**Lithium-ion battery – A regression problem:**

Analysis of the Problem:

a. Understand the Problem: Predict properties of new materials based on their composition and crystal structure.

b. Problem Type: Multivariate regression problem, as you're trying to predict multiple continuous properties (Formation Energy, E Above Hull, Band Gap) given the input features.

c. Measure of Loss/Success: You can use Mean Absolute Error (MAE) or Mean Squared Error (MSE) for each property. The goal would be to minimize these errors.

d. Evaluation Protocol: Use K-fold cross-validation to optimize your model during training.

Data Exploration and Preparation:

a. Check for missing values, NaN values or features, and handle them if necessary.

b. Use statistical tools to understand the data distribution and correlations between features.

c. Encode categorical features like Spacegroup and Crystal System using one-hot encoding or other suitable encoding techniques.

d. Scale continuous features to a common range, such as [0, 1] or [-1, 1].

Testing Phase I: Baseline Models

a. Use linear regression, decision trees, and other non-neural network models to establish a baseline performance.

b. Experiment with dimensionality reduction techniques (e.g., PCA) and clustering algorithms to observe their impact on the predictive accuracy.

Testing Phase II: Model Development

a. Design a base neural network architecture (NN/DNN) with appropriate activation functions, loss functions, and optimizer.

b. Train the model using the prepared data and K-fold cross-validation.

Testing Phase III: Model Regularization and Hyperparameter optimization

a. Use regularization techniques like L1, L2, or dropout to reduce overfitting.

b. Perform hyperparameter tuning (e.g., learning rate, batch size, number of layers, number of neurons per layer) using grid search, random search, or Bayesian optimization.

Evaluation of the model predictions

a. Compare the performance of the baseline models and the neural network models in terms of MAE or MSE.

b. Analyze and discuss the performance improvements, if any, achieved by the neural network models.

Lessons Learnt and Conclusions

a. Summarize the findings and insights gained during the project.

b. Discuss possible future improvements or extensions to the project.

**Lithium-ion batteries: A Classification problem.**

In this section, it will be provided an in-depth explanation of each step in the "Analysis of the Problem" stage, discuss the importance of predicting the crystal system, and explore its applications in the industry.

# **Analysis of the Problem:**

1. **Understand the Problem**:

The primary objective of this project is to predict the crystal system of materials, specifically lithium-ion (Li-ion) silicate cathodes, based on their composition, structure, and other properties. Accurate prediction of the crystal system can provide insights into the performance and stability of Li-ion batteries, which are essential components in various industries, such as consumer electronics, electric vehicles, and renewable energy storage systems.

* 1. **Problem Type:**

The task involves classifying materials into one of the three crystal systems: monoclinic, orthorhombic, or triclinic. As such, it is a multiclass classification problem, a common type of machine learning problem in which an algorithm must assign an input to one of multiple categories.

* 1. **Measure of Loss/Success:**

In order to evaluate the performance of the classification models, it will be used standard metrics like accuracy, precision, recall, confusion matrix, ROC curve and F1-score for each class.

* + 1. **Confusion Matrix**:

A confusion matrix is a table that is often used to describe the performance of a classification model on a set of data for which the true values are known. It summarizes the correct and incorrect predictions made by the model, showing the number of true positive, true negative, false positive, and false negative predictions for each class.

* Convenient way to describe the performance
* Basis for different measures
* Good for balanced classes

Diagram

Description automatically generated with low confidence

* + 1. **ROC Curve**:

The Receiver Operating Characteristic (ROC) curve is a graphical representation of the performance of a binary classifier system as its discrimination threshold is varied. It plots the true positive rate (sensitivity) against the false positive rate (1 - specificity) for different threshold values. The area under the ROC curve (AUC-ROC) is a popular performance measure for binary classification problems.

Chart, line chart

Description automatically generated

* Receiver Operating Characteristics
* (i) Confusion Matrix 🡪 based on a score threshold of 50 %
* (ii) 0 % - 100 % Threshold 🡪 TP & TN values would change
* (iii) ROC 🡪 Plot for every threshold value
* Closer to the left, better it is
  + 1. **Precision and Recall:**

Precision is the ratio of true positive predictions to the sum of true positive and false positive predictions. It measures the model's ability to correctly identify positive instances.

