.data API for faster training time.We also group our data into group of batches for further faster training and also zip our data into train dataset and test dataset.

```
[ ] import tensorflow as tf

#slices the first dimension
train_features_dataset=tf.data.Dataset.from_tensor_slices(X_train)
train_labels_dataset=tf.data.Dataset.from_tensor_slices(y_train)


test_features_dataset=tf.data.Dataset.from_tensor_slices(X_test)
test_labels_dataset=tf.data.Dataset.from_tensor_slices(y_test)

#combining features and labels
train_dataset=tf.data.Dataset.zip(train_features_dataset,train_labels_dataset)
test_dataset=tf.data.Dataset.zip(test_features_dataset,test_labels_dataset)

#preprocessing and batch
BATCH_SIZE=1024
train_dataset=train_dataset.batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)
test_dataset=test_dataset.batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)
train_dataset,test_dataset

(<_PrefetchDataset element_spec=(TensorSpec(shape=(None, 7), dtype=tf.float64, name=None), TensorSpec(shape=(None,), dtype=tf.float64, name=None))>,
 <_PrefetchDataset element_spec=(TensorSpec(shape=(None, 7), dtype=tf.float64, name=None), TensorSpec(shape=(None,), dtype=tf.float64, name=None))>)
```

Setting up the hyperparameters:

We define the hyperparameter as given in the NBeats paper for using them in our model

```
[ ] import tensorflow as tf

#slices the first dimension
train_features_dataset=tf.data.Dataset.from_tensor_slices(X_train)
train_labels_dataset=tf.data.Dataset.from_tensor_slices(y_train)


test_features_dataset=tf.data.Dataset.from_tensor_slices(X_test)
test_labels_dataset=tf.data.Dataset.from_tensor_slices(y_test)

#combining features and labels
train_dataset=tf.data.Dataset.zip(train_features_dataset,train_labels_dataset)
test_dataset=tf.data.Dataset.zip(test_features_dataset,test_labels_dataset)

#preprocessing and batch
BATCH_SIZE=1024
train_dataset=train_dataset.batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)
test_dataset=test_dataset.batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)
train_dataset,test_dataset

(<_PrefetchDataset element_spec=(TensorSpec(shape=(None, 7), dtype=tf.float64, name=None), TensorSpec(shape=(None,), dtype=tf.float64, name=None))>,
 <_PrefetchDataset element_spec=(TensorSpec(shape=(None, 7), dtype=tf.float64, name=None), TensorSpec(shape=(None,), dtype=tf.float64, name=None))>)
```

2.1.6 NBEATS Generic Model

We create NBeats Block class with the logic given for the block. First we intialize the class objects that we are going to be using by init method. As given in paper input is passed through 4 full connected layers ,so we use dense layers with relu activation. We also intialize theta layer which outputs our backcast and forecast outputs. Then by call method we pass the data through 4 dense layers by looping and the resultant data is passed through theta layer. The final output is segregated to return us the backcast and forecast.

```
class NBeatsBlock(tf.keras.layers.Layer):
  def __init__(self,
               input_size:int,
               theta_size:int,
               horizon:int,
               n_neurons:int,
               n_layers:int,
               **kwargs):
    super().__init__(**kwargs)
    self.input_size=input_size
    self.theta_size=theta_size
    self.horizon=horizon
    self.n_neurons=n_neurons
    self.n_layers=n_layers

    #4 fully connected layers with relu activation
    self.hidden=[tf.keras.layers.Dense(n_neurons,activation="relu") for _ in range(n_layers)]

    #output of block:theta layer with linear activation
    self.theta_layer=tf.keras.layers.Dense(theta_size,activation="relu",name="theta")

  def call(self,inputs):
    x=inputs
    for layer in self.hidden:
      x=layer(x)
    theta=self.theta_layer(x)

    backcast,forecast=theta[:,:self.input_size],theta[:,-self.horizon:]
    return backcast,forecast
```

Building,Compiling and Running NBeats model:

Creating N-BEATS block:

