STEPS TO WRITE CODE LINE BY LINE FOR THIS PROJECT OF CHICKEN DISEASE

1. Create a Git repository (add readme, git ignore file as python and license as MIT)
2. Open Git bash, navigate to your cwd and get clone the repo

git clone (paste the repo link)

1. Create a template.py file to setup the file structure and write the following code in it.
2. import os
3. from pathlib import Path  #this will help to recognise that it is windows path and ignore the "/"
4. import logging
5. logging.basicConfig(level=logging.INFO, format='[%(asctime)s]: %(message)s: %(lineno)s')
6. project\_name = "cnnClassifier"
7. list\_of\_files=[
8. ".github/workflows/.gitkeep",       #used when deploying the app, writing all CI/CD commands (gitkeep is used, when if you commit the code and any file is empty)
9. f"src/{project\_name}/\_\_init\_\_.py",  #\_\_init\_\_.py is a constructor file which is needed to use that file methods in another file
10. f"src/{project\_name}/components/\_\_init\_\_.py",
11. f"src/{project\_name}/utils/\_\_init\_\_.py",
12. f"src/{project\_name}/config/\_\_init\_\_.py",
13. f"src/{project\_name}/config/configuration/\_\_init\_\_.py",
14. f"src/{project\_name}/pipeline/\_\_init\_\_.py",
15. f"src/{project\_name}/entity/\_\_init\_\_.py",
16. f"src/{project\_name}/constants/\_\_init\_\_.py",
17. "config/config.yaml",
18. "dvc.yaml",
19. "params.yaml",
20. "requirements.txt",
21. "setup.py",
22. "research/trails.ipynb",
23. "templates/index.html"
24. ]
25. for filepath in list\_of\_files:
26. filepath = Path(filepath)
27. filedir, filename = os.path.split(filepath)
28. if filedir != "":
29. os.makedirs(filedir, exist\_ok=True)
30. logging.info(f"Creating directory: {filedir} for the file: {filename}")
31. if (not os.path.exists(filepath)) or (os.path.getsize(filepath)==0):
32. with open(filepath, "w") as f:
33. pass
34. logging.info(f"Creating empty file: {filepath}")
35. else:
36. logging.info(f"{filename} is already existing")
37. Write the requirements.txt and add the below code
38. tensorflow
39. pandas
40. dvc
41. notebook
42. numpy
43. matplotlib
44. seaborn
45. python-box==6.0.2
46. pyYAML
47. tqdm
48. ensure==1.0.2
49. joblib
50. types-PyYAML
51. scipy
52. FLask
53. Flask-Cors

5. So after writing these required libraries, we need to create our new environment variable for our project, so write the following code in the terminal

conda create –n “name\_for\_variable(chicken” python=3.8 –y

6. Activate the env : conda activate chicken

7. Now install the requirements using the following code-

pip install –r requirements.txt

8. We need to log our actions so for that inside the src/cnnClassifier contructor file made, write the below code

import os

import sys

import logging

logging\_str = "[%(asctime)s: %(levelname)s: %(module)s: %(message)s]"

log\_dir = "logs"

log\_filepath = os.path.join(log\_dir, "running\_logs.log")

os.makedirs(log\_dir, exist\_ok=True)

logging.basicConfig(

    level= logging.INFO,

    format= logging\_str,

    handlers= [

        logging.FileHandler(log\_filepath),          #this will add the dir to the given path

        logging.StreamHandler(sys.stdout)           #tthis wil show the log on your terminal as well

    ]

)

logger = logging.getLogger("cnnClassifierLogger")

This will create a file named logs and will store every logging information inside it.

9. We have to create utils, just to store the repetitive functions used in the projects, so create common.py inside the utils folder

\* ensure\_annotations – This is used to indicate error such as when int tries to multiply with str and which python allows normally but using this decorator, we can debug the code and understand the issue.