Recall (also known as sensitivity or true positive rate) is the ratio of true positive predictions to the sum of true positive and false negative predictions. It measures the proportion of actual positive instances that were identified.

Precision is the ratio of True Positives to the sum of True Positives and False Positives.

Recall (also known as Sensitivity or True Positive Rate) is the ratio of True Positives to the sum of True Positives and False Negatives.

* Precision: How often, when a model makes a positive prediction, the prediction is correct
* Recall: How many of the True Cases are correctly predicted by the model.

Shape

Description automatically generatedA picture containing diagram

Description automatically generated

* + 1. **Accuracy**

Accuracy is the ratio of correct predictions to the total number of predictions made. It is the most commonly used metric for classification problems and provides an overall assessment of how well the model is performing.

Accuracy can be calculated using the following formula:

Where:

True Positives (TP) are the instances correctly classified as positive.

True Negatives (TN) are the instances correctly classified as negative.

False Positives (FP) are the instances incorrectly classified as positive.

False Negatives (FN) are the instances incorrectly classified as negative.

* + 1. **F1-score:**

The F1-score is the harmonic mean of precision and recall. It provides a single metric for model comparison when both precision and recall are important. The F1-score ranges from 0 to 1, where a score of 1 indicates perfect precision and recall, and 0 indicates the worst performance.

F1-score can be calculated using the following formula:

* + 1. **Evaluation Protocol:**

K-fold cross-validation as used in the lecture, will be utilized to optimize the model during training. In this technique, the dataset is divided into K equal-sized subsets. The model is trained on K-1 subsets and validated on the remaining subset. This process is repeated K times, with each subset being used as the validation set once. The average performance metric obtained from all the folds will be used as the final evaluation metric.

1. **Importance of Predicting the Crystal System in the Context of E-Mobility, ESS, and EV:**

Predicting the crystal system of Li-ion silicate cathodes is crucial for several reasons, especially in the context of e-mobility, energy storage systems (ESS), and electric vehicles (EV)

a. Understanding the relationships between the crystal structure and electrochemical properties of the materials can lead to the development of improved battery performance and longevity, which is essential for the e-mobility sector and EVs to meet consumer expectations.

b. Knowledge of the crystal system can facilitate the design of new materials with targeted properties, leading to more efficient and sustainable energy storage solutions. This is particularly important for ESS, which play a critical role in integrating renewable energy sources into the grid.

c. Accurate predictions of the crystal system can help identify promising materials for further experimental investigation, thus reducing the time and cost of materials discovery. This accelerates the development and deployment of advanced batteries for e-mobility and ESS applications.

1. **Applications in the Industry:**

The prediction of the crystal system in Li-ion silicate cathodes has numerous industrial applications, specifically in e-mobility, ESS, and EVs:

a. In the consumer electronics industry, improved batteries can lead to longer-lasting and more efficient devices, such as smartphones, laptops, and wearables, which are essential components of the e-mobility ecosystem.

b. The electric vehicle (EV) industry can benefit from better batteries, as they can offer increased driving range, faster charging, and longer lifespan. This contributes to the widespread adoption of EVs and advances the transition towards sustainable transportation.

c. Renewable energy storage systems, such as solar and wind farms, can utilize more efficient batteries to store and distribute energy more effectively. This is vital for grid stability and the integration of renewable energy sources, ultimately contributing to a greener and more sustainable future.

1. **Dataset Provenance**

This dataset was collected from the Materials Project [1] for research purposes. The calculations were performed using VASP (Vienna Ab initio Simulation Package) software, a widely used software package for atomistic simulations [2]. The crystal structure data was obtained through the Inorganic Crystal Structure Database (ICSD) [3], a comprehensive collection of crystal structure data for inorganic compounds.

[1] Jain et al., "The Materials Project: A Materials Genome Approach to Accelerating Materials Innovation", APL Materials, vol. 1, no. 1, 2013.

