In [32]:
```python
# Import necessary libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.metrics import accuracy_score, classification_report
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import matplotlib.pyplot as plt
from sklearn.preprocessing import OneHotEncoder
```

In [33]:
```python
# Load datasets
logs = pd.read_csv(r"C:\Users\junai\OneDrive - Middlesex University\Applied Data Ar
grades = pd.read_csv(r"C:\Users\junai\OneDrive - Middlesex University\Applied Data
```

In [34]:
```python
logs.head(8)
```

Out[34]:

	StudentId	Time	Type	Action
0	72af	28/05/23, 10:51	User report	Grade user report viewed
1	72af	28/05/23, 10:51	System	Course viewed
2	c426	27/05/23, 15:53	System	Course viewed
3	0326	26/05/23, 22:22	System	Course viewed
4	8b7a	26/05/23, 21:52	System	Course viewed
5	8b7a	26/05/23, 21:52	Open Grader	Open Grader viewed
6	8b7a	26/05/23, 21:52	System	Course viewed
7	bde7	26/05/23, 20:06	System	Course viewed

In [35]:
```python
grades.head(8)
```

Out[35]:

	StudentId	Grade
0	c426	2nd
1	8de3	2nd
2	d969	2nd
3	6d29	1st
4	1dd9	1st
5	f63c	1st
6	0a2e	3rd
7	06f3	3rd

Created a dictionary (grade_mapping) to map grade categories to numeric values. This will replace the values in the 'Grade' column with their corresponding numeric values according to the grade_mapping dictionary.

In [37]:
```python
# Map grade categories to numeric values
grade_mapping = {'1st': 1, '2nd': 2, '3rd': 3, 'Fail': 0}
```

In [38]:
```
# Apply the mapping to the 'grade' column
grades['Grade'] = grades['Grade'].map(grade_mapping)
```

In [39]:
```
# Now you can use nlargest on the numeric 'Grade' column
top_grades = grades['Grade'].nlargest(10)
print(top_grades)
```

```
6     3
7     3
9     3
10    3
13    3
14    3
15    3
19    3
22    3
25    3
Name: Grade, dtype: int64
```

Merging two DataFrames (logs and grades) on the 'StudentId' column using the merge function.

In [40]:
```
# Merge datasets on 'StudentId'
data = pd.merge(logs, grades, on='StudentId', how='inner')
```

In [41]:
```
data.head(5)
```

Out[41]:

	StudentId	Time	Type	Action	Grade
0	72af	28/05/23, 10:51	User report	Grade user report viewed	1
1	72af	28/05/23, 10:51	System	Course viewed	1
2	72af	26/05/23, 09:58	User report	Grade user report viewed	1
3	72af	26/05/23, 09:58	System	Course viewed	1
4	72af	22/05/23, 16:15	User report	Grade user report viewed	1

In [42]:
```
# Missing Values
print(logs.isnull().sum())
```

```
StudentId    0
Time         0
Type         0
Action       0
dtype: int64
```

In [43]:
```
print(grades.isnull().sum())
```

```
StudentId    0
Grade        0
dtype: int64
```

Preprocessing the 'Time' column in your DataFrame data to extract additional features such as 'hour', 'weekday', 'day', and 'month'. data['Time'] = pd.to_datetime(data['Time']): This line converts the 'Time' column to a pandas datetime format, which allows for easier extraction of various time-related features.

data['hour'] = data['Time'].dt.hour: This line extracts the hour component from the 'Time' column and creates a new column named 'hour' in the DataFrame.

data['weekday'] = data['Time'].dt.weekday: This line extracts the weekday (0 = Monday, 1 = Tuesday, ..., 6 = Sunday) and creates a new column named 'weekday' in the DataFrame.

data['day'] = data['Time'].dt.day: This line extracts the day of the month and creates a new column named 'day' in the DataFrame.

data['month'] = data['Time'].dt.month: This line extracts the month and creates a new column named 'month' in the DataFrame.

In [44]:

```
# Data Preprocessing
# Assuming 'Time' is a timestamp
data['Time'] = pd.to_datetime(data['Time'])
data['hour'] = data['Time'].dt.hour
data['weekday'] = data['Time'].dt.weekday
data['day'] = data['Time'].dt.day
data['month'] = data['Time'].dt.month
```

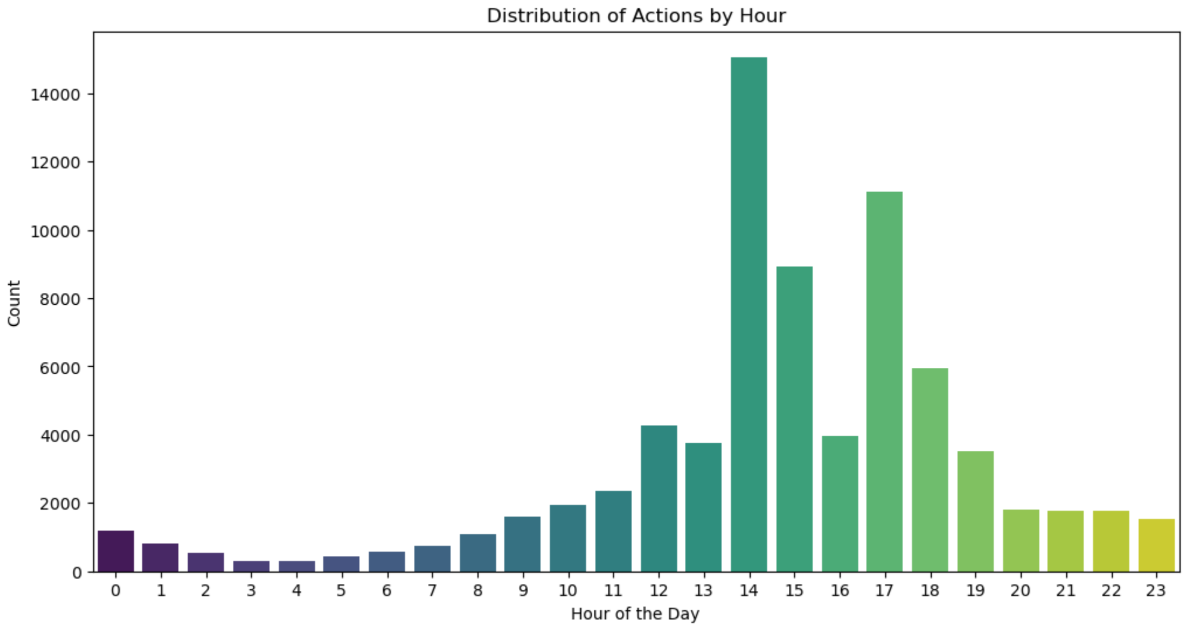
```
C:\Users\junai\AppData\Local\Temp\ipykernel_19588\32314884.py:3: UserWarning: Coul
d not infer format, so each element will be parsed individually, falling back to `
dateutil`. To ensure parsing is consistent and as-expected, please specify a forma
t.
  data['Time'] = pd.to_datetime(data['Time'])
```

