NNDL-LAB3

Name: Kushal Sourav

Regno: 2347125

Data Preprocessing

In [3]:
```
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.preprocessing.image import ImageDataGenerator

#
(x_train, y_train), (x_test, y_test) = cifar10.load_data()


x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0


y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

datagen = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True
)

datagen.fit(x_train)


train_data, test_data = (x_train, y_train), (x_test, y_test)
```

Network Architecture Design , Activation Functions

In [7]:
```
from tensorflow.keras.layers import Activation

model = Sequential()

model.add(Flatten(input_shape=(32, 32, 3)))

model.add(Dense(512))
model.add(Activation('relu'))

model.add(Dense(256))
model.add(Activation('tanh'))

model.add(Dense(10, activation='softmax'))
model.summary()
```

Model: "sequential_3"

Layer (type)	Output Shape	Param #
flatten_3 (Flatten)	(None, 3072)	0
dense_9 (Dense)	(None, 512)	1,573,376
activation (Activation)	(None, 512)	0
dense_10 (Dense)	(None, 256)	131,328
activation_1 (Activation)	(None, 256)	0
dense_11 (Dense)	(None, 10)	2,570

Total params: 1,707,274 (6.51 MB)

Trainable params: 1,707,274 (6.51 MB)

Non-trainable params: 0 (0.00 B)

Justification:

Input Layer:

- Flatten the 32x32x3 image to a 1D vector (3072) so it can be processed by fully connected layers.

Hidden Layer 1 (512 Neurons, ReLU):

- 512 neurons to capture complex features.
- ReLU activation to avoid vanishing gradients and speed up learning.

Hidden Layer 2 (256 Neurons, tanh):

- 256 neurons to reduce complexity and focus on finer details.
- tanh activation for smoother gradient flow and centered outputs (-1 to 1).

Output Layer (10 Neurons, Softmax):

- 10 neurons for each CIFAR-10 class.
- Softmax activation to output class probabilities for multi-class classification.

Role of Activation Functions in Backpropagation:

- Activation functions introduce non-linearity into the network, allowing it to learn from complex patterns in the data.
- In backpropagation, gradients are computed from the loss and propagated backward through the layers. ReLU allows gradients to flow when the input is positive, while tanh provides smoother gradient changes.

- If the activation function saturates , gradients can become very small (vanishing gradient problem), hindering learning. ReLU addresses this by providing gradients that don't vanish for positive inputs

4. Loss Function and Optimizer:

Categorical Cross-Entropy:

- **This is the standard loss function for multi-class classification tasks.**
- **It compares the predicted probability distribution with the true distribution (one-hot encoded labels).**

Mean Squared Error (MSE):

- **Usually used for regression tasks, but we will compare it here for classification.**
- **MSE is the average squared difference between the predicted and true values.**

Hinge Loss:

- **Used for "maximum-margin" classification, primarily in Support Vector Machines (SVMs).**
- **Hinge loss focuses on maximizing the margin between predicted and true classes.**

In [8]:
```
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accur

model.compile(optimizer='adam', loss='mean_squared_error', metrics=['accuracy'])

model.compile(optimizer='adam', loss='hinge', metrics=['accuracy'])
```

Optimizer: Adam

- Adam (Adaptive Moment Estimation) is a popular choice because it adjusts the learning rate for each parameter dynamically based on the first and second moments of gradients.
- It combines the advantages of both SGD and RMSProp, making it robust and efficient for many problems.

Learning Rate:

- The learning rate controls how large the steps taken during optimization are.
- If the learning rate is too high, the network may not converge, bouncing around the minima.
- If the learning rate is too low, training can be too slow and may get stuck in local minima.

Code for Adam Optimizer with Learning Rate:

In [9]:
```
from tensorflow.keras.optimizers import Adam

adam_optimizer = Adam(learning_rate=0.001)
```

```
modelCompiled = model.compile(optimizer=adam_optimizer, loss='categorical_crosse
```

How does the choice of optimizer and learning rate influence the convergence of the network? How would you adjust the learning rate if the model is not converging properly?