Instance of the NBeats block class as nbeats block layer and input layer is defined. By passing the input data to the block layer we obtain our forecast and backcast. Then for the remaining stacks we use loop to remove the backcast data(stored as residuals) and add the block forecast to give as summed up forecast(stored as forecast). The we can define our model with functional method, compile and fit with the hyperparameter given. We also use two callbacks namely ReduceLROnPlateau which reduce learning rate to stop overfitting and EarlyStopping which stops fitting one it reached patience limit given for our ease.

```
%%time
import tensorflow as tf
from tensorflow.keras import layers
tf.random.set_seed(42)

nbeats_block_layer=NBeatsBlock(input_size=INPUT_SIZE,
                               theta_size=THETA_SIZE,
                               horizon=HORIZON,
                               n_neurons=N_NEURONS,
                               n_layers=N_LAYERS)

stack_input=layers.Input(shape=INPUT_SIZE,name="stack_input")

residuals,forecast=nbeats_block_layer(stack_input)

for i,_ in enumerate(range(N_STACKS-1)):
  backcast,block_forecast=NBeatsBlock(
      input_size=INPUT_SIZE,
      theta_size=THETA_SIZE,
      horizon=HORIZON,
      n_neurons=N_NEURONS,
      n_layers=N_LAYERS,
      name=f"NBeatsBlock{i}")(residuals)

  residuals=layers.subtract([residuals,backcast],name=f"Sub{i}")
  forecast=layers.add([forecast,block_forecast],name=f"Add{i}")

model_g=tf.keras.Model(inputs=stack_input,outputs=forecast,name="NBeatsModel")
model_g.compile(loss="mae",
              optimizer=tf.keras.optimizers.Adam())
model_g.fit(train_dataset,
          epochs=N_EPOCHS,
          validation_data=test_dataset,
          verbose=1,
          callbacks=[tf.keras.callbacks.EarlyStopping(monitor="val_loss",
                                                      patience=200,
                                                      restore_best_weights=True),
                     tf.keras.callbacks.ReduceLROnPlateau(monitor="val_loss",
                                                          patience=100,
                                                          verbose=1)])
```

Evaluating and Forecasting

```
model_g.evaluate(test_dataset)

2/2 [==============================] - 0s 13ms/step - loss: 4.0842
4.084219932556152
```

We now evaluate our learnt model on test dataset and we see loss is not huge.

```
[ ] def make_preds(model,input_data):
      forecast=model.predict(input_data)
      return forecast
```

```
[ ] model_g_preds=make_preds(model_g,test_dataset)
    model_g_preds

2/2 [==============================] - 1s 17ms/step
array([[485.5035 ],
       [483.9712 ],
       [482.83878],
       ...,
       [484.83792],
       [487.43436],
       [492.1323 ]], dtype=float32)
```

We define MASE and sMAPE metrics function as tensorflow does not have these built-in.

```
def mean_absolute_scaled_error(y_true,y_pred):
  mae=tf.keras.metrics.mean_absolute_error(y_true,y_pred)
  mae_naive=tf.keras.metrics.mean_absolute_error(y_true[1:],y_pred[:-1])
  return mae/mae_naive
```

```
def smape(y_true,y_pred):
  smape=tf.math.scalar_mul(200,tf.math.divide(tf.math.abs(tf.math.subtract(y_true,y_pred)),tf.math.add(y_true,y_pred)))
  smape=tf.math.reduce_mean(smape)
  return smape
```

Now we use metrics like MAE,MSE,RMSE,MAPE,MASE,sMAPE to evaluate the prediction made by our model.Now we define evaluate preds function.

```
def evaluate_preds(y_true,y_pred):
  y_true=tf.cast(y_true,tf.float32)
  y_pred=tf.cast(y_pred,tf.float32)

  mae=tf.keras.metrics.mean_absolute_error(y_true,y_pred)
  mse=tf.keras.metrics.mean_squared_error(y_true,y_pred)
  rmse=tf.sqrt(mse)
  mape=tf.keras.metrics.mean_absolute_percentage_error(y_true,y_pred)
  mase=mean_absolute_scaled_error(y_true,y_pred)
  sMAPE=smape(y_true,y_pred)


  return {"mae":mae.numpy(),
          "mse":mse.numpy(),
          "rmse":rmse.numpy(),
          "mape":mape.numpy(),
          "mase":mase.numpy(),
          "smape":sMAPE.numpy()}
```

```
model_g_results=evaluate_preds(y_test,tf.squeeze(model_g_preds))
model_g_results
```

```
{'mae': 4.08422,
 'mse': 106.2029,
 'rmse': 10.305479,
 'mape': 1.3445683,
 'mase': 0.6810401,
 'smape': 1.3086859}
```