\* Config\_box – using this we can access the keys of dictionaries using “.” Operator, we have import Box and then write while defining a dictionary = ConfigBox({‘key’:’value’})

The below code is needed to put in common.py folder

import os

from box.exceptions import BoxValueError

import yaml

from cnnClassifier import logger

import json

import joblib

from ensure import ensure\_annotations

from box import ConfigBox

from pathlib import Path

from typing import Any

import base64

@ensure\_annotations

def read\_yaml(path\_to\_yaml: Path) -> ConfigBox:

    """read yaml file and returns

    Args:

        path\_to\_yaml(str): path like input

    Raises:

        ValueError: if yaml file is empty

        e: empty file

    Returns:

        ConfigBox: ConfigBox type

    """

    try:

        with open(path\_to\_yaml) as yaml\_file:

            content =  yaml.safe\_load(yaml\_file)

            logger.info(f"yaml file: {path\_to\_yaml} loaded successfully")

            return ConfigBox(content)

    except BoxValueError:

        raise ValueError("yaml is empty")

    except Exception as e:

        raise e

@ensure\_annotations

def create\_directories(path\_to\_directories: list, verbose=True):            #this is used to make the directories in the artifacts folder i.e.data ingestion, model, etc

    """"creaate list of directories

    Args:

        path\_to\_directories (list): list of path of directories

        ignore\_log (bool, optional): ignore if multiple dirs is to be created. Defaults to False

    """

    for path in path\_to\_directories:

        os.makedirs(path, exist\_ok=True)

        if verbose:

            logger.infor(f"Created directory at: {path}")

def save\_json(path: Path, data:dict):           #this json file is created to save the accuracy and loss after evalaution model.

    """ save json data

    Args:

        path (Path): path to json file

        data (dict): data to be saved in json file.

    """

    with open(path, "w") as f:

        json.dump(data, f, indent=4)

    logger.info(f"json file saved at {path}")

@ensure\_annotations

def load\_json(path: Path) -> ConfigBox:

    """ load json files data

    Args:

        path (Path): path to json file

    Returns:

        ConfigBox: data as class attributes instead of dict

    """

    with open(path) as f:

        content=json.load(f)

    logger.info(f"json file loaded successfully from {path}")

    return ConfigBox(content)

@ensure\_annotations

def save\_bin(data: Any, path: Path) -> ConfigBox:

    """ save binary file

    Args:

        data (Any): data to be saved as binary

        path (Path): path to binary file

    """

    joblib.dump(value=data, filename=path)

    logger.info(f"binary file saved at {path}")

@ensure\_annotations

def load\_bin(path: Path) -> ConfigBox:

    """ load binary data

    Args:

        path (Path): path to binary file

    Returns:

        data (Any): object stored in the file

    """

    data = joblib.load(path)

    logger.info(f"binary file loaded from {path}")

    return data

@ensure\_annotations

def get\_size(path: Path) -> ConfigBox:

    """ get size in Kb

    Args:

        path (Path): path of the file

    Returns:

        str: size in Kb

    """

    size\_in\_kb = round(os.path.getsize(path)/1024)

    return f"~ {size\_in\_kb} KB"

def decodeImage (imgstring, filename):

    imgdata = base64.b64decode(imgstring)

    with open(filename, 'wb') as f:

        f.write(imgdata)

        f.close()

def enodeImageIntoBase64(croppedImagePath):

    with open(croppedImagePath, "rb") as f:

        return base64.b64dencode(f.read())

10. Now we have discussed the workflow, that which file is needed to start with

##Workflows

1. Update config.yaml

2. Update secrets.yaml [optional]

3. update params.yaml

4. update the entity

5. update the configuration manager in src config

6. Update the components

7. Update the pipeline

8. Update the main.py

9. update the dvc.yaml

11. After that we drag and drop the zip dataset file in github

12. After this we have to prepare the data ingestion notebook and write the code there, and once when the code perfect we have to put it in modular coding format

13. Inside confgi.yaml, we need to write the artifacts folder and it’s subfolders name because even if you want to change any folder names, you change it from here.