- **Optimizer:** Adam automatically adjusts the learning rate during training, speeding up convergence by reducing oscillations. SGD might take longer without momentum.
- **Learning Rate:** If the learning rate is too high, the model might overshoot the optimal solution; if too low, it converges slowly.

Adjusting Learning Rate if Model Doesn't Converge:

- start with a larger learning rate and reduce it if the model struggles to converge.
- Use a learning rate scheduler to decrease the rate during training if the loss plateaus.

5: Training the Model

we train the model using backpropagation, which involves the following:

- Computing the loss by comparing predictions with the actual labels.
- Calculating gradients of the loss with respect to the model weights using backpropagation.
- Updating the weights in the direction of reducing the loss, scaled by the learning rate.

In [10]:
```
from tensorflow.keras.callbacks import LearningRateScheduler


def lr_schedule(epoch):
    initial_lr = 0.001
    drop = 0.5
    epochs_drop = 10
    lr = initial_lr * (drop ** (epoch // epochs_drop))
    return lr

lr_scheduler = LearningRateScheduler(lr_schedule)

history = model.fit(datagen.flow(x_train, y_train, batch_size=64),
                    epochs=50,
                    validation_data=(x_test, y_test),
                    callbacks=[lr_scheduler],
                    verbose=1)
```

```
Epoch 1/50
c:\Users\USER\AppData\Local\Programs\Python\Python311\Lib\site-packages\keras\src
\trainers\data_adapters\py_dataset_adapter.py:122: UserWarning: Your `PyDataset`
class should call `super().__init__(**kwargs)` in its constructor. `**kwargs` can
include `workers`, `use_multiprocessing`, `max_queue_size`. Do not pass these arg
uments to `fit()`, as they will be ignored.
  self._warn_if_super_not_called()
```

```
782/782 ━━━━━━━━━━━━━━━━━━━━ 49s 60ms/step - accuracy: 0.2206 - loss: 2.1385 - va
l_accuracy: 0.3482 - val_loss: 1.8089 - learning_rate: 0.0010
Epoch 2/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 41s 52ms/step - accuracy: 0.3267 - loss: 1.8553 - va
l_accuracy: 0.3876 - val_loss: 1.7067 - learning_rate: 0.0010
Epoch 3/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 41s 52ms/step - accuracy: 0.3694 - loss: 1.7454 - va
l_accuracy: 0.4262 - val_loss: 1.6131 - learning_rate: 0.0010
Epoch 4/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 42s 54ms/step - accuracy: 0.3948 - loss: 1.6743 - va
l_accuracy: 0.4372 - val_loss: 1.5643 - learning_rate: 0.0010
Epoch 5/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 43s 55ms/step - accuracy: 0.4127 - loss: 1.6358 - va
l_accuracy: 0.4406 - val_loss: 1.5519 - learning_rate: 0.0010
Epoch 6/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 45s 57ms/step - accuracy: 0.4256 - loss: 1.6016 - va
l_accuracy: 0.4660 - val_loss: 1.4938 - learning_rate: 0.0010
Epoch 7/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 43s 54ms/step - accuracy: 0.4286 - loss: 1.5879 - va
l_accuracy: 0.4557 - val_loss: 1.5282 - learning_rate: 0.0010
Epoch 8/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 42s 54ms/step - accuracy: 0.4389 - loss: 1.5643 - va
l_accuracy: 0.4722 - val_loss: 1.4762 - learning_rate: 0.0010
Epoch 9/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 40s 51ms/step - accuracy: 0.4442 - loss: 1.5520 - va
l_accuracy: 0.4638 - val_loss: 1.4928 - learning_rate: 0.0010
Epoch 10/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 45s 57ms/step - accuracy: 0.4439 - loss: 1.5407 - va
l_accuracy: 0.4557 - val_loss: 1.5092 - learning_rate: 0.0010
