DEEP LEARNING PROJECT PLAN – HEART DISEASE PREDICTION
(Local machine + VS Code + 3-person team)

Context:
You will work on your LOCAL machine (not Colab) and you want the EASIEST possible workflow.
You will use VS Code as your main IDE. The project uses a meaningful medical dataset from Kaggle
to predict heart disease.

Final objective:
Build a simple deep learning model in Keras/TensorFlow that predicts whether a patient has heart disease,
evaluate it with clear metrics, save the model, and write a short report that explains all steps.

At the end of the project you must have:
- A working Python environment on your computer (Python 3.9) inside VS Code.
- One main notebook (or script) that goes from data loading to model evaluation.
- A saved model file (for example: models/heart_failure_model.h5).
- A short written report (2–5 pages) that you can export as PDF.

==
SECTION 0 – Team organization (3 persons, in parallel)
==

To save time, the project is split into 3 logical parts that can be done in parallel.
Each person has clear responsibilities and deliverables, and everything is easy to merge at the end.

Person A – Data & Setup (Phases 1–4)
- Prepares project folder structure and Python virtual environment.
- Sets up VS Code for the project.
- Downloads the Kaggle dataset and places it in data/.
- Performs basic EDA (exploratory data analysis).
- Implements the full preprocessing pipeline:
  handling missing values, encoding categorical features, scaling numeric features.
- Provides a simple Python function load_and_preprocess() that returns X_scaled, y, feature_names.
- Writes short markdown explanations of the preprocessing choices.

Person B – Model & Training (Phases 5–7)
- Uses Person A’s load_and_preprocess() function.
- Creates train/validation/test splits.
- Builds a simple feedforward neural network with Keras.
- Trains the model with early stopping.
- Optionally tries 1–2 simple hyperparameter variations (batch size or hidden layer size).
- Saves the final trained model in models/heart_failure_model.h5.
- Summarizes the final architecture and hyperparameters.

Person C – Evaluation & Report (Phases 8–10)
- Loads the final model and evaluates it on the test set.
- Produces metrics: accuracy, classification report, confusion matrix.
- Plots training/validation curves using the history object.
- Interprets the results in simple language.
- Owns the project report: integrates notes from A and B into a clean document.
- Exports the final report as a PDF into report/.

Integration strategy:
- All 3 work inside the same VS Code workspace (same folder and virtual environment).

- There is ONE main notebook (notebooks/heart_disease_project.ipynb) where everything is finally assembled:
  Person A fills sections for setup/EDA/preprocessing,
  Person B fills sections for split/model/training,
  Person C fills sections for evaluation/results/conclusion.

==

PHASE 1 – Local environment, VS Code and project structure

==

Goal of this phase:
Prepare a clean, simple workspace on your computer (inside VS Code) so that everything is organized and easy to run.

Main responsible: Person A.

Step 1.1 – Create the project folder
- Create a folder on your machine, for example: heart_disease_dl_project
- Inside it, create these subfolders:
  - data/ → will contain the Kaggle CSV file
  - notebooks/ → will contain your notebook(s)
  - models/ → will contain the saved model file(s)
  - report/ → will contain your written report

Step 1.2 – Open the project in VS Code
- Launch VS Code.
- Click “File” → “Open Folder…” and select heart_disease_dl_project.
- This will be your VS Code workspace for the project.

Step 1.3 – Create a Python virtual environment (Windows, Python 3.9)
- In VS Code, open the integrated terminal (View → Terminal).
- Make sure the current directory is the project root (heart_disease_dl_project).
- Run:

```
python -m venv .venv
```

- Activate the virtual environment in the terminal:

```
.venv\Scripts\activate
```

- You should see (.venv) at the start of the terminal line.

Step 1.4 – Select the virtual environment in VS Code
- In VS Code, press Ctrl+Shift+P and type “Python: Select Interpreter”.
- Choose the interpreter from .venv (it will look like .venv\Scripts\python.exe).
- This ensures all code in VS Code uses this environment.

Step 1.5 – Install required Python libraries (easiest choices)
With the virtual environment activated, in the VS Code terminal, run:

```
pip install --upgrade pip
pip install tensorflow pandas numpy scikit-learn matplotlib jupyter
```

Minimal explanation:
- tensorflow → building and training the deep learning model
- pandas → loading and manipulating the CSV data
- numpy → numerical operations
- scikit-learn → train/test split, metrics, scaling
- matplotlib → simple plots for EDA
- jupyter → allows notebooks to run inside VS Code

Step 1.6 – Install VS Code extensions (if not already installed)
- Open the Extensions panel (Ctrl+Shift+X).