Insights: This plot shows the distribution of actions throughout the day. You can identify peak hours of activity and periods of low activity. It helps you understand when students are most engaged with the platform.

In [45]:

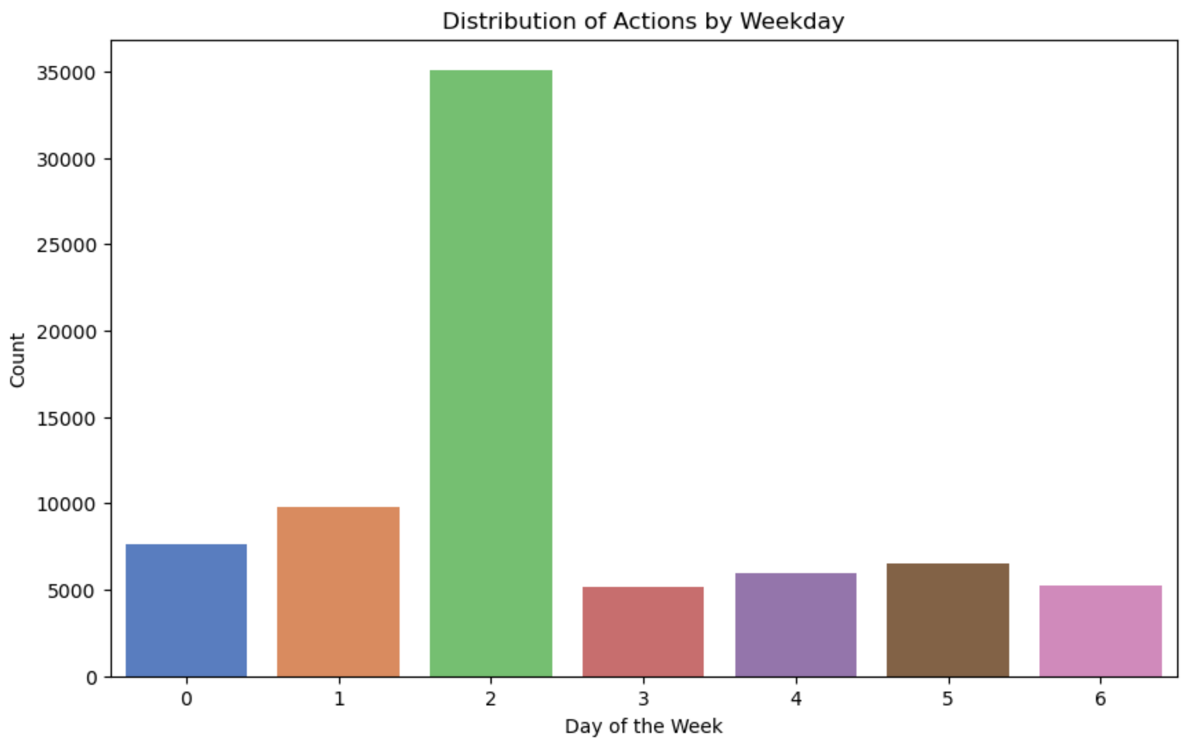
```
import matplotlib.pyplot as plt
import seaborn as sns

# Example: Distribution of actions by hour
plt.figure(figsize=(12, 6))
sns.countplot(x='hour', data=data, palette='viridis')
plt.title('Distribution of Actions by Hour')
plt.xlabel('Hour of the Day')
plt.ylabel('Count')
plt.show()
```



This plot visualizes how actions are distributed across weekdays. It can help you identify if there are certain days of the week when students are more active or less active.