artifacts\_root: artifacts

data\_ingestion:

  root\_dir: artifacts/data\_ingestion

  source\_URL: https://github.com/uv0037/Chicken\_disease\_classification/raw/main/Chicken-fecal-images.zip

  local\_data\_file: artifacts/data\_ingestion/data.zip

  unzip\_dir: artifacts/data\_ingestion

14. Now to write the code in data ingestion file.

from dataclasses import dataclass

from pathlib import Path

@dataclass(frozen=True)

class DataIngestionConfig:

    root\_dir: Path

    source\_URL: str

    local\_data\_file: Path

    unzip\_dir: Path

from cnnClassifier.constants import \*

from cnnClassifier.utils.common import read\_yaml, create\_directories

class ConfigurationManager:

    def \_\_init\_\_(

        self,

        config\_filepath = CONFIG\_FILE\_PATH,

        params\_filepath = PARAMS\_FILE\_PATH):

        self.config = read\_yaml(config\_filepath)

        self.params = read\_yaml(PARAMS\_FILE\_PATH)

        create\_directories([self.config.artifacts\_root])

    def get\_data\_ingestion\_config(self) -> DataIngestionConfig:

        config = self.config.data\_ingestion

        create\_directories([config.root\_dir])

        data\_ingestion\_config = DataIngestionConfig(

            root\_dir = config.root\_dir,

            source\_URL = config.source\_URL,

            local\_data\_file = config.local\_data\_file,

            unzip\_dir = config.unzip\_dir

        )

        return data\_ingestion\_config

import os

import urllib.request as request

import zipfile

from cnnClassifier import logger

from cnnClassifier.utils.common import get\_size

class DataIngestion:

    def \_\_init\_\_(self, config: DataIngestionConfig):

        self.config=config

    def download\_file(self):

        if not os.path.exists(self.config.local\_data\_file):

            filename, headers = request.urlretrieve(

                url = self.config.source\_URL,

                filename = self.config.local\_data\_file

            )

            logger.info(f"{filename} download with the below headers \n{headers}")

        else:

            logger.info(f"File already exists of size: {get\_size(Path(self.config.local\_data\_file))}")

    def extract\_zip\_file(self):

        unzip\_path = self.config.unzip\_dir

        os.makedirs(unzip\_path, exist\_ok=True)

        with zipfile.ZipFile(self.config.local\_data\_file, 'r') as zip\_ref:

            zip\_ref.extractall(unzip\_path)

try:

    config = ConfigurationManager()

    data\_ingestion\_config = config.get\_data\_ingestion\_config()

    data\_ingestion = DataIngestion(config=data\_ingestion\_config)

    data\_ingestion.download\_file()

    data\_ingestion.extract\_zip\_file()

except Exception as e:

    raise e

The above code will create the folder artifacts, download the dataset zip file from github link and then unzip it.

15. Create a new file in entity as config\_entity.py and paste the following code  
  
from dataclasses import dataclass

from pathlib import Path

@dataclass(frozen=True)

class DataIngestionConfig:

    root\_dir: Path

    source\_URL: str

    local\_data\_file: Path

    unzip\_dir: Path

16. paste the following code in configurationmanager() code in the config/configuration.py  
from cnnClassifier.constants import \*

from cnnClassifier.utils.common import read\_yaml, create\_directories

from cnnClassifier.entity.config\_entity import DataIngestionConfig

class ConfigurationManager:

    def \_\_init\_\_(

        self,

        config\_filepath = CONFIG\_FILE\_PATH,

        params\_filepath = PARAMS\_FILE\_PATH):

        self.config = read\_yaml(config\_filepath)

        self.params = read\_yaml(PARAMS\_FILE\_PATH)

        create\_directories([self.config.artifacts\_root])

    def get\_data\_ingestion\_config(self) -> DataIngestionConfig:

        config = self.config.data\_ingestion

        create\_directories([config.root\_dir])

        data\_ingestion\_config = DataIngestionConfig(

            root\_dir = config.root\_dir,

            source\_URL = config.source\_URL,

            local\_data\_file = config.local\_data\_file,

            unzip\_dir = config.unzip\_dir

        )

        return data\_ingestion\_config

and now paste the following code in stage\_01\_data\_ingestion.py made in components

