Problem Statement

Continuing with the same scenario, now that you have been able to successfully predict each student GPA, now you will classify each Student based on they probability to have a successful GPA score.

The different classes are:

- Low : Students where final GPA is predicted to be between: 0 and 2
- Medium : Students where final GPA is predicted to be between: 2 and 3.5
- High : Students where final GPA is predicted to be between: 3.5 and 5

1) Import Libraries

First let's import the following libraries, if there is any library that you need and is not in the list bellow feel free to include it

In [35]:
```
# Librerías estándar
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Librerías de sklearn
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder

# Librerías de TensorFlow y Keras
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Conv1D, MaxPooling1D, Fl
from tensorflow.keras.regularizers import l2
from tensorflow.keras.optimizers import RMSprop, Adamax
```

2) Load Data

- You will use the same file from the previous activity (Student Performance Data)

In [36]:
```
data = pd.read_csv("M2_A3_StudentPerformanceData.csv")
```

3) Add a new column called 'Profile' this column will have the following information

Based on the value of GPA for each student:

- If GPA values between 0 and 2 will be labeled 'Low',
- Values between 2 and 3.5 will be 'Medium',

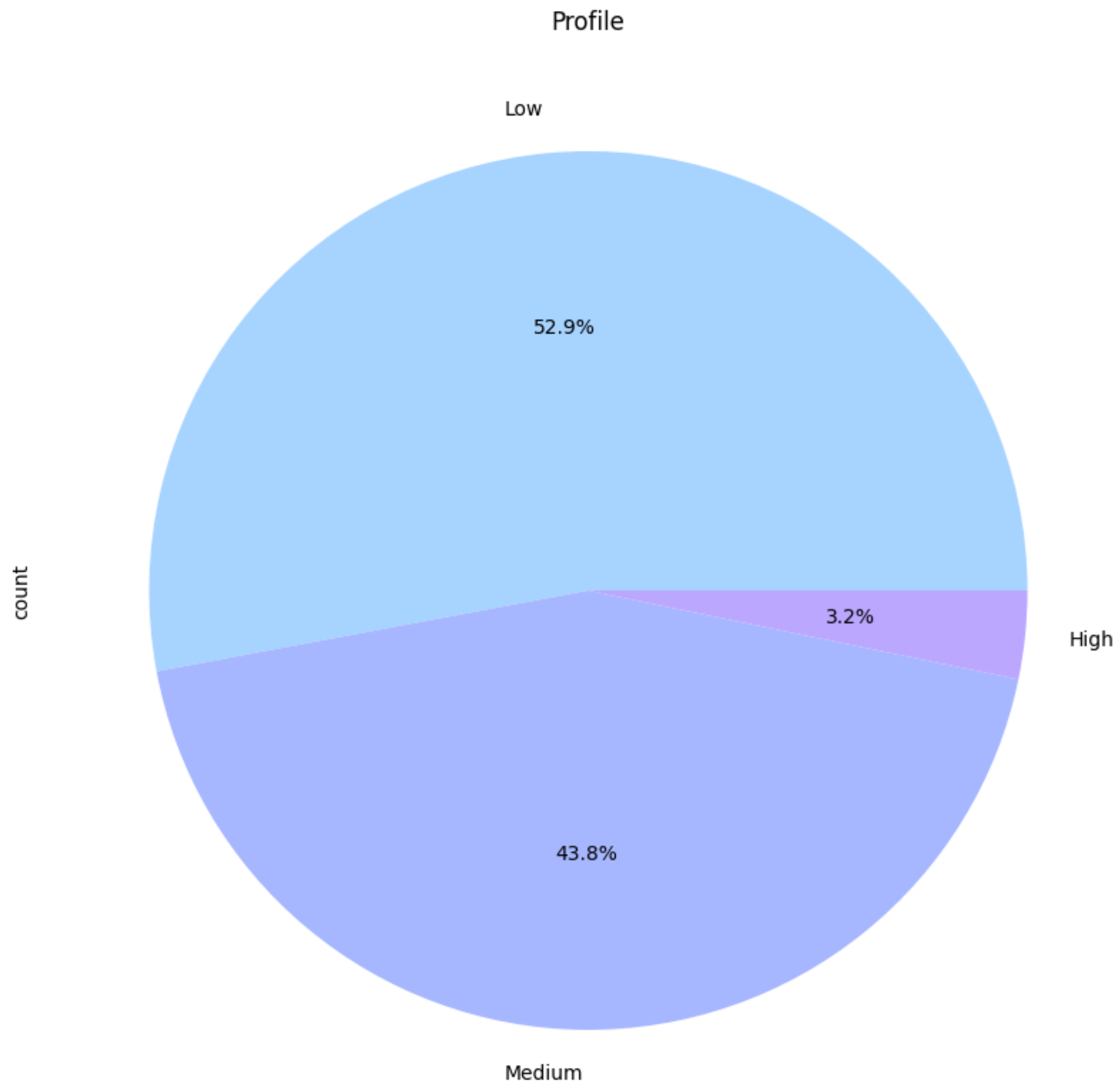
- And values between 3.5 and 5 will be 'High'.

In [37]:
```
data['Profile'] = pd.cut(data['GPA'], bins=[0, 2, 3.5, 5], labels=['Low', 'M
```

4) Use Matplotlib to show a Pie chart to show the percentage of students in each profile.

- Title: Students distribution of Profiles
- Graph Type: pie

In [38]:
```
plt.figure(figsize=(10, 10))
colors = ['#a7d4ff', '#a7b7ff', '#bca7ff']
data['Profile'].value_counts().plot.pie(autopct='%1.1f%%', colors=colors)
plt.title('Profile')
plt.show()
```



5) Convert the Profile column into a Categorical Int

You have already created a column with three different values: 'Low', 'Medium', 'High'. These are Categorical values. But, it is important to notice that Neural Networks works better with numbers, since we apply mathematical operations to them.

Next you need to convert Profile values from Low, Medium and High, to 0, 1 and 2. IMPORTANT, the order does not matter, but make sure you always assign the same number to Low, same number to Medium and same number to High.

Make sure to use the fit_transform method from LabelEncoder.