Epoch 11/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 47s 59ms/step - accuracy: 0.4641 - loss: 1.4893 - va
l_accuracy: 0.5087 - val_loss: 1.3920 - learning_rate: 5.0000e-04
Epoch 12/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 46s 58ms/step - accuracy: 0.4780 - loss: 1.4547 - va
l_accuracy: 0.4916 - val_loss: 1.4259 - learning_rate: 5.0000e-04
Epoch 13/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 42s 53ms/step - accuracy: 0.4823 - loss: 1.4458 - va
l_accuracy: 0.4981 - val_loss: 1.4088 - learning_rate: 5.0000e-04
Epoch 14/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 49s 62ms/step - accuracy: 0.4825 - loss: 1.4393 - va
l_accuracy: 0.4991 - val_loss: 1.3963 - learning_rate: 5.0000e-04
Epoch 15/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 46s 58ms/step - accuracy: 0.4811 - loss: 1.4373 - va
l_accuracy: 0.5016 - val_loss: 1.3865 - learning_rate: 5.0000e-04
Epoch 16/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 42s 54ms/step - accuracy: 0.4890 - loss: 1.4252 - va
l_accuracy: 0.5103 - val_loss: 1.3652 - learning_rate: 5.0000e-04
Epoch 17/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 40s 51ms/step - accuracy: 0.4869 - loss: 1.4235 - va
l_accuracy: 0.5096 - val_loss: 1.3692 - learning_rate: 5.0000e-04
Epoch 18/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 40s 51ms/step - accuracy: 0.4993 - loss: 1.4107 - va
l_accuracy: 0.5119 - val_loss: 1.3731 - learning_rate: 5.0000e-04
Epoch 19/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 41s 52ms/step - accuracy: 0.4936 - loss: 1.4147 - va
l_accuracy: 0.5111 - val_loss: 1.3645 - learning_rate: 5.0000e-04
Epoch 20/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 42s 53ms/step - accuracy: 0.4983 - loss: 1.3982 - va
l_accuracy: 0.5153 - val_loss: 1.3679 - learning_rate: 5.0000e-04
Epoch 21/50
```

```
782/782 ━━━━━━━━━━━━━━━━━━━━ 39s 49ms/step - accuracy: 0.5130 - loss: 1.3666 - va
l_accuracy: 0.5196 - val_loss: 1.3380 - learning_rate: 2.5000e-04
Epoch 22/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 39s 50ms/step - accuracy: 0.5122 - loss: 1.3593 - va
l_accuracy: 0.5157 - val_loss: 1.3409 - learning_rate: 2.5000e-04
Epoch 23/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 41s 52ms/step - accuracy: 0.5147 - loss: 1.3588 - va
l_accuracy: 0.5254 - val_loss: 1.3285 - learning_rate: 2.5000e-04
Epoch 24/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 42s 53ms/step - accuracy: 0.5131 - loss: 1.3518 - va
l_accuracy: 0.5253 - val_loss: 1.3259 - learning_rate: 2.5000e-04
Epoch 25/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 45s 57ms/step - accuracy: 0.5187 - loss: 1.3475 - va
l_accuracy: 0.5296 - val_loss: 1.3152 - learning_rate: 2.5000e-04
Epoch 26/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 42s 54ms/step - accuracy: 0.5224 - loss: 1.3380 - va
l_accuracy: 0.5265 - val_loss: 1.3220 - learning_rate: 2.5000e-04
Epoch 27/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 43s 54ms/step - accuracy: 0.5173 - loss: 1.3531 - va
l_accuracy: 0.5283 - val_loss: 1.3184 - learning_rate: 2.5000e-04
Epoch 28/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 44s 57ms/step - accuracy: 0.5218 - loss: 1.3362 - va
l_accuracy: 0.5268 - val_loss: 1.3225 - learning_rate: 2.5000e-04
Epoch 29/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 42s 54ms/step - accuracy: 0.5241 - loss: 1.3341 - va
l_accuracy: 0.5277 - val_loss: 1.3231 - learning_rate: 2.5000e-04
Epoch 30/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 50s 64ms/step - accuracy: 0.5251 - loss: 1.3285 - va
l_accuracy: 0.5285 - val_loss: 1.3080 - learning_rate: 2.5000e-04