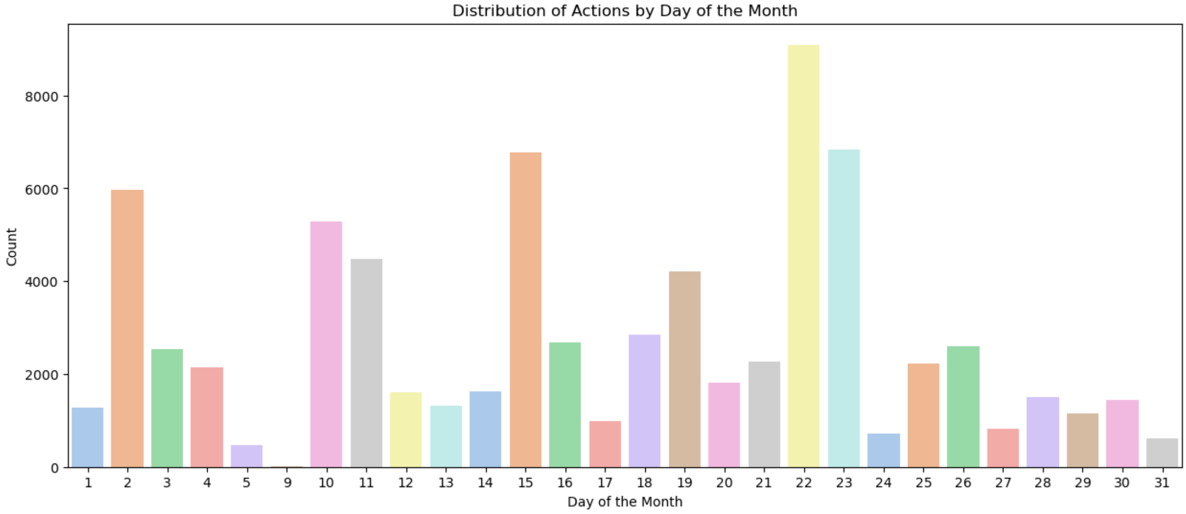
In [46]:
```
# Example: Distribution of actions by weekday
plt.figure(figsize=(10, 6))
sns.countplot(x='weekday', data=data, palette='muted')
plt.title('Distribution of Actions by Weekday')
plt.xlabel('Day of the Week')
plt.ylabel('Count')
plt.show()
```



Examining the distribution of actions by day of the month can reveal any patterns or spikes in activity. For example, you might notice increased activity around assignment due dates.

In [47]:
```
# Example: Distribution of actions by day of the month
plt.figure(figsize=(15, 6))
sns.countplot(x='day', data=data, palette='pastel')
plt.title('Distribution of Actions by Day of the Month')
```

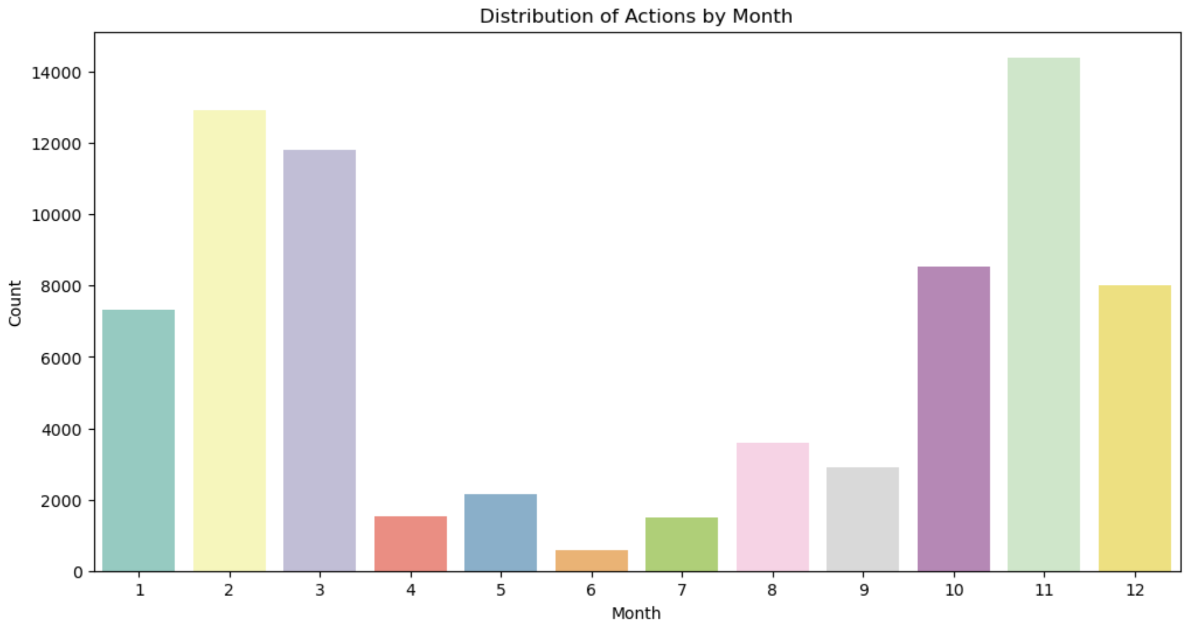
```
plt.xlabel('Day of the Month')
plt.ylabel('Count')
plt.show()
```



This plot shows the monthly distribution of actions. It can help you identify if there are specific months with higher or lower engagement levels. For example, there might be increased activity during exam months.

In [48]:

```
# Example: Distribution of actions by month
plt.figure(figsize=(12, 6))
sns.countplot(x='month', data=data, palette='Set3')
plt.title('Distribution of Actions by Month')
plt.xlabel('Month')
plt.ylabel('Count')
plt.show()
```



This code is performing one-hot encoding on the 'Type' and 'Action' columns, assuming they are categorical variables.

One-hot encoding is a technique used to convert categorical variables into a binary matrix (0s and 1s). For each unique category in the original column, a new binary column is created. The binary column corresponding to the category of each row is marked with a '1', and all other binary columns are marked with '0'.

If the 'Type' column had categories like 'Assignment', 'Quiz', and 'Discussion', and the 'Action' column had categories like 'Viewed', 'Created', and 'Submitted', after one-hot encoding, you might have columns like 'Type_Assignment', 'Type_Quiz', 'Type_Discussion', 'Action_Viewed', 'Action_Created', 'Action_Submitted', etc.