In [39]:
```
data['Profile'] = data['Profile'].replace({'Low': 0, 'Medium': 1, 'High': 2})
data = data.drop(['GPA'], axis=1)
data.dropna(inplace=True)
data['Profile'] = data['Profile'].astype(int)
```

```
<ipython-input-39-fc9e1053ecc5>:1: FutureWarning: Downcasting behavior in `r
eplace` is deprecated and will be removed in a future version. To retain the
old behavior, explicitly call `result.infer_objects(copy=False)`. To opt-in
to the future behavior, set `pd.set_option('future.no_silent_downcasting', T
rue)`
  data['Profile'] = data['Profile'].replace({'Low': 0, 'Medium': 1, 'High':
2})
<ipython-input-39-fc9e1053ecc5>:1: FutureWarning: The behavior of Series.rep
lace (and DataFrame.replace) with CategoricalDtype is deprecated. In a futur
e version, replace will only be used for cases that preserve the categories.
To change the categories, use ser.cat.rename_categories instead.
  data['Profile'] = data['Profile'].replace({'Low': 0, 'Medium': 1, 'High':
2})
```

In [40]:
```
data
```

Out[40]:

	StudentID	Age	Gender	Ethnicity	ParentalEducation	StudyTimeWeekly	Abser
0	1001	17	1	0	2	19.833723	
1	1002	18	0	0	1	15.408756	
2	1003	15	0	2	3	4.210570	
3	1004	17	1	0	3	10.028829	
4	1005	17	1	0	2	4.672495	
...	...	...	...	...	...	...	
2387	3388	18	1	0	3	10.680555	
2388	3389	17	0	0	1	7.583217	
2389	3390	16	1	0	2	6.805500	
2390	3391	16	1	1	0	12.416653	
2391	3392	16	1	0	2	17.819907	

2376 rows × 15 columns

6) Select the columns for your model.

Same as the last excersice we need a dataset for features and a dataset for label.

- Create the following dataset:
  - A dataset with the columns for the model.
  - From that data set generate the 'X' dataset. This dataset will have all the features (make sure Profile is NOT in this dataset)
  - Generate a second 'y' dataset, This dataset will only have our label column, which is 'Profile'.
  - Generate the Train and Test datasets for each X and y:
    - X_train with 80% of the data
    - X_test with 20% of the data
    - y_train with 80% of the data
    - y_test with 20% of the data

In [41]:

```
X = data.drop(['Profile'], axis=1)
y = data['Profile']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, ran
```

7) All Feature datasets in the same scale.

Use StandardScaler to make sure all features in the X_train and X_test datasets are on the same scale.

Standardization transforms your data so that it has a mean of 0 and a standard deviation of 1. This is important because many machine learning algorithms perform better when the input features are on a similar scale.

Reason for Using StandardScaler:

- Consistent Scale: Features with different scales (e.g., age in years, income in dollars) can bias the model. StandardScaler ensures all features contribute equally.
- Improved Convergence: Algorithms like gradient descent converge faster with standardized data.
- Regularization: Helps in achieving better performance in regularization methods like Ridge and Lasso regression.

In [42]:
```
scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

8. Define your Deep Neural Network.

- This will be a Sequential Neural Network.
- With a Dense input layer with 64 units, and input dimention based on the X_train size and Relu as the activation function.
- A Dense hidden layer with 32 units, and Relu as the activation function.
- And a Dense output layer with the number of different values in the y dataset, activation function = to sofmax

This last part of the output layer is super important, since we want to do a classification and not a regression, we will use activation functions that fits better a classification scenario.

In [43]:
```
model = Sequential()
model.add(Dense(64, input_dim=X_train.shape[1], activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(y_train.nunique(), activation='softmax'))
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: U
serWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. Wh
en using Sequential models, prefer using an `Input(shape)` object as the fir
st layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

9. Compile your Neural Network

- Choose Adam as the optimizer
- And sparse_categorical_crossentropy as the Loss function
- Also add the following metrics: accuracy

In [44]:
```
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metr
```

10. Fit (or train) your model

- Use the X_train and y_train datasets for the training
- Do 50 data iterations
- Choose the batch size = 10
- Also select a validation_split of 0.2
- Save the result of the fit function in a variable called 'history'

In [45]:
```
history = model.fit(X_train, y_train, epochs=50, batch_size=10, validation_s
```

```
Epoch 1/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.6627 - loss: 0.7972 -
val_accuracy: 0.8816 - val_loss: 0.3778
Epoch 2/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9189 - loss: 0.2861 -
val_accuracy: 0.9237 - val_loss: 0.2406
Epoch 3/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9476 - loss: 0.1636 -
val_accuracy: 0.9289 - val_loss: 0.1918
Epoch 4/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9592 - loss: 0.1247 -
val_accuracy: 0.9474 - val_loss: 0.1635
Epoch 5/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9772 - loss: 0.0858 -
val_accuracy: 0.9526 - val_loss: 0.1431
Epoch 6/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9839 - loss: 0.0735 -
val_accuracy: 0.9474 - val_loss: 0.1399
Epoch 7/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9809 - loss: 0.0654 -
val_accuracy: 0.9553 - val_loss: 0.1301
Epoch 8/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9857 - loss: 0.0648 -
val_accuracy: 0.9579 - val_loss: 0.1232
Epoch 9/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9866 - loss: 0.0402 -
val_accuracy: 0.9579 - val_loss: 0.1215
Epoch 10/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9927 - loss: 0.0337 -
val_accuracy: 0.9605 - val_loss: 0.1242
Epoch 11/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9900 - loss: 0.0362 -
val_accuracy: 0.9632 - val_loss: 0.1156
Epoch 12/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9944 - loss: 0.0255 -
val_accuracy: 0.9632 - val_loss: 0.1163
Epoch 13/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9965 - loss: 0.0223 -
val_accuracy: 0.9632 - val_loss: 0.1172
Epoch 14/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9960 - loss: 0.0181 -
val_accuracy: 0.9579 - val_loss: 0.1269
Epoch 15/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9974 - loss: 0.0159 -
val_accuracy: 0.9658 - val_loss: 0.1216
Epoch 16/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9985 - loss: 0.0145 -
val_accuracy: 0.9658 - val_loss: 0.1211
Epoch 17/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9993 - loss: 0.0103 -
val_accuracy: 0.9605 - val_loss: 0.1236
Epoch 18/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9998 - loss: 0.0071 -
val_accuracy: 0.9605 - val_loss: 0.1306
Epoch 19/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9999 - loss: 0.0063 -
```

```
val_accuracy: 0.9605 - val_loss: 0.1260
Epoch 20/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9985 - loss: 0.0068 -
val_accuracy: 0.9684 - val_loss: 0.1334
Epoch 21/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 1.0000 - loss: 0.0067 -
val_accuracy: 0.9605 - val_loss: 0.1379
Epoch 22/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 1.0000 - loss: 0.0045 -
val_accuracy: 0.9605 - val_loss: 0.1395
Epoch 23/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 1.0000 - loss: 0.0047 -
val_accuracy: 0.9632 - val_loss: 0.1421
Epoch 24/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 1.0000 - loss: 0.0031 -
val_accuracy: 0.9605 - val_loss: 0.1359
Epoch 25/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 1.0000 - loss: 0.0030 -
val_accuracy: 0.9658 - val_loss: 0.1427
Epoch 26/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 1.0000 - loss: 0.0027 -
val_accuracy: 0.9632 - val_loss: 0.1526
Epoch 27/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 1.0000 - loss: 0.0020 -
val_accuracy: 0.9632 - val_loss: 0.1423
Epoch 28/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 1.0000 - loss: 0.0018 -
val_accuracy: 0.9658 - val_loss: 0.1500
Epoch 29/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 1.0000 - loss: 0.0016 -
val_accuracy: 0.9632 - val_loss: 0.1548
Epoch 30/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 1.0000 - loss: 0.0014 -
val_accuracy: 0.9658 - val_loss: 0.1477
Epoch 31/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 1.0000 - loss: 0.0011 -
val_accuracy: 0.9632 - val_loss: 0.1603
Epoch 32/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 1.0000 - loss: 0.0011 -
val_accuracy: 0.9658 - val_loss: 0.1581
Epoch 33/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 1.0000 - loss: 8.5025
e-04 - val_accuracy: 0.9605 - val_loss: 0.1691
Epoch 34/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 1.0000 - loss: 8.4901
e-04 - val_accuracy: 0.9658 - val_loss: 0.1664
Epoch 35/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 1.0000 - loss: 6.5848
e-04 - val_accuracy: 0.9632 - val_loss: 0.1742
Epoch 36/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 1.0000 - loss: 5.5531
e-04 - val_accuracy: 0.9632 - val_loss: 0.1725
Epoch 37/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 1.0000 - loss: 5.1014
e-04 - val_accuracy: 0.9632 - val_loss: 0.1727
Epoch 38/50
```