Epoch 31/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 46s 59ms/step - accuracy: 0.5342 - loss: 1.3056 - va
l_accuracy: 0.5367 - val_loss: 1.3013 - learning_rate: 1.2500e-04
Epoch 32/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 40s 51ms/step - accuracy: 0.5304 - loss: 1.3091 - va
l_accuracy: 0.5358 - val_loss: 1.3057 - learning_rate: 1.2500e-04
Epoch 33/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 46s 59ms/step - accuracy: 0.5303 - loss: 1.3086 - va
l_accuracy: 0.5316 - val_loss: 1.3119 - learning_rate: 1.2500e-04
Epoch 34/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 43s 55ms/step - accuracy: 0.5359 - loss: 1.2994 - va
l_accuracy: 0.5381 - val_loss: 1.2946 - learning_rate: 1.2500e-04
Epoch 35/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 42s 54ms/step - accuracy: 0.5395 - loss: 1.3042 - va
l_accuracy: 0.5396 - val_loss: 1.2951 - learning_rate: 1.2500e-04
Epoch 36/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 45s 57ms/step - accuracy: 0.5373 - loss: 1.2998 - va
l_accuracy: 0.5358 - val_loss: 1.3005 - learning_rate: 1.2500e-04
Epoch 37/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 39s 50ms/step - accuracy: 0.5374 - loss: 1.2960 - va
l_accuracy: 0.5381 - val_loss: 1.2958 - learning_rate: 1.2500e-04
Epoch 38/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 40s 51ms/step - accuracy: 0.5372 - loss: 1.3013 - va
l_accuracy: 0.5353 - val_loss: 1.2977 - learning_rate: 1.2500e-04
Epoch 39/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 36s 46ms/step - accuracy: 0.5375 - loss: 1.3031 - va
l_accuracy: 0.5367 - val_loss: 1.3007 - learning_rate: 1.2500e-04
Epoch 40/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 39s 50ms/step - accuracy: 0.5411 - loss: 1.2973 - va
l_accuracy: 0.5419 - val_loss: 1.2951 - learning_rate: 1.2500e-04
Epoch 41/50
```

```
782/782 ━━━━━━━━━━━━━━━━━━━━ 42s 54ms/step - accuracy: 0.5451 - loss: 1.2828 - va
l_accuracy: 0.5440 - val_loss: 1.2859 - learning_rate: 6.2500e-05
Epoch 42/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 40s 51ms/step - accuracy: 0.5429 - loss: 1.2853 - va
l_accuracy: 0.5375 - val_loss: 1.2862 - learning_rate: 6.2500e-05
Epoch 43/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 40s 51ms/step - accuracy: 0.5484 - loss: 1.2759 - va
l_accuracy: 0.5408 - val_loss: 1.2840 - learning_rate: 6.2500e-05
Epoch 44/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 43s 55ms/step - accuracy: 0.5444 - loss: 1.2880 - va
l_accuracy: 0.5413 - val_loss: 1.2865 - learning_rate: 6.2500e-05
Epoch 45/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 40s 50ms/step - accuracy: 0.5406 - loss: 1.2815 - va
l_accuracy: 0.5423 - val_loss: 1.2829 - learning_rate: 6.2500e-05
Epoch 46/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 41s 53ms/step - accuracy: 0.5422 - loss: 1.2837 - va
l_accuracy: 0.5415 - val_loss: 1.2821 - learning_rate: 6.2500e-05
Epoch 47/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 41s 53ms/step - accuracy: 0.5464 - loss: 1.2794 - va
l_accuracy: 0.5473 - val_loss: 1.2818 - learning_rate: 6.2500e-05
Epoch 48/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 40s 51ms/step - accuracy: 0.5443 - loss: 1.2796 - va
l_accuracy: 0.5408 - val_loss: 1.2818 - learning_rate: 6.2500e-05
Epoch 49/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 45s 58ms/step - accuracy: 0.5504 - loss: 1.2725 - va
l_accuracy: 0.5413 - val_loss: 1.2876 - learning_rate: 6.2500e-05
Epoch 50/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 45s 58ms/step - accuracy: 0.5381 - loss: 1.2869 - va
l_accuracy: 0.5452 - val_loss: 1.2769 - learning_rate: 6.2500e-05
```