[2] Kresse and J. Furthmüller, "Efficient Iterative Schemes for Ab Initio Total-Energy Calculations Using a Plane-Wave Basis Set", Physical Review B, vol. 54, no. 16, 1996.

[3] Bergerhoff, R. Hundt, R. Sievers, and I. Brown, "The Inorganic Crystal Structure Database", Journal of Chemical Information and Computer Sciences, vol. 23, no. 2, pp. 66-69, 1983.

1. **Features Definition**

**Materials Id:** A unique identifier assigned to each material in the dataset. This ID helps in indexing, searching, and cross-referencing materials within the database.

**Formula:** The chemical formula of the material, represented by the symbols of the constituent elements along with their respective stoichiometric ratios. The formula provides information on the elemental composition of the material.

**Spacegroup:** The space group represents the symmetry of the crystal lattice, which is the arrangement of atoms in a crystalline material. It describes the periodic arrangement of atoms in three-dimensional space **and is essential for understanding the structural and physical properties of the material.**

**Formation Energy (eV):** The formation energy is the energy change associated with the formation of a compound from its constituent elements in their most stable reference states. It is measured in electron volts (eV) and is a key indicator of the thermodynamic stability of the material. A lower formation energy implies a more stable compound.

**E Above Hull (eV):** The energy above the convex hull is the difference between the formation energy of a material and the lowest formation energy of all materials with the same composition in the database. It indicates the stability of a material with respect to other materials of the same composition. A value of zero means the material is on the convex hull and is thermodynamically stable, while positive values indicate metastable materials.

**Band Gap (eV):** The band gap is the energy difference between the highest occupied molecular orbital (HOMO) and the lowest unoccupied molecular orbital (LUMO) in a material. It determines the material's electrical conductivity, with larger band gaps corresponding to insulators and smaller band gaps corresponding to conductors or semiconductors.

**Nsites:** The number of lattice sites in the unit cell of the crystal structure. It represents the total number of atomic positions in the repeating unit of the crystal lattice.

**Density (gm/cc):** The mass density of the material, measured in grams per cubic centimeter (gm/cc). It is the ratio of the mass of the material to its volume. Density is an important property for various applications, as it influences the mechanical, thermal, and electrical properties of materials.

**Volume**: The volume of the unit cell of the crystal structure, which is the smallest repeating unit of the lattice. It provides information about the packing efficiency of atoms in the material and can influence properties such as thermal expansion and compressibility.

**Has Bandstructure:** A binary attribute indicating whether a calculated electronic band structure is available for the material. The band structure provides detailed information about the electronic properties of the material, such as the distribution of energy levels and the nature of electronic transitions.

**Crystal System:** The crystal system is a classification of crystal structures based on the lattice type and symmetry. In this case, the materials are classified into three major crystal systems: monoclinic, orthorhombic, and triclinic. The crystal system is important for understanding the arrangement of atoms in a material and can influence its physical and chemical properties.

# **Data Exploration and Preparation:**

a. Check for missing values, NaN values or features, and handle them if necessary.

b. Use statistical tools to understand the data distribution and correlations between features.

c. Encode categorical features like Spacegroup using one-hot encoding or other suitable encoding techniques.

d. Scale continuous features to a common range, such as [0, 1] or [-1, 1].

# **Testing Phase I: Baseline Models**

a. Use logistic regression, decision trees, random forests, and other non-neural network models to establish a baseline performance.(Random Forest, XGBoost, LightGBM)

b. Experiment with dimensionality reduction techniques (e.g., PCA) and clustering algorithms to observe their impact on the predictive accuracy.

Logistic Regression: Logistic regression is a probabilistic discriminative model that can be extended to handle multi-class problems using the One-vs-Rest (OvR) or Softmax strategies. In OvR, a separate logistic regression model is trained for each class, treating it as the positive class and the rest as negative. In Softmax, logistic regression is generalized to handle multiple classes directly.

