In [36]:
```python
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import cifar10
from sklearn.model_selection import train_test_split
import tensorflow as tf
```

In [2]:
```python
#load the dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 ━━━━━━━━━━━━━━━━━━━━ 11s 0us/step
```

In [3]:
```python
#Combine the train and test sets for a new 80-20 split
x_combined = np.concatenate((x_train, x_test), axis=0)
y_combined = np.concatenate((y_train, y_test), axis=0)
```

In [10]:
```python
#Dislay 5 sample images with labels
labels = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship','truck']
plt.figure(figsize=(10, 2))
for i in range(5):
    plt.subplot(1, 5, i+1)
    plt.imshow(x_combined[i])
    plt.title(labels[int(y_combined[i])])
    plt.axis('off')
plt.tight_layout()
plt.show()
```

```
/tmp/ipython-input-10-3047913360.py:7: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is d
eprecated, and will error in future. Ensure you extract a single element from your array before performing this
operation. (Deprecated NumPy 1.25.)
  plt.title(labels[int(y_combined[i])])
```

In [11]:
```python
#Print the shape of the dataset and count unique labels
print("Shape of the dataset:", x_combined.shape)
print("Shape of the labels:", y_combined.shape)
print("Number of unique labels:", len(np.unique(y_combined)))
print("Unique labels:", np.unique(y_combined))
```

```
Shape of the dataset: (60000, 32, 32, 3)
Shape of the labels: (60000, 1)
Number of unique labels: 10
Unique labels: [0 1 2 3 4 5 6 7 8 9]
```

In [12]:
```python
#Normalise the pixel value to [0,1]
x_combined = x_combined.astype('float32') / 255.0
```

In [13]:
```python
#Split the train and test to (80%,20%)
x_train, x_test, y_train, y_test = train_test_split(x_combined, y_combined, test_size=0.2, random_state=42)
```

In [14]:
```python
print("Shape of the train set:", x_train.shape)
print("Shape of the test set:", x_test.shape)
```

```
Shape of the train set: (48000, 32, 32, 3)
Shape of the test set: (12000, 32, 32, 3)
```

In [16]:
```python
#CNN model architecture
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.utils import to_categorical
```

In [17]:
```python
#convert labels to one-hot encoded vectors
y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)
```

In [21]:
```python
#build a simple CNN MODEL
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    MaxPooling2D((2, 2)),
    Dropout(0.25),
    Conv2D(64, (3, 3), activation='relu'),
```

```
    MaxPooling2D((2, 2)),
    Dropout(0.25),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(10, activation='softmax')
])

#Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
#Display model summary
model.summary()
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
conv2d_4 (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d_4 (MaxPooling2D)	(None, 15, 15, 32)	0
dropout_6 (Dropout)	(None, 15, 15, 32)	0
conv2d_5 (Conv2D)	(None, 13, 13, 64)	18,496
max_pooling2d_5 (MaxPooling2D)	(None, 6, 6, 64)	0
dropout_7 (Dropout)	(None, 6, 6, 64)	0
flatten_2 (Flatten)	(None, 2304)	0
dense_4 (Dense)	(None, 128)	295,040
dropout_8 (Dropout)	(None, 128)	0
dense_5 (Dense)	(None, 10)	1,290

Total params: 315,722 (1.20 MB)

Trainable params: 315,722 (1.20 MB)

Non-trainable params: 0 (0.00 B)

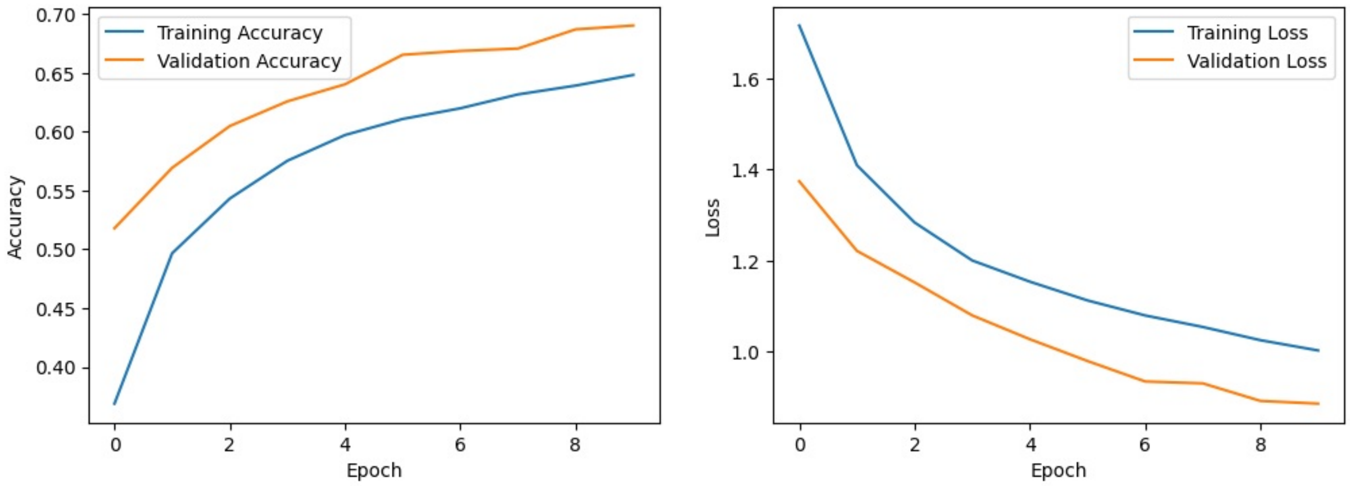
In [20]:
```
#Insights
#1.Well-structured Cnn: Conv->Pool->Dropout repeated twice, followed by Dense layers
#2.parameter-Efficient:315k parameters- suitable for CIFAR-10(smallimages)
#3.Output layer has 10 units with softmax(correct for CIFAR-10's 10 classes)
```

In [22]:
```
#Train the model
history = model.fit(x_train, y_train, epochs=10, batch_size=64, validation_data=(x_test, y_test))
```