How Backpropagation Updates Weights:

- **Forward Pass:** The model makes a prediction using the current weights.
- **Loss Calculation:** The loss function calculates the error between predictions and actual labels.
- **Gradient Calculation:** Gradients of the loss are computed w.r.t. each weight using backpropagation.
- **Weight Update:** The optimizer updates weights in the opposite direction of the gradient, scaled by the learning rate.

Role of Learning Rate in Backpropagation:

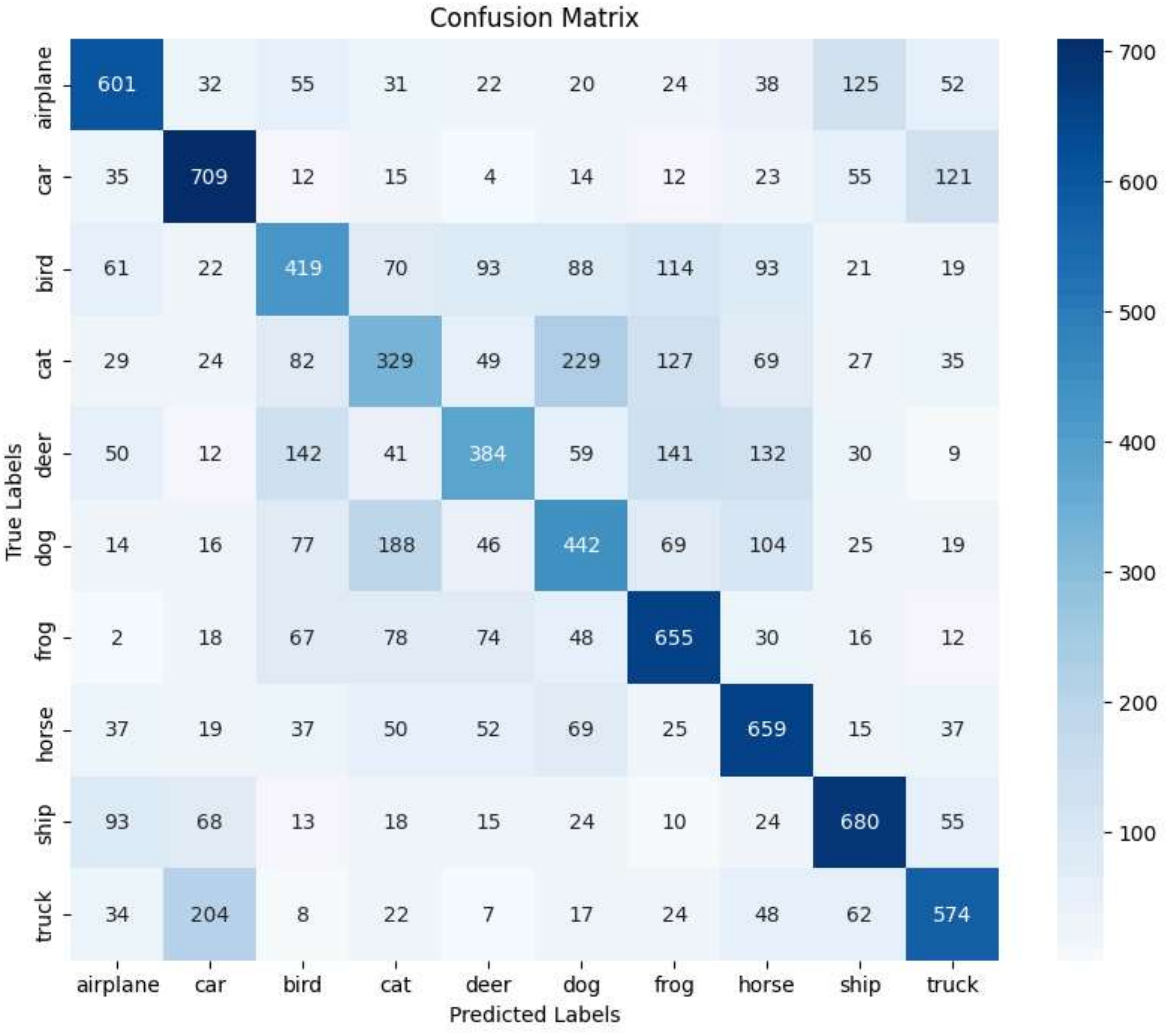
The learning rate determines the step size during weight updates. A higher learning rate results in larger updates and faster training but risks overshooting minima, while a lower rate ensures stable convergence but can slow down training.