In [49]:
```
# Assuming 'Type' and 'Action' are categorical variables
data = pd.get_dummies(data, columns=['Type', 'Action'])
```

In [50]:
```
print(data.columns)
```

```
Index(['StudentId', 'Time', 'Grade', 'hour', 'weekday', 'day', 'month',
       'Type_Assignment', 'Type_File', 'Type_File submissions', 'Type_Folder',
       'Type_Forum', 'Type_Kaltura Video Resource', 'Type_Open Grader',
       'Type_Overview report', 'Type_Page', 'Type_Questionnaire', 'Type_Quiz',
       'Type_Scheduler', 'Type_System', 'Type_Turnitin Assignment 2',
       'Type_URL', 'Type_User report', 'Type_User tours',
       'Action_A file has been uploaded.',
       'Action_A submission has been submitted.', 'Action_Add Submission',
       'Action_Calendar event created', 'Action_Calendar event deleted',
       'Action_Course activity completion updated',
       'Action_Course module instance list viewed',
       'Action_Course module viewed', 'Action_Course searched',
       'Action_Course user report viewed', 'Action_Course viewed',
       'Action_Discussion created', 'Action_Discussion subscription created',
       'Action_Discussion subscription deleted', 'Action_Discussion viewed',
       'Action_Grade overview report viewed',
       'Action_Grade user report viewed',
       'Action_Individual Responses report viewed', 'Action_List Submissions',
       'Action_Open Grader viewed', 'Action_Post created',
       'Action_Post updated', 'Action_Quiz attempt reviewed',
       'Action_Quiz attempt started', 'Action_Quiz attempt submitted',
       'Action_Quiz attempt summary viewed', 'Action_Quiz attempt viewed',
       'Action_Recent activity viewed',
       'Action_Remove submission confirmation viewed.',
       'Action_Responses submitted', 'Action_Scheduler booking added',
       'Action_Scheduler booking form viewed',
       'Action_Scheduler booking removed',
       'Action_Some content has been posted.', 'Action_Step shown',
       'Action_Submission created.', 'Action_Submission form viewed.',
       'Action_Submission updated.',
       'Action_The status of the submission has been updated.',
       'Action_The status of the submission has been viewed.',
       'Action_Tour ended', 'Action_Tour started', 'Action_User graded',
       'Action_User list viewed', 'Action_User profile viewed',
       'Action_Video resource viewed',
       'Action_Zip archive of folder downloaded'],
      dtype='object')
```

In [51]:
```
print(data.columns.tolist())
```

```
['StudentId', 'Time', 'Grade', 'hour', 'weekday', 'day', 'month', 'Type_Assignmen
t', 'Type_File', 'Type_File submissions', 'Type_Folder', 'Type_Forum', 'Type_Kaltu
ra Video Resource', 'Type_Open Grader', 'Type_Overview report', 'Type_Page', 'Type
_Questionnaire', 'Type_Quiz', 'Type_Scheduler', 'Type_System', 'Type_Turnitin Assi
gnment 2', 'Type_URL', 'Type_User report', 'Type_User tours', 'Action_A file has b
een uploaded.', 'Action_A submission has been submitted.', 'Action_Add Submissio
n', 'Action_Calendar event created', 'Action_Calendar event deleted', 'Action_Cour
se activity completion updated', 'Action_Course module instance list viewed', 'Act
ion_Course module viewed', 'Action_Course searched', 'Action_Course user report vi
ewed', 'Action_Course viewed', 'Action_Discussion created', 'Action_Discussion sub
scription created', 'Action_Discussion subscription deleted', 'Action_Discussion v
iewed', 'Action_Grade overview report viewed', 'Action_Grade user report viewed',
'Action_Individual Responses report viewed', 'Action_List Submissions', 'Action_Op
en Grader viewed', 'Action_Post created', 'Action_Post updated', 'Action_Quiz atte
mpt reviewed', 'Action_Quiz attempt started', 'Action_Quiz attempt submitted', 'Ac
tion_Quiz attempt summary viewed', 'Action_Quiz attempt viewed', 'Action_Recent ac
tivity viewed', 'Action_Remove submission confirmation viewed.', 'Action_Responses
submitted', 'Action_Scheduler booking added', 'Action_Scheduler booking form viewe
d', 'Action_Scheduler booking removed', 'Action_Some content has been posted.', 'A
ction_Step shown', 'Action_Submission created.', 'Action_Submission form viewed.',
'Action_Submission updated.', 'Action_The status of the submission has been update
d.', 'Action_The status of the submission has been viewed.', 'Action_Tour ended',
'Action_Tour started', 'Action_User graded', 'Action_User list viewed', 'Action_Us
er profile viewed', 'Action_Video resource viewed', 'Action_Zip archive of folder
downloaded']
```

Label encoding is a technique used to convert categorical values into numerical labels. Each unique category is assigned a unique integer label.

The 'Grade' column is typically the target variable in a machine learning problem, representing the output or prediction variable.

In [52]:
```python
# Assuming 'Grade' is the target variable
label_encoder = LabelEncoder()
data['Grade'] = label_encoder.fit_transform(data['Grade'])
```

In [ ]:
```python

```

In [53]:
```python
# Convert time to datetime and extract useful features
data['Time'] = pd.to_datetime(data['Time'])
data['hour'] = data['Time'].dt.hour
data['day'] = data['Time'].dt.day
data['month'] = data['Time'].dt.month
data['weekday'] = data['Time'].dt.weekday
```

In [80]:
```python
# Train-Test Split
X = data.drop(['StudentId', 'Time'], axis=1)  # Features
y = data['Grade']  # Target
```

In [83]:
```python
# Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta
```

In [84]:
```python
# Convert string labels to numerical labels
label_encoder = LabelEncoder()
y_train_encoded = label_encoder.fit_transform(y_train)
y_test_encoded = label_encoder.transform(y_test)
```

In [85]:
```python
from keras.utils import to_categorical

# Assuming y is your label data
```

```
y_one_hot = to_categorical(y)

# Convert data types to float32
X = X.astype('float32')
y_one_hot = to_categorical(y)
```

One-hot encoding is a representation of categorical variables as binary vectors. It's commonly used when the categories don't have an ordinal relationship (i.e., the order doesn't matter), and you want to represent each category as a separate binary column. The output is a binary matrix representation (one-hot encoding) of the input labels.