Gradient Descent: Gradient descent is not a classification model but an optimization algorithm often used to train models like logistic regression, neural networks, etc. It is used to minimize the loss function by iteratively updating the model's parameters.

Decision Trees: Decision trees are a type of model that recursively splits the input space based on feature values, resulting in a tree-like structure. They can handle multi-class problems directly.

Target Variable Preparation: Convert the target variable into numerical labels using LabelEncoder.

Random Forest: Random forests are ensembles of decision trees that improve the performance of individual trees by averaging their predictions. They can handle multi-class problems directly.

Target Variable Preparation: Convert the target variable into numerical labels using LabelEncoder.

XGBoost: XGBoost is a gradient boosting algorithm that trains an ensemble of weak learners (typically decision trees) sequentially, with each new learner trying to correct the errors of the previous one. It can handle multi-class problems using the Softmax objective.

Target Variable Preparation: Convert the target variable into numerical labels using LabelEncoder.

LightGBM: LightGBM is another gradient boosting algorithm similar to XGBoost, but it uses a different tree-building strategy, making it faster and more memory-efficient. It can handle multi-class problems using the Softmax objective.

Target Variable Preparation: Convert the target variable into numerical labels using LabelEncoder.

Neural Networks: Neural networks are a powerful class of models that can approximate complex functions. They consist of layers of interconnected nodes or neurons. For multi-class problems, you can use a Softmax activation function in the output layer.

Target Variable Preparation: Convert the target variable into one-hot encoded vectors.

Support Vector Machines (SVM): SVMs are a class of discriminative models that find the best separating hyperplane between classes. For multi-class problems, you can use the One-vs-One (OvO) or One-vs-Rest (OvR) strategies.

Target Variable Preparation: Convert the target variable into numerical labels using LabelEncoder.

k-Nearest Neighbors (k-NN): k-NN is a simple instance-based learning algorithm that predicts the class label of a new instance based on the majority vote of its k-nearest neighbors in the training set. It can handle multi-class problems directly.

Target Variable Preparation: Convert the target variable into numerical labels using LabelEncoder.

Each method has its advantages and drawbacks. For your problem, I would recommend trying Random Forest, XGBoost, LightGBM, and Neural Networks, as they are generally more robust and perform well on a wide range of problems. Remember to split your data into training and validation sets and evaluate model performance using appropriate metrics like accuracy, F1-score, or confusion matrix.

# **Testing Phase II:** Model Development

Based on the results we have got so far, it seems like the LogisticRegression and SGDClassifier models are performing quite well on the test data. However, to move forward with Phase II and III, the following steps could be developed:

**Phase II: Model Development**

Start with a simple feedforward neural network architecture using Keras. This is a good choice for a small dataset with two dimensions. You can experiment with different architectures by changing the number of layers and units in each layer.

Choose an appropriate loss function and activation functions for the problem at hand. For example, if it's a binary classification problem, use binary\_crossentropy as the loss function and sigmoid activation for the output layer. For multi-class problems, use categorical\_crossentropy and softmax activation.

Initialize the network using appropriate techniques like He initialization or Glorot initialization.

Choose an optimizer like Adam, RMSProp, or SGD with momentum.

Split the data into training and validation sets, and train the model for a reasonable number of epochs (e.g., 50-100) while monitoring the training and validation loss.

# **Testing Phase III: Model Regularization and Hyperparameter optimization**

**Phase III:** Model Regularization and Hyperparameter Optimization

Apply regularization techniques like L1, L2, or dropout to prevent overfitting.

Perform hyperparameter optimization using techniques like grid search, random search, or Bayesian optimization to find the best combination of hyperparameters for your model. Some hyperparameters to consider tuning include learning rate, batch size, number of hidden layers, number of units in each layer, dropout rate, and regularization strength.

Retrain the model using the best hyperparameters found in the previous step.

Evaluate the final model using the test set and compare its performance with the baseline models from Phase I.

If the performance is not satisfactory, you can try other neural network architectures like convolutional neural networks (CNNs) or recurrent neural networks (RNNs) if they are suitable for your problem.

a. Use regularization techniques like L1, L2, or dropout to reduce overfitting.

b. Perform hyperparameter tuning (e.g., learning rate, batch size, number of layers, number of neurons per layer) using grid search, random search, or Bayesian optimization.