- Install:
  - Python (Microsoft)
  - Jupyter (Microsoft)

These allow you to run notebooks directly inside VS Code.

Step 1.7 – Create the main notebook
- In the Explorer, right-click on notebooks/ → “New File”.
- Name it: heart_disease_project.ipynb
- Open it in VS Code.
- Create top-level markdown sections:
  1. Setup
  2. Data loading & EDA
  3. Preprocessing
  4. Train/validation/test split
  5. Model design
  6. Training and tuning
  7. Evaluation
  8. Model saving
  9. Conclusion and report notes

Later, Persons A, B, C will fill their parts in these sections.

==
PHASE 2 – Dataset acquisition and basic inspection
==

Goal of this phase:
Download the medical dataset, put it in the right folder, and quickly check its structure.

Main responsible: Person A.

Step 2.1 – Download the dataset from Kaggle
- Go to Kaggle and search for: "Heart Failure Prediction Dataset" (author: fedesoriano).
- Download the CSV file (usually named heart.csv).
- Move or copy heart.csv into the data/ folder of your project.

Step 2.2 – Load the dataset in the notebook
In the “Setup” and “Data loading & EDA” sections of heart_disease_project.ipynb:

- Import the main libraries:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report, confusion_matrix

import tensorflow as tf
from tensorflow import keras
```

- Load the CSV file:

```
df = pd.read_csv("../data/heart.csv")
df.head()
```

Step 2.3 – Basic inspection

- Check the shape:
```
df.shape
```
- Check information on columns and types:
```
df.info()
```
- Quick statistics:
```
df.describe()
```

Write down for your report:
- Number of rows (patients).
- Number of columns (features + target).
- Name of the target column (for example: HeartDisease).

==
PHASE 3 – Exploratory Data Analysis (EDA)
==

Goal of this phase:
Understand the data quickly: distributions, types, missing values, and the balance of the target class.

Main responsible: Person A.

Step 3.1 – Check for missing values
- Use:
```
df.isna().sum()
```

- Decide simple rules:
  - If there are few missing values in numeric columns → fill with median.
  - If there are missing values in categorical columns → fill with the most frequent category.

Step 3.2 – Identify target and features
- Assume the target column is called HeartDisease (0 = no, 1 = yes).
- Confirm this with:
```
df["HeartDisease"].value_counts()
```

- Note in your report the percentage of patients with and without heart disease.

Step 3.3 – Quick visualizations (very simple)
- Plot the distribution of the target:

```
df["HeartDisease"].value_counts().plot(kind="bar")
plt.title("Distribution of HeartDisease")
plt.xlabel("Class")
plt.ylabel("Count")
```

- Optionally, plot a few histograms for key numeric features (Age, RestingBP, Cholesterol, MaxHR, Oldpeak).

You do not need many plots, just enough to show you looked at the data.

In the notebook, add markdown cells under the plots with 2–3 bullet points describing what you observe.

==
PHASE 4 – Data preprocessing pipeline
==

Goal of this phase:
Transform the raw DataFrame into a clean numeric matrix X and a target vector y,

ready to be used by the neural network.

Main responsible: Person A.

Step 4.1 – Handle missing values
- For each numeric column with missing values:
  - Replace NaN with the median of that column.
- For each categorical column with missing values:
  - Replace NaN with the most frequent value (mode).

Example pattern (to adapt to actual columns):

```
df["Age"].fillna(df["Age"].median(), inplace=True)
```

Step 4.2 – Separate features and target
- Define the target column name, e.g.:
  target = "HeartDisease"
- Create y:
  y = df[target].values
- Drop the target from the feature table:
  X = df.drop(columns=[target])

Step 4.3 – Identify categorical and numeric columns
Typical example (to adapt after you print df.columns):
- Numeric: Age, RestingBP, Cholesterol, FastingBS, MaxHR, Oldpeak
- Categorical: Sex, ChestPainType, RestingECG, ExerciseAngina, ST_Slope

Write this classification in your report (markdown cell).

Step 4.4 – Encode categorical variables
- Use one-hot encoding with pandas get_dummies to keep things simple:

```
X = pd.get_dummies(
    X,
    columns=["Sex", "ChestPainType", "RestingECG", "ExerciseAngina", "ST_Slope"],
    drop_first=True
)
```

This turns categories into binary columns (0/1).

Step 4.5 – Scale numeric features
- Create a StandardScaler and fit it on all features:

```
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

- X_scaled is now a numeric NumPy array.