In [86]:

```
# Convert numerical labels to one-hot encoding
y_train_one_hot = to_categorical(y_train_encoded, num_classes=len(label_encoder.cla
y_test_one_hot = to_categorical(y_test_encoded, num_classes=len(label_encoder.class
```

In [ ]:

In [87]:

```
# Check and convert data types
print(X.dtypes)
print(y.dtypes)
```

```
Grade                                          float32
hour                                           float32
weekday                                        float32
day                                            float32
month                                          float32
                                                ...
Action_User graded                             float32
Action_User list viewed                        float32
Action_User profile viewed                     float32
Action_Video resource viewed                   float32
Action_Zip archive of folder downloaded        float32
Length: 69, dtype: object
int64
```

In [ ]:

X_train_scaled = scaler.fit_transform(X_train): Fitting the scaler on the training data (X_train) and transforming it. This ensures that the scaling parameters (mean and standard deviation) are computed from the training data and then applied to both the training and testing data.

X_test_scaled = scaler.transform(X_test): Transforming the testing data (X_test) using the parameters (mean and standard deviation) computed from the training data. This ensures consistency in scaling between the training and testing sets.

StandardScaler standardizes features by removing the mean and scaling to unit variance. It's a common preprocessing step, especially for algorithms that are sensitive to the scale of input features, such as neural networks.

Dropout is a regularization technique commonly used in neural networks to prevent overfitting. It randomly drops a fraction of input units during training, which helps prevent the model from relying too much on specific nodes.

In [88]:

```
from sklearn.preprocessing import StandardScaler
from keras.layers import Dropout

# Feature Scaling
```

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Sequential is a linear stack of layers for building a neural network model. It allows you to create models layer-by-layer in a step-by-step fashion.

Dense is a fully connected layer, which means that each neuron in one layer is connected to every neuron in the next layer. The first Dense layer with 128 neurons and ReLU activation: This layer is the input layer. It takes the input data with a number of features equal to the number of columns in your dataset. Two hidden layers with 64 and 32 neurons and ReLU activation: These layers capture complex patterns in the data. The number of neurons is a hyperparameter that can be adjusted based on the complexity of the problem.

In [89]:
```
# Build the Model
model = Sequential()
#model.add(Dense(320, input_dim=X_train_scaled.shape[1], activation='relu'))
#model.add(Dropout(0.5))  # Adding dropout for regularization
#model.add(Dense(256, activation='relu'))
#model.add(Dropout(0.5))  # Adding dropout for regularization
model.add(Dense(128, activation='relu'))

model.add(Dense(64, activation='relu'))
model.add(Dense(32, activation='relu'))
#model.add(Dense(16, activation='relu'))
model.add(Dense(len(label_encoder.classes_), activation='softmax'))
```

Adam is an optimization algorithm that is widely used for training deep learning models. The algorithm computes adaptive learning rates for each parameter by considering both the average of past gradients (momentum) and the average of past squared gradients (RMSprop)

The learning rate is a hyperparameter that determines the step size at each iteration while moving toward a minimum of the loss function. By setting learning_rate=0.0001, I am using a lower learning rate compared to default values.

In [90]:
```
# Use a lower learning rate
from keras.optimizers import Adam

optimizer = Adam(learning_rate=0.0001)
```

The loss parameter is set to 'categorical_crossentropy'. This is a common loss function used for multiclass classification problems when the target variable is one-hot encoded. It measures the difference between the true distribution and the predicted distribution.

The optimizer parameter is set to the Adam optimizer with a specified learning rate (0.0001), as defined earlier. The optimizer is responsible for updating the model's weights based on the computed gradients.

The metrics parameter is set to ['accuracy']. During training, the accuracy metric will be computed and displayed.

Here's a brief summary of each parameter:

Loss Function (loss): Measures the error during training. Optimizer (optimizer): Determines how the model's weights are updated based on the calculated gradients. Metrics (metrics): Additional metrics to monitor during training. In this case, it's accuracy.

In [112…
```
model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['accur
```

In [113…
```
# Modify the output layer to have as many neurons as classes
#model.add(Dense(len(label_encoder.classes_), activation='softmax'))
```

In [114…
```
# Train the Model
history = model.fit(X_train_scaled, y_train_one_hot, epochs=10, batch_size=32, verb
```

```
Epoch 1/10
1882/1882 [==============================] - 5s 2ms/step - loss: 0.0029 - accurac
y: 0.9994
Epoch 2/10
1882/1882 [==============================] - 4s 2ms/step - loss: 0.0029 - accurac
y: 0.9994
Epoch 3/10
1882/1882 [==============================] - 4s 2ms/step - loss: 0.0022 - accurac
y: 0.9995
Epoch 4/10
1882/1882 [==============================] - 4s 2ms/step - loss: 0.0018 - accurac
y: 0.9995
Epoch 5/10
1882/1882 [==============================] - 4s 2ms/step - loss: 0.0032 - accurac
y: 0.9994
Epoch 6/10
1882/1882 [==============================] - 4s 2ms/step - loss: 0.0015 - accurac
y: 0.9996
Epoch 7/10
1882/1882 [==============================] - 5s 3ms/step - loss: 0.0022 - accurac
y: 0.9995
Epoch 8/10
1882/1882 [==============================] - 4s 2ms/step - loss: 0.0022 - accurac
y: 0.9995
Epoch 9/10
1882/1882 [==============================] - 4s 2ms/step - loss: 0.0020 - accurac
y: 0.9997
Epoch 10/10
1882/1882 [==============================] - 4s 2ms/step - loss: 0.0012 - accurac
y: 0.9998
```

Evaluation Method:

The evaluate method is used to evaluate the model on the test data. Inputs:

X_test_scaled: The scaled feature data of the test set.