2.1.7 NBEATS Interpretable Model

Setting up trend and seasonality hyperparameters: We define our trend and seasonality hyperparameter in dictionary which we can use later as key word arguments. We set batch size as 1024,window size as 7 and horizon as 1.

```
trend_params = {
    'width': 256,
    'degree': 2,
    'num_blocks': 3,
    'num_block_layers': 4
}

seasonality_params = {
    'width': 2048,
    'harmonics': 4,
    'num_blocks': 3,
    'num_block_layers': 4
}

batch_size=1024
window_size=7
horizon=1
```

Defining basis expansion function We now define polynomial basis function and forier basis function which we will projecting to get trend and seasonality componenents of the output respectively.

```
def polynomial_basis(x,degree):
  x=tf.expand_dims(x,axis=-1)
  powers=tf.range(degree+1,dtype=tf.float32)
  return tf.pow(x,powers)


def fourier_basis(x,harmonics):
  x=tf.expand_dims(x,axis=-1)
  sin_terms=[tf.sin(2*np.pi*x*i) for i in range(0,harmonics)]
  cos_terms=[tf.cos(2*np.pi*x*i) for i in range(0,harmonics)]
  return tf.concat(sin_terms+cos_terms, axis=-1)
```

Trend Block layer:

We difine Trend block class which captures the trend and outputs the trend forecast. We intialize the class objects . We also create a function which build multiple blocks and return it which we will use it later in call method. We intially set trend forecast to null tensor with required shape. Input data is passed through block and stored as block output. Then trend forecast is updated by adding to the intialized null matrix with the result of polynomial basis multiplied with block output. Then we return the mean of trend forecast.

```
class TrendBlock(layers.Layer):
  def __init__(self,
               width,
               degree,
               num_blocks,
               num_block_layers):
    super(TrendBlock,self).__init__()
    self.num_blocks=num_blocks
    self.num_block_layers=num_block_layers
    self.degree=degree
    self.blocks=[self.build_block(width) for _ in range(num_blocks)]

  def build_block(self,width):
    return keras.Sequential([layers.Dense(width,activation="relu") for _ in range(self.num_block_layers)]+
                            [layers.Dense(self.degree+1,activation=None)])

  def call(self,inputs):
    trend_forecast=tf.zeros([tf.shape(inputs)[0],tf.shape(inputs)[1],self.degree+1],dtype=tf.float32)
    for block in self.blocks:
      block_output= tf.expand_dims(block(inputs),axis=1)
      trend_forecast+= polynomial_basis(inputs,self.degree) * block_output
    trend_forecast= tf.reduce_sum(trend_forecast,axis=-1)
    return tf.reduce_mean(trend_forecast,axis=-1)
```

Seasonality Block layer:

Similary to trend block layer we start with defining seasonality forecast as null tensor which get update later in loop by adding to it the product of ouput of fourier basis and block output. Then the mean of seasonality forecast is returned.

```
class SeasonalityBlock(layers.Layer):
  def __init__(self,
               width,
               harmonics,
               num_blocks,
               num_block_layers):
    super(SeasonalityBlock,self).__init__()
    self.num_blocks=num_blocks
    self.num_block_layers= num_block_layers
    self.harmonics=harmonics
    self.blocks= [self.build_block(width) for _ in range(num_blocks)]

  def build_block(self,width):
    return keras.Sequential([layers.Dense(width,activation="relu") for _ in range(self.num_block_layers)]+

                            [layers.Dense(2* self.harmonics,activation = None)])

  def call(self,inputs):
    seasonality_forecast = tf.zeros([tf.shape(inputs)[0],tf.shape(inputs)[1],2 * self.harmonics],dtype=tf.float32)
    for block in self.blocks:
      block_output = tf.expand_dims(block(inputs),axis=1)
      seasonality_forecast+= fourier_basis(inputs,self.harmonics) * block_output
    seasonality_forecast = tf.reduce_sum(seasonality_forecast, axis=-1)
    return tf.reduce_mean(seasonality_forecast,axis=-1)
```

NBeats interpretable class:

Here in init funciton two stacks namely trend and seasonality stack class objects are defined to which trend and seasonality parameters are passed. The call function trend and seasonality objects are passed with input data and the sum of trend forecast and seasonality forecast is returned.

```
[ ] class NBeatsInterpretableModel(keras.Model):
      def __init__(self,
                   trend_params, seasonality_params):
        super(NBeatsInterpretableModel,self).__init__()
        self.trend_stack= TrendBlock(**trend_params)
        self.seasonality_stack=SeasonalityBlock(**seasonality_params)

      def call(self,inputs):
        trend_forecast =self.trend_stack(inputs)
        seasonality_forecast= self.seasonality_stack(inputs)
        return trend_forecast + seasonality_forecast
```

