SWS3009 Lab 3 Introduction to Deep Learning

Name:	**YANG RUNKANG**
Name:	WEN SIJIE

This lab should be done by both Deep Learning members of the team. Please ensure that you fill in the names of **both** team members in the spaces above. Answer **all** your questions on **this Python Notebook.**

Submission Instructions

SUBMISSION DEADLINE: Thursday July 4 2024, 2359 hours (11.59 pm). Folder will close by 00:15 hours on July 5 afterwhich no submission will be allowed.

Please submit this Python notebook to Canvas on the deadline provided.

Marks will be awarded as follows:

0 marks: No/empty/Non-English submission

1 mark : Poor submission

2 marks: Acceptable submission

3 marks: Good submission

1. Introduction

We will achieve the following objectives in this lab:

```
1. An understanding of the practical limitations of using dense networks in complex tasks
2. Hands-on experience in building a deep learning neural network to solve a relatively complex task.
```

Each step may take a long time to run. You and your partner may want to work out how to do things simultaneously, but please do not miss out on any learning opportunities.

2. Submission Instructions

Please submit your answer book to Canvas by the deadline.

3. Creating a Dense Network for CIFAR-10

We will now begin building a neural network for the CIFAR-10 dataset. The CIFAR-10 dataset consists of 50,000 32x32x3 (32x32 pixels, RGB channels) training images and 10,000 testing images (also 32x32x3), divided into the following 10 categories:

```
1. Airplane
2. Automobile
3. Bird
4. Cat
5. Deer
6. Dog
7. Frog
8. Horse
9. Ship
10. Truck
```

In the first two parts of this lab we will create a classifier for the CIFAR-10 dataset.

3.1 Loading the Dataset

We begin firstly by creating a Dense neural network for CIFAR-10. The code below shows how we load the CIFAR-10 dataset:

In [32]:
```python
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.datasets import cifar10

def load_cifar10():
    (train_x, train_y), (test_x, test_y) = cifar10.load_data()
    train_x = train_x.reshape(train_x.shape[0], 3072) # Question 1
    test_x = test_x.reshape(test_x.shape[0], 3072) # Question 1
    train_x = train_x.astype('float32')
    test_x = test_x.astype('float32')
    train_x /= 255.0
    test_x /= 255.0
    ret_train_y = to_categorical(train_y,10)
    ret_test_y = to_categorical(test_y, 10)

    return (train_x, ret_train_y), (test_x, ret_test_y)


(train_x, train_y), (test_x, test_y) = load_cifar10()
```

Question 1

Explain what the following two statements do, and where the number "3072" came from:

```
train_x = train_x.reshape(train_x.shape[0], 3072) # Question 1
test_x = test_x.reshape(test_x.shape[0], 3072) # Question 1
```

Train_x and test_x are arrays where each image has the shape (number_of_samples, 32, 32, 3). This means there are a number of images, each being 32x32 pixels with 3 color channels (RGB). The .reshape method changes the shape of these arrays. train_x.shape[0] and test_x.shape[0] give the number of images in the datasets. The number 3072 is used to flatten each image from a 32x32x3 matrix into a single vector of length 3072. This is necessary because dense neural networks work with 1D vectors rather than 3D images.

3.2 Building the MLP Classifier

In the code box below, create a new fully connected (dense) multilayer perceptron classifier for the CIFAR-10 dataset. To begin with, create a network with one hidden layer of 1024 neurons, using the SGD optimizer. You should output the training and validation accuracy at every epoch, and train for 50 epochs:

In [33]:
```
"""
Write your code to build an MLP with one hidden layer of 1024 neurons,
with an SGD optimizer. Train for 50 epochs, and output the training and
validation accuracy at each epoch.
"""
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.regularizers import l2

model = Sequential()
model.add(Dense(1024, input_shape=(3072,), activation='relu'))
# model.add(Dense(512, activation='relu'))
model.add(Dense(10, activation='softmax'))

sgd = SGD(learning_rate=0.001, momentum=0.9)
model.compile(optimizer=sgd, loss='categorical_crossentropy', metrics=['accuracy'])

history = model.fit(train_x, train_y, epochs=50, validation_data=(test_x, test_y), batch_size=32, verbose=2)

final_train_accuracy = history.history['accuracy'][-1]
final_val_accuracy = history.history['val_accuracy'][-1]
print(f"Final training accuracy: {final_train_accuracy}")
print(f"Final validation accuracy: {final_val_accuracy}")
```

```
Epoch 1/50
1563/1563 - 8s - 5ms/step - accuracy: 0.3458 - loss: 1.8454 - val_accuracy: 0.3821 - val_loss: 1.7319
Epoch 2/50
1563/1563 - 7s - 5ms/step - accuracy: 0.4173 - loss: 1.6666 - val_accuracy: 0.4288 - val_loss: 1.6168
Epoch 3/50
1563/1563 - 8s - 5ms/step - accuracy: 0.4430 - loss: 1.5900 - val_accuracy: 0.4599 - val_loss: 1.5509
Epoch 4/50
1563/1563 - 7s - 5ms/step - accuracy: 0.4634 - loss: 1.5337 - val_accuracy: 0.4515 - val_loss: 1.5468
Epoch 5/50
1563/1563 - 8s - 5ms/step - accuracy: 0.4804 - loss: 1.4926 - val_accuracy: 0.4753 - val_loss: 1.4959
Epoch 6/50
1563/1563 - 8s - 5ms/step - accuracy: 0.4955 - loss: 1.4549 - val_accuracy: 0.4681 - val_loss: 1.4857
Epoch 7/50
1563/1563 - 8s - 5ms/step - accuracy: 0.5041 - loss: 1.4238 - val_accuracy: 0.4811 - val_loss: 1.4612
Epoch 8/50
1563/1563 - 8s - 5ms/step - accuracy: 0.5146 - loss: 1.3944 - val_accuracy: 0.4850 - val_loss: 1.4547
Epoch 9/50
1563/1563 - 8s - 5ms/step - accuracy: 0.5213 - loss: 1.3673 - val_accuracy: 0.4943 - val_loss: 1.4282
Epoch 10/50
1563/1563 - 8s - 5ms/step - accuracy: 0.5327 - loss: 1.3453 - val_accuracy: 0.5036 - val_loss: 1.4064
Epoch 11/50
1563/1563 - 8s - 5ms/step - accuracy: 0.5388 - loss: 1.3246 - val_accuracy: 0.5013 - val_loss: 1.4095
Epoch 12/50
1563/1563 - 8s - 5ms/step - accuracy: 0.5466 - loss: 1.3032 - val_accuracy: 0.5072 - val_loss: 1.3983
Epoch 13/50
1563/1563 - 8s - 5ms/step - accuracy: 0.5540 - loss: 1.2812 - val_accuracy: 0.5180 - val_loss: 1.3665
Epoch 14/50
1563/1563 - 8s - 5ms/step - accuracy: 0.5605 - loss: 1.2635 - val_accuracy: 0.5205 - val_loss: 1.3543
Epoch 15/50
1563/1563 - 8s - 5ms/step - accuracy: 0.5666 - loss: 1.2450 - val_accuracy: 0.5216 - val_loss: 1.3487
Epoch 16/50
1563/1563 - 8s - 5ms/step - accuracy: 0.5729 - loss: 1.2299 - val_accuracy: 0.5024 - val_loss: 1.3891
Epoch 17/50
1563/1563 - 8s - 5ms/step - accuracy: 0.5791 - loss: 1.2134 - val_accuracy: 0.5153 - val_loss: 1.3658
Epoch 18/50
1563/1563 - 8s - 5ms/step - accuracy: 0.5855 - loss: 1.1964 - val_accuracy: 0.5315 - val_loss: 1.3284
Epoch 19/50
1563/1563 - 8s - 5ms/step - accuracy: 0.5858 - loss: 1.1855 - val_accuracy: 0.5273 - val_loss: 1.3415
Epoch 20/50
1563/1563 - 8s - 5ms/step - accuracy: 0.5948 - loss: 1.1650 - val_accuracy: 0.5219 - val_loss: 1.3506
Epoch 21/50
1563/1563 - 8s - 5ms/step - accuracy: 0.5984 - loss: 1.1531 - val_accuracy: 0.5232 - val_loss: 1.3503
Epoch 22/50
1563/1563 - 8s - 5ms/step - accuracy: 0.6066 - loss: 1.1359 - val_accuracy: 0.5269 - val_loss: 1.3329
Epoch 23/50
1563/1563 - 8s - 5ms/step - accuracy: 0.6099 - loss: 1.1269 - val_accuracy: 0.5280 - val_loss: 1.3293
Epoch 24/50
1563/1563 - 8s - 5ms/step - accuracy: 0.6172 - loss: 1.1091 - val_accuracy: 0.5324 - val_loss: 1.3284
Epoch 25/50
1563/1563 - 8s - 5ms/step - accuracy: 0.6220 - loss: 1.0948 - val_accuracy: 0.5265 - val_loss: 1.3413
Epoch 26/50
1563/1563 - 8s - 5ms/step - accuracy: 0.6263 - loss: 1.0805 - val_accuracy: 0.5357 - val_loss: 1.3320
Epoch 27/50
1563/1563 - 8s - 5ms/step - accuracy: 0.6308 - loss: 1.0654 - val_accuracy: 0.5349 - val_loss: 1.3296
Epoch 28/50
1563/1563 - 8s - 5ms/step - accuracy: 0.6377 - loss: 1.0515 - val_accuracy: 0.5321 - val_loss: 1.3518
Epoch 29/50
1563/1563 - 8s - 5ms/step - accuracy: 0.6408 - loss: 1.0387 - val_accuracy: 0.5332 - val_loss: 1.3333
Epoch 30/50
1563/1563 - 8s - 5ms/step - accuracy: 0.6446 - loss: 1.0262 - val_accuracy: 0.5450 - val_loss: 1.3147
Epoch 31/50
1563/1563 - 8s - 5ms/step - accuracy: 0.6488 - loss: 1.0158 - val_accuracy: 0.5508 - val_loss: 1.2945
Epoch 32/50
1563/1563 - 8s - 5ms/step - accuracy: 0.6549 - loss: 1.0018 - val_accuracy: 0.5358 - val_loss: 1.3328
Epoch 33/50
1563/1563 - 8s - 5ms/step - accuracy: 0.6588 - loss: 0.9858 - val_accuracy: 0.5306 - val_loss: 1.3374
Epoch 34/50
1563/1563 - 8s - 5ms/step - accuracy: 0.6646 - loss: 0.9729 - val_accuracy: 0.5369 - val_loss: 1.3474
Epoch 35/50
1563/1563 - 7s - 5ms/step - accuracy: 0.6708 - loss: 0.9622 - val_accuracy: 0.5348 - val_loss: 1.3358
Epoch 36/50
1563/1563 - 8s - 5ms/step - accuracy: 0.6757 - loss: 0.9457 - val_accuracy: 0.5399 - val_loss: 1.3353
Epoch 37/50
1563/1563 - 8s - 5ms/step - accuracy: 0.6816 - loss: 0.9315 - val_accuracy: 0.5443 - val_loss: 1.3194
Epoch 38/50
1563/1563 - 8s - 5ms/step - accuracy: 0.6864 - loss: 0.9200 - val_accuracy: 0.5399 - val_loss: 1.3474
Epoch 39/50
1563/1563 - 8s - 5ms/step - accuracy: 0.6886 - loss: 0.9083 - val_accuracy: 0.5324 - val_loss: 1.3331
Epoch 40/50
1563/1563 - 8s - 5ms/step - accuracy: 0.6924 - loss: 0.8974 - val_accuracy: 0.5378 - val_loss: 1.3505
Epoch 41/50
1563/1563 - 8s - 5ms/step - accuracy: 0.6953 - loss: 0.8902 - val_accuracy: 0.5303 - val_loss: 1.3720
Epoch 42/50
1563/1563 - 8s - 5ms/step - accuracy: 0.6995 - loss: 0.8745 - val_accuracy: 0.5262 - val_loss: 1.3858
Epoch 43/50
1563/1563 - 7s - 5ms/step - accuracy: 0.7043 - loss: 0.8629 - val_accuracy: 0.5301 - val_loss: 1.3840
```

```
Epoch 44/50
1563/1563 - 8s - 5ms/step - accuracy: 0.7106 - loss: 0.8477 - val_accuracy: 0.5392 - val_loss: 1.3627
Epoch 45/50
1563/1563 - 8s - 5ms/step - accuracy: 0.7143 - loss: 0.8388 - val_accuracy: 0.5422 - val_loss: 1.3510
Epoch 46/50
1563/1563 - 7s - 5ms/step - accuracy: 0.7220 - loss: 0.8207 - val_accuracy: 0.5317 - val_loss: 1.4007
Epoch 47/50
1563/1563 - 7s - 5ms/step - accuracy: 0.7207 - loss: 0.8154 - val_accuracy: 0.5460 - val_loss: 1.3355
Epoch 48/50
1563/1563 - 7s - 5ms/step - accuracy: 0.7271 - loss: 0.8020 - val_accuracy: 0.5361 - val_loss: 1.3958
Epoch 49/50
1563/1563 - 7s - 5ms/step - accuracy: 0.7335 - loss: 0.7882 - val_accuracy: 0.5456 - val_loss: 1.3520
Epoch 50/50
1563/1563 - 7s - 5ms/step - accuracy: 0.7360 - loss: 0.7776 - val_accuracy: 0.5308 - val_loss: 1.4404
Final training accuracy: 0.7359600067138672
Final validation accuracy: 0.5307999849319458
```