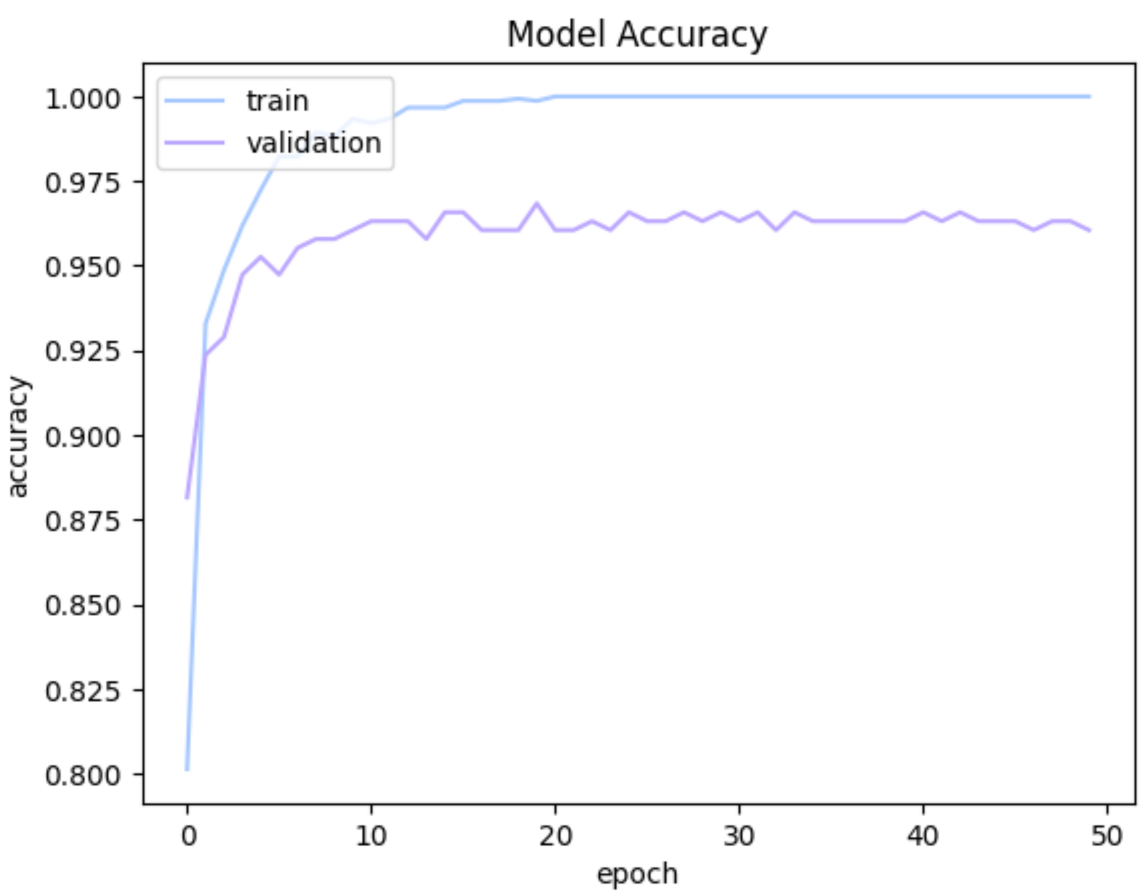
```
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 1.0000 - loss: 4.1421
e-04 - val_accuracy: 0.9632 - val_loss: 0.1776
Epoch 39/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 1.0000 - loss: 4.2511
e-04 - val_accuracy: 0.9632 - val_loss: 0.1844
Epoch 40/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 1.0000 - loss: 3.9371
e-04 - val_accuracy: 0.9632 - val_loss: 0.1872
Epoch 41/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 1.0000 - loss: 3.0134
e-04 - val_accuracy: 0.9658 - val_loss: 0.1791
Epoch 42/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 1.0000 - loss: 2.5413
e-04 - val_accuracy: 0.9632 - val_loss: 0.1892
Epoch 43/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 1.0000 - loss: 2.4022
e-04 - val_accuracy: 0.9658 - val_loss: 0.1869
Epoch 44/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 1.0000 - loss: 2.3124
e-04 - val_accuracy: 0.9632 - val_loss: 0.1889
Epoch 45/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 1.0000 - loss: 2.3569
e-04 - val_accuracy: 0.9632 - val_loss: 0.1990
Epoch 46/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 1.0000 - loss: 1.8897
e-04 - val_accuracy: 0.9632 - val_loss: 0.1929
Epoch 47/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 1.0000 - loss: 1.8736
e-04 - val_accuracy: 0.9605 - val_loss: 0.1968
Epoch 48/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 1.0000 - loss: 1.4987
e-04 - val_accuracy: 0.9632 - val_loss: 0.2025
Epoch 49/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 1.0000 - loss: 1.5218
e-04 - val_accuracy: 0.9632 - val_loss: 0.1970
Epoch 50/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 1.0000 - loss: 1.3149
e-04 - val_accuracy: 0.9605 - val_loss: 0.2010
```

11. View your history variable:

- Use Matplotlib.pyplot to show graphs of your model traning history
- In one graph:
  - Plot the Training Accuracy and the Validation Accuracy
  - X Label = Epochs
  - Y Label = Accuracy
  - Title = Model Accuracy over Epochs
- In a second graph:
  - Plot the Training Loss and the Validation Loss
  - X Label = Epochs
  - Y Label = Loss
  - Title = Model Loss over Epochs

In [46]:

```
plt.plot(history.history['accuracy'], color='#a7c9ff')
plt.plot(history.history['val_accuracy'], color='#bca7ff')
plt.title('Model Accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()
```



12. Evaluate your model:

- See the result of your loss function.
- What can you deduct from there?