The following code is a standalone code means we can run the data ingestion separately as well . this is a pipeline we have created .  
from cnnClassifier.config.configuration import ConfigurationManager

from cnnClassifier.components.data\_ingestion import DataIngestion

from cnnCLassifier import logger

STAGE\_NAME = "Data Ingestion stage"

class DataIngestionTrainingPipeline:

    def \_\_init\_\_(self):

        pass

    def main(self):

        config = ConfigurationManager()

        data\_ingestion\_config = config.get\_data\_ingestion\_config()

        data\_ingestion = DataIngestion(config=data\_ingestion\_config)

        data\_ingestion.download\_file()

        data\_ingestion.extract\_zip\_file()

if \_\_name\_\_ == '\_\_main\_\_':

    try:

        logger.info(f">>> stage {STAGE\_NAME} started <<<")

        obj = DataIngestionTrainingPipeline()

        obj.main()

        logger.info(f">>> stage {STAGE\_NAME} completed <<<")

    except Exception as e:

        raise e

17. This pipeline we have to call it in the main.py so we need to import the pipeline class in main.py as per given code

from cnnClassifier import logger

from cnnClassifier.pipeline.stage\_01\_data\_ingestion import DataIngestionTrainingPipeline

STAGE\_NAME ="Data Ingestion stage"

try:

    logger.info(f">>> stage {STAGE\_NAME} started<<<")

    data\_ingestion = DataIngestionTrainingPipeline()

    data\_ingestion.main()

    logger.info(f">>> stage {STAGE\_NAME} completed <<<")

except Exception as e:

    logger.exception(e)

    raise e

18. Now we have to prepare the base model which in we are going to use Keras’s VGG16

So same like the data ingestion notebook, we have create prepare\_base\_model notebook

And update param.yaml as well

Params.yaml

AUGMENTATION: True

IMAGE\_SIZE: [224, 224, 3]

BATCH\_SIZE: 16

INCLUDE\_TOP: False

EPOCHS: 1

CLASSES: 2

WEIGHTS: imagenet

LEARNING\_RATE: 0.01

19. After this we need to do the same just the way we did for data ingestion

In 02.prepare\_base\_model.ipynb write the given code

from dataclasses import dataclass

from pathlib import Path

@dataclass(frozen=True)

class PrepareBaseModelConfig:

    root\_dir: Path

    base\_model\_path: Path

    updated\_base\_model\_path: Path

    params\_image\_size: list

    params\_learning\_rate: float

    params\_include\_top: bool

    params\_weights: str

    params\_classes: int

from cnnClassifier.constants import \*

from cnnClassifier.utils.common import read\_yaml, create\_directories

class ConfigurationManager:

    def \_\_init\_\_(

        self,

        config\_filepath = CONFIG\_FILE\_PATH,

        params\_filepath = PARAMS\_FILE\_PATH):

        self.config = read\_yaml(config\_filepath)

        self.params = read\_yaml(params\_filepath)

        create\_directories([self.config.artifacts\_root])

    def get\_prepare\_base\_model\_config(self) -> PrepareBaseModelConfig:

        config = self.config.prepare\_base\_model

        create\_directories([config.root\_dir])

        prepare\_base\_model\_config = PrepareBaseModelConfig(

            root\_dir = Path(config.root\_dir),

            base\_model\_path = Path(config.base\_model\_path),

            updated\_base\_model\_path = Path(config.updated\_base\_model\_path),

            params\_image\_size = self.params.IMAGE\_SIZE,

            params\_learning\_rate = self.params.LEARNING\_RATE,

            params\_include\_top = self.params.INCLUDE\_TOP,

            params\_weights = self.params.WEIGHTS,

            params\_classes = self.params.CLASSES

        )

        return prepare\_base\_model\_config

import os

import urllib.request as request

from zipfile import ZipFile

import tensorflow as tf

class PrepareBaseModel:

    def \_\_init\_\_(self, config:PrepareBaseModelConfig):

        self.config = config

    def get\_base\_model(self):

        self.model = tf.keras.applications.vgg16.VGG16(

            input\_shape=self.config.params\_image\_size,

            weights=self.config.params\_weights,

            include\_top=self.config.params\_include\_top

        )

        self.save\_model(path=self.config.base\_model\_path, model=self.model)