Question 2

Complete the following table on the design choices for your MLP:

Hyperparameter	What I used	Why?
Optimizer	SGD	Specified in question
# of hidden layers	1	Specified in question
# of hidden neurons	1024	Specified in question
Hid layer activation	ReLU	they are simple, introduce non-linearity, enable sparse activation, and avoid the vanishing gradient problem
# of output neurons	10	There are 10 classes in the CIFAR-10 dataset
Output activation	Softmax	Softmax activation is used for multi-class classification
learning_rate	0.001	Common initial learning rate for SGD
momentum	0.9	Helps accelerate SGD in the relevant direction and dampen oscillations
loss		Common loss function for multi-class classification

Question 3:

What was your final training accuracy? Validation accuracy? Is there overfitting / underfitting? Explain your answer:

The Final training accuracy is 0.7359600067138672, the Final validation accuracy is 0.5307999849319458. There seems to be overfitting since the training accuracy is high while the validation accuracy is low.

3.3 Experimenting with the MLP

Cut and paste your code from Section 3.2 to the box below (you may need to rename your MLP). Experiment with the number of hidden layers, the number of neurons in each hidden layer, the optimization algorithm, etc. See Keras Optimizers for the types of optimizers and their parameters. **Train for 100 epochs.**

In [35]:
```python
"""
Cut and paste your code from Section 3.2 below, then modify it to get
much better results than what you had earlier. E.g. increase the number of
nodes in the hidden layer, increase the number of hidden layers,
change the optimizer, etc.

Train for 100 epochs.

"""
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam

model = Sequential()
model.add(Dense(2048, input_shape=(3072,), activation='relu'))
# model.add(Dense(1024, activation='relu', input_shape=(3072,)))
model.add(Dense(1024, activation='relu'))
model.add(Dense(512, activation='relu'))
model.add(Dense(256, activation='relu'))
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))

adam = Adam(learning_rate=0.001)
model.compile(optimizer=adam, loss='categorical_crossentropy', metrics=['accuracy'])

history = model.fit(train_x, train_y, epochs=100, validation_data=(test_x, test_y), batch_size=32, verbose=1)
```

```python
final_train_accuracy = history.history['accuracy'][-1]
final_val_accuracy = history.history['val_accuracy'][-1]
print(f"Final training accuracy: {final_train_accuracy}")
print(f"Final validation accuracy: {final_val_accuracy}")
```

```
Epoch 1/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 49s 31ms/step - accuracy: 0.2509 - loss: 2.0488 - val_accuracy: 0.3360 - val_loss: 1.9329
Epoch 2/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.3694 - loss: 1.7483 - val_accuracy: 0.3956 - val_loss: 1.6759
Epoch 3/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 48s 30ms/step - accuracy: 0.4103 - loss: 1.6553 - val_accuracy: 0.4262 - val_loss: 1.6168
Epoch 4/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 48s 30ms/step - accuracy: 0.4268 - loss: 1.5939 - val_accuracy: 0.4481 - val_loss: 1.5498
Epoch 5/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 48s 31ms/step - accuracy: 0.4451 - loss: 1.5481 - val_accuracy: 0.4504 - val_loss: 1.5318
Epoch 6/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 48s 31ms/step - accuracy: 0.4636 - loss: 1.4956 - val_accuracy: 0.4515 - val_loss: 1.5503
Epoch 7/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.4719 - loss: 1.4699 - val_accuracy: 0.4672 - val_loss: 1.4998
Epoch 8/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 51s 32ms/step - accuracy: 0.4807 - loss: 1.4413 - val_accuracy: 0.4634 - val_loss: 1.5170
Epoch 9/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 48s 31ms/step - accuracy: 0.4933 - loss: 1.4088 - val_accuracy: 0.4683 - val_loss: 1.5208
Epoch 10/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 49s 31ms/step - accuracy: 0.5046 - loss: 1.3790 - val_accuracy: 0.4698 - val_loss: 1.4900
Epoch 11/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.5109 - loss: 1.3617 - val_accuracy: 0.4705 - val_loss: 1.4879
Epoch 12/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 48s 31ms/step - accuracy: 0.5205 - loss: 1.3346 - val_accuracy: 0.4746 - val_loss: 1.4932
Epoch 13/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 49s 31ms/step - accuracy: 0.5288 - loss: 1.3140 - val_accuracy: 0.4819 - val_loss: 1.4756
Epoch 14/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 49s 31ms/step - accuracy: 0.5385 - loss: 1.2867 - val_accuracy: 0.4880 - val_loss: 1.4826
Epoch 15/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 48s 30ms/step - accuracy: 0.5464 - loss: 1.2647 - val_accuracy: 0.4937 - val_loss: 1.4825
Epoch 16/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 48s 31ms/step - accuracy: 0.5557 - loss: 1.2369 - val_accuracy: 0.4967 - val_loss: 1.4516
Epoch 17/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 48s 31ms/step - accuracy: 0.5605 - loss: 1.2223 - val_accuracy: 0.4937 - val_loss: 1.4726
Epoch 18/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.5723 - loss: 1.1886 - val_accuracy: 0.4756 - val_loss: 1.5219
Epoch 19/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 48s 31ms/step - accuracy: 0.5799 - loss: 1.1658 - val_accuracy: 0.4794 - val_loss: 1.5630
Epoch 20/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 48s 31ms/step - accuracy: 0.5835 - loss: 1.1577 - val_accuracy: 0.4873 - val_loss: 1.5417
Epoch 21/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.5922 - loss: 1.1215 - val_accuracy: 0.4876 - val_loss: 1.5664