```
Epoch 1/10
750/750 ━━━━━━━━━━━━━━━━━━━━ 66s 85ms/step - accuracy: 0.2846 - loss: 1.9258 - val_accuracy: 0.5178 - val_loss:
1.3734
Epoch 2/10
750/750 ━━━━━━━━━━━━━━━━━━━━ 81s 84ms/step - accuracy: 0.4853 - loss: 1.4360 - val_accuracy: 0.5691 - val_loss:
1.2206
Epoch 3/10
750/750 ━━━━━━━━━━━━━━━━━━━━ 64s 86ms/step - accuracy: 0.5361 - loss: 1.2991 - val_accuracy: 0.6046 - val_loss:
1.1514
Epoch 4/10
750/750 ━━━━━━━━━━━━━━━━━━━━ 81s 85ms/step - accuracy: 0.5760 - loss: 1.2051 - val_accuracy: 0.6256 - val_loss:
1.0790
Epoch 5/10
750/750 ━━━━━━━━━━━━━━━━━━━━ 64s 85ms/step - accuracy: 0.5937 - loss: 1.1549 - val_accuracy: 0.6401 - val_loss:
1.0266
Epoch 6/10
750/750 ━━━━━━━━━━━━━━━━━━━━ 83s 86ms/step - accuracy: 0.6074 - loss: 1.1164 - val_accuracy: 0.6652 - val_loss:
0.9788
Epoch 7/10
750/750 ━━━━━━━━━━━━━━━━━━━━ 80s 83ms/step - accuracy: 0.6178 - loss: 1.0825 - val_accuracy: 0.6684 - val_loss:
0.9339
Epoch 8/10
750/750 ━━━━━━━━━━━━━━━━━━━━ 84s 86ms/step - accuracy: 0.6324 - loss: 1.0483 - val_accuracy: 0.6704 - val_loss:
0.9297
Epoch 9/10
750/750 ━━━━━━━━━━━━━━━━━━━━ 80s 84ms/step - accuracy: 0.6397 - loss: 1.0221 - val_accuracy: 0.6867 - val_loss:
0.8912
Epoch 10/10
750/750 ━━━━━━━━━━━━━━━━━━━━ 62s 83ms/step - accuracy: 0.6499 - loss: 0.9927 - val_accuracy: 0.6900 - val_loss:
0.8852
```

In [23]:
```
#Insights
#we choose 10 epoch because after this layers we got a signs of overfitting so we decided to drop epoch to 10
```

In [24]:
```
#Plot Accuracy and loss
plt.figure(figsize=(12, 4))
#Accuracy
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