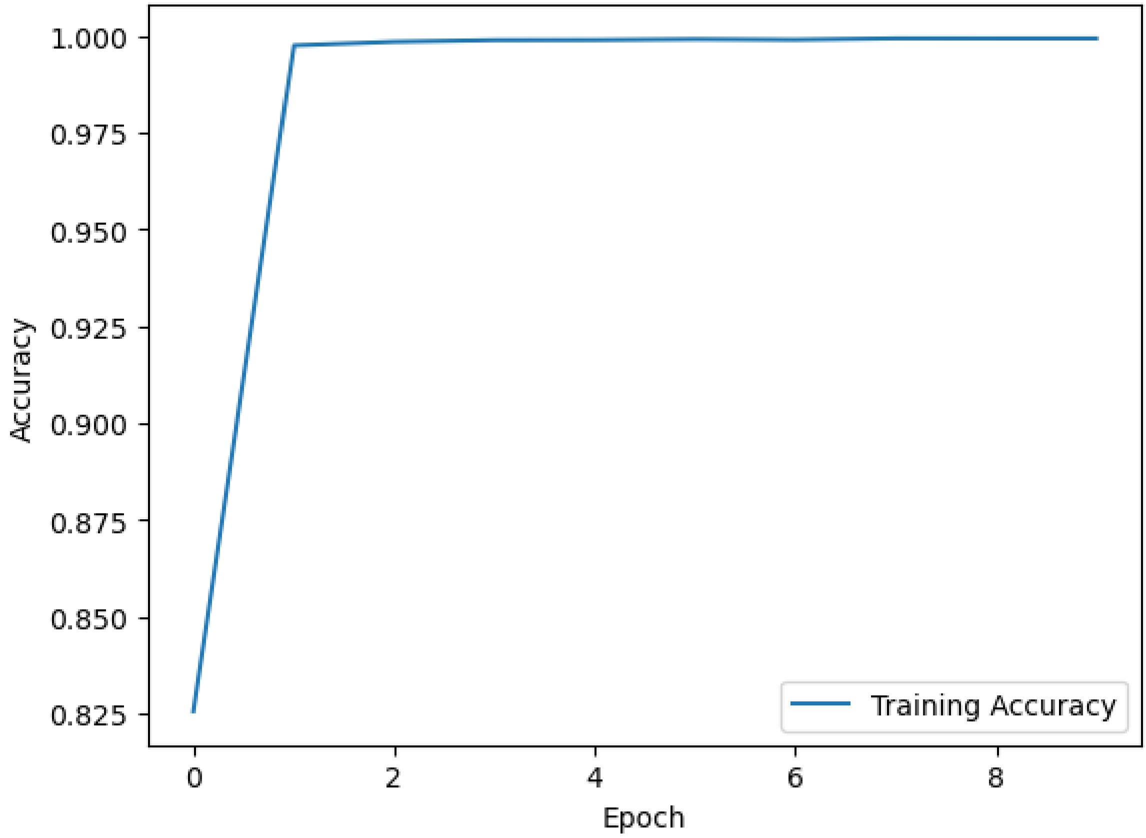
y_test_one_hot: The one-hot encoded labels of the test set. Outputs:

The evaluate method returns a tuple containing the loss value and the specified metrics during the compilation of the model. In this case, you are interested in the accuracy.

In [115…
```
# Evaluate the Model
loss, accuracy = model.evaluate(X_test_scaled, y_test_one_hot)
print(f'Accuracy on the test set: {accuracy * 100:.2f}%')
```

```
471/471 [==============================] - 1s 1ms/step - loss: 0.0063 - accuracy:
0.9996
Accuracy on the test set: 99.96%
```

In [95]:
```python
plt.plot(history.history['accuracy'], label='Training Accuracy')
# Uncomment the line below if you have validation data
# plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```



In [96]:
```python
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 128)               8960

 dense_1 (Dense)             (None, 64)                8256

 dense_2 (Dense)             (None, 32)                2080

 dense_3 (Dense)             (None, 4)                 132

=================================================================
Total params: 19428 (75.89 KB)
Trainable params: 19428 (75.89 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

This information can be useful for understanding the internal parameters of your neural network, especially during the training process. The weights represent the strength of connections between neurons, and the biases allow the model to capture patterns and relationships in the data.

Extracting Weights and Biases:

model.layers[0].get_weights() returns a list containing the weights and biases of the first layer. model.layers[0].get_weights()[0] extracts the weights. model.layers[0].get_weights()[1] extracts the biases.