Epoch 22/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 49s 31ms/step - accuracy: 0.5966 - loss: 1.0998 - val_accuracy: 0.4845 - val_loss: 1.5777
Epoch 23/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.6072 - loss: 1.0780 - val_accuracy: 0.4824 - val_loss: 1.5723
Epoch 24/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.6209 - loss: 1.0448 - val_accuracy: 0.4864 - val_loss: 1.5986
Epoch 25/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 48s 31ms/step - accuracy: 0.6295 - loss: 1.0199 - val_accuracy: 0.4746 - val_loss: 1.6578
Epoch 26/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 48s 30ms/step - accuracy: 0.6394 - loss: 0.9941 - val_accuracy: 0.4859 - val_loss: 1.6620
Epoch 27/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.6389 - loss: 0.9784 - val_accuracy: 0.4829 - val_loss: 1.7094
Epoch 28/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 46s 30ms/step - accuracy: 0.6492 - loss: 0.9590 - val_accuracy: 0.4756 - val_loss: 1.7493
Epoch 29/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.6578 - loss: 0.9396 - val_accuracy: 0.4880 - val_loss: 1.7733
Epoch 30/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 46s 29ms/step - accuracy: 0.6689 - loss: 0.9127 - val_accuracy: 0.4712 - val_loss: 1.8152
Epoch 31/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.6698 - loss: 0.9067 - val_accuracy: 0.4882 - val_loss: 1.8023
Epoch 32/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 46s 30ms/step - accuracy: 0.6767 - loss: 0.8891 - val_accuracy: 0.4839 - val_loss: 1.8435
Epoch 33/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 46s 30ms/step - accuracy: 0.6886 - loss: 0.8536 - val_accuracy: 0.4781 - val_loss: 1.9429
Epoch 34/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 46s 29ms/step - accuracy: 0.6912 - loss: 0.8527 - val_accuracy: 0.4756 - val_loss: 1.9261
Epoch 35/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.7029 - loss: 0.8256 - val_accuracy: 0.4850 - val_loss: 1.8902
Epoch 36/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 49s 32ms/step - accuracy: 0.7018 - loss: 0.8183 - val_accuracy: 0.4707 - val_loss: 2.0613
Epoch 37/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.7118 - loss: 0.7885 - val_accuracy: 0.4675 - val_loss: 2.0418
Epoch 38/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.7223 - loss: 0.7720 - val_accuracy: 0.4778 - val_loss: 1.9910
Epoch 39/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.7278 - loss: 0.7528 - val_accuracy: 0.4726 - val_loss: 2.0993
Epoch 40/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.7304 - loss: 0.7425 - val_accuracy: 0.4744 - val_loss: 2.0616
Epoch 41/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.7343 - loss: 0.7395 - val_accuracy: 0.4710 - val_loss: 2.2115
Epoch 42/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 48s 31ms/step - accuracy: 0.7398 - loss: 0.7283 - val_accuracy: 0.4704 - val_loss: 2.2287
Epoch 43/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.7524 - loss: 0.6881 - val_accuracy: 0.4740 - val_loss: 2.2429
```

```
Epoch 44/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 46s 29ms/step - accuracy: 0.7507 - loss: 0.6930 - val_accuracy: 0.4725 - val_loss: 2.3431
Epoch 45/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.7582 - loss: 0.6612 - val_accuracy: 0.4801 - val_loss: 2.2627
Epoch 46/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.7583 - loss: 0.6636 - val_accuracy: 0.4771 - val_loss: 2.2827
Epoch 47/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.7669 - loss: 0.6436 - val_accuracy: 0.4636 - val_loss: 2.4050
Epoch 48/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 48s 31ms/step - accuracy: 0.7699 - loss: 0.6451 - val_accuracy: 0.4746 - val_loss: 2.4115
Epoch 49/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 46s 30ms/step - accuracy: 0.7752 - loss: 0.6260 - val_accuracy: 0.4648 - val_loss: 2.4176
Epoch 50/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 46s 29ms/step - accuracy: 0.7831 - loss: 0.6099 - val_accuracy: 0.4727 - val_loss: 2.4755
Epoch 51/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 46s 29ms/step - accuracy: 0.7846 - loss: 0.6090 - val_accuracy: 0.4674 - val_loss: 2.5776
Epoch 52/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 46s 29ms/step - accuracy: 0.7898 - loss: 0.5859 - val_accuracy: 0.4665 - val_loss: 2.6249
Epoch 53/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 46s 29ms/step - accuracy: 0.7886 - loss: 0.5928 - val_accuracy: 0.4708 - val_loss: 2.4781
Epoch 54/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 46s 30ms/step - accuracy: 0.8003 - loss: 0.5593 - val_accuracy: 0.4595 - val_loss: 2.5049
Epoch 55/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.7936 - loss: 0.5725 - val_accuracy: 0.4667 - val_loss: 2.8473
Epoch 56/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 48s 31ms/step - accuracy: 0.7986 - loss: 0.5594 - val_accuracy: 0.4695 - val_loss: 2.7812
Epoch 57/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 48s 30ms/step - accuracy: 0.7990 - loss: 0.5565 - val_accuracy: 0.4688 - val_loss: 2.7805
Epoch 58/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.8076 - loss: 0.5330 - val_accuracy: 0.4612 - val_loss: 2.8936
Epoch 59/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.8053 - loss: 0.5501 - val_accuracy: 0.4598 - val_loss: 2.8542
Epoch 60/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.8156 - loss: 0.5225 - val_accuracy: 0.4639 - val_loss: 2.9502
Epoch 61/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.8202 - loss: 0.5063 - val_accuracy: 0.4616 - val_loss: 2.9687
Epoch 62/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 48s 30ms/step - accuracy: 0.8212 - loss: 0.5122 - val_accuracy: 0.4600 - val_loss: 2.8646