You can save the names of the final columns in case you want to reuse the model later:

```
feature_names = X.columns.tolist()
```

Step 4.6 – Expose a helper function for others
Create a cell with a function that Person B and C can call:

```
def load_and_preprocess(path="../data/heart.csv"):
    df = pd.read_csv(path)
    # repeat the same steps: cleaning, encoding, scaling
    # ...
```

```
        return X_scaled, y, feature_names
```

This is the main interface between Person A and the rest of the team.

==
PHASE 5 – Train / validation / test split
==

Goal of this phase:
Create three sets to train, tune and evaluate the model in a clean way.

Main responsible: Person B.

Step 5.1 – Call the preprocessing from Person A
In the notebook, after Person A's section:

```
    X_scaled, y, feature_names = load_and_preprocess()
```

Step 5.2 – First split: train vs temp (validation + test)
- Use stratified split to keep the same class balance:

```
    X_train, X_temp, y_train, y_temp = train_test_split(
        X_scaled, y,
        test_size=0.3,
        stratify=y,
        random_state=42
    )
```

Step 5.3 – Second split: validation vs test
- Split X_temp into validation and test:

```
    X_val, X_test, y_val, y_test = train_test_split(
        X_temp, y_temp,
        test_size=0.5,
        stratify=y_temp,
        random_state=42
    )
```

Now you have approximately:
- 70 percent training data
- 15 percent validation data
- 15 percent test data

Step 5.4 – Optional: convert to tf.data.Datasets (simple)
To keep the Keras training easy and efficient:

```
    batch_size = 32

    train_ds = tf.data.Dataset.from_tensor_slices((X_train, y_train)).batch(batch_size)
    val_ds   = tf.data.Dataset.from_tensor_slices((X_val, y_val)).batch(batch_size)
    test_ds  = tf.data.Dataset.from_tensor_slices((X_test, y_test)).batch(batch_size)
```

You can also train directly on NumPy arrays, but using Dataset is clean and easy.

Add a small markdown block explaining why you split the data this way.

==
PHASE 6 – Model design (simple feedforward network)
==

Goal of this phase:
Define a very simple but correct neural network for binary classification.

Main responsible: Person B.

Step 6.1 – Choose model architecture
We choose a small Sequential model with:
- Input layer: number of features (X_train.shape[1])
- Hidden layer 1: 32 neurons, activation relu
- Dropout: 0.2 (to reduce overfitting)
- Hidden layer 2: 16 neurons, activation relu
- Output layer: 1 neuron, activation sigmoid (for probability between 0 and 1)

Step 6.2 – Implement the model in Keras

```
input_dim = X_train.shape[1]

model = keras.Sequential([
    keras.layers.Input(shape=(input_dim,)),
    keras.layers.Dense(32, activation="relu"),
    keras.layers.Dropout(0.2),
    keras.layers.Dense(16, activation="relu"),
    keras.layers.Dense(1, activation="sigmoid")
])
```

Step 6.3 – Compile the model
Use the simplest, standard choices:

- Loss: binary_crossentropy
- Optimizer: Adam with learning rate 0.001
- Metrics: accuracy

```
model.compile(
    loss="binary_crossentropy",
    optimizer=keras.optimizers.Adam(learning_rate=1e-3),
    metrics=["accuracy"]
)
```

Add a markdown cell where you briefly explain these choices (why binary_crossentropy, why Adam, why accuracy).

==
PHASE 7 – Training, monitoring and basic tuning
==

Goal of this phase:
Train the model, avoid overfitting using early stopping, and perform very light hyperparameter tuning (just a couple of tries).

Main responsible: Person B.

Step 7.1 – Set up early stopping
- Stop training when validation loss stops improving:

```
early_stop = keras.callbacks.EarlyStopping(
    monitor="val_loss",
    patience=5,
    restore_best_weights=True
    )
```

Step 7.2 – Train the model

If you use tf.data.Datasets:

```
    history = model.fit(
        train_ds,
        validation_data=val_ds,
        epochs=50,
        callbacks=[early_stop]
    )
```

If you use NumPy arrays directly, replace train_ds and val_ds by (X_train, y_train) and (X_val, y_val).