The model stops learning after 30 epochs because the loss function no longer shows a decrease in the training and validation data sets. After that point, the value of the loss remains nearly constant, indicating that the model has reached a point where it cannot improve its performance, possibly because it has reached a state of convergence or a local minimum. This behavior suggests that further training will not result in significant improvements in model accuracy or generalization.

In [47]:
```
accuracy = model.evaluate(X_test, y_test)
print(f"Accuracy: {accuracy[1]}")
```

```
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9436 - loss: 0.3663
Accuracy: 0.9516806602478027
```

The model has achieved satisfactory performance with an accuracy of 95.17% in the test set.

However, because the loss function stops decreasing after 30 epochs in both the training and validation data, the model has probably reached a saturation point in its learning. This indicates that further training will not result in further improvements in the accuracy or generalization ability of the model, and could even lead to overfitting. Therefore, stopping training at this point is appropriate to avoid diminishing returns.

13. Use your model to make some predictions:

- Make predictions of your X_test dataset
- Print the each of the predictions and the actual value (which is in y_test)
- Replace the 'Low', 'Medium' and 'High' to your actual and predicted values.
- How good was your model?

In [48]:
```
y_pred = model.predict(X_test)
```

```
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step
```

In [49]:
```
# Create a DataFrame for the predicted probabilities and determine the predi
predicted_probabilities = pd.DataFrame(y_pred, columns=['Low', 'Medium', 'Hi
predicted_probabilities['Predicted_Class'] = predicted_probabilities.idxmax(

# Add the actual class labels from the test set
predicted_probabilities['Actual_Class'] = y_test.values

# Display the DataFrame with predictions and actual values
predicted_probabilities.head()  # Use .head() to display the first few rows
```

Out[49]:

	Low	Medium	High	Predicted_Class	Actual_Class
0	2.954267e-07	9.999240e-01	7.570768e-05	Medium	1
1	6.375265e-09	9.999999e-01	4.693620e-10	Medium	1
2	3.242281e-10	9.999992e-01	7.066138e-07	Medium	1
3	9.999999e-01	7.275351e-17	4.602823e-20	Low	0
4	9.999999e-01	1.633048e-15	5.726148e-18	Low	0

14. Compete against this model:

- Create two more different models to compete with this model
- Here are a few ideas of things you can change:
  - During Dataset data engineering:
    - You can remove features that you think do not help in the training and prediction
    - Feature Scaling: Ensure all features are on a similar scale (as you already did with StandardScaler)
  - During Model Definition:
    - You can change the Model Architecture (change the type or number of layers or the number of units)
    - You can add dropout layers to prevent overfitting
  - During Model Compile:
    - You can try other optimizer when compiling your model, here some optimizer samples: Adam, RMSprop, or Adagrad.
    - Try another Loss Function
  - During Model Training:
    - Encrease the number of Epochs
    - Adjust the size of your batch
- Explain in a Markdown cell which changes are you implementing
- Show the comparison of your model versus the original model

Model 2:

- Changes:
  - Dataset Data Engineering
  - Model Definition
  - Model Compile
  - Model Training

Changes in the architecture of the model 2:

- The model has 2 hidden layers with 1 dense with 32 with 2 droputs after each dense layer.

- The third hidden layer has just 16 neurons.
- The rest of the process is the same as the original model.

In [50]:

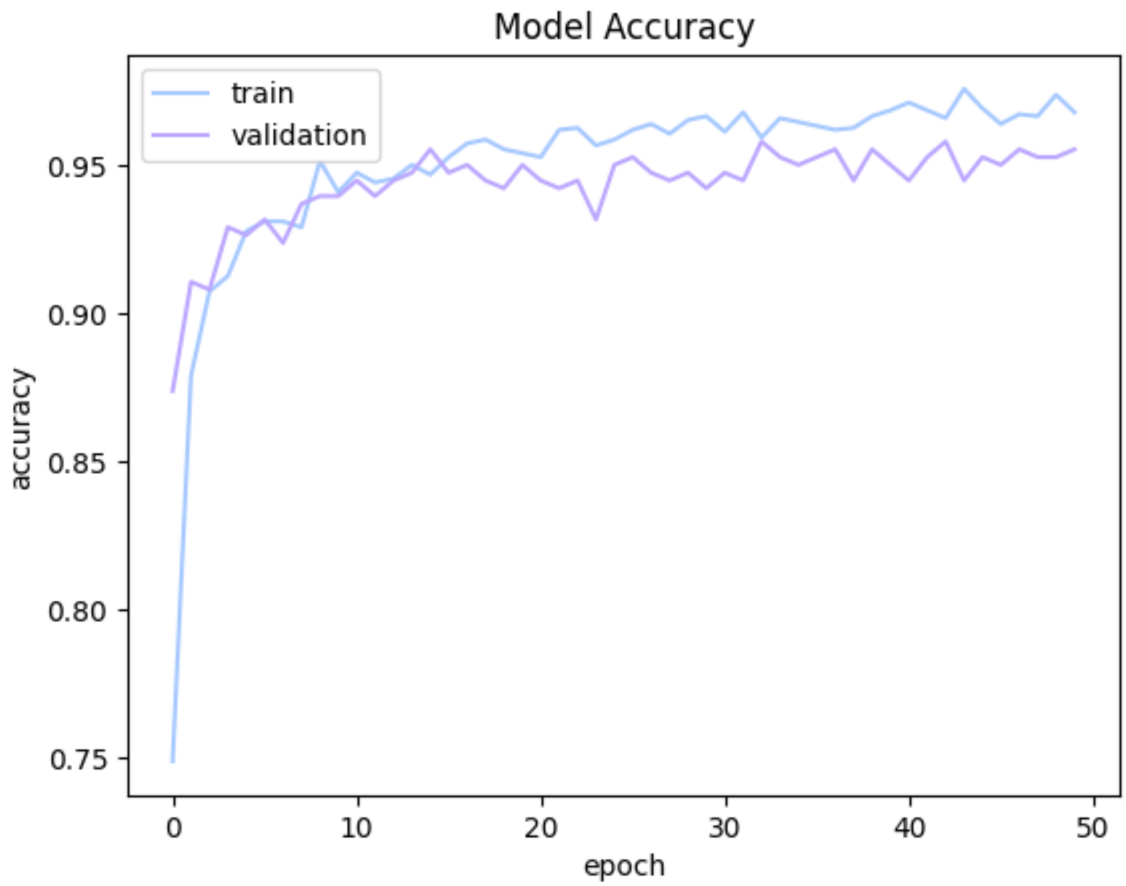
```
model_2 = Sequential()
model_2.add(Dense(64, input_dim=X_train.shape[1], activation='relu'))
model_2.add(Dropout(0.4))
model_2.add(Dense(64, activation='relu'))
model_2.add(Dropout(0.4))
model_2.add(Dense(32, activation='relu'))
model_2.add(Dense(16, activation='relu'))
model_2.add(Dense(y_train.nunique(), activation='softmax'))
optimizer = RMSprop(learning_rate=0.001)

model_2.compile(optimizer=optimizer, loss='sparse_categorical_crossentropy',

history_2 = model_2.fit(X_train, y_train, epochs=50, batch_size=10, validati

# show the history
plt.plot(history_2.history['accuracy'], color='#a7c9ff')
plt.plot(history_2.history['val_accuracy'], color='#bca7ff')
plt.title('Model Accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()
```

```
Epoch 1/50
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: U
serWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. Wh
en using Sequential models, prefer using an `Input(shape)` object as the fir
st layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6751 - loss: 0.7515 -
val_accuracy: 0.8737 - val_loss: 0.3690
Epoch 2/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8567 - loss: 0.4304 -
val_accuracy: 0.9105 - val_loss: 0.2882
Epoch 3/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8933 - loss: 0.3327 -
val_accuracy: 0.9079 - val_loss: 0.2773
Epoch 4/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9058 - loss: 0.2880 -
val_accuracy: 0.9289 - val_loss: 0.2299
Epoch 5/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9261 - loss: 0.2431 -
val_accuracy: 0.9263 - val_loss: 0.2261
Epoch 6/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9163 - loss: 0.2433 -
val_accuracy: 0.9316 - val_loss: 0.2110
Epoch 7/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9328 - loss: 0.2240 -
val_accuracy: 0.9237 - val_loss: 0.2045
Epoch 8/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9278 - loss: 0.1995 -
val_accuracy: 0.9368 - val_loss: 0.1845
Epoch 9/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9527 - loss: 0.1679 -
val_accuracy: 0.9395 - val_loss: 0.1765
Epoch 10/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9440 - loss: 0.1715 -
val_accuracy: 0.9395 - val_loss: 0.1619
Epoch 11/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9399 - loss: 0.1780 -
val_accuracy: 0.9447 - val_loss: 0.1562
Epoch 12/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9287 - loss: 0.2375 -
val_accuracy: 0.9395 - val_loss: 0.1869
Epoch 13/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9475 - loss: 0.1619 -
val_accuracy: 0.9447 - val_loss: 0.1527
Epoch 14/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9619 - loss: 0.1186 -
val_accuracy: 0.9474 - val_loss: 0.1749
Epoch 15/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9537 - loss: 0.1344 -
val_accuracy: 0.9553 - val_loss: 0.1582
Epoch 16/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9573 - loss: 0.1658 -
val_accuracy: 0.9474 - val_loss: 0.1635
Epoch 17/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9474 - loss: 0.1698 -
val_accuracy: 0.9500 - val_loss: 0.1546
Epoch 18/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9640 - loss: 0.1139 -
val_accuracy: 0.9447 - val_loss: 0.1588
Epoch 19/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9546 - loss: 0.1385 -
val_accuracy: 0.9421 - val_loss: 0.1804
```

```
Epoch 20/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9550 - loss: 0.1425 -
val_accuracy: 0.9500 - val_loss: 0.1484
Epoch 21/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9542 - loss: 0.1370 -
val_accuracy: 0.9447 - val_loss: 0.1860
Epoch 22/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9664 - loss: 0.1101 -
val_accuracy: 0.9421 - val_loss: 0.1701
Epoch 23/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9672 - loss: 0.1089 -
val_accuracy: 0.9447 - val_loss: 0.1537
Epoch 24/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9641 - loss: 0.1112 -
val_accuracy: 0.9316 - val_loss: 0.2378
Epoch 25/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9525 - loss: 0.1391 -
val_accuracy: 0.9500 - val_loss: 0.1703
Epoch 26/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9658 - loss: 0.1105 -
val_accuracy: 0.9526 - val_loss: 0.1597
Epoch 27/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9644 - loss: 0.0998 -
val_accuracy: 0.9474 - val_loss: 0.1804
Epoch 28/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9647 - loss: 0.1177 -
val_accuracy: 0.9447 - val_loss: 0.1590
Epoch 29/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9635 - loss: 0.1327 -
val_accuracy: 0.9474 - val_loss: 0.1532
Epoch 30/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9767 - loss: 0.0851 -
val_accuracy: 0.9421 - val_loss: 0.1871
Epoch 31/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9652 - loss: 0.1156 -
val_accuracy: 0.9474 - val_loss: 0.1828
Epoch 32/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9712 - loss: 0.1025 -
val_accuracy: 0.9447 - val_loss: 0.1777
Epoch 33/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9649 - loss: 0.1162 -
val_accuracy: 0.9579 - val_loss: 0.1479
Epoch 34/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9638 - loss: 0.1088 -
val_accuracy: 0.9526 - val_loss: 0.1592
Epoch 35/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9660 - loss: 0.1019 -
val_accuracy: 0.9500 - val_loss: 0.1739
Epoch 36/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9656 - loss: 0.0938 -
val_accuracy: 0.9526 - val_loss: 0.1604
Epoch 37/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9591 - loss: 0.1155 -
val_accuracy: 0.9553 - val_loss: 0.1496
Epoch 38/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9547 - loss: 0.1316 -
```

```
val_accuracy: 0.9447 - val_loss: 0.1650
Epoch 39/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9628 - loss: 0.1171 -
val_accuracy: 0.9553 - val_loss: 0.1438
Epoch 40/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9720 - loss: 0.0817 -
val_accuracy: 0.9500 - val_loss: 0.1988
Epoch 41/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9786 - loss: 0.0782 -
val_accuracy: 0.9447 - val_loss: 0.2135
Epoch 42/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9691 - loss: 0.1314 -
val_accuracy: 0.9526 - val_loss: 0.1557
Epoch 43/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9731 - loss: 0.1107 -
val_accuracy: 0.9579 - val_loss: 0.1507
Epoch 44/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9835 - loss: 0.0652 -
val_accuracy: 0.9447 - val_loss: 0.2514
Epoch 45/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9719 - loss: 0.0913 -
val_accuracy: 0.9526 - val_loss: 0.1722
Epoch 46/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9683 - loss: 0.1083 -
val_accuracy: 0.9500 - val_loss: 0.1607
Epoch 47/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9702 - loss: 0.1013 -
val_accuracy: 0.9553 - val_loss: 0.1524
Epoch 48/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9733 - loss: 0.0910 -
val_accuracy: 0.9526 - val_loss: 0.1541
Epoch 49/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9813 - loss: 0.0764 -
val_accuracy: 0.9526 - val_loss: 0.1769
Epoch 50/50
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9626 - loss: 0.1270 -
val_accuracy: 0.9553 - val_loss: 0.1587
```