#Loss
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

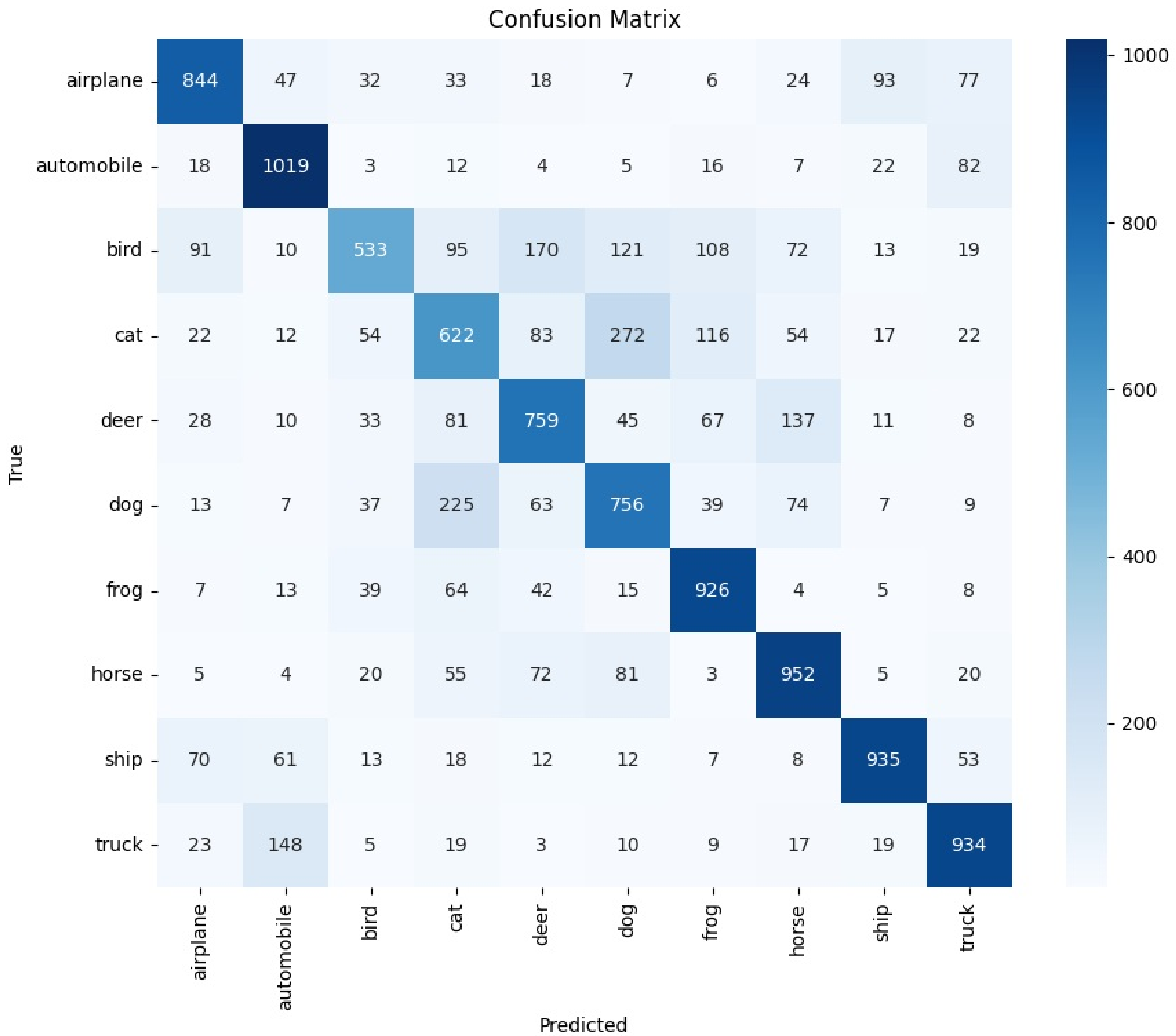


In [25]:
```
#Evaluate the trained model
test_loss, test_acc = model.evaluate(x_test, y_test)
print('Test accuracy:', test_acc)
```

```
375/375 ━━━━━━━━━━━━━━━━━━━━ 5s 13ms/step - accuracy: 0.6916 - loss: 0.8893
Test accuracy: 0.6899999976158142
```