Build,compile and fit:

The model is built,compiled and fit by the given hyperparameters.

```
model_i = NBeatsInterpretableModel(trend_params, seasonality_params)

model_i.compile(loss="mae",
                optimizer="adam")

model_i.fit(train_dataset,
            epochs=N_EPOCHS,
            validation_data=test_dataset,
            callbacks=[tf.keras.callbacks.EarlyStopping(monitor="val_loss",
                                                        patience=200,
                                                        restore_best_weights=True),
                       tf.keras.callbacks.ReduceLROnPlateau(monitor="val_loss",
                                                            patience=100,
                                                            verbose=1)])
```

Evaluating and Forecasting

```
model_i.evaluate(test_dataset)
```

```
2/2 [==============================] - 0s 8ms/step - loss: 136.7376
136.73756408691406
```

```
[ ] model_i_preds=make_preds(model_i,test_dataset)
    model_i_preds
```

```
2/2 [==============================] - 1s 16ms/step
array([477.3837 , 428.91446, 394.91125, ..., 514.48615, 404.87445,
       829.708  ], dtype=float32)
```

The model is then evaluated on test data and we also predict the price with trained model.

```
[ ] model_i_results=evaluate_preds(y_test,tf.squeeze(model_i_preds))
    model_i_results
```

```
{'mae': 136.73756,
 'mse': 429261.94,
 'rmse': 655.18085,
 'mape': 44.417965,
 'mase': 0.9990549,
 'smape': 24.14808}
```

The predicted price is then evaluated with metric defined earlier in evaluate preds function.

2.1.8 Ensemble model

We can further improve the accuracy of our model by ensemble method. In this our model is fitted on three different metrics: sMAPE, MASE, and MAPE. We also use several number of model and get the result of thus obatained ensemble model. An **ensemble** involves training and combining multiple different models on the same problem.
Lets create SMAPE AND MASE loss function which we are going to use in our model as tensorflow doesn't have them built in .

```
from tensorflow.keras import layers,Model
```

```
[38] import tensorflow as tf
     from tensorflow.keras.losses import Loss

     class SMAPE(Loss):
         def __init__(self, **kwargs):
             super(SMAPE, self).__init__(**kwargs)

         def call(self, y_true, y_pred):
             epsilon = tf.keras.backend.epsilon()
             numerator = tf.abs(y_true - y_pred)
             denominator = tf.abs(y_true) + tf.abs(y_pred) + epsilon
             smape = tf.reduce_mean(numerator / denominator)
             return smape
     class MASE(Loss):
         def __init__(self, seasonal_period=1, **kwargs):
             super(MASE, self).__init__(**kwargs)
             self.seasonal_period = seasonal_period

         def call(self, y_true, y_pred):
             epsilon = tf.keras.backend.epsilon()


             naive_forecast = tf.roll(y_true, shift=self.seasonal_period, axis=0)
             naive_errors = tf.abs(y_true - naive_forecast)
             mae_naive = tf.reduce_mean(naive_errors)

             errors = tf.abs(y_true - y_pred)
             mae_model = tf.reduce_mean(errors)

             mase = mae_model / (mae_naive + epsilon)
             return mase
```

Now we create a function which loops every loss function for num_iter times and appends a model with different loss function each time to ensemble_models list and finally the function returns the list of ensemble models.

We call the function get_ensemble_models and store the output in ensemble_models.

```
ensemble_models=get_ensemble_models(num_iter=2,
                                    num_epochs=10)
```

```
Optimizing model by reducing: <__main__.MASE object at 0x7b748cee4e20> for 10 epochs, model number: 0
Optimizing model by reducing: mape for 10 epochs, model number: 0
Optimizing model by reducing: <__main__.SMAPE object at 0x7b748cee4b80> for 10 epochs, model number: 0
Optimizing model by reducing: <__main__.MASE object at 0x7b748cee4e20> for 10 epochs, model number: 1
Optimizing model by reducing: mape for 10 epochs, model number: 1
Optimizing model by reducing: <__main__.SMAPE object at 0x7b748cee4b80> for 10 epochs, model number: 1
```

To make a prediction out of our ensemble model we create a function as follows.

```
def make_ensemble_preds(ensemble_models,data):
  ensemble_preds=[]
  for  model in ensemble_models:
    preds=model.predict(data)
    ensemble_preds.append(preds)
  return tf.constant(tf.squeeze(ensemble_preds))
```