    @staticmethod

    def \_prepare\_full\_model(model, classes, freeze\_all, freeze\_till, learning\_rate):

        if freeze\_all:

            for layers in model.layers:

                model.trainable = False

        elif (freeze\_till is not None) and (freeze\_till > 0):

            for layers in model.layers[:-freeze\_till]:

                model.trainable = False

        flatten\_in = tf.keras.layers.Flatten()(model.output)

        prediction = tf.keras.layers.Dense(

            units=classes,

            activation="softmax"

        )(flatten\_in)

        full\_model = tf.keras.models.Model(

            inputs = model.input,

            outputs = prediction

        )

        full\_model.compile(

            optimizer = tf.keras.optimizers.SGD(learning\_rate=learning\_rate),

            loss = tf.keras.losses.CategoricalCrossentropy(),

            metrics = ["accuracy"]

        )

        full\_model.summary()

        return full\_model

    def update\_base\_model(self):

        self.full\_model=self.\_prepare\_full\_model(

            model = self.model,

            classes = self.config.params\_classes,

            freeze\_all = True,

            freeze\_till = None,

            learning\_rate = self.config.params\_learning\_rate

        )

        self.save\_model(path=self.config.updated\_base\_model\_path, model=self.full\_model)

    @staticmethod

    def save\_model(path: Path, model: tf.keras.Model):

        model.save(path)

try:

    config = ConfigurationManager()

    prepare\_base\_model\_config = config.get\_prepare\_base\_model\_config()

    prepare\_base\_model = PrepareBaseModel(config=prepare\_base\_model\_config)

    prepare\_base\_model.get\_base\_model()

    prepare\_base\_model.update\_base\_model()

except Exception as e:

    raise e

this will be the rough code for preparing the model, now we have to convert this rough code into modular coding.

1. Update the config\_entity.py

from dataclasses import dataclass

from pathlib import Path

@dataclass(frozen=True)

class DataIngestionConfig:

    root\_dir: Path

    source\_URL: str

    local\_data\_file: Path

    unzip\_dir: Path

@dataclass(frozen=True)

class PrepareBaseModelConfig:

    root\_dir: Path

    base\_model\_path: Path

    updated\_base\_model\_path: Path

    params\_image\_size: list

    params\_learning\_rate: float

    params\_include\_top: bool

    params\_weights: str

    params\_classes: int

1. Update the configuration.py file as well as import the class the config\_entity.
2. from cnnClassifier.constants import \*
3. from cnnClassifier.utils.common import read\_yaml, create\_directories
4. from cnnClassifier.entity.config\_entity import (DataIngestionConfig,
5. PrepareBaseModelConfig)
6. class ConfigurationManager:
7. def \_\_init\_\_(
8. self,
9. config\_filepath = CONFIG\_FILE\_PATH,
10. params\_filepath = PARAMS\_FILE\_PATH):
11. self.config = read\_yaml(config\_filepath)
12. self.params = read\_yaml(PARAMS\_FILE\_PATH)
13. create\_directories([self.config.artifacts\_root])
14. def get\_data\_ingestion\_config(self) -> DataIngestionConfig:
15. config = self.config.data\_ingestion
16. create\_directories([config.root\_dir])
17. data\_ingestion\_config = DataIngestionConfig(
18. root\_dir = config.root\_dir,
19. source\_URL = config.source\_URL,
20. local\_data\_file = config.local\_data\_file,
21. unzip\_dir = config.unzip\_dir
22. )
23. return data\_ingestion\_config
25. def get\_prepare\_base\_model\_config(self) -> PrepareBaseModelConfig:
26. config = self.config.prepare\_base\_model
28. create\_directories([config.root\_dir])
29. prepare\_base\_model\_config = PrepareBaseModelConfig(
30. root\_dir = Path(config.root\_dir),
31. base\_model\_path = Path(config.base\_model\_path),
32. updated\_base\_model\_path = Path(config.updated\_base\_model\_path),
33. params\_image\_size = self.params.IMAGE\_SIZE,
34. params\_learning\_rate = self.params.LEARNING\_RATE,
35. params\_include\_top = self.params.INCLUDE\_TOP,
36. params\_weights = self.params.WEIGHTS,
37. params\_classes = self.params.CLASSES
38. )
39. return prepare\_base\_model\_config