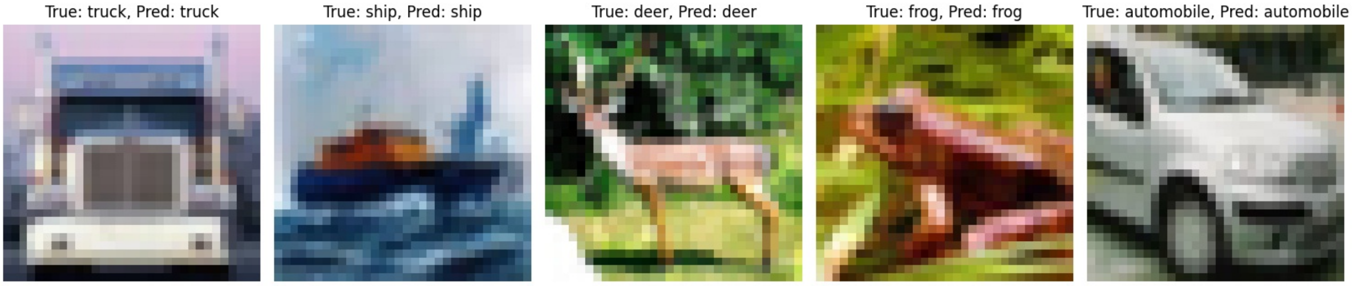
In [26]:
```
#Confusion Matrix and classification report
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns

#Get model predictions
y_pred = model.predict(x_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_test_classes = np.argmax(y_test, axis=1)

#Classification report
print(classification_report(y_test_classes, y_pred_classes))

#Confusion matrix
cm = confusion_matrix(y_test_classes, y_pred_classes)

#plot confusion matrix
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=labels, yticklabels=labels)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

```
375/375 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step
              precision    recall  f1-score   support

           0       0.75      0.71      0.73      1181
           1       0.77      0.86      0.81      1188
           2       0.69      0.43      0.53      1232
           3       0.51      0.49      0.50      1274
           4       0.62      0.64      0.63      1179
           5       0.57      0.61      0.59      1230
           6       0.71      0.82      0.77      1123
           7       0.71      0.78      0.74      1217
           8       0.83      0.79      0.81      1189
           9       0.76      0.79      0.77      1187

    accuracy                           0.69     12000
   macro avg       0.69      0.69      0.69     12000
weighted avg       0.69      0.69      0.69     12000
```



In [27]:
```
#Insights
#1. we got an accuracy of 69
#2. from confusion matrix we can see out predictions
```

In [29]:
```
#Plot
#Identify correct and incorrect predictions
correct_preds = np.where(y_pred_classes == y_test_classes)[0]
incorrect_preds = np.where(y_pred_classes != y_test_classes)[0]

#Plot 5 correctly classified images
plt.figure(figsize=(15, 5))
for i in range(5):
    plt.subplot(1, 5, i+1)
    plt.imshow(x_test[correct_preds[i]])
    plt.title(f'True: {labels[y_test_classes[correct_preds[i]]]}, Pred: {labels[y_pred_classes[correct_preds[i]
    plt.axis('off')

plt.tight_layout()
plt.show()
```



In [30]:
```
#Top 5 incorrectly classified images
plt.figure(figsize=(15, 5))
for i in range(5):
    plt.subplot(1, 5, i+1)
    plt.imshow(x_test[incorrect_preds[i]])
    plt.title(f'True: {labels[y_test_classes[incorrect_preds[i]]]}, Pred: {labels[y_pred_classes[incorrect_pred:
    plt.axis('off')

plt.tight_layout()
plt.show()
```

In [34]:
```
#To improve model
def build_and_train_model(optimizer, epochs=12, batch_size=64):
    model = Sequential([
        Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
        MaxPooling2D((2, 2)),
        Dropout(0.25),