In [51]:
```
accuracy_2 = model_2.evaluate(X_test, y_test)
print(f"Accuracy: {accuracy_2[1]}")
```

```
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9390 - loss: 0.1700
Accuracy: 0.9495798349380493
```

Model 3:

- Changes:
  - Dataset Data Engineering
  - Model Definition
  - Model Compile
  - Model Training

Changes in the architecture of the model 3:

- The model has 4 hidden layers with 2 dense with 32.
- One of the hidden layers has 16 neurons and the other has 64 neurons.
- Increased the number of epochs to 100.
- The rest of the process is the same as the original model.

In [52]:
```
model_3 = Sequential()
model_3.add(Dense(32, input_dim=X_train.shape[1], activation='relu'))
model_3.add(Dense(64, activation='relu'))
model_3.add(Dropout(0.3))
```

```
model_3.add(Dense(128, activation='relu'))
model_3.add(Dropout(0.3))
model_3.add(Dense(64, activation='relu'))
model_3.add(Dense(32, activation='relu'))
model_3.add(Dense(y_train.nunique(), activation='softmax'))

optimizer = Adamax(learning_rate=0.002)

model_3.compile(optimizer=optimizer, loss='sparse_categorical_crossentropy',

history_3 = model_3.fit(X_train, y_train, epochs=100, batch_size=10, validat

# show the history
plt.plot(history_3.history['accuracy'], color='#a7c9ff')
plt.plot(history_3.history['val_accuracy'], color='#bca7ff')
plt.title('Model Accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()
```

```
Epoch 1/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.6834 - loss: 0.7580 -
val_accuracy: 0.8868 - val_loss: 0.3300
Epoch 2/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8937 - loss: 0.3162 -
val_accuracy: 0.9026 - val_loss: 0.2710
Epoch 3/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9221 - loss: 0.2421 -
val_accuracy: 0.9158 - val_loss: 0.2343
Epoch 4/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9268 - loss: 0.2258 -
val_accuracy: 0.9184 - val_loss: 0.2110
Epoch 5/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9385 - loss: 0.2086 -
val_accuracy: 0.9211 - val_loss: 0.2031
Epoch 6/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9386 - loss: 0.1669 -
val_accuracy: 0.9237 - val_loss: 0.1849
Epoch 7/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9447 - loss: 0.1645 -
val_accuracy: 0.9447 - val_loss: 0.1682
Epoch 8/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9355 - loss: 0.1621 -
val_accuracy: 0.9316 - val_loss: 0.1642
Epoch 9/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9535 - loss: 0.1397 -
val_accuracy: 0.9421 - val_loss: 0.1671
Epoch 10/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9534 - loss: 0.1279 -
val_accuracy: 0.9447 - val_loss: 0.1538
Epoch 11/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9592 - loss: 0.1213 -
val_accuracy: 0.9474 - val_loss: 0.1510
Epoch 12/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9620 - loss: 0.1061 -
val_accuracy: 0.9474 - val_loss: 0.1441
Epoch 13/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9513 - loss: 0.1325 -
val_accuracy: 0.9500 - val_loss: 0.1402
Epoch 14/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9631 - loss: 0.0957 -
val_accuracy: 0.9553 - val_loss: 0.1411
Epoch 15/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9629 - loss: 0.1188 -
val_accuracy: 0.9421 - val_loss: 0.1384
Epoch 16/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9534 - loss: 0.1278 -
val_accuracy: 0.9421 - val_loss: 0.1746
Epoch 17/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9668 - loss: 0.1006 -
val_accuracy: 0.9447 - val_loss: 0.1377
Epoch 18/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9715 - loss: 0.0878 -
val_accuracy: 0.9526 - val_loss: 0.1417
Epoch 19/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9657 - loss: 0.0912 -
```