23. Now in components folder, make new file for prepare\_base\_model.py and write the following code  
import os

import urllib.request as request

from zipfile import ZipFile

import tensorflow as tf

from pathlib import Path

from cnnClassifier.entity.config\_entity import PrepareBaseModelConfig

class PrepareBaseModel:

    def \_\_init\_\_(self, config:PrepareBaseModelConfig):

        self.config = config

    def get\_base\_model(self):

        self.model = tf.keras.applications.vgg16.VGG16(

            input\_shape=self.config.params\_image\_size,

            weights=self.config.params\_weights,

            include\_top=self.config.params\_include\_top

        )

        self.save\_model(path=self.config.base\_model\_path, model=self.model)

    @staticmethod

    def \_prepare\_full\_model(model, classes, freeze\_all, freeze\_till, learning\_rate):

        if freeze\_all:

            for layers in model.layers:

                model.trainable = False

        elif (freeze\_till is not None) and (freeze\_till > 0):

            for layers in model.layers[:-freeze\_till]:

                model.trainable = False

        flatten\_in = tf.keras.layers.Flatten()(model.output)

        prediction = tf.keras.layers.Dense(

            units=classes,

            activation="softmax"

        )(flatten\_in)

        full\_model = tf.keras.models.Model(

            inputs = model.input,

            outputs = prediction

        )

        full\_model.compile(

            optimizer = tf.keras.optimizers.SGD(learning\_rate=learning\_rate),

            loss = tf.keras.losses.CategoricalCrossentropy(),

            metrics = ["accuracy"]

        )

        full\_model.summary()

        return full\_model

    def update\_base\_model(self):

        self.full\_model=self.\_prepare\_full\_model(

            model = self.model,

            classes = self.config.params\_classes,

            freeze\_all = True,

            freeze\_till = None,

            learning\_rate = self.config.params\_learning\_rate

        )

        self.save\_model(path=self.config.updated\_base\_model\_path, model=self.full\_model)

    @staticmethod

    def save\_model(path: Path, model: tf.keras.Model):

        model.save(path)

24. Now to update the pipeline, the write(update) the following code in main.py and don’t forget to import the classes  
from cnnClassifier import logger

from cnnClassifier.pipeline.stage\_01\_data\_ingestion import DataIngestionTrainingPipeline

from cnnClassifier.pipeline.stage\_02\_prepare\_base\_model import PrepareBaseModelTrainingPipeline

STAGE\_NAME ="Data Ingestion stage"

try:

    logger.info(f">>> stage {STAGE\_NAME} started<<<")

    data\_ingestion = DataIngestionTrainingPipeline()

    data\_ingestion.main()

    logger.info(f">>> stage {STAGE\_NAME} completed <<<")

except Exception as e:

    logger.exception(e)

    raise e

STAGE\_NAME ="Prepare base model stage"

try:

    logger.info("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

    logger.info(f">>> stage {STAGE\_NAME} started<<<")

    prepare\_base\_model = PrepareBaseModelTrainingPipeline()

    prepare\_base\_model.main()

    logger.info(f">>> stage {STAGE\_NAME} completed <<<")

except Exception as e:

    logger.exception(e)

    raise e

25. Now let start with next component prepare\_callbacks notebook, this will save the logs regarding to the model performance, the epochs with which the model is performing the best. (checkpoints in this is referred to the point where training of specific number of epochs so basically you can start training from that checkpoint too.)