6. Model Evaluation

In [13]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_sc
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

class_names = ['airplane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',

y_pred = model.predict(x_test)
```

```
y_pred_classes = np.argmax(y_pred, axis=1)
y_true = np.argmax(y_test, axis=1)

accuracy = accuracy_score(y_true, y_pred_classes)
print(f'Accuracy: {accuracy:.4f}')

precision = precision_score(y_true, y_pred_classes, average='micro')
recall = recall_score(y_true, y_pred_classes, average='micro')
f1 = f1_score(y_true, y_pred_classes, average='micro')

print(f'Precision: {precision:.4f}')
print(f'Recall: {recall:.4f}')
print(f'F1-Score: {f1:.4f}')

conf_matrix = confusion_matrix(y_true, y_pred_classes)


plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=class_na
plt.ylabel('True Labels')
plt.xlabel('Predicted Labels')
plt.title('Confusion Matrix')
plt.show()
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
Accuracy: 0.5452
Precision: 0.5452
Recall: 0.5452
F1-Score: 0.5452
```



Explanation of the Metrics:

- **Accuracy:** The proportion of correct predictions out of the total predictions.
- **Precision:** The proportion of true positive predictions out of all positive predictions. It measures how precise the model is when it predicts a certain class.
- **Recall:** The proportion of true positives out of all actual positive instances. It indicates how well the model captures all relevant instances.
- **F1-Score:** The harmonic mean of precision and recall, which balances both metrics.
- **Confusion Matrix:** A matrix showing actual vs. predicted class labels, providing insights into where the model is misclassifying.

How can you further improve model performance if the accuracy is low?

- **Data Augmentation:** Use random flips, rotations, and shifts with ImageDataGenerator to increase data diversity.
- **Model Architecture:** Add more layers or neurons, or switch to CNNs for better handling of image data.
- **Regularization:** Apply Dropout layers and L2 regularization to reduce overfitting.
- **Learning Rate Tuning:** Adjust the learning rate, possibly lower it, for better convergence.
- **Early Stopping:** Stop training when validation loss stops improving to avoid overfitting.
- **Ensemble Methods:** Combine predictions from multiple models to boost accuracy

7. Optimization Strategies:

- **Early Stopping:** Halt training when validation performance deteriorates to prevent overfitting.
- **Learning Rate Scheduling:** Adjust the learning rate during training for smoother convergence and to escape local minima.
- **Weight Initialization:** Start with small random weights to prevent saturation of activation functions and improve convergence speed.

Why is weight initialization important, and how does it impact the convergence of your network?

- Proper weight initialization helps avoid issues like vanishing/exploding gradients, leading to faster and more stable convergence of the network.