        Conv2D(64, (3, 3), activation='relu'),
        MaxPooling2D((2, 2)),
        Dropout(0.25),


        Flatten(),
        Dense(128, activation='relu'),
        Dropout(0.5),
        Dense(10, activation='softmax')
    ])

    model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])
    history = model.fit(x_train, y_train, epochs=epochs, batch_size=batch_size, validation_data=(x_test, y_test

    test_loss, test_acc = model.evaluate(x_test, y_test)
    print(f'Test accuracy with {optimizer}: {test_acc}')

    return history,test_acc
```

In [39]:
```
#run with different optimizers
optimizers={
    'Adam':tf.keras.optimizers.Adam(),
    'SGD':tf.keras.optimizers.SGD(),
    'RMSprop':tf.keras.optimizers.RMSprop()

}
results={}
for name,opt in optimizers.items():
    history,acc=build_and_train_model(opt)
    results[name]=acc
```

```
Epoch 1/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 68s 89ms/step - accuracy: 0.3008 - loss: 1.8891 - val_accuracy: 0.5120 - val_loss:
1.3632
Epoch 2/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 80s 86ms/step - accuracy: 0.4868 - loss: 1.4217 - val_accuracy: 0.5867 - val_loss:
1.1750
Epoch 3/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 83s 88ms/step - accuracy: 0.5430 - loss: 1.2814 - val_accuracy: 0.6208 - val_loss:
1.0856
```

```
Epoch 4/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 82s 88ms/step - accuracy: 0.5810 - loss: 1.1920 - val_accuracy: 0.6452 - val_loss:
1.0219
Epoch 5/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 81s 87ms/step - accuracy: 0.5947 - loss: 1.1397 - val_accuracy: 0.6566 - val_loss:
0.9747
Epoch 6/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 80s 85ms/step - accuracy: 0.6164 - loss: 1.0964 - val_accuracy: 0.6688 - val_loss:
0.9439
Epoch 7/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 64s 85ms/step - accuracy: 0.6265 - loss: 1.0710 - val_accuracy: 0.6758 - val_loss:
0.9220
Epoch 8/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 65s 87ms/step - accuracy: 0.6409 - loss: 1.0331 - val_accuracy: 0.6799 - val_loss:
0.9093
Epoch 9/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 79s 83ms/step - accuracy: 0.6544 - loss: 0.9880 - val_accuracy: 0.6942 - val_loss:
0.8813
Epoch 10/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 63s 84ms/step - accuracy: 0.6546 - loss: 0.9851 - val_accuracy: 0.7018 - val_loss:
0.8601
Epoch 11/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 84s 87ms/step - accuracy: 0.6683 - loss: 0.9544 - val_accuracy: 0.6852 - val_loss:
0.8916
Epoch 12/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 64s 85ms/step - accuracy: 0.6674 - loss: 0.9540 - val_accuracy: 0.7082 - val_loss:
0.8447
375/375 ━━━━━━━━━━━━━━━━━━━━ 5s 14ms/step - accuracy: 0.7083 - loss: 0.8441
Test accuracy with <keras.src.optimizers.adam.Adam object at 0x788983838cd0>: 0.7082499861717224
Epoch 1/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 65s 86ms/step - accuracy: 0.1325 - loss: 2.2787 - val_accuracy: 0.2496 - val_loss:
2.0588
Epoch 2/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 81s 84ms/step - accuracy: 0.2437 - loss: 2.0542 - val_accuracy: 0.3185 - val_loss:
1.9199
Epoch 3/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 84s 86ms/step - accuracy: 0.2953 - loss: 1.9238 - val_accuracy: 0.3629 - val_loss:
1.7846
Epoch 4/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 80s 84ms/step - accuracy: 0.3390 - loss: 1.8089 - val_accuracy: 0.4011 - val_loss:
1.6952
Epoch 5/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 82s 84ms/step - accuracy: 0.3715 - loss: 1.7284 - val_accuracy: 0.4218 - val_loss:
1.6093
Epoch 6/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 84s 85ms/step - accuracy: 0.3955 - loss: 1.6629 - val_accuracy: 0.4565 - val_loss:
1.5366
Epoch 7/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 66s 88ms/step - accuracy: 0.4158 - loss: 1.6039 - val_accuracy: 0.4717 - val_loss:
1.4806
Epoch 8/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 78s 83ms/step - accuracy: 0.4317 - loss: 1.5713 - val_accuracy: 0.4866 - val_loss:
1.4392
Epoch 9/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 83s 84ms/step - accuracy: 0.4529 - loss: 1.5206 - val_accuracy: 0.4932 - val_loss:
1.4275
Epoch 10/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 63s 84ms/step - accuracy: 0.4631 - loss: 1.4947 - val_accuracy: 0.5202 - val_loss:
1.3753
Epoch 11/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 64s 86ms/step - accuracy: 0.4701 - loss: 1.4726 - val_accuracy: 0.5225 - val_loss:
1.3625
Epoch 12/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 82s 86ms/step - accuracy: 0.4816 - loss: 1.4455 - val_accuracy: 0.5290 - val_loss:
1.3381
375/375 ━━━━━━━━━━━━━━━━━━━━ 6s 15ms/step - accuracy: 0.5253 - loss: 1.3408
Test accuracy with <keras.src.optimizers.sgd.SGD object at 0x7889be0d2650>: 0.5289999842643738
Epoch 1/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 66s 86ms/step - accuracy: 0.2953 - loss: 1.9259 - val_accuracy: 0.5207 - val_loss:
1.3499
Epoch 2/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 83s 87ms/step - accuracy: 0.4837 - loss: 1.4506 - val_accuracy: 0.5706 - val_loss:
1.2156
Epoch 3/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 80s 85ms/step - accuracy: 0.5384 - loss: 1.3067 - val_accuracy: 0.6172 - val_loss:
1.1173
Epoch 4/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 64s 85ms/step - accuracy: 0.5796 - loss: 1.2010 - val_accuracy: 0.6301 - val_loss:
1.0592
Epoch 5/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 84s 87ms/step - accuracy: 0.5981 - loss: 1.1521 - val_accuracy: 0.6533 - val_loss:
1.0201
Epoch 6/12
```