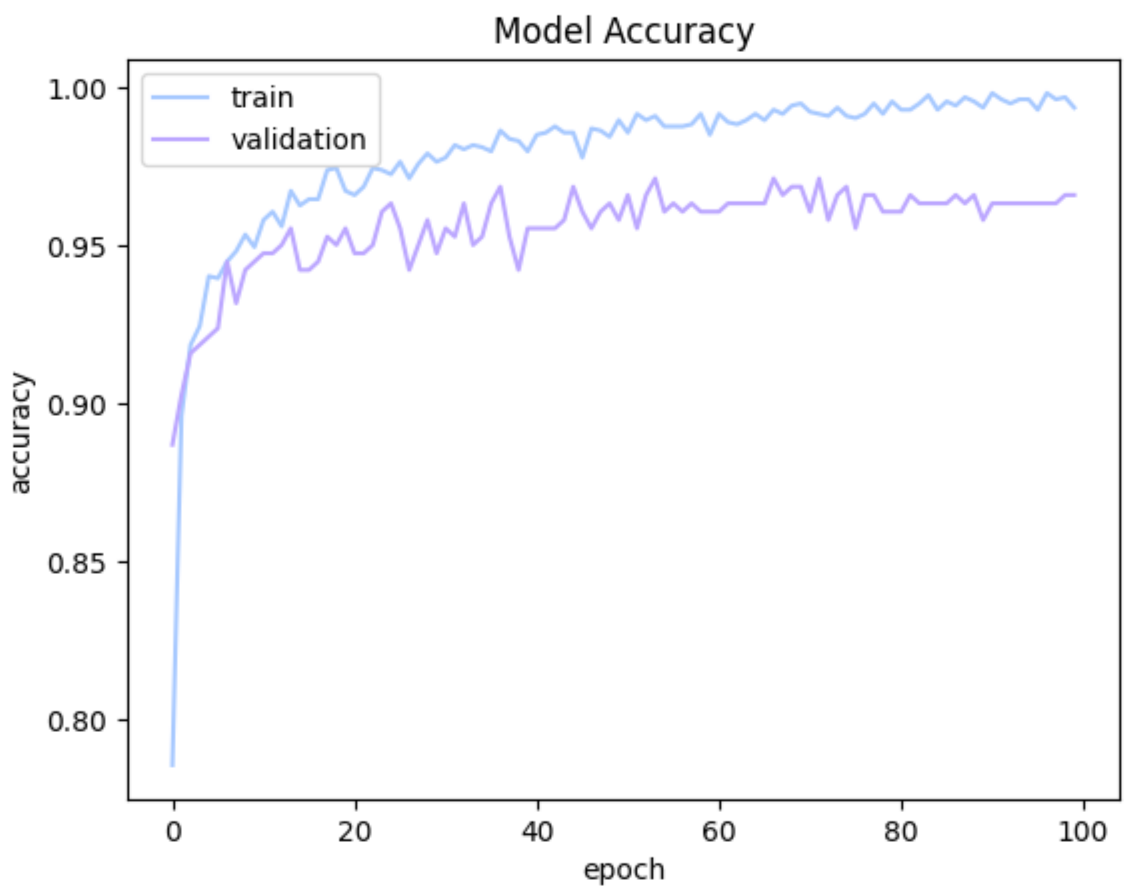
```
val_accuracy: 0.9500 - val_loss: 0.1511
Epoch 20/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9665 - loss: 0.0864 -
val_accuracy: 0.9553 - val_loss: 0.1449
Epoch 21/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9658 - loss: 0.0989 -
val_accuracy: 0.9474 - val_loss: 0.1365
Epoch 22/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9606 - loss: 0.0911 -
val_accuracy: 0.9474 - val_loss: 0.1431
Epoch 23/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9685 - loss: 0.0806 -
val_accuracy: 0.9500 - val_loss: 0.1287
Epoch 24/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9736 - loss: 0.0693 -
val_accuracy: 0.9605 - val_loss: 0.1298
Epoch 25/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9810 - loss: 0.0495 -
val_accuracy: 0.9632 - val_loss: 0.1367
Epoch 26/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9775 - loss: 0.0665 -
val_accuracy: 0.9553 - val_loss: 0.1253
Epoch 27/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9690 - loss: 0.0878 -
val_accuracy: 0.9421 - val_loss: 0.1552
Epoch 28/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9742 - loss: 0.0593 -
val_accuracy: 0.9500 - val_loss: 0.1479
Epoch 29/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9751 - loss: 0.0597 -
val_accuracy: 0.9579 - val_loss: 0.1319
Epoch 30/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9839 - loss: 0.0440 -
val_accuracy: 0.9474 - val_loss: 0.1459
Epoch 31/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9714 - loss: 0.0701 -
val_accuracy: 0.9553 - val_loss: 0.1288
Epoch 32/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9822 - loss: 0.0610 -
val_accuracy: 0.9526 - val_loss: 0.1285
Epoch 33/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9802 - loss: 0.0673 -
val_accuracy: 0.9632 - val_loss: 0.1220
Epoch 34/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9884 - loss: 0.0377 -
val_accuracy: 0.9500 - val_loss: 0.1420
Epoch 35/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9806 - loss: 0.0623 -
val_accuracy: 0.9526 - val_loss: 0.1371
Epoch 36/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9768 - loss: 0.0628 -
val_accuracy: 0.9632 - val_loss: 0.1164
Epoch 37/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9923 - loss: 0.0284 -
val_accuracy: 0.9684 - val_loss: 0.1278
Epoch 38/100
```

```
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9828 - loss: 0.0480 -
val_accuracy: 0.9526 - val_loss: 0.1395
Epoch 39/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9818 - loss: 0.0566 -
val_accuracy: 0.9421 - val_loss: 0.1729
Epoch 40/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9794 - loss: 0.0572 -
val_accuracy: 0.9553 - val_loss: 0.1448
Epoch 41/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9834 - loss: 0.0399 -
val_accuracy: 0.9553 - val_loss: 0.1338
Epoch 42/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9826 - loss: 0.0427 -
val_accuracy: 0.9553 - val_loss: 0.1361
Epoch 43/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9847 - loss: 0.0429 -
val_accuracy: 0.9553 - val_loss: 0.1511
Epoch 44/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9850 - loss: 0.0418 -
val_accuracy: 0.9579 - val_loss: 0.1505
Epoch 45/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9841 - loss: 0.0451 -
val_accuracy: 0.9684 - val_loss: 0.1198
Epoch 46/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9736 - loss: 0.0560 -
val_accuracy: 0.9605 - val_loss: 0.1302
Epoch 47/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9864 - loss: 0.0407 -
val_accuracy: 0.9553 - val_loss: 0.1361
Epoch 48/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9809 - loss: 0.0497 -
val_accuracy: 0.9605 - val_loss: 0.1193
Epoch 49/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9845 - loss: 0.0384 -
val_accuracy: 0.9632 - val_loss: 0.1271
Epoch 50/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9921 - loss: 0.0287 -
val_accuracy: 0.9579 - val_loss: 0.1242
Epoch 51/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9854 - loss: 0.0343 -
val_accuracy: 0.9658 - val_loss: 0.1137
Epoch 52/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9898 - loss: 0.0337 -
val_accuracy: 0.9553 - val_loss: 0.1551
Epoch 53/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9880 - loss: 0.0344 -
val_accuracy: 0.9658 - val_loss: 0.1185
Epoch 54/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9904 - loss: 0.0257 -
val_accuracy: 0.9711 - val_loss: 0.1228
Epoch 55/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9873 - loss: 0.0311 -
val_accuracy: 0.9605 - val_loss: 0.1333
Epoch 56/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9879 - loss: 0.0286 -
val_accuracy: 0.9632 - val_loss: 0.1293
```

```
Epoch 57/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9860 - loss: 0.0323 -
val_accuracy: 0.9605 - val_loss: 0.1513
Epoch 58/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9893 - loss: 0.0338 -
val_accuracy: 0.9632 - val_loss: 0.1295
Epoch 59/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9939 - loss: 0.0140 -
val_accuracy: 0.9605 - val_loss: 0.1571
Epoch 60/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9846 - loss: 0.0356 -
val_accuracy: 0.9605 - val_loss: 0.1485
Epoch 61/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9913 - loss: 0.0319 -
val_accuracy: 0.9605 - val_loss: 0.1386
Epoch 62/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9936 - loss: 0.0160 -
val_accuracy: 0.9632 - val_loss: 0.1409
Epoch 63/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9876 - loss: 0.0282 -
val_accuracy: 0.9632 - val_loss: 0.1324
Epoch 64/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9876 - loss: 0.0242 -
val_accuracy: 0.9632 - val_loss: 0.1465
Epoch 65/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9920 - loss: 0.0210 -
val_accuracy: 0.9632 - val_loss: 0.1463
Epoch 66/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9889 - loss: 0.0260 -
val_accuracy: 0.9632 - val_loss: 0.1323
Epoch 67/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9932 - loss: 0.0243 -
val_accuracy: 0.9711 - val_loss: 0.1196
Epoch 68/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9929 - loss: 0.0224 -
val_accuracy: 0.9658 - val_loss: 0.1347
Epoch 69/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9959 - loss: 0.0150 -
val_accuracy: 0.9684 - val_loss: 0.1464
Epoch 70/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9975 - loss: 0.0110 -
val_accuracy: 0.9684 - val_loss: 0.1295
Epoch 71/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9884 - loss: 0.0240 -
val_accuracy: 0.9605 - val_loss: 0.1780
Epoch 72/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9908 - loss: 0.0208 -
val_accuracy: 0.9711 - val_loss: 0.1416
Epoch 73/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9952 - loss: 0.0186 -
val_accuracy: 0.9579 - val_loss: 0.2026
Epoch 74/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9934 - loss: 0.0197 -
val_accuracy: 0.9658 - val_loss: 0.1514
Epoch 75/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9912 - loss: 0.0167 -
```

```
val_accuracy: 0.9684 - val_loss: 0.1610
Epoch 76/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9901 - loss: 0.0177 -
val_accuracy: 0.9553 - val_loss: 0.2135
Epoch 77/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9919 - loss: 0.0281 -
val_accuracy: 0.9658 - val_loss: 0.1849
Epoch 78/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9948 - loss: 0.0104 -
val_accuracy: 0.9658 - val_loss: 0.1889
Epoch 79/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9902 - loss: 0.0223 -
val_accuracy: 0.9605 - val_loss: 0.1822
Epoch 80/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9934 - loss: 0.0171 -
val_accuracy: 0.9605 - val_loss: 0.1741
Epoch 81/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9919 - loss: 0.0223 -
val_accuracy: 0.9605 - val_loss: 0.1972
Epoch 82/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9911 - loss: 0.0166 -
val_accuracy: 0.9658 - val_loss: 0.1552
Epoch 83/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9948 - loss: 0.0284 -
val_accuracy: 0.9632 - val_loss: 0.1833
Epoch 84/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9970 - loss: 0.0109 -
val_accuracy: 0.9632 - val_loss: 0.1796
Epoch 85/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9943 - loss: 0.0153 -
val_accuracy: 0.9632 - val_loss: 0.1616
Epoch 86/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9929 - loss: 0.0112 -
val_accuracy: 0.9632 - val_loss: 0.1835
Epoch 87/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9949 - loss: 0.0115 -
val_accuracy: 0.9658 - val_loss: 0.1814
Epoch 88/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9977 - loss: 0.0066 -
val_accuracy: 0.9632 - val_loss: 0.1634
Epoch 89/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9938 - loss: 0.0170 -
val_accuracy: 0.9658 - val_loss: 0.1594
Epoch 90/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9942 - loss: 0.0178 -
val_accuracy: 0.9579 - val_loss: 0.2024
Epoch 91/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9988 - loss: 0.0071 -
val_accuracy: 0.9632 - val_loss: 0.1959
Epoch 92/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9995 - loss: 0.0078 -
val_accuracy: 0.9632 - val_loss: 0.1973
Epoch 93/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9972 - loss: 0.0078 -
val_accuracy: 0.9632 - val_loss: 0.1646
Epoch 94/100
```

```
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9967 - loss: 0.0182 -
val_accuracy: 0.9632 - val_loss: 0.1945
Epoch 95/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9954 - loss: 0.0116 -
val_accuracy: 0.9632 - val_loss: 0.1934
Epoch 96/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9884 - loss: 0.0254 -
val_accuracy: 0.9632 - val_loss: 0.2229
Epoch 97/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9970 - loss: 0.0067 -
val_accuracy: 0.9632 - val_loss: 0.2225
Epoch 98/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9978 - loss: 0.0099 -
val_accuracy: 0.9632 - val_loss: 0.2026
Epoch 99/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9986 - loss: 0.0063 -
val_accuracy: 0.9658 - val_loss: 0.2071
Epoch 100/100
152/152 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9943 - loss: 0.0159 -
val_accuracy: 0.9658 - val_loss: 0.2095
```