Epoch 63/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.8257 - loss: 0.4837 - val_accuracy: 0.4658 - val_loss: 2.9333
Epoch 64/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 49s 31ms/step - accuracy: 0.8290 - loss: 0.4804 - val_accuracy: 0.4581 - val_loss: 3.0474
Epoch 65/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 50s 32ms/step - accuracy: 0.8266 - loss: 0.4875 - val_accuracy: 0.4651 - val_loss: 2.8110
Epoch 66/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 48s 31ms/step - accuracy: 0.8378 - loss: 0.4634 - val_accuracy: 0.4595 - val_loss: 2.9936
Epoch 67/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 48s 31ms/step - accuracy: 0.8345 - loss: 0.4677 - val_accuracy: 0.4653 - val_loss: 3.0550
Epoch 68/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.8376 - loss: 0.4608 - val_accuracy: 0.4634 - val_loss: 3.0683
Epoch 69/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.8399 - loss: 0.4498 - val_accuracy: 0.4646 - val_loss: 3.1429
Epoch 70/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.8371 - loss: 0.4697 - val_accuracy: 0.4668 - val_loss: 3.1092
Epoch 71/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 46s 30ms/step - accuracy: 0.8480 - loss: 0.4318 - val_accuracy: 0.4586 - val_loss: 3.0990
Epoch 72/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.8465 - loss: 0.4417 - val_accuracy: 0.4630 - val_loss: 3.3696
Epoch 73/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 46s 30ms/step - accuracy: 0.8497 - loss: 0.4320 - val_accuracy: 0.4536 - val_loss: 3.1646
Epoch 74/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.8478 - loss: 0.4313 - val_accuracy: 0.4616 - val_loss: 3.3456
Epoch 75/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.8544 - loss: 0.4180 - val_accuracy: 0.4679 - val_loss: 3.3116
Epoch 76/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 48s 31ms/step - accuracy: 0.8566 - loss: 0.4159 - val_accuracy: 0.4584 - val_loss: 2.9796
Epoch 77/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 48s 31ms/step - accuracy: 0.8557 - loss: 0.4065 - val_accuracy: 0.4604 - val_loss: 3.1718
Epoch 78/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.8530 - loss: 0.4184 - val_accuracy: 0.4600 - val_loss: 3.2999
Epoch 79/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 50s 32ms/step - accuracy: 0.8535 - loss: 0.4223 - val_accuracy: 0.4588 - val_loss: 3.3378
Epoch 80/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 49s 31ms/step - accuracy: 0.8688 - loss: 0.3789 - val_accuracy: 0.4618 - val_loss: 3.3883
Epoch 81/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 48s 31ms/step - accuracy: 0.8546 - loss: 0.4189 - val_accuracy: 0.4621 - val_loss: 3.3343
Epoch 82/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.8618 - loss: 0.3955 - val_accuracy: 0.4583 - val_loss: 3.6197
Epoch 83/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 46s 30ms/step - accuracy: 0.8622 - loss: 0.3936 - val_accuracy: 0.4620 - val_loss: 3.5825
Epoch 84/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 48s 31ms/step - accuracy: 0.8696 - loss: 0.3789 - val_accuracy: 0.4635 - val_loss: 3.3331
Epoch 85/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 46s 30ms/step - accuracy: 0.8714 - loss: 0.3745 - val_accuracy: 0.4597 - val_loss: 3.6038
Epoch 86/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 46s 29ms/step - accuracy: 0.8698 - loss: 0.3791 - val_accuracy: 0.4592 - val_loss: 3.6030
```

```
Epoch 87/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 46s 30ms/step - accuracy: 0.8695 - loss: 0.3725 - val_accuracy: 0.4592 - val_loss: 3.5405
Epoch 88/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.8689 - loss: 0.3915 - val_accuracy: 0.4587 - val_loss: 3.7449
Epoch 89/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 50s 32ms/step - accuracy: 0.8812 - loss: 0.3499 - val_accuracy: 0.4614 - val_loss: 3.6788
Epoch 90/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 50s 32ms/step - accuracy: 0.8732 - loss: 0.3736 - val_accuracy: 0.4617 - val_loss: 3.6647
Epoch 91/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 49s 31ms/step - accuracy: 0.8724 - loss: 0.3785 - val_accuracy: 0.4569 - val_loss: 3.5190
Epoch 92/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 50s 32ms/step - accuracy: 0.8766 - loss: 0.3659 - val_accuracy: 0.4606 - val_loss: 3.6524
Epoch 93/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 51s 32ms/step - accuracy: 0.8748 - loss: 0.3644 - val_accuracy: 0.4531 - val_loss: 4.1067
Epoch 94/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 50s 32ms/step - accuracy: 0.8898 - loss: 0.3308 - val_accuracy: 0.4604 - val_loss: 3.8145
Epoch 95/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 49s 31ms/step - accuracy: 0.8811 - loss: 0.3531 - val_accuracy: 0.4608 - val_loss: 3.8496
Epoch 96/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 50s 32ms/step - accuracy: 0.8868 - loss: 0.3358 - val_accuracy: 0.4423 - val_loss: 3.6440
Epoch 97/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.8850 - loss: 0.3388 - val_accuracy: 0.4602 - val_loss: 3.6943
Epoch 98/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 46s 29ms/step - accuracy: 0.8860 - loss: 0.3399 - val_accuracy: 0.4582 - val_loss: 3.7256
Epoch 99/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 47s 30ms/step - accuracy: 0.8896 - loss: 0.3289 - val_accuracy: 0.4528 - val_loss: 3.8763
Epoch 100/100
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 48s 31ms/step - accuracy: 0.8839 - loss: 0.3492 - val_accuracy: 0.4658 - val_loss: 3.9429
Final training accuracy: 0.8792399764060974
Final validation accuracy: 0.4657999873161316
```