Step 7.3 – Plot training curves
- Plot train vs validation loss:

```
    plt.plot(history.history["loss"], label="train_loss")
    plt.plot(history.history["val_loss"], label="val_loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend()
    plt.title("Training and validation loss")
```

You can also plot accuracies if you want.

Step 7.4 – Simple hyperparameter tuning (optional)
To satisfy the idea of "tuning" without spending too much time, you can try:
- Run one training with batch_size = 16
- Run another with batch_size = 64
- Optionally change the number of neurons (for example 16 and 8)

Compare validation accuracy and keep the best configuration.
Note the final choice in your report (markdown cell).

==
PHASE 8 – Final evaluation on the test set
==

Goal of this phase:
Evaluate the final model on the untouched test set and interpret the results.

Main responsible: Person C.

Step 8.1 – Evaluate accuracy on test set
Using the final model and test_ds (or X_test, y_test):

```
    test_loss, test_acc = model.evaluate(test_ds)
    print("Test accuracy:", test_acc)
```

Step 8.2 – Compute predictions and classification report

```
    y_pred_proba = model.predict(X_test)
    y_pred = (y_pred_proba > 0.5).astype(int)

    print(classification_report(y_test, y_pred))
    print(confusion_matrix(y_test, y_pred))
```

Step 8.3 – Interpret the results
In your report/notebook, explain in simple words:
- The final test accuracy.
- What the confusion matrix means (how many true positives, false negatives, etc.).
- Why recall and precision are important in a medical context
  (false negatives are risky because they mean sick patients are not detected).

==
PHASE 9 – Save the model and prepare for reuse
==

Goal of this phase:
Save the trained model to a file so it can be reused by another application later
(web app, API, etc.).

Main responsible: Person B (but Person C should mention it in the report).

Step 9.1 – Save the model to the models/ folder

```
    model.save("../models/heart_failure_model.h5")
```

Step 9.2 – Test reloading (to show it works)

```
    loaded_model = keras.models.load_model("../models/heart_failure_model.h5")
    loaded_model.evaluate(test_ds)
```

Mention in the report that the model is saved and can be loaded in any Python
script that has the same libraries installed.

==
PHASE 10 – Write the project report
==

Goal of this phase:
Produce a short, clear report (2–5 pages) that explains the entire workflow in
simple words.

Main responsible: Person C (but everyone contributes content).

Suggested structure:

1. Introduction
   - Context: heart disease is a major health issue.
   - Objective: build a simple deep learning model to predict heart disease risk
     from the Kaggle dataset.

2. Problem framing
   - Type of problem: binary classification.
   - Target variable: HeartDisease (0 or 1).
   - Chosen metrics: accuracy, plus precision, recall, F1-score.

3. Dataset description
   - Source: Kaggle Heart Failure Prediction Dataset.
   - Number of rows and columns.
   - Short description of the main features (Age, RestingBP, Cholesterol, etc.).
   - Class balance (percentage of patients with heart disease).

4. Preprocessing

   - Handling of missing values.
   - Encoding of categorical features (one-hot encoding).
   - Scaling of numeric features.
   - Train/validation/test split and proportions.

5. Model and training
   - Architecture of the neural network (layers, activations).
   - Loss, optimizer, learning rate.
   - Early stopping strategy.
   - Brief mention of simple hyperparameter tuning (batch size or number of neurons).

6. Results
   - Test accuracy.
   - Classification report (precision, recall, F1-score).
   - Confusion matrix (and a short interpretation).
   - One or two training curves (loss vs epochs).

7. Model saving and possible future work
   - Mention that the model is saved as models/heart_failure_model.h5.
   - Ideas for improvement if you had more time (more tuning, more features, cross-validation).

8. Conclusion
   - Summary of what you implemented.
   - What you learned from this project.

Once the report is written in Word, Google Docs or LaTeX, export it as PDF and put it
inside the report/ folder.

==
END OF PLAN
==

This plan is designed to be as simple as possible while still covering all the phases
your professor expects: environment, data loading, preprocessing, modeling, training,
evaluation, saving, reporting, and now also a clear 3-person team split that allows
everyone to work in parallel inside VS Code.