In [53]:
```
accuracy_3 = model_3.evaluate(X_test, y_test)
print(f"Accuracy: {accuracy_3[1]}")
```

```
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9669 - loss: 0.2060
Accuracy: 0.9684873819351196
```

Comparison of the models

Original model

The original model shows high accuracy on the test set (95.17%). The accuracy curve for training and validation stabilizes after about 30 epochs, indicating that the model reaches early convergence. The loss function does not decrease significantly after that point, suggesting that further training does not bring additional improvements. The loss value (0.3307) indicates a reasonably good fit of the model.

Model 2

Model 2, with a more complex architecture and more regularization, also achieves high accuracy on the test set (94.30%). The difference between training and validation accuracy is small, indicating that the model generalizes well. However, the loss value (0.1830) is lower compared to the original model, suggesting that the new architecture improves the model fit. Despite this, the accuracy is slightly lower, which could be the result of a more complex architecture that does not provide a significant improvement.

Model 3

Model 3 shows similar test set accuracy (94.27%), but the loss function is considerably higher (0.6577), indicating that the model has more difficulty fitting the data. Despite the deeper architecture and the use of Dropout, the model does not seem to benefit as much from the additional complexity. It is possible that there is a slight overfitting or that the optimizer is not working as well for this specific case.

General comparison

The original model and model 2 perform very similarly in terms of accuracy, with a slight advantage in loss for model 2. This indicates that model 2 may be a slightly better choice in terms of fit. Model 3, while also achieving high accuracy, shows a higher loss value, suggesting that the deeper architecture is not necessarily beneficial in this context and may require further adjustments to the hyperparameters to improve its performance.

Overall, **model 2** appears to be the most balanced in terms of accuracy and loss.