We get the ensemble predictions and evaluate.We take mean the ensemble predictions for evaluation.

```
ensemble_mean=tf.reduce_mean(ensemble_preds,axis=0)
ensemble_mean
```

```
<tf.Tensor: shape=(1060,), dtype=float32, numpy=
array([480.975  , 481.34134, 481.80722, ..., 480.12332, 482.8339 ,
       485.6414 ], dtype=float32)>
```

```
ensemble_results=evaluate_preds(y_true=y_test,
                                y_pred=ensemble_mean)
```

```
ensemble_results
```

```
{'mae': 6.185615,
 'mse': 162.99251,
 'rmse': 12.766851,
 'mape': 2.112101,
 'mase': 0.806018,
 'smape': 2.0389025}
```

We can also consider median of the prediction which is best choice as there is less deviation.

```
import numpy as np
```

```
ensemble_median=np.median(ensemble_preds,axis=0)
ensemble_median,len(ensemble_median)
```

```
(array([479.98215, 479.9565 , 480.47675, ..., 479.19424, 482.35233,
        484.89847], dtype=float32),
 1060)
```

```
ensemble_results=evaluate_preds(y_true=y_test,
                                y_pred=ensemble_median)
```

```
ensemble_results
```

```
{'mae': 6.1856146,
 'mse': 162.99246,
 'rmse': 12.76685,
 'mape': 2.1121006,
 'mase': 0.80601794,
 'smape': 2.0389023}
```

We get the result close to the result obtained in the generic model with just 2 num_iter and 10 epochs. We can get much better results with more epochs and more number of iterations.We got this accuracy with less epochs compared to generic model which had 5000 epochs .

Test case 2

```
%%time
ensemble_models=get_ensemble_models(num_iter=3,
                                    num_epochs=10)
```

```
Optimizing model by reducing: <__main__.MASE object at 0x7b141b764730> for 10 epochs, model number: 0
Optimizing model by reducing: mape for 10 epochs, model number: 0
Optimizing model by reducing: <__main__.SMAPE object at 0x7b141b7645e0> for 10 epochs, model number: 0
Optimizing model by reducing: <__main__.MASE object at 0x7b141b764730> for 10 epochs, model number: 1
Optimizing model by reducing: mape for 10 epochs, model number: 1
Optimizing model by reducing: <__main__.SMAPE object at 0x7b141b7645e0> for 10 epochs, model number: 1
Optimizing model by reducing: <__main__.MASE object at 0x7b141b764730> for 10 epochs, model number: 2
Optimizing model by reducing: mape for 10 epochs, model number: 2
Optimizing model by reducing: <__main__.SMAPE object at 0x7b141b7645e0> for 10 epochs, model number: 2
CPU times: user 9min 46s, sys: 13 s, total: 9min 59s
Wall time: 11min 17s
```

```
ensemble_results=evaluate_preds(y_true=y_test,
                                y_pred=ensemble_mean)
ensemble_results
```

```
{'mae': 6.0420976,
 'mse': 154.26141,
 'rmse': 12.420201,
 'mape': 1.9720353,
 'mase': 0.79090637,
 'smape': 1.9250772}
```

```
ensemble_results=evaluate_preds(y_true=y_test,
                                y_pred=ensemble_median)
```

```
ensemble_results
```

```
{'mae': 6.0420985,
 'mse': 154.2614,
 'rmse': 12.420201,
 'mape': 1.9720355,
 'mase': 0.7909064,
 'smape': 1.9250777}
```

3. Conclusion

We have seen the various models with different results and accuracy and tried to forecast the price of NIFTY50 data using deep learning with tensorflow with two different configurations of NBEATs algorithm:NBeats Generic and NBeats Interpretable model. Each has its own benefits of being interpretable and giving us the accurate results. Generic model gave us the accurate results while Interpretable model was giving us the extra benefit of investigating our output.We also used the concept of ensemble for improving our result further and indeed see that pure DL methods without any specific time series or hybrid model we are able to produce the good results which was the aim of the NBEATS paper . This is a reference to the N-BEATS paper [1].

References

[1] Oreshkin, Boris N., Dmitri Carpov, Nicolas Chapados, and Yoshua Bengio: *N-beats: Neural basis expansion analysis for interpretable time series forecasting*. arXiv preprint arXiv:1905.10437, 2019. https://arxiv.org/pdf/1905.10437.