Question 4:

Complete the following table with your final design (you may add more rows for the # neurons (layer1) etc. to detail how many neurons you have in each hidden layer). Likewise you may replace the learning_rate, momentum etc rows with parameters more appropriate to the optimizer that you have chosen.

Hyperparameter	What I used	Why?
Optimizer	Adam	Adam optimizer generally provides good performance on various tasks
# of hidden layers	5	Increasing the depth of the network can improve its capacity to learn complex patterns
# neurons(layer1)	2048	Increasing the number of neurons increases the model's capacity
Hid layer1 activation	ReLU	they are simple, introduce non-linearity, enable sparse activation, and avoid the vanishing gradient problem
# neurons(layer2)	1024	Increasing the number of neurons increases the model's capacity
Hid layer2 activation	ReLU	they are simple, introduce non-linearity, enable sparse activation, and avoid the vanishing gradient problem
# of output neurons	10	There are 10 classes in the CIFAR-10 dataset
Output activation	Softmax	Softmax activation is used for multi-class classification
learning_rate	0.001	Lower learning rate can provide more stable convergence
momentum	N/A	Not applicable for Adam optimizer
loss	Categorical Crossentropy	Common loss function for multi-class classification

Question 5

What is the final training and validation accuracy that you obtained after 150 epochs. Is there considerable improvement over Section 3.2? Are there still signs of underfitting or overfitting? Explain your answer.

The Final training accuracy is 0.8792399764060974, the Final validation accuracy is 0.4657999873161316. There seems to be still overfitting too, since the training accuracy is high while the validation accuracy is low.

Question 6

Write a short reflection on the practical difficulties of using a dense MLP to classsify images in the CIFAR-10 datasets.

The CIFAR-10 dataset consists of 60,000 32x32 color images in 10 different classes, meaning each image has 3 color channels, resulting in 3072 input features (32x32x3) for a dense MLP. This high dimensionality poses significant challenges. The vast number of input features leads to an explosion in the number of parameters in the network, making it computationally expensive and memory-intensive to train. Each neuron in the hidden layer is connected to all 3072 input features, and with multiple layers, the number of parameters increases exponentially, requiring substantial computational resources and increasing the risk of overfitting, as the model may memorize the training data rather than generalizing to unseen data. Dense

MLPs do not exploit the spatial structure of the images, unlike CNNs, which use convolutional layers to learn spatial hierarchies by focusing on local patterns and gradually combining them to understand more complex features. This lack of spatial awareness makes dense MLPs less effective at capturing the local and hierarchical patterns in the images, leading to poorer performance in image classification tasks. Additionally, the training process of dense MLPs on such high-dimensional data can be challenging due to vanishing and exploding gradient problems. As the error signal propagates back through many layers, it can diminish exponentially, making it difficult for the network to learn, or if the gradients are too large, they can cause numerical instability and hinder convergence.

---

4. Creating a CNN for the MNIST Data Set

In this section we will now create a convolutional neural network (CNN) to classify images in the MNIST dataset that we used in the previous lab. Let's go through each part to see how to do this.

4.1 Loading the MNIST Dataset

As always we will load the MNIST dataset, scale the inputs to between 0 and 1, and convert the Y labels to one-hot vectors. However unlike before we will not flatten the 28x28 image to a 784 element vector, since CNNs can inherently handle 2D data.

In [19]:
```python
from keras.datasets import mnist
from keras.utils import to_categorical

def load_mnist():
    (train_x, train_y),(test_x, test_y) = mnist.load_data()
    train_x = train_x.reshape(train_x.shape[0], 28, 28, 1)
    test_x = test_x.reshape(test_x.shape[0], 28, 28, 1)

    train_x=train_x.astype('float32')
    test_x = test_x.astype('float32')

    train_x /= 255.0
    test_x /= 255.0

    train_y = to_categorical(train_y, 10)
    test_y = to_categorical(test_y, 10)

    return (train_x, train_y), (test_x, test_y)
```

4.2 Building the CNN

We will now build the CNN. Unlike before we will create a function to produce the CNN. We will also look at how to save and load Keras models using "checkpoints", particularly "ModelCheckpoint" that saves the model each epoch.

Let's begin by creating the model. We call os.path.exists to see if a model file exists, and call "load_model" if it does. Otherwise we create a new model.

In [24]:
```python
# load_model loads a model from a hd5 file.
from keras.models import Sequential, load_model
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D
import os

# MODEL_NAME = 'mnist-cnn.hd5'
MODEL_NAME = 'mnist-cnn.keras'


def buildmodel(model_name):
    if os.path.exists(model_name):
        model = load_model(model_name)
    else:
        model = Sequential()
        model.add(Conv2D(32, kernel_size=(5,5),
        activation='relu',
        input_shape=(28, 28, 1), padding='same')) # Question 7

        model.add(MaxPooling2D(pool_size=(2,2), strides=2)) # Question 8
        model.add(Conv2D(64, kernel_size=(5,5), activation='relu'))
        model.add(Conv2D(128, kernel_size=(5,5), activation='relu'))
        model.add(Conv2D(64, kernel_size=(5,5), activation='relu'))
        model.add(MaxPooling2D(pool_size=(2,2), strides=2))
        model.add(Flatten()) # Question 9
        model.add(Dense(1024, activation='relu'))
        model.add(Dropout(0.1))
        model.add(Dense(10, activation='softmax'))

    return model
```

Question 7

The first layer in our CNN is a 2D convolution kernel, shown here:

```
model.add(Conv2D(32, kernel_size=(5,5),
activation='relu',
input_shape=(28, 28, 1), padding='same')) # Question 7
```

Why is the input_shape set to (28, 28, 1)? What does this mean? What does "padding = 'same'" mean?

The input_shape is set to (28, 28, 1) because the MNIST dataset consists of 28x28 pixel grayscale images. The 28, 28 represents the height and width of the images, and 1 represents the single color channel (grayscale).

padding='same' means that the convolutional layer will apply padding to the input images so that the output has the same height and width as the original image. This involves adding zeroes around the borders of the image before applying the convolution operation.

Question 8

The second layer is the MaxPooling2D layer shown below:

```
model.add(MaxPooling2D(pool_size=(2,2), strides=2)) # Question 8
```

What other types of pooling layers are available? What does 'strides = 2' mean?

Other types of pooling layers include AveragePooling2D and GlobalAveragePooling2D.

- AveragePooling2D: Computes the average value for each patch of the feature map.
- GlobalAveragePooling2D: Reduces each feature map to a single value by taking the average over all spatial dimensions.

strides=2 means that the pooling window will move 2 pixels at a time across the input feature map. This reduces the spatial dimensions of the feature map by a factor of 2.

Question 9

What does the "Flatten" layer here do? Why is it needed?

```
model.add(Flatten()) # Question 9
```

The Flatten layer converts the multi-dimensional output of the previous layers into a one-dimensional vector.

It is needed to transform the 2D matrix of features into a 1D vector so that it can be used as input to the fully connected (dense) layers that follow. Fully connected layers require a 1D input.

4.3 Training the CNN

Let's now train the CNN. In this example we introduce the idea of a "callback", which is a routine that Keras calls at the end of each epoch. Specifically we look at two callbacks:

```
1. ModelCheckpoint: When called, Keras saves the model to the specified filename.

2. EarlyStopping: When called, Keras checks if it should stop the training prematurely.
```

Let's look at the code to see how training is done, and how callbacks are used.

In [25]:
```
from keras.optimizers import SGD
from keras.callbacks import EarlyStopping, ModelCheckpoint

def train(model, train_x, train_y, epochs, test_x, test_y, model_name):

    model.compile(optimizer=SGD(learning_rate=0.01, momentum=0.7),
                  loss='categorical_crossentropy', metrics=['accuracy'])

    savemodel = ModelCheckpoint(model_name)
    stopmodel = EarlyStopping(min_delta=0.001, patience=10) # Question 10

    print("Starting training.")

    model.fit(x=train_x, y=train_y, batch_size=32,
    validation_data=(test_x, test_y), shuffle=True,
    epochs=epochs,
    callbacks=[savemodel, stopmodel])

    print("Done. Now evaluating.")
    loss, acc = model.evaluate(x=test_x, y=test_y)
    print("Test accuracy: %3.2f, loss: %3.2f"%(acc, loss))
```

Notice that there isn't very much that is unusual going on; we compile the model with our loss function and optimizer, then call fit, and finally evaluate to look at the final accuracy for the test set. The only thing unusual is the "callbacks" parameter here in the fit function call

```
model.fit(x=train_x, y=train_y, batch_size=32,
validation_data=(test_x, test_y), shuffle=True,
epochs=epochs,
callbacks=[savemodel, stopmodel])
```

Question 10.

What does do the min_delta and patience parameters do in the EarlyStopping callback, as shown below? (2 MARKS)

```
stopmodel = EarlyStopping(min_delta=0.001, patience=10) # Question 10
```

- min_delta: This parameter specifies the minimum change in the monitored quantity (e.g., validation loss) that qualifies as an improvement. If the change in the monitored quantity is less than min_delta, it is not considered as an improvement. In this case, min_delta=0.001 means that only changes in the validation loss greater than 0.001 will be considered as improvements.
- patience: This parameter specifies the number of epochs with no improvement after which training will be stopped. If the monitored quantity does not improve for patience number of epochs, training is halted. Here, patience=10 means that if there is no improvement in the validation loss for 10 consecutive epochs, training will stop.

4.4 Putting it together.

Now let's run the code and see how it goes (Note: To save time we are training for only 5 epochs; we should train much longer to get much better results):

In [26]:
```
    (train_x, train_y),(test_x, test_y) = load_mnist()
    model = buildmodel(MODEL_NAME)
    train(model, train_x, train_y, 5, test_x, test_y, MODEL_NAME)
```

```
Starting training.
Epoch 1/5
c:\Users\Y\AppData\Local\Programs\Python\Python312\Lib\site-packages\keras\src\layers\convolutional\base_conv.py:107: UserW
arning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input
(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 13s 7ms/step - accuracy: 0.7412 - loss: 0.7860 - val_accuracy: 0.9714 - val_loss: 0.0890
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 13s 7ms/step - accuracy: 0.9739 - loss: 0.0811 - val_accuracy: 0.9791 - val_loss: 0.0600
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 13s 7ms/step - accuracy: 0.9850 - loss: 0.0476 - val_accuracy: 0.9847 - val_loss: 0.0459
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 13s 7ms/step - accuracy: 0.9884 - loss: 0.0369 - val_accuracy: 0.9880 - val_loss: 0.0355
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 13s 7ms/step - accuracy: 0.9909 - loss: 0.0284 - val_accuracy: 0.9862 - val_loss: 0.0447
Done. Now evaluating.
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9826 - loss: 0.0604
Test accuracy: 0.99, loss: 0.04
```

Question 11.

Compare the relative advantages and disadvantages of CNN vs. the Dense MLP that you build in sections 3.2 and 3.3. What makes CNNs better (or worse)?

Dense MLPs consist of layers where each neuron is connected to every neuron in the subsequent layer. This dense connectivity allows them to learn complex patterns in the data but at the cost of a large number of parameters, making them computationally expensive and prone to overfitting, especially with high-dimensional input data like images.

CNNs are particularly well-suited for image and spatial data due to their ability to capture spatial hierarchies and local features. This makes CNNs generally better at image classification and object detection tasks compared to Dense MLPs. However, CNNs can be less effective for tasks where spatial relationships are less important or for data that isn't grid-like, where Dense MLPs might still be preferable.

5. Making a CNN for the CIFAR-10 Dataset

Now comes the fun part: Using the example above for creating a CNN for the MNIST dataset, now create a CNN in the box below for the CIFAR-10 dataset. At the end of each epoch save the model to a file called "cifar.hd5" (note: the .hd5 is added automatically for you).

Question 12.