In [97]:
```python
# Get the weights of the first layer
weights = model.layers[0].get_weights()[0]
# Get the biases of the first layer
biases = model.layers[0].get_weights()[1]
print("Weights of the first layer\n")
print(weights)
print("\nBiases of the first level\n")
print(biases)
```

```
Weights of the first layer

[[ 0.46848583 -0.18607074 -0.5455124  ...  0.03239947 -0.5302831
  -0.06399445]
 [-0.03054515 -0.16996475 -0.05956086 ...  0.05958569  0.01632309
   0.02799055]
 [-0.03788987 -0.09450658 -0.00382218 ...  0.17748445  0.06884288
   0.02540101]
 ...
 [-0.22449996  0.039084   -0.1937931  ...  0.07263734  0.05527788
   0.05911088]
 [-0.2370843  -0.35716784  0.07542636 ...  0.11076396  0.08335129
  -0.02965326]
 [-0.29569137 -0.27955738  0.04989143 ... -0.21279298  0.08283481
   0.12173562]]

Biases of the first level

[ 0.19528754  0.19796793  0.03983392  0.16874222  0.00661214  0.07347823
 -0.03719745 -0.09429593  0.03676533 -0.05721852  0.12064415  0.08303937
  0.1643042  -0.00934983 -0.04512561  0.18841523  0.18908131  0.16308562
  0.10737846  0.00175962 -0.05867509 -0.03127384  0.07537002 -0.00330109
 -0.08985464  0.18910514  0.09246244  0.09894574 -0.03823512  0.12077427
  0.18819714 -0.00172306  0.00122479  0.07690366  0.1824104   0.13397451
  0.10045917  0.23538135  0.19358419  0.28064474  0.00094311  0.27843174
  0.12239853 -0.05936019 -0.01281102  0.15952076  0.19801283 -0.07177682
  0.11830488  0.11628141  0.01020171  0.01357711 -0.03670569  0.13088153
  0.19450758  0.11039422  0.12575667  0.07133302  0.17936927  0.21017496
 -0.0236094   0.17501047  0.15452494 -0.0320926   0.16595687 -0.07427726
  0.15850575  0.17095959 -0.06469747  0.09485559  0.04154339 -0.00525209
  0.02028094  0.19164741  0.2662655   0.05977794  0.1268835   0.13148999
  0.04891406 -0.03415893  0.21198069  0.14846185  0.09295031  0.12961897
  0.10200744  0.17340864  0.08279346 -0.03887123 -0.02157377  0.13171262
 -0.03243947 -0.05615916  0.09338974  0.07999895  0.15725173  0.07982496
  0.17357734  0.10434949  0.3369604  -0.04317952 -0.00103304  0.04424524
 -0.05244878  0.1500974   0.19304645  0.19939995  0.04254669  0.10320267
  0.0665808   0.10898437 -0.00975554  0.12899508  0.12389743  0.15108027
 -0.07390274  0.10638418  0.10892602 -0.01425621  0.10564844  0.06308206
  0.14231287  0.26285344  0.2580639   0.2055598   0.15120374  0.09531008
  0.18788302  0.04762898]
```

This process helps you understand which features contribute more or less to the model's predictions based on the learned weights in the first layer. Features with higher importance scores are considered more influential in making predictions.

In [116…

```
# Calculate feature importance scores by taking the absolute sum of weights for eac
feature_importance = abs(weights).sum(axis=1)
# Normalize the scores to make them comparable
feature_importance /= feature_importance.sum()
# Printing feature importance
print(feature_importance)
```

```
[0.03909452 0.0081053  0.00810598 0.00819447 0.00765113 0.01214892
 0.01407641 0.01228763 0.01219119 0.01109537 0.0152094  0.0153222
 0.01420546 0.0146049  0.01137726 0.01141478 0.01282163 0.0139439
 0.01102014 0.01430057 0.01396684 0.01222073 0.01719929 0.01497674
 0.01508307 0.01775998 0.01525527 0.01331144 0.01624338 0.01320891
 0.0155398  0.01441157 0.01377455 0.01618857 0.01595647 0.01709406
 0.01384991 0.01602863 0.01379579 0.01613057 0.01115484 0.01577337
 0.01544549 0.01638397 0.01215703 0.01666734 0.01298027 0.01256215
 0.01211488 0.01636898 0.01455528 0.01405308 0.01720833 0.01532549
 0.01475263 0.01639021 0.01544288 0.01685438 0.01575327 0.01591116
 0.01549044 0.01398954 0.01632228 0.01465239 0.01420402 0.01629254
 0.01437008 0.01302689 0.01463007]
```

In [117…

```
import pandas as pd
from sklearn.preprocessing import OneHotEncoder
```

In [118…

```
unseen_data = pd.read_csv(r"C:\Users\junai\OneDrive - Middlesex University\Applied
```

encoder = OneHotEncoder(sparse=False, handle_unknown='ignore') creates an instance of the OneHotEncoder class.

sparse=False ensures that the encoded result is a dense array, and handle_unknown='ignore' allows the encoder to handle unknown categories gracefully.

encoder.fit_transform(unseen_data[['StudentId']]) fits the encoder to the 'StudentId' column in the unseen data and transforms it into a one-hot encoded representation.

The result is stored in the variable student_ids_encoded. This array now contains the one-hot encoded representation of the 'StudentId' column in the unseen data.

In [101…

```
# Use the same encoder that you used during training
encoder = OneHotEncoder(sparse=False, handle_unknown='ignore')
student_ids_encoded = encoder.fit_transform(unseen_data[['StudentId']])
```

```
C:\Users\junai\anaconda3\Lib\site-packages\sklearn\preprocessing\_encoders.py:972:
FutureWarning: `sparse` was renamed to `sparse_output` in version 1.2 and will be
removed in 1.4. `sparse_output` is ignored unless you leave `sparse` to its defaul
t value.
  warnings.warn(
```

In [119…

```
# Check the number of features in the encoded data
print(f'Number of features in encoded data: {student_ids_encoded.shape[1]}')
```

```
Number of features in encoded data: 10
```

Set the Expected Number of Features:

num_features_expected = 69 specifies the number of features expected by the model.

Check and Adjust:

if student_ids_encoded.shape[1] < num_features_expected: checks if the encoded student IDs have fewer features than expected. If true, it adds zero columns to the right of the existing data to make up the difference using np.zeros. If false, it truncates the extra columns on the right to match the expected number of features.

In [103...

```
# Assuming your model expects 128 features, adjust the encoded data accordingly
num_features_expected = 69
if student_ids_encoded.shape[1] < num_features_expected:
    # If the encoded data has fewer features, add zero columns to make up the diffe
    adjusted_student_ids_encoded = np.hstack([student_ids_encoded, np.zeros((studer
else:
    # If the encoded data has more features, truncate the extra columns
    adjusted_student_ids_encoded = student_ids_encoded[:, :num_features_expected]
```

This code uses the trained model to make predictions on the adjusted and encoded student IDs. The model.predict function takes the adjusted student IDs as input and produces predictions as output. The predictions are then printed to the console. The output is a probability distribution across different classes for each student ID, indicating the model's confidence in each class.