Evaluation of the model predictions

a. Compare the performance of the baseline models and the neural network models in terms of accuracy, precision, recall, and F1-score.

b. Analyze and discuss the performance improvements, if any, achieved by the neural network models.

Lessons Learnt and Conclusions

a. Summarize the findings and insights gained during the project.

b. Discuss possible future improvements or extensions to the project.

# Define the parameter grid for hyperparameter tuning

param\_grid\_nn = {

    "batch\_size": [10, 50, 100],

    "epochs": [10, 50, 100],

    "learning\_rate": np.linspace(0.001, 0.1, 5),

}

# Modify the create\_model function to accept learning\_rate as an argument

def create\_model(input\_dim, output\_dim, learning\_rate=0.001):

    model = Sequential()

    model.add(Dense(64, input\_dim=input\_dim, activation='relu', kernel\_initializer='he\_normal'))

    model.add(Dense(32, activation='relu', kernel\_initializer='he\_normal'))

    model.add(Dense(output\_dim, activation='softmax', kernel\_initializer='glorot\_normal'))

    model.compile(loss='sparse\_categorical\_crossentropy', optimizer=Adam(learning\_rate=learning\_rate), metrics=['accuracy'])

    return model

n\_features = X\_train.shape[1]

n\_classes = len(np.unique(y\_train))

# Define a function to create and compile a Keras model with the specified learning rate

def create\_model\_nn(learning\_rate):

    return create\_model(n\_features, n\_classes, learning\_rate)

# Create a KerasClassifier model for hyperparameter tuning

model\_nn = KerasClassifier(build\_fn=create\_model\_nn, verbose=0)

# Create a GridSearchCV object with the Keras model

grid\_search\_nn = GridSearchCV(estimator=model\_nn, param\_grid=param\_grid\_nn, scoring="neg\_log\_loss", cv=10, n\_jobs=2, verbose=2)

# Perform the grid search by training the model with different hyperparameters

grid\_search\_nn.fit(X\_train, y\_train.values.ravel())

# Retrieve the best model

best\_model\_nn = grid\_search\_nn.best\_estimator\_

print("Best parameters found for Neural Network:", grid\_search\_nn.best\_params\_)

print("Best score found for NN:", grid\_search\_nn.best\_score\_)

[C:\Users\rofer\AppData\Local\Temp\ipykernel\_24136\475250556.py:27](file:///C:\Users\rofer\AppData\Local\Temp\ipykernel_24136\475250556.py:27): DeprecationWarning: KerasClassifier is deprecated, use Sci-Keras (<https://github.com/adriangb/scikeras>) instead. See <https://www.adriangb.com/scikeras/stable/migration.html> for help migrating.

model\_nn = KerasClassifier(build\_fn=create\_model\_nn, verbose=0)

Fitting 10 folds for each of 45 candidates, totalling 450 fits

[c:\Users\rofer\anaconda3\envs\Thesis\lib\site-packages\joblib\externals\loky\process\_executor.py:700](file:///C:\Users\rofer\anaconda3\envs\Thesis\lib\site-packages\joblib\externals\loky\process_executor.py:700): UserWarning: A worker stopped while some jobs were given to the executor. This can be caused by a too short worker timeout or by a memory leak.

warnings.warn(

# If you want to use other search algorithms like Hyperband,

# you can replace RandomSearch with Hyperband in the code above.

# Also, don't forget to adjust the search space and the number of trials according to your requirements and computational resources.

# Loading the data

cwd = os.getcwd()

DATA\_PATH = os.path.join(cwd, '..', 'data')

RAW\_DATA\_PATH = os.path.join(DATA\_PATH, 'raw')

hyper\_DATA\_PATH = os.path.join(DATA\_PATH, 'interim')

# Define the model-building function with hyperparameters, X\_train, and y\_train as arguments

def build\_model(hp, X\_train, y\_train):

    # Determine the input dimension based on the input data

    input\_dim = X\_train.shape[1]

    # Determine the output dimension based on the number of unique classes in the target data

    output\_dim = len(np.unique(y\_train))

    # Create a sequential model

    model = keras.Sequential()

    # Add an input layer with the input dimension

    model.add(layers.Input(shape=(input\_dim,)))

    # Add a flatten layer to reshape the input data

    # The Flatten layer is typically used when working with multidimensional data, such as images,

    # to reshape the input into a 2D tensor before passing it through dense layers.

    # model.add(layers.Flatten())

    # Add a variable number of dense layers determined by the hyperparameter 'num\_layers'

    for i in range(hp.Int("num\_layers", 1, 3)):

        model.add(

            layers.Dense(

                # Set the number of units in each dense layer, determined by the hyperparameter 'units\_{i}'

                units=hp.Int(f"units\_{i}", min\_value=32, max\_value=128, step=8),

                # Set the activation function of each dense layer, determined by the hyperparameter 'activation'

                activation=hp.Choice("activation", ["relu", "tanh"]),

            )

        )

    # Add a dropout layer if the hyperparameter 'dropout' is True

    if hp.Boolean("dropout"):

        model.add(layers.Dropout(rate=0.25))

    # Add the output layer with the softmax activation function and the output dimension

    model.add(layers.Dense(output\_dim, activation="softmax"))

    # Define the learning rate using the hyperparameter 'lr'

    learning\_rate = hp.Float("lr", min\_value=1e-4, max\_value=1e-2, sampling="log")

    # Compile the model with the Adam optimizer, sparse\_categorical\_crossentropy loss, and accuracy metric

    model.compile(

        optimizer=keras.optimizers.Adam(learning\_rate=learning\_rate),

        loss="sparse\_categorical\_crossentropy",

        metrics=["accuracy"],

    )

    # Return the compiled model

    return model

# Call the build\_model function with HyperParameters and the training data

build\_model(keras\_tuner.HyperParameters(), X\_train, y\_train)

tuner.search\_space\_summary()

Search space summary

Default search space size: 5

num\_layers (Int)

{'default': None, 'conditions': [], 'min\_value': 1, 'max\_value': 3, 'step': 1, 'sampling': 'linear'}

units\_0 (Int)

{'default': None, 'conditions': [], 'min\_value': 32, 'max\_value': 128, 'step': 8, 'sampling': 'linear'}

activation (Choice)

{'default': 'relu', 'conditions': [], 'values': ['relu', 'tanh'], 'ordered': False}

dropout (Boolean)

{'default': False, 'conditions': []}

lr (Float)

{'default': 0.0001, 'conditions': [], 'min\_value': 0.0001, 'max\_value': 0.01, 'step': None, 'sampling': 'log'}

tuner.search(X\_train, y\_train, epochs=50, validation\_split=0.2, verbose=1)

Trial 5 Complete [00h 00m 23s]

val\_accuracy: 1.0

Best val\_accuracy So Far: 1.0

Total elapsed time: 00h 01m 55s

INFO:tensorflow:Oracle triggered exit

tuner.results\_summary()

Results summary

Results in c:\Users\rofer\WS2022\_Project\_Template\_lithium\_ion\_batteries\lithium\_ion\_battery\_classification\notebooks\..\data\interim\keras\_tuner\_example

Showing 10 best trials

Objective(name="val\_accuracy", direction="max")

Trial 0 summary

Hyperparameters:

num\_layers: 2

units\_0: 128

activation: tanh

dropout: False

lr: 0.0005334545424135214

units\_1: 32

Score: 1.0

Trial 1 summary

Hyperparameters:

num\_layers: 1

units\_0: 120

activation: tanh

dropout: True

lr: 0.0008788815784436897

units\_1: 104

Score: 1.0

Trial 2 summary

Hyperparameters:

num\_layers: 2

units\_0: 88

activation: tanh

dropout: False

lr: 0.0018019381898324188

units\_1: 96

Score: 1.0

Trial 3 summary

Hyperparameters:

num\_layers: 3

units\_0: 32

activation: tanh

dropout: False

lr: 0.0001888154526950113

units\_1: 48

units\_2: 32

Score: 1.0

Trial 4 summary

Hyperparameters:

num\_layers: 1

units\_0: 56

activation: relu

dropout: True

lr: 0.0059195072811734474

units\_1: 80

units\_2: 96

Score: 1.0

Then we have updated the code, we have:

1. Increased the range of **num\_layers** to 1-5 for a deeper model.
2. Added more activation functions (e.g., ELU and Swish) to the choice of **activation**.
3. Added a **BatchNormalization** layer, controlled by the **batch\_norm** hyperparameter, to improve model training.
4. Modified the **dropout** hyperparameter to be unique for each layer and added a variable **dropout\_rate** for each dropout layer.
5. Expanded the range of the **units\_{i}** and **lr** hyperparameters.
6. Introduced a **weight\_decay** hyperparameter to add L2 regularization to the Adam optimizer.

# If you want to use other search algorithms like Hyperband,

# you can replace RandomSearch with Hyperband in the code above.

# Also, don't forget to adjust the search space and the number of trials according to your requirements and computational resources.

# Loading the data

cwd = os.getcwd()

DATA\_PATH = os.path.join(cwd, '..', 'data')

RAW\_DATA\_PATH = os.path.join(DATA\_PATH, 'raw')

hyper\_DATA\_PATH = os.path.join(DATA\_PATH, 'interim')

# Define the model-building function with hyperparameters, X\_train, and y\_train as arguments

def build\_model(hp, X\_train, y\_train):

    # Determine the input dimension based on the input data

    input\_dim = X\_train.shape[1]

    # Determine the output dimension based on the number of unique classes in the target data

    output\_dim = len(np.unique(y\_train))

    # Create a sequential model

    model = keras.Sequential()

    # Add an input layer with the input dimension

    # model.add(layers.Input(shape=(input\_dim,)))

    # Add a flatten layer to reshape the input data

    # The Flatten layer is typically used when working with multidimensional data, such as images,

    # to reshape the input into a 2D tensor before passing it through dense layers.

    # model.add(layers.Flatten())

    # Add a variable number of dense layers determined by the hyperparameter 'num\_layers'

    for i in range(hp.Int("num\_layers", 1, 5)):

        model.add(

            layers.Dense(

                # Set the number of units in each dense layer, determined by the hyperparameter 'units\_{i}'

                units=hp.Int(f"units\_{i}", min\_value=16, max\_value=256, step=8),

                # Set the activation function of each dense layer, determined by the hyperparameter 'activation'

                activation=hp.Choice("activation", ["relu", "tanh", "elu", "swish"]),

            )

        )

        # Add a variable dropout rate for each dense layer determined by the hyperparameter 'dropout\_rate\_{i}'

        if hp.Boolean(f"dropout\_{i}"):

            model.add(layers.Dropout(rate=hp.Float(f"dropout\_rate\_{i}", min\_value=0.1, max\_value=0.5, step=0.05)))

    model.add(layers.Dense(output\_dim, activation="softmax"))

    # Add the output layer with the softmax activation function and the output dimension

    model.add(layers.Dense(output\_dim, activation="softmax"))

    # Define the learning rate using the hyperparameter 'lr'

    learning\_rate = hp.Float("lr", min\_value=1e-5, max\_value=1e-2, sampling="log")  # Extend the learning rate range

    weight\_decay = hp.Float("weight\_decay", min\_value=1e-6, max\_value=1e-2, sampling="log")  # Add weight decay hyperparameter

    # Compile the model with the Adam optimizer, sparse\_categorical\_crossentropy loss, and accuracy metric

    model.compile(

        optimizer=keras.optimizers.Adam(learning\_rate=learning\_rate, decay=weight\_decay),

        loss="sparse\_categorical\_crossentropy",

        metrics=["accuracy"],

    )

    return model

# Call the build\_model function with HyperParameters and the training data

build\_model(keras\_tuner.HyperParameters(), X\_train, y\_train)