26. make new file 03.prepare\_callbacks.ipynb in research folder and firstly go the main folder by os.chdir(“../”)

27. Update the config.yaml as below

artifacts\_root: artifacts

data\_ingestion:

  root\_dir: artifacts/data\_ingestion

  source\_URL: https://github.com/uv0037/Chicken\_disease\_classification/raw/main/Chicken-fecal-images.zip

  local\_data\_file: artifacts/data\_ingestion/data.zip

  unzip\_dir: artifacts/data\_ingestion

prepare\_base\_model:

  root\_dir: artifacts/prepare\_base\_model

  base\_model\_path: artifacts/prepare\_base\_model/base\_model.h5

  updated\_base\_model\_path: artifacts/prepare\_base\_model/base\_model\_updated.h5

prepare\_callbacks:

  root\_dir: artifacts/prepare\_callbacks

  tensorboard\_root\_log\_dir: artifacts/prepare\_callbacks/tensorboard\_log\_dir

  checkpoint\_model\_filepath: artifacts/prepare\_callbacks/checkpoint\_dir/model.h5

28. Now write the following code in the callback notebook

def \_\_init\_\_(

        self,

        config\_filepath = CONFIG\_FILE\_PATH,

        params\_filepath = PARAMS\_FILE\_PATH):

        self.config = read\_yaml(config\_filepath)

        self.params = read\_yaml(params\_filepath)

        create\_directories([self.config.artifacts\_root])

from cnnClassifier.constants import \*

from cnnClassifier.utils.common import read\_yaml, create\_directories

class ConfigurationManager:

    def \_\_init\_\_(

        self,

        config\_filepath = CONFIG\_FILE\_PATH,

        params\_filepath = PARAMS\_FILE\_PATH):

        self.config = read\_yaml(config\_filepath)

        self.params = read\_yaml(params\_filepath)

        create\_directories([self.config.artifacts\_root])

    def get\_prepare\_callback\_config(self) -> PrepareCallbacksConfig:

        config = self.config.prepare\_callbacks

        model\_ckpt\_dir = os.path.dirname(config.checkpoint\_model\_filepath)

        create\_directories([

            Path(model\_ckpt\_dir),

            Path(config.tensorboard\_root\_log\_dir)

        ])

        prepare\_callback\_config = PrepareCallbacksConfig(

            root\_dir=Path(config.root\_dir),

            tensorboard\_root\_log\_dir=Path(config.tensorboard\_root\_log\_dir),

            checkpoint\_model\_filepath=Path(config.checkpoint\_model\_filepath)

        )

        return prepare\_callback\_config

import os

import urllib.request as request

from zipfile import ZipFile

import tensorflow as tf

import time

class PrepareCallbacks:

    def \_\_init\_\_(self, config: PrepareCallbacksConfig):

        self.config = config

    @property

    def \_create\_tb\_callbacks(self):

        timestamp = time.strftime("%Y-%M-%d-%H:%M:%S")

        tb\_running\_log\_dir = os.path.join(

            self.config.tensorboard\_root\_log\_dir,

            f"tb\_logs\_at{timestamp}"

        )

        return tf.keras.callbacks.TensorBoard(log\_dir=tb\_running\_log\_dir)

    @property

    def \_create\_ckpt\_callbacks(self):

        return tf.keras.callbacks.ModelCheckpoint(

            filepath = self.config.checkpoint\_model\_filepath,

            save\_best\_only=True

        )

    def get\_tb\_ckpt\_callbacks(self):

        return [

            self.\_create\_tb\_callbacks,

            self.get\_tb\_ckpt\_callbacks

        ]

try:

    config = ConfigurationManager()

    prepare\_callbacks\_config = config.get\_prepare\_callback\_config()

    prepare\_callbacks = PrepareCallbacks(config=prepare\_callbacks\_config)

    callback\_list = prepare\_callbacks.get\_tb\_ckpt\_callbacks()

except Exception as e:

    raise e

29. Copy the entity and configuration code to its respective folder to convert this into modular coding.

30. For this component we don’t need to prepare a .py file in pipeline folder because this will be called and used during training the model.

31. .