In [104...

```
# Now, you can use your model to make predictions
predictions = model.predict(adjusted_student_ids_encoded)
print(predictions)
```

```
1/1 [==============================] - 0s 76ms/step
[[5.2982719e-05 1.0949338e-01 8.7676823e-01 1.3685446e-02]
 [9.0356416e-06 6.2821336e-02 9.3044215e-01 6.7275264e-03]
 [5.1900506e-06 5.0959170e-02 9.4199926e-01 7.0364103e-03]
 [2.2682711e-05 8.9942887e-02 9.0162337e-01 8.4110536e-03]
 [4.8387819e-06 4.7429185e-02 9.4631624e-01 6.2497715e-03]
 [2.0982118e-04 1.4011358e-01 8.4223586e-01 1.7440693e-02]
 [9.9529070e-06 6.5448932e-02 9.2792058e-01 6.6205352e-03]
 [1.3478639e-05 9.7656198e-02 8.9422774e-01 8.1026116e-03]
 [1.0798721e-04 1.4069888e-01 8.4607989e-01 1.3113267e-02]
 [3.8732394e-15 5.9092231e-10 8.0390346e-06 9.9999201e-01]]
```

In [105...

```
predicted_classes = np.argmax(predictions, axis=1)
```

In [106...

```
# Map class indices to grades
grade_mapping = {
    0: 'Fail',
    1: 'Grade 1st',
    2: 'Grade 2nd',
    3: 'Grade 3rd'
}
```

In [107...

```
# Map predicted class indices to grades
predicted_grades = [grade_mapping[idx] for idx in predicted_classes]
```

In [108...

```
# Create a DataFrame with StudentId and predicted grades
predicted_data = pd.DataFrame({'StudentId': unseen_data['StudentId'], 'PredictedGra
```

In [109...

```
# Display the predicted data
print(predicted_data)
```

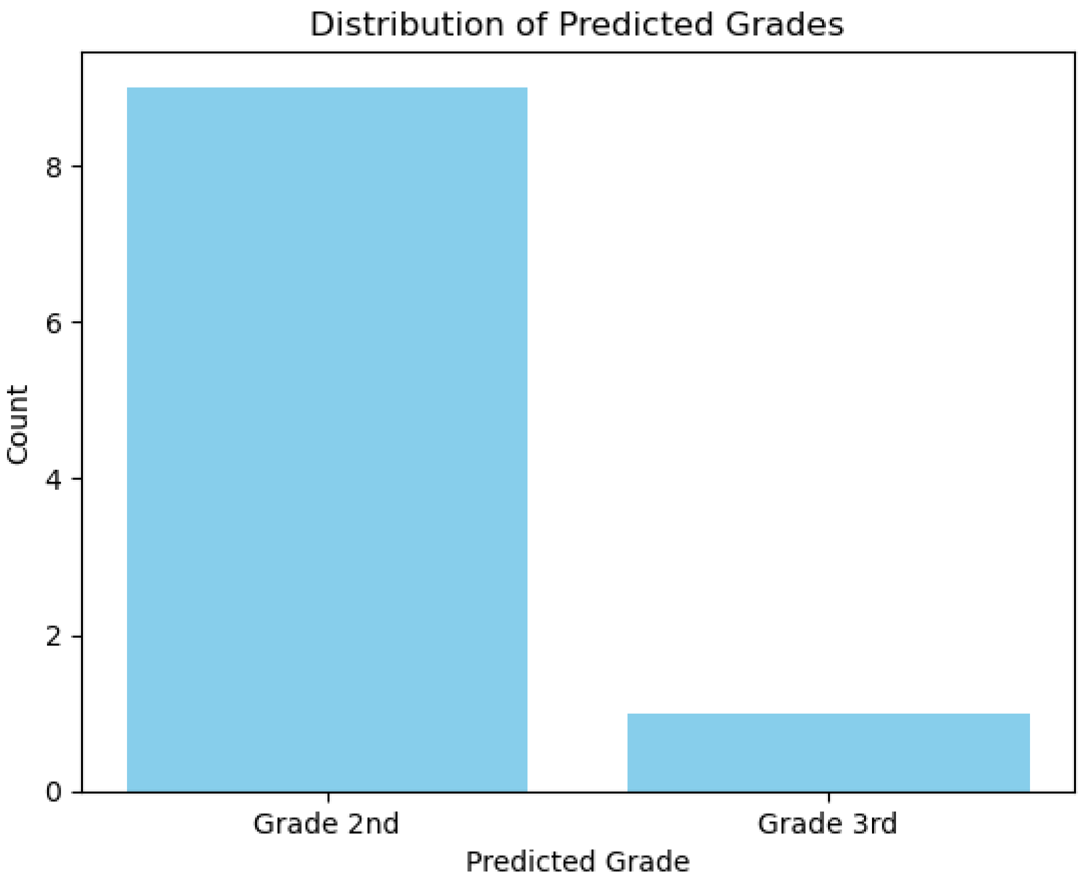
```
  StudentId PredictedGrade
0      aca3      Grade 2nd
1      4f2c      Grade 2nd
2      295e      Grade 2nd
3      d1d7      Grade 2nd
4      6cd6      Grade 2nd
5      c0a8      Grade 2nd
6      2e3f      Grade 2nd
7      cad7      Grade 2nd
8      ade7      Grade 2nd
9      05cf      Grade 3rd
```

In [110…

```
# Count the occurrences of each predicted grade
grade_counts = predicted_data['PredictedGrade'].value_counts()
```

In [111…

```
# Plot the bar chart
plt.bar(grade_counts.index, grade_counts.values, color='skyblue')
plt.xlabel('Predicted Grade')
plt.ylabel('Count')
plt.title('Distribution of Predicted Grades')
plt.show()
```



In [ ]:

In [ ]: