**N-BEATS:**

N-BEATS (Non-linear forecasting with basis functions) is a neural network architecture for time series forecasting.

The main idea behind N-BEATS is to decompose a complex time series into a sum of simpler functions, called basis functions, which can be learned by a neural network.

N-BEATS consists of two main components: the first is a stack of fully connected layers called the "backcast" model, and the second is a stack of fully connected layers called the "forecast" model. The backcast model is responsible for learning the history of the time series up to the current time step, while the forecast model is responsible for predicting future values.

The backcast and forecast models are trained separately, but they share the same set of weights. This allows the model to learn complex interactions between the past and the future of the time series.

The backcast model is composed of a stack of fully connected layers, called "blocks", each one with a specific number of neurons. Each block takes as input the concatenation of the past values of the time series and the output of the previous block. The output of the last block is then used as input for the forecast model.

The forecast model is also composed of a stack of fully connected layers, called "heads". Each head takes as input the output of the backcast model and predicts a future value of the time series. The final output of the N-BEATS is the sum of the predictions of all heads.

By this way the model learns the non-linear function that best approximate the target series, it can handle multiple input and output sequences, and is particularly well-suited for long-term forecasting.

**HYPERPARAMETER TUNNING:**

The specific hyperparameters that you should tune for your N-BEATS model will depend on the characteristics of your dataset and the specific problem you're trying to solve. However, here are a few hyperparameters that are commonly tuned for N-BEATS and similar models:

Number of blocks: The number of blocks in the backcast model of the N-BEATS architecture. This parameter controls the depth of the model.

Number of neurons per block: The number of neurons per block in the backcast model of the N-BEATS architecture. This parameter controls the capacity of the model.

Number of heads: The number of heads in the forecast model of the N-BEATS architecture. This parameter controls the number of parallel outputs of the model.

Number of neurons per head: The number of neurons per head in the forecast model of the N-BEATS architecture. This parameter controls the capacity of each output of the model.

Learning rate: The learning rate of the optimizer used during the training process. This parameter controls the step size of the gradient descent algorithm.

Dropout rate: The dropout rate applied to the input of each block and head in the N-BEATS architecture. This parameter helps to prevent overfitting.

Number of training epochs: The number of times the model is trained on the entire dataset.

It's also important to keep in mind that the performance of the model will depend on many factors, such as the quality of the data, the preprocessing techniques used, and the hyperparameters of the model. Therefore, it would be a good idea to use techniques such as cross validation and grid search to find the best set of hyperparameters for your specific dataset.

Input & Output Layers:

- The sizes of the input and output layers should be adequate to assign a node to each feature.

- The input chunk length should not be smaller than the order of seasonality.

- For efficient memory usage, set them to a power of 2

Batch Size:

- Number of observations the model will process before it updates its matrix weights.

- For efficient memory usage, set them to a power of 2.

- Very large batch sizes may mislead the gradient descent in a single direction (sub-optimal minimum)

- Smaller batch sizes will cause the gradient descent to bounce around in different directions

- This reduces accuracy, but can prevent overfitting.

- Usually you should choose an initial batch size of 32.

Epochs:

- Tells the model how many training cycles it is supposed to run.

- During each epoch, the model will process the entire training set, making one forward and one

backward pass.

The product of these hyperparameters defines the tensor size of the model.

- Large parameter values can make it bump against the memory limit of your system and will lead to

exponentially longer processing times.

- Small parameter values may turn out to be inadequate to mirror complex patterns in the source data.

def optimize\_nbeats(params):

# create the N-BEATS model with the given set of hyperparameters

nbeats = NBEATSModel(

backcast\_length=params['backcast\_length'],

forecast\_length=params['forecast\_length'],

stack\_types=[params['stack\_type']],

nb\_blocks\_per\_stack=params['nb\_blocks\_per\_stack'],

share\_weights\_in\_stack=params['share\_weights\_in\_stack'],

hidden\_layer\_units=params['hidden\_layer\_units']

)

# fit the model on the training data

nbeats.fit(

x\_train,

y\_train,

batch\_size=params['batch\_size'],

epochs=params['epochs'],

verbose=0

)

# evaluate the model on the validation data

y\_pred = nbeats.predict(x\_val)

mse = mean\_squared\_error(y\_val, y\_pred)

# return the negative of the evaluation metric

return -mse

search\_space = {

'backcast\_length': hp.quniform('backcast\_length', 10, 100, 10),

'forecast\_length': hp.quniform('forecast\_length', 10, 100, 10),

'stack\_type': hp.choice('stack\_type', ['linear', 'dense']),

'nb\_blocks\_per\_stack': hp.quniform('nb\_blocks\_per\_stack', 1, 3, 1),

'share\_weights\_in\_stack': hp.choice('share\_weights\_in\_stack', [True, False]),

'hidden\_layer\_units': hp.quniform('hidden\_layer\_units', 10, 100, 10),

'batch\_size': hp.quniform('batch\_size', 32, 256, 32),

'epochs': hp.quniform('epochs', 10, 100, 10)

}

best = fmin(fn=optimize\_nbeats, space=search\_space, algo=None)

**DOCKER, AWS, EC2 & S3:**

Docker is a platform that allows developers to easily create, deploy, and run applications in containers. Containers are a lightweight, portable, and self-sufficient way of packaging software and dependencies together, making it easy to run the same application on different environments and platforms.

Docker uses a technology called containerization, which allows developers to package an application and its dependencies together in a container. This container can then be run on any machine that has the Docker runtime installed, regardless of the underlying operating system or infrastructure.

Docker provides several benefits over traditional virtualization or deployment methods, such as:

Portability: Docker containers can be easily moved between different environments, such as development, staging, and production, without the need to make any changes to the application or its dependencies.

Scalability: Docker containers can be easily scaled up or down to handle changes in traffic or load.

Isolation: Docker containers provide a level of isolation between different applications, which helps to prevent conflicts and improve security.

Efficiency: Docker containers are more lightweight and efficient than traditional virtual machines, which means they can be run on more resources-constrained environments, such as laptops or servers with limited resources.

Docker is used in a wide range of applications, from web development and data science to machine learning and big data. It's also widely used in cloud computing, as it makes it easy to deploy and scale applications on cloud platforms such as AWS, Azure, and Google Cloud.

In summary, Docker is a platform and toolset that allows developers to create, deploy, and run applications in containers, which provides a lightweight, portable, and self-sufficient way of packaging software and dependencies together.

Docker, AWS EC2 and S3 can be used together to deploy and run a Python application in the following way:

Dockerize the application: First, you'll need to create a Docker image of your Python application. This involves creating a Dockerfile that defines the dependencies and configurations needed to run the application in a container. Once the Dockerfile is created, you can use the docker build command to create an image of your application.

Push the image to a container registry: Next, you'll need to push the image to a container registry, such as Docker Hub or Amazon Elastic Container Registry (ECR). This will allow you to easily access the image from any machine that has access to the registry.

Launch an EC2 instance: Next, you'll need to launch an EC2 instance on AWS. This is a virtual machine that you can use to run your container. You can choose the instance type and size based on your application's resource requirements.

Run the container on the EC2 instance: Once the EC2 instance is running, you can use the docker run command to start your container on the instance. This will make your application accessible to users via the public IP address or the DNS name of the EC2 instance.

Store data in S3: If your application needs to store data, you can use Amazon S3 to store files, images, and other data. You can use the AWS SDK for Python (boto3) to interact with S3 and store or retrieve data from your application.

Automate the process: To automate the process of building, pushing, and deploying your application, you can use AWS CodeBuild, CodeCommit, and CodePipeline. These tools can be used together to automate the process of building your application, testing it, and deploying it to your EC2 instances.

Monitoring and scaling: Finally, you can use AWS tools such as CloudWatch to monitor your application's performance, and Auto Scaling to automatically scale your EC2 instances based on demand.

In summary, with the combination of Docker, AWS EC2, and S3, you can containerize your Python application, store it in a container registry, launch an EC2 instance on AWS, run the container on that instance and make the application accessible to users, store data in S3, automate the process with CodeBuild, CodeCommit, and CodePipeline, and monitor and scale your application using CloudWatch and Auto Scaling.