```
750/750 ━━━━━━━━━━━━━━━━━━━━ 84s 89ms/step - accuracy: 0.6141 - loss: 1.1098 - val_accuracy: 0.6597 - val_loss:
1.0207
Epoch 7/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 79s 86ms/step - accuracy: 0.6290 - loss: 1.0789 - val_accuracy: 0.6678 - val_loss:
0.9761
Epoch 8/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 82s 86ms/step - accuracy: 0.6401 - loss: 1.0496 - val_accuracy: 0.6447 - val_loss:
1.0299
Epoch 9/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 64s 85ms/step - accuracy: 0.6471 - loss: 1.0350 - val_accuracy: 0.6720 - val_loss:
0.9424
Epoch 10/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 83s 87ms/step - accuracy: 0.6483 - loss: 1.0289 - val_accuracy: 0.6404 - val_loss:
1.0699
Epoch 11/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 64s 85ms/step - accuracy: 0.6592 - loss: 1.0143 - val_accuracy: 0.6900 - val_loss:
0.9521
Epoch 12/12
750/750 ━━━━━━━━━━━━━━━━━━━━ 82s 86ms/step - accuracy: 0.6580 - loss: 1.0011 - val_accuracy: 0.7001 - val_loss:
0.9029
375/375 ━━━━━━━━━━━━━━━━━━━━ 4s 12ms/step - accuracy: 0.6983 - loss: 0.9039
Test accuracy with <keras.src.optimizers.rmsprop.RMSprop object at 0x788983f47890>: 0.700083315372467
```

In [40]:
```
#performance comparison table
import pandas as pd
results_df=pd.DataFrame(list(results.items()),columns=['Optimizer','Test Accuracy'])
results_df
```

Out[40]:

	Optimizer	Test Accuracy
0	Adam	0.708250
1	SGD	0.529000
2	RMSprop	0.700083

In [41]:
```
#Insights
#1.adam generally gives the best accuracy for CNNs on CIFAR-10 due to its adaptive learning rate
#2.SGD lags in performance
#3.RMSprop is also performing well
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js