Summarize your design in the table below (the actual coding cell comes after this):

Hyperparameter	What I used	Why?
Optimizer	Adam	Adam optimizer generally provides good performance
Input shape	(32, 32, 3)	CIFAR-10 images are 32x32 pixels with 3 color channels
First layer	Conv2D(32)	Convolutional layer with 32 filters and ReLU activation
Second layer	MaxPooling2D(2,2)	Pooling layer to reduce spatial dimensions
Third layer	Conv2D(64)	Convolutional layer with 64 filters and ReLU activation
Fourth layer	MaxPooling2D(2,2)	Pooling layer to reduce spatial dimensions
Fifth layer	Conv2D(128)	Convolutional layer with 128 filters and ReLU activation
Sixth layer	MaxPooling2D(2,2)	Pooling layer to reduce spatial dimensions
Dense layer	Dense(512)	Fully connected layer with 512 neurons and ReLU activation
Output layer	Dense(10)	Fully connected layer with 10 neurons for 10 classes

In [27]:
```python
"""
Write your code for your CNN for the CIFAR-10 dataset here.

Note: train_x, train_y, test_x, test_y were changed when we called
load_mnist in the previous section. You will now need to call load_cifar10
again.

"""

from keras.models import Sequential, load_model
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D
from keras.optimizers import Adam
from keras.callbacks import EarlyStopping, ModelCheckpoint
import os

# Define the filename to save the model
MODEL_NAME = 'cifar.keras'

# Function to load CIFAR-10 dataset
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.datasets import cifar10

def load_cifar10():
    (train_x, train_y), (test_x, test_y) = cifar10.load_data()
    train_x = train_x.reshape(train_x.shape[0], 32, 32, 3)
    test_x = test_x.reshape(test_x.shape[0], 32, 32, 3)
    train_x = train_x.astype('float32')
    test_x = test_x.astype('float32')
    train_x /= 255.0
    test_x /= 255.0
    train_y = to_categorical(train_y, 10)
    test_y = to_categorical(test_y, 10)

    return (train_x, train_y), (test_x, test_y)

(train_x, train_y), (test_x, test_y) = load_cifar10()

# Function to build the CNN model
```

```
def build_model(model_name):
    if os.path.exists(model_name):
        model = load_model(model_name)
    else:
        model = Sequential()
        model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(32, 32, 3), padding='same'))
        model.add(MaxPooling2D(pool_size=(2, 2), strides=2))
        model.add(Conv2D(64, kernel_size=(3, 3), activation='relu', padding='same'))
        model.add(MaxPooling2D(pool_size=(2, 2), strides=2))
        model.add(Conv2D(128, kernel_size=(3, 3), activation='relu', padding='same'))
        model.add(MaxPooling2D(pool_size=(2, 2), strides=2))
        model.add(Flatten())
        model.add(Dense(512, activation='relu'))
        model.add(Dropout(0.5))
        model.add(Dense(10, activation='softmax'))

    return model

# Function to train the CNN model
def train(model, train_x, train_y, epochs, test_x, test_y, model_name):
    model.compile(optimizer=Adam(learning_rate=0.001), loss='categorical_crossentropy', metrics=['accuracy'])

    savemodel = ModelCheckpoint(model_name, save_best_only=True)
    stopmodel = EarlyStopping(min_delta=0.001, patience=10)

    print("Starting training.")

    model.fit(x=train_x, y=train_y, batch_size=32, validation_data=(test_x, test_y), shuffle=True, epochs=epochs, callback

    print("Done. Now evaluating.")
    loss, acc = model.evaluate(x=test_x, y=test_y)
    print("Test accuracy: %3.2f, loss: %3.2f" % (acc, loss))

# Build and train the model
model = build_model(MODEL_NAME)
train(model, train_x, train_y, 50, test_x, test_y, MODEL_NAME)
```

```
Starting training.
Epoch 1/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 12s 7ms/step - accuracy: 0.3740 - loss: 1.7110 - val_accuracy: 0.5686 - val_loss: 1.2052
Epoch 2/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 11s 7ms/step - accuracy: 0.6226 - loss: 1.0674 - val_accuracy: 0.6832 - val_loss: 0.9077
Epoch 3/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 11s 7ms/step - accuracy: 0.6880 - loss: 0.8853 - val_accuracy: 0.7194 - val_loss: 0.8319
Epoch 4/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 12s 7ms/step - accuracy: 0.7333 - loss: 0.7599 - val_accuracy: 0.7208 - val_loss: 0.8049
Epoch 5/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 12s 7ms/step - accuracy: 0.7620 - loss: 0.6717 - val_accuracy: 0.7374 - val_loss: 0.7648
Epoch 6/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 11s 7ms/step - accuracy: 0.7821 - loss: 0.6054 - val_accuracy: 0.7422 - val_loss: 0.7882
Epoch 7/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 11s 7ms/step - accuracy: 0.8104 - loss: 0.5413 - val_accuracy: 0.7510 - val_loss: 0.7473
Epoch 8/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 11s 7ms/step - accuracy: 0.8277 - loss: 0.4821 - val_accuracy: 0.7482 - val_loss: 0.7792
Epoch 9/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 11s 7ms/step - accuracy: 0.8450 - loss: 0.4342 - val_accuracy: 0.7503 - val_loss: 0.7876
Epoch 10/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 11s 7ms/step - accuracy: 0.8572 - loss: 0.3998 - val_accuracy: 0.7539 - val_loss: 0.8120
Epoch 11/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 11s 7ms/step - accuracy: 0.8733 - loss: 0.3583 - val_accuracy: 0.7484 - val_loss: 0.8768
Epoch 12/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 11s 7ms/step - accuracy: 0.8822 - loss: 0.3270 - val_accuracy: 0.7584 - val_loss: 0.8753
Epoch 13/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 11s 7ms/step - accuracy: 0.8912 - loss: 0.2983 - val_accuracy: 0.7579 - val_loss: 0.8994
Epoch 14/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 11s 7ms/step - accuracy: 0.8996 - loss: 0.2838 - val_accuracy: 0.7342 - val_loss: 0.9973
Epoch 15/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 11s 7ms/step - accuracy: 0.9005 - loss: 0.2717 - val_accuracy: 0.7569 - val_loss: 0.9466
Epoch 16/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 11s 7ms/step - accuracy: 0.9110 - loss: 0.2482 - val_accuracy: 0.7605 - val_loss: 0.9508
Epoch 17/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 11s 7ms/step - accuracy: 0.9141 - loss: 0.2412 - val_accuracy: 0.7575 - val_loss: 0.9931
Done. Now evaluating.
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.7582 - loss: 0.9709
Test accuracy: 0.76, loss: 0.99
```

```
In [ ]:
```

```
